A Monitoring Approach for Runtime Service Discovery

Authors: K. Mahbub, G. Spanoudakis, A. Zisman

Affiliation: School of Informatics, City University London

Phone: +44 (0) 20 7040 4042

Fax: +44 (0) 20 7040 0244

Email: {k.mahbub | g.spanoudakis | a.zisman}@soi.city.ac.uk

Abstract. Effective runtime service discovery requires identification of services based on different service characteristics such as structural, behavioral, quality, and contextual characteristics. However, current service registries guarantee services described in terms of structural and sometimes quality characteristics and, therefore, it is not always possible to assume that services in them will have all the characteristics required for effective service discovery. In this paper, we describe a monitor-based runtime service discovery framework called MoRSeD. The framework supports service discovery in both push and pull modes of query execution. The push mode of query execution is performed in parallel to the execution of a service-based system, in a proactive way. Both types of queries are specified in a query language called SerDiQueL that allows the representation of structural, behavioural, quality, and contextual conditions of services to be identified. The framework uses a monitor component to verify if behavioural and contextual conditions in the queries can be satisfied by services, based on translations of these conditions into properties represented in event calculus, and verification of the satisfiability of these properties against services. The monitor is also used to support identification that services participating in a service-based system are unavailable, and identification of changes in the behavioural and contextual characteristics of the services. A prototype implementation of the framework has been developed. The framework has been evaluated in terms of comparison of its performance when using and when not using the monitor component.

Keywords: runtime service discovery, proactive discovery, service monitoring, query

1 Introduction

Runtime service discovery has been recognised as an important aspect to support service-based systems. More specifically, runtime service discovery, also known as *dynamic service discovery*, is concerned with the identification of services that can replace services participating in service-based systems during the execution of these systems. Several approaches have been proposed to support runtime service discovery (see [13][14][29] for example). However, most of these approaches do not consider different characteristics of a service ranging from structural, to behavioural, quality, and contextual aspects when trying to identify the services. Moreover, existing approaches for runtime service discovery support the discovery process in pull mode in a reactive way, where services are identified when there is a need to do so.

There are several situations that may trigger the need for runtime service discovery including, (i) unavailability or malfunctioning of a participating service, (ii) changes in the structure, behaviour, quality, or context characteristics of a participating service, (iii) changes in the context of the service-based system, or (iv) availability of a “better” service due to the provision of a new service or changes in the characteristics of an existing service. Given the above situations and the need to provide better precision when identifying services to replace existing services at runtime, it is necessary to consider different characteristics of the services such as structural, behavioral, quality, or contextual characteristics. However, it is not possible to assume that services will always be described in terms of all the above characteristics. Current approaches for service registries guarantee the existence of structural descriptions of services, typically in the form ofWSDL [2] specifications [19][24][25]. In order to fulfill the need to identify services based on other criteria and not only structural characteristics, it is necessary to have a mechanism to verify the behavioural and contextual characteristics of services even when there are no available behavioural specifications and up-to-date contextual values of distinct aspects of the services (e.g., location, availability, response time) in the registries.

In this paper, we describe a monitor-based runtime service discovery framework (MoRSeD) that we have develop to address the above limitation. This framework extends our previous runtime service discovery framework [78][79][80] with a monitor component to support (i) the identification of service unavailability, (ii) the identification of changes in the behavioural and contextual characteristics of the services participating in a service-based system and services that are candidates for replacing services in the system, and (iii) the evaluation of behavioural and contextual criteria in service discovery queries against candidate services. The monitor verifies the satisfiability of behavioural and contextual properties of the services against messages exchanged between a service-based system and the services deployed by it. The behavioral properties represent the existence, order, dependency, and conditional iterations of functionalities of the services (e.g., all users need to be authenticated before having access to the service); while contextual properties represent contextual values of the various functionalities in a service (e.g., the service should not take more than 10 seconds to display a map of the area where the suer is located). The monitor is based on our previous work [4][6][59] supporting monitoring of behavioural and contextual characteristics of service-based systems.

Our runtime service discovery framework supports identification of services based on service discovery queries that can be executed in a pull and/or push mode. In the push mode of query execution, service discovery is performed in parallel to the execution of the service-based system using pre-subscribed queries and services. The subscribed queries are associated with specific service binding points in the service based system and aim to maintain up-to-date sets of candidate replacement services for these binding points. In pull mode, a query associated with a binding point is executed anytime that the service bound to this point becomes unavailable and, therefore, needs to be replaced with an alternative service. Regardless of the mode of their execution, queries in MoRSeD are specified in an XML-based query language, called SerDiQueL [80] that allows the expression of logical combinations of conditions about various characteristics of services to be identified, namelystructural, behavioural, quality, and contextual conditions.

To illustrate the work described in this paper, and the circumstances in which it is necessary to replace services participating in service-based systems, we present a scenario of a service-based system that will be used throughout the paper.

The service-based system used in this scenario is called *Route-Planner* and allows users to request information from a PDA about optimal routes to be taken when driving. More specifically the system offers services that (a) identify the exact current location of a user, (b) allow users to find an optimal route for a certain location given the exact location of the user by using a *Global Positioning System* (GPS), (c) 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, (d) 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* (RTS), and (e) compute new routes at regular intervals due to traffic changes. The *Route-Planner* system has been implemented as a BPEL2 [7] process and all the services used in the system (i.e., GPS, e-AZ Map, RTS) have been implemented as Axis2 [8] web services.

Consider a user of the *Route-Planner* system trying to find an optimal route from his current position "CP" to destination "DP". However, after invoking *Route-Planner* to find an optimal route to destination "DP", *Route-Planner* fails to provide the optimal route since it cannot access the service that identifies the user's exact current location (the service is unavailable). In this case, it is necessary to identify an alternative service for calculating the user's current location so that *Route-Planner* can continue its execution. After such service is identified and bound to *Route-Planner*, the user is presented with the route to be taken. The user starts following the route, but during the journey the response time of the *Road Traffic Service* becomes very slow as RTS is a free of charge service and there are many applications using this service at the time (there is a change in the context of the service). Thus, it is necessary to identify a service that can provide traffic information with an acceptable response time so that the overall performance of the system is not compromised. Following the identification of an adequate replacement service for RTS and its binding to *Route-Planner*, the user continues its journey. After using the system for a while, however, *Route-Planner* realizes that the battery of the PDA where it runs is running out of power (there is a change in the context of the system environment). As the user does not have any battery charger in the car in order to save battery consumption, it is necessary to identify a service that provides monochrome electronic instead of colored maps, since the display of colored maps consumes more electric power. Assuming that such service is identified and used to replace the *e-AZ Map* service in the system. The user continues the journey and destination "DP" is reached without any more problems. After a couple of days, however, a new service that provides electronic maps in both monochrome and colored formats, at a cheaper price than the one being used by *Route-Planner*, becomes available. Given that a decrease in the cost of using *Route-Planner* system and the possibility of switching between colored and monochromatic maps when necessary are important requirements of the system, the new available service is identified and bound to the system. Hence, the next time that the user accesses *Route-Planner*, he/she will pay less whilstbeing able to see maps in different display modes.

The remainder of this paper is structured as follows. In Section 2 we present the monitor-based runtime service discovery framework with its main components, the service discovery process, and the service monitoring process. In Section 3 we describe how service queries represented in SerDiQueL are translated into monitoring properties. In Section 4 we discuss implementation and evaluation aspects of our work. In Section 5 we give an account of related work. Finally, in Section 6 we discuss concluding remarks and future work.

2 Monitor-based Runtime Service Discovery Framework

Figure 1 shows the overall architecture of the monitor-based runtime service discovery (MoRSeD) framework and its main components. These components are described below.

The **Web Service Interface** supports the interaction between a service-based system (client) and the framework. For this purpose, it exposes interfaces through which client systems can send service discovery queries and create service and query subscriptions in MoRSeD and receive back results of executing these queries.

The **Service Requestor** orchestrates the functionality offered by the other components in the framework. This component prepares the service query submitted to MoRSeD for evaluation; invokes other components of the framework in order to execute a query, organises the results of the query and returns these results to a client application through the web service interface. The Service Requestor also manages query subscriptions for push mode execution, and receives information that can trigger the execution of such queries in the push mode, including (a) information from service registry listeners about services that become available in them, and (b) information from the monitor about the services deployed in a client service-based system or their replacement candidate service that become unavailable, changes in their characteristics and/or context, changes in the context of the client service-based system, and behavioural and contextual matchings of potential candidate services during the execution of a query.

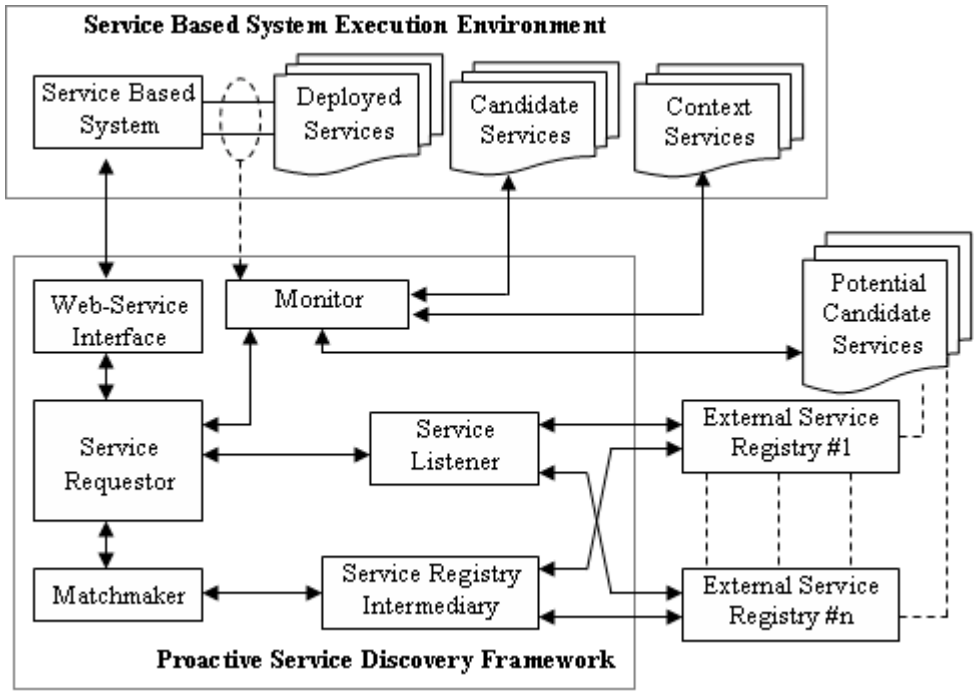

Service Based System Execution Environment
Service Based System
Deployed Services
Candidate Services
Context Services
Web-Service Interface
Monitor
Potential Candidate Services
Service Requestor
Service Listener
External Service Registry #1
Matchmaker
Service Registry Intermediary
External Service Registry #n
Proactive Service Discovery Framework

Fig. 1 Architecture of the proactive service discovery framework

The **Service Matchmaker** is responsible for executing queries. To do this, the service matchmaker parses structural and quality characteristics of a query evaluates them against structural and quality of service specifications, and computes the distances between a query and candidate services based on the evaluation of structural, behavioural, contextual, and quality characteristics of a service (see Sect. 2.1),. The evaluation of the behavioural and contextual characteristics is assisted by the monitor of MoRSeD.

The **Service Registry Intermediary** supports the use of different service registries by providing an interface for accessing services in them. MoRSeD supports services from registries that are based on a faceted service specification scheme developed in the SeCSE project [1]. In this scheme facets are used to specify different aspects of a service (e.g., service inteface, quality, and behaviour). In the framework, we assume services described in terms of structural facets describing the operations of services with their data types using WSDL [2], quality of service facets describing quality aspects of services represented in XML-based schema, and context facets represented in XML format which describe context operations associated with service operations that are executed at runtime in order to generate context values. MoRSeD framework uses the SeCSE service registry as the external service registry. This registry has been implemented using an eXist database [3] backend that is accessed by the service registry intermediary through remote method invocation (RMI).

The **Service Listener** notifies the service requestor about new services that become available and changes in the structural and quality descriptions of existing services in the registry. These notifications are based on subscriptions for structural and quality information made by the service requestor.

The **Monitor** is responsible for identifying services that become unavailable, changes in the behavioural or contextual characteristics of the services deployed in a service-based system and their replacement candidate services, and changes in the context of the service-based system. It is also responsible for verifying whether the behavioural and contextual conditions specified in service discovery queries are satisfied by services.

In the following we describe the service discovery and monitoring processes used by MoRSeD framework.

2.1 Service Discovery Process

MoRSeD can execute service discovery in both pull and push modes. The pull mode of query execution is performed to (a) identify services that may be initially bound to a service-based system, (b) as a first step in the push mode of query execution, (c) due to changes in the context of an application environment, and (d) when a client application requests a service to be identified. On the other hand, the push mode of query execution is performed in parallel to the execution of a service-based system, in a proactive way, in order to identify services due to any of situations (i) to (iv) described in Section 1.

In both pull and push modes of query execution, services are identified based on matching of service discovery queries specified in SerDiQueL [80] against services. A service discovery query represents the characteristics of a service to be identified. SerDiQueL allows for the representation of different criteria, namely: (a) structural, describing the interface of a required service, (b) behavioural, describing the functionality of a required service, and (c) constraints, describing additional conditions for a service. These additional conditions may be concerned with quality characteristics of a service (e.g., the time or cost to execute an operation in a service), or interface or functional characteristics of a service that cannot be described in terms of the structural and behavioural descriptions used in SerDiQueL (e.g., the provider of a service or the receiver of a message). The constraints in a query can be classified as *contextual* and *non-contextual*. A contextual constraint is concerned with information that changes dynamically during the execution of a service-based system or its participating services. A non-contextual constraint is concerned with information that does not change dynamically. The constraints can also be *hard* or *soft*. A hard constraint needs to be satisfied by all candidate services for a query, while a soft constraint does not need to be satisfied by all candidate services but is used to rank a service with respect to a query. A detailed description of SerDiQueL is out of the scope of this paper, but can be found in [80]. In Section 3, we present description of parts of SerDiQueL to facilitate understanding of the material described in this section.

The matching of service discovery queries against services is executed in a two-stage process. The first stage is called *filtering stage*. In this stage, hard non-contextual constraints in a query are evaluated against service specifications and a set of candidate services that comply with these constraints are identified. The matchmaker requests the service registry intermediary to retrieve services from the external service registries that match the hard constraints of a query, when they are present, or to retrieve all the services when the hard constraints are not present in a query. The results of the first-stage matching process are sent to the service requestor.

The second stage in the matching process is a *ranking stage*. This stage is executed in a three sub-stage process based on the computation of partial distances concerned with each sub-stage and the computation of an overall distance that considers all partial distances and weight associated with these distances. In the first sub-stage, structural and behavioural characteristics of a query are evaluated against the candidate services returned by the first stage in the process. In this case, a *structural-behavioural partial distance* is computed for each candidate service with respect to a query. When the query does not have hard constraints, the structural-behavioural partial distances are evaluated for all the services in the registry. In the second sub-stage, soft non-contextual constraints are evaluated against the set of candidate services and a *soft non-*

contextual partial distance is computed for each candidate service. In the third sub-stage, contextual constraints are evaluated against the candidate services and *contextual partial distances* are computed for each candidate service. The overall distance between a query and each candidate service is calculated by the average of all the partial distances. The result of the ranking stage is a set of candidate services that have an overall distance with a query that is below a certain distance threshold. This distance threshold is either specified in the query or has a default value of 0.5. The candidate services that comply with the threshold are sorted by ascending order of distances between them and the query. The results of the second-stage matching process are sent to the service requestor that organises them in the necessary format required by the web-service interface.

In MoRSeD framework, in order to identify services that match a query, it is necessary that the query have at least structural criteria. This is important given that services cannot be identified for an application unless the interface through which the application will use the service is known. If any of the other criteria, or any of their combinations, is not present in a query (e.g., hard, behavioural, non-contextual, and contextual constraints), the respective stage of the matching process is not executed and the overall distance is calculated based on the criteria present in the query.

The structural matching between a query and services is performed by evaluating the structural criteria in a query and structural specifications of services expressed in WSDL [2] by comparing signatures of service and query operations. In this process, an operation in a candidate service matches a query operation only if the two operations have the same number of input and output parameters, the data types of the input parameters of the service operation are super-types of the input parameters of the query operation, and the data types of the output parameters of the service operation are subtypes of the output parameters of the query operation. To evaluate the conditions regarding the parameters of query and service operations, MoRSeD compares graphs representing the data types of the parameters of each operation and the linguistic distances of the names of the parameters. This matching process uses a variant of the VF2 algorithm [82] for detecting graph morphisms that we have developed for static service discovery [84][85]. The structural distance between a query operation and a service operation is computed by the average of the linguistic distance of the names of the operations, the names of the parameters based on WordNet lexicon [83], the distance between the types of input parameters, and the distance between the types of the output parameters. The exact formula for the structural distance and more details of the data type graph construction and graph morphism detection can be found in [79].

The behavioural matching between a query and services is performed by verifying if the behavioural criteria in a query can be satisfied by a service. This verification is executed by the monitor component (see Figure 1) based on requests received by the service requestor for monitoring specific properties. More specifically, the behavioural properties to be monitored are derived from the translation of behavioural criteria in a SerDiQueL query into event calculus (EC) [5] in terms of events and fluents. A detailed description of this translation is presented in Section 3. The satisfiability of properties by the services is verified by the *analyzer* component of the monitor based on invocations of the services by the *service client* component and the events collected for these services by the *event collector* component (see Subsection 2.2 for details about the monitor). The monitor deploys a service client for each service that needs to be monitored. The service client component is responsible for the invocation of services and the generation of runtime events intercepted by the event collector. The event collector component is responsible to gather runtime information during the execution of services and to make this information available during the verification of the different properties. The result of a behavioural matching between a service and a query is a binary value indicating whether a specific service satisfies the behavioural property (value zero) or a service does not satisfy the behavioural property (value one). The services that do not satisfy the behavioural properties are not considered as possible candidate services during the computation of the ranking stage.

Based on the evaluation of the structural criteria and behavioural properties, structural-behavioural partial distances between a query Q and possible candidate services that satisfy the behavioural properties are computed.

The matching of soft non-contextual constraints between a query and services is executed by evaluating the soft non-contextual criteria in a query and service specification facets. More specifically, the soft non-contextual constraint expressions in a query are evaluated against elements in service specifications. The result of this evaluation is a binary value indicating whether a specific constraint is satisfied (value zero) or not (value one). After the evaluation of individual constraints, a partial soft non-contextual distance constraint partial distance is calculated for a query Q and a service S taking into consideration weights associated with all soft con-contextual constraints in a query. The function used to calculate the soft non-contextual constraint partial distance is detailed in [79].

The matching of contextual constraints between a query and service is executed by evaluating the context constraints of a query and context information of a service. As in the case of behavioural matching, this evaluation is executed by the monitor component (see Figure 1) based on requests received by the service requestor for monitoring specific contextual properties. The contextual properties to be monitored are derived from the translation of contextual criteria in a SerDiQueL query into event calculus (EC) [5] (see Section 3 for details of this translation).

Context information of each service is provided by *context services* based on the execution of context operations at runtime. Context operations are associated with service operations and are specified in context facets represented in XML format in the service registries. A context operation is defined in terms of (i) the associated service operation, (ii) the name of the context operation, (iii) the identifier of the context service that offers the context value for the operation, (iv) a value indicating for how long the context value returned by the execution of the context operation is valid, and (v) the semantic category of the context operation instead of its signature. The semantic category of an operation is specified in terms of ontologies. The use of semantic category is to support the fact that it is possible to have different signatures for context operations in the context services.

The monitor deploys a service client for each context service that needs to be monitored. As in the case of behavioral matching, the service client component is responsible for the invocation of context services and the generation of runtime events intercepted by the event collector. The event collector component is responsible to gather runtime information during the execution of the context services and to make this information available during the verification of the different context properties. The result of this evaluation is also a binary value indicating whether a specific contextual constraint is satisfied (value zero) or not (value one). After the evaluation of individual constraints, a partial contextual constraint distance is calculated for a query Q and a service S taking into consideration weights associated with all contextual constraints in a query. The result of the partial contextual constraint distance is passed to the matchmaker to compute the overall distance between a service and a query. The function used to calculate the contextual constraint partial distance is detailed in [79]. The evaluation of the contextual constraints is based on the work described in [81].

In the pull mode of query execution, an application instructs the framework to locate replacement services. More specifically, a service-based system issues a query in SerDiQueL to the web-service interface. This query is passed to the service requestor subsystem with the necessary characteristics of the service that needs to be replaced in the system, including the context of the application environment. The service requestor sends the query to the matchmaker to execute the query against the services as described by the matching process above. The results of the matching process are returned to the client application.

The push mode of query execution is performed in a proactive way in parallel to the execution of the service-based system. More specifically, the push mode of query execution can be performed when there is the need to replace a service due to any of cases (a) to (d) described in Section 1. The push mode of query execution is based on the subscriptions of the services participating in a service-based system, the application environment, and the queries associated with the participating services. The need for these subscriptions is to allow services to be discovered when changes in the subscribed services, or the application environment, are identified. These changes can be identified by the monitor (when there are behavioural and contextual changes in the services) or the service listeners (when there are structural and quality changes in the services, or a new service becomes available).

The push mode of query execution for a subscribed service and its queries requires an initial execution of the query in pull mode in order to create an initial set of candidate services for this service and queries. These set of candidate services will be maintained up-to-date in parallel to the execution of the service-based system, so that if there is a need to replace a subscribed service in the system, the replacement service will be selected from this up-to-date set. The candidate services in the set are maintained in ascending order of their distance with a respective query.

We describe below the service discovery process for replacing a service in an application when (a) a subscribed service becomes unavailable, (b) there are changes in the structure, behavioural, quality, or contextual characteristics of a subscribed service, (c) there are changes in the context of the system's environment, and (d) a new service becomes available or an existing service in the registry has its characteristics changed.

In order to follow the description of the above cases consider S1 one of the services participating in service-based system RP and Q1 a query for S1. Assume that a pull mode service discovery process has been executed for S1 and Q1. Suppose Set_S1 the set of candidate services returned by the execution of the pull mode process (services that match the structural, behavioural, quality and context constraints of Q1). Note that Set_S1 also includes service S1. Consider that RP, service S1, query Q1, and the candidate services in Set_S1 are subscribed.

In cases (b), (c), and (d), it is possible that the replacement of a service in a service-based system is not executed immediately after the identification of a replacement service. This may happen when it is better to delay the replacement process instead of replacing a service that may be running in the system (e.g., a new better service for a subscribed service S' is identified, but an operation of S' is being invoked). The framework uses *replacement policies* to assist with the decision of when to execute the replacement of a service in a service-based system for the different cases.

The actual replacement of a service can be realized by applying different techniques. Examples of these techniques are concerned with (i) mechanisms for dynamic replacement of partner web services in a WS-BPEL process by binding partner links at runtime as offered in WS-BPEL specification [49], (ii) extension of the execution environment of web service composition to enable dynamic reconfiguration of web service composition [48][47], and (iii) proxy service based framework to deploy adaptable web service compositions [44][45][46]. In the current implementation of the framework we make use of a proxy mechanism to replace a service in a service-based system, when necessary, following the replacement policies for the different cases. The details of the replacement policies and the mechanism to replace a service in a service-based system are beyond the scope of this paper, but have been described in [94].

Case 1: A subscribed service (S1) is unavailable

In this case, the monitor informs the service requestor that *S1* is unavailable. The service requestor takes necessary actions to replace *S1* by a service *S2* in *Set_S1,* which has the smaller distance with *Q1* from all the services in *Set_S1*. Service *S1* is removed from *Set_S1* and unsubscribed.

Case 2: Changes in the structure, behavioural, quality, or context characteristics of a subscribed service (S1)

In this case, the new version of service S1 needs to be evaluated against query Q1 to see if S1 continues to match Q1. In a negative case, a new service that matches Q1 needs to be identified to replace S1 in the system. This case is divided into two sub-cases depending if the monitor or service listener components are used, as described below.

Case 2.a: Changes in the context or behaviour of service (S1)

In this situation, the monitor informs the service requestor that S1 is no longer satisfying the behavioural criteria or contextual constraints of Q1. The service requestor takes necessary action to replace service S1 in the system by a service S2 in Set_S1. This set is being constantly updated since its services have been subscribed for changes in its functional, quality, behavioural, and context characteristics. The result of the process is passed to the web-service interface subsystem and subsequently to the client of the service-based system. In the situation in which Set_S1 is empty (i.e., there are no services in the registries that can fulfill the query), the web-service interface informs the service-based system that there are no available services to replace service S1.

Case 2.b: Changes in the structure or quality characteristics of a subscribed service (S1)

In this situation, the service listener informs the service requestor that a change has occurred in S1 together with the type of the criteria that has been changed (i.e., structural or quality). The service requestor passes query Q1 and information about the criteria type that has changed in S1 to the matchmaker. The matchmaker evaluates Q1 against the new version of the service specification of S1. The new version of the service specification is accessed from the external service registries through the service registry intermediary. The result of the matching process is passed to the service requestor. In this case S1 does not need to be matched against behavioural or context constraints since S1 has not been changed with respect to these aspects.

If the new specification of S1 matches Q1, and S1 has the smallest distance with Q1 when compared to the other services in Set_S1, nothing needs to be done. However, when the new version of S1 matches Q1 but does not have the smallest distance with Q1 when compared with other services in Set_S1, or S1 does not match Q1, a service S2 in Set_S1 is used to replace S1 in the application, such that S2 has the smallest distance with Q1 when compared to the other services in Set_S1. The result of the process is passed to the web-service interface subsystem and subsequently to the application client. Similarly to case 2.a, if Set_S1 is empty (i.e., there are no services in the registries that can fulfill the query), the web-service interface informs the application that there are no available services.

Case 3: Changes in the context of the application environment

The monitor informs the service requestor that a change has occurred in the context characteristics of the application environment. In this case, the context constraint of Q1 is modified. Assume Q1' the new version of Q1 with the modified context constraint. Service S1, as well as the other services in Set_S1, need to be evaluated against the new context constraints in Q1'. The services in Set_S1 already match the other constraints of the query that have not been modified.

The service requester subsystem sends the contextual constraints of Q1' (translated into EC) to the monitor in order to evaluate the context conditions of Q1' against Set_S1. In this case, if a new set of services Set_S3 that is a sub-set of (or equal to) Set_S1 that match the context conditions of Q1' is identified, this set will replace Set_S1. Moreover, if S1 is in Set_S3 and S1 has the smallest distance with Q1' when compared to the other services in Set_S3, nothing else needs to be done. However, if S1 is in Set_S3 but S1 does not have the smallest distance with Q1' when compared to the other services in Set_S3 or if S1 is not in Set_S3, a service S2 from Set_S3 is used to replace S1 in the application such that S2 has the smallest distance with Q1' when compared to the other services in Set_S3.

In the case in which no service in Set_S1 match the context conditions of Q1', the new version of the query is evaluated against the services in the external registries in order to build a new set of candidate services for Q1'. In this case, a service in this new set is used to replace S1 in the application and all the services in the set are subscribed together with query Q1' for future push mode service discovery iterations.

The result of the process is passed to the web-service interface subsystem and subsequently to the application client. In the situation in which there are no services in the registries that can fulfill Q1', the web-service interface informs the application that there are no available services.

Case 4: A new service becomes available or a service in the registry has its characteristics changed

Suppose that a new service S3 becomes available. In this case, the service listener informs the service requestor that S3 is available. The service requestor sends the structural and quality criteria of Q1 to the matchmaker in order to evaluate these criteria against the specification of S3. The service requestor also sends the behavioural and contextual criteria of Q1 (translated into EC) to the monitor in order to evaluate these criteria against S3.

In the case that S3 matches the constraints of Q1, S3 is added to Set_S1 and is subscribed. In addition, the service requestor compares the results of the matching of Q1 against S3 and against S1 in order to see if the former is better than the latter. In positive case, S3 is used to replace S1 in the system.

2.2 Monitoring Process

MoRSeD supports the monitoring of services in order to (a) identify whether services become unavailable, (b) identify that there are changes in the behavioural or contextual characteristics of the services participating in service-based system or replacement candidate services, (c) identify that there are contextual changes in the service-based system environment, and (d) verify if behavioural and contextual properties specified in service discovery queries are satisfied by services.

The monitor receives requests from the service requestor to verify at regular intervals the satisfiability of specific properties of a service-based system, its constituent services, or context services; or to verify the satisfiability of the properties with potential candidate services during the matching process. The monitor intercepts all the runtime messages exchanged between the service-based system and its constituent services and verifies the satisfiability of the properties against these messages. It also invokescontext services to verify the satisfiability of the contextual properties specified in queries. When the monitor detects violation of a property, it notifies the service requestor about this violation, which takes the necessary actions to identify replacement services either in pull mode or proactive push mode. The monitor also invokes potential candidate services to verify the satisfiability of behavioural and contextual properties.

The monitor of MoRSeD has been adapted from the monitor approach discussed in [4, 6]. It consists of three components namely (a) service client, (b) event collector, and (c) analyzer described below.

Service Client: The service client is responsible for the invocation of a service. The monitor deploys a service client for each service that needs to be monitored (i.e., candidate services, context services, and potential candidate services) in order to generate runtime events that are used to verify properties (i.e., different behavioural and contextual constraints expressed in a query).

More specifically, given a WSDL specification of a service, the respective service client produces all possible sequences of operations described in the WSDL specification. In order to produce the alternative sequences of operations, the service client considers those sequences that contain the order of operations specified in the behavioural query. This reduces the number of sequences of operations to be executed by the client. After producing the alternative sequence of operations, the service client invokes the operations in the order they appear in each sequence. The client uses the Web Service Invocation Framework (WSIF) [86] to generate random values for the input parameters of each operation in a sequence and to invoke the operations. It should be noted that in MoRSeD a behavioural property for a service expresses the existence of certain operations in the service or the desired order of execution of operations in the service. Hence the values of the input and output parameters of these operations are not relevant to the monitoring process. The framework also assumes that if an operation in a service is not invoked in the correct order of execution of the operations of that service, the service will generate an exception. With this assumption, the service client stops invoking the operations in one of the sequences of operations as soon as it receives an exception from the service, and continues to execute the next of sequence of operations.

Event Collector. The event collector is responsible to gather (a) information during the execution of service-based system and the services deployed by the service based system, or (b) information exchanged between the service client and its respective service. In case (a), the event collector intercepts the SOAP messages exchanged between the service-based system and its constituent services. In case (b), the event collector intercepts the SOAP messages exchanged between the service client and the respective services. The intercepted SOAP messages are then transformed into a form that is understood by the analyzer component and recorded in an event database in the monitor.

Analyzer. The analyzer checks the satisfiability of the properties against the runtime events. The properties to be monitored may be related to the behaviour or quality properties that should be provided by an individual service of the service-based system, or groups of such services. These properties are expressed in event calculus (EC) [5] in terms of *events* and *fluents*.

An event is something that occurs at a specific instance of time, has instantaneous duration, and may change the state of a system (e.g., invocation of an operation, response returned following the execution of an operation or assignment of a value to a variable). In our framework we consider the following type of events that may occur during the execution of service-based systems:

(i) The invocation of an operation by the composition process of the service-based system in one of its partner services or the return from this execution.

(ii) The invocation of an operation in the composition process of the service-based system by another external service or the reply following this execution.

The occurrence of an event *e* of the above types at time *t* is expressed by the Event Calculus (EC) predicate ***Happens***$(e,t,\Re(t1,t2))$ where $\Re(t1,t2)$ signifies the expected time range for t).

Fluents signify system states as conditions over the values of specific variables of the composition process of a service-based system or its constituent services. The states represented by fluents are initiated and terminated by events. Fluent initiation and termination are expressed by the following predicates of EC:

- ***Initiates****(e,f,t)* – This signifies that a fluent f starts to hold after the event e at time t.
- ***Terminates****(e,f,t)* – This signifies that a fluent f ceases to hold after the event e occurs at time t.

A fluent holds from its initiation until its termination. The existence of a fluent *f* at time *t* is expressed by the predicate ***HoldsAt****(f,t)*.

In MoRSeD, we use the following terms to represent event and fluents:

- *ic:PartnerService:OperationName(_oId, _ip1,_ip2…_ipn)* – This term signifies the invocation (denoted by the prefix *ic*) of an operation by the composition process of a service based system in one of its partner services, or the invocation of an operation in the composition process of the service based system by another service. In this expression *oId* is a variable whose value identifies the exact instance of invocation made to the operation and *_ip1,_ip2…_ipn* are variables that indicate the values of the input parameters of the operation at the time of its invocation.
- *ir:PartnerService:OperationName(_oId, _op1,_op2…_opn)* – This term signifies the return from the execution of an operation (denoted by the prefix *ir*) invoked by the composition process in a partner service, or the return following the execution of an operation that was invoked by another service in the composition process. In this expression *oId* is a variable whose value identifies the exact instance of invocation made to the operation to which this response corresponds to and *_op1,_op2…_opn* are variables that indicate the values of the output parameters of the operation at the time of its response.
- *valueOf(fluent_expression, value_expression)* – This term signifies a fluent, where *fluent_expression* denotes a typed variable in a service based system or its constituent services, and the *value_expression* is a term that either represents an EC variable or signifies a call to an operation that returns an object of some type. The operation called by *value_expression* may be an internal operation that is provided by the monitoring subsystem or an operation that is provided by an external web-service. An operation call in the monitoring subsystem takes one of the following terms:
    - *oc:S:O(_P1,…,_Pn)* that signifies the invocation of an operation O in an external service S.
    - *oc:self:O(_P1,…,_Pn)* that signifies the invocation of the built-in operation O of the monitor.

  In the above forms, _P1, …, _Pn are variables that indicate the values of the input parameters of the operation O at the time of its invocation.

In addition to the EC predicates discussed above, in the property specifications we use relational predicates to enable comparison among values of variables, return values of operation calls, and constant values by using standard relational operators.

An example of a property for the behaviour of the *GPS* service specified using event calculus is shown in Figure 2. The formula C1 in this figure expresses that following a request for the location from a client to *GPS* at time t1 (see literal **Happens**(ic:getLocation(ID),t1,R(t1,t1))) and the response of this request at time t2 (see literal **Happens**(ir:getLocation(ID,latitude,longitude),t2,R(t1,t2)) the latitude and longitude of the returned location cannot be negative.

It should be noted that in MoRSeD the EC properties to be monitored are automatically translated from the behavioural and contextual constraint in the queries expressed in SerDiQueL (see Section 3).

(C1) (∀ t1:Time, ∃ t2: Time[1])

[1] In all the EC formulas in this paper, "Time" denotes the set of time stamps that are recorded in the trace of the events captured during the execution of a service based system.

```
Happens(ic:getLocation(ID),t1,R(t1,t1)) ^
Happens(ir:getLocation(ID,lat,long),t2,R(t1,t2)) ⇒ lat > 0 ^ long > 0
```

Fig. 2 Example property in EC

The monitor checks the satisfiability of a property against the runtime events recorded by the *event collector*. More specifically, the satisfiability of a property is checked by verifying whether the set of the recorded events entail the negation of a property *p* or, formally, if:

$$\{ER(T)\} \models_{nf} \neg p$$

where ER(T) is the set of the events that have been recorded by the event collector from the start of the monitoring process until time T, and $\models_{\mathrm{nf}}$ is the logical entailment using the normal rules of inference of first-order logic and the principle of negation as failure.

At runtime, the monitor maintains templates that represent different instantiations of the formulas that specify the behavioural properties and assumptions for a system. The templates maintained by the monitor store the state of different instantiations of a property f, including:

- The identifier (FID) of *f*.
- The bindings (VB) of the non-time variables of all the predicates in f, and
- For each predicate p in f :
    - The qualifier of the time variable (Q) and signature (SG) of p.
    - A time range (LB,UB) that indicates when p should occur. The boundaries of this range are set according to the time constraint of p in f.
    - The truth-value (TV) of p which can be: UN (if the truth-value of p has not been established), True (if p is true), or False (if p is false).
    - A time stamp (TS) that indicates the time in which the truth-value of p is established (TS is set to the time variable of the predicate initially).
    - The source (SC) of the evidence for the truth value of p which can be: UN (if the truth value of p has not been established), RE (if the truth value of p is established by a recorded event unified with it), or NF (if the truth value of p is established by the principle of negation as failure)

The monitor picks events from the *Event Database* and checks if there are instances of templates that should be updated by the events. Updates may be made if the signature, the event variable bindings, and the time of the event comply with the predicate signature, the predicate variable bindings, and the time range of the predicate in a template instance, respectively. If a predicate is updated, the bindings of the predicate's variables in the template are also updated. New instances of templates may also be generated if the event corresponds to an unconstrained predicate of a template (i.e., a predicate whose time variable is not constrained by the time variable of another predicate), or the variable bindings of the predicate have values that are different from the event variable bindings values. The truth-value of a predicate in a template instance may also be updated by applying the principle negation as failure.

```
L1  : Happens(ic:getLocation(op1),1,ℜ(1,1))
L2  : Happens(ir:getLocation(op1, 120, 210),4,ℜ(4,4))
L3  : Happens(ic:getTraffic(op2, 120, 210),5,ℜ(5,5))
L4  : Happens(ir:getTraffic(op2, trafficInfo),9,ℜ(9,9))
L5  . . . . ..
L6  . . . .. . .
L7  . . . . .. .
L8  : Happens(ic:getLocation(op19),29,ℜ(29,29))
L9  : Happens(ir:getLocation(op19, -50, 210),32,ℜ(32,32))
L10 . . . . .
```

```
L11 . . . .
```

Fig. 3 Event log of *Route Planner* process

For example, consider the property *C1* shown in Figure 2 and the event log of the *Route Planner* application in Figure 3. As shown in the figure, following the occurrence of the event signified by literal L8 at T=29, the monitor will create the following template for formula *C1*:

ID	C1						
VB	(oID, op19)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:getLocation(oID),t1,R(t1,t1))	29	29	29	True	RE
2	∃	**Happens**(ir:getLocation(oID,lat,long),t2, R(t1,t2))	t2	29	t2	UN	UN
3	∃	lat > 0	t2	29	t2	UN	UN
4	∃	long > 0	t2	29	t2	UN	UN

Subsequently, when the event signified by literal L9 in Figure 3 is encountered, the above template will be updated and take the following form:

ID	C1						
VB	(oID, op19) (lat, -50) (long, 210)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:getLocation(oID),t1,R(t1,t1))	29	29	29	True	RE
2	∃	**Happens**(ir:getLocation(oID,lat,long),t2, R(t1,t2))	32	29	32	True	RE
3	∃	lat > 0	32	29	32	False	RE
4	∃	long > 0	32	29	32	True	RE

When the truth values of all predicates in a formula template have been established, a check for possible formula violations is performed according to the criteria described in [4, 6]. For example, if the truth-value of all the predicates in the template is true the formula is satisfied, and if the truth-value of all the unconstrained predicates in the formula is true and the truth-value of at least one constrained predicate is false and the source of all predicates is *RE* or *NF*, the formula is marked as inconsistent with the recorded behaviour of the system.

3 Translations from SerDiQueL to EC Properties

In MoRSeD the behavioral and contextual properties to be verified by the analyzer component are translated from service discovery queries represented in SerDiQueL. In these queries, behavioral properties may refer to: (i) the existence and order of certain functionalities in a service, (ii) dependencies between functionalities of a service, and (iii) conditional iterations of sequences of service functionalities. Contextual properties can represent contextual values that need to be checked for the various functionalities in a service.

To translate behavioural criteria and contextual constraints specified in SerDiQueL queries into event calculus, we use a set of *translation patterns* and our framework applies these patterns to generate the required translations automatically. The translation patterns used and the process realised by MoRSeD are described in the following.

3.1 SerDiQueL Behavioural and Contextual Sub-queries

An example of a SerDiQueL query is shown in Figure 4. This query (Q1) has been specified to identify services that could replace the *GPS* service in the *Route Planner* application (see Section 1). Recall that the GPS service provides the location of the PDA where *Route Planner* operates after receiving payment. According to Q1, a replacement service for GPS will need to satisfy the following conditions:

(a) It should authenticate its user before allowing access to its functionality;
(b) It should not take more than 10 seconds to provide the location of a user;
(c) It should receive payment from the user before the provision of location information.

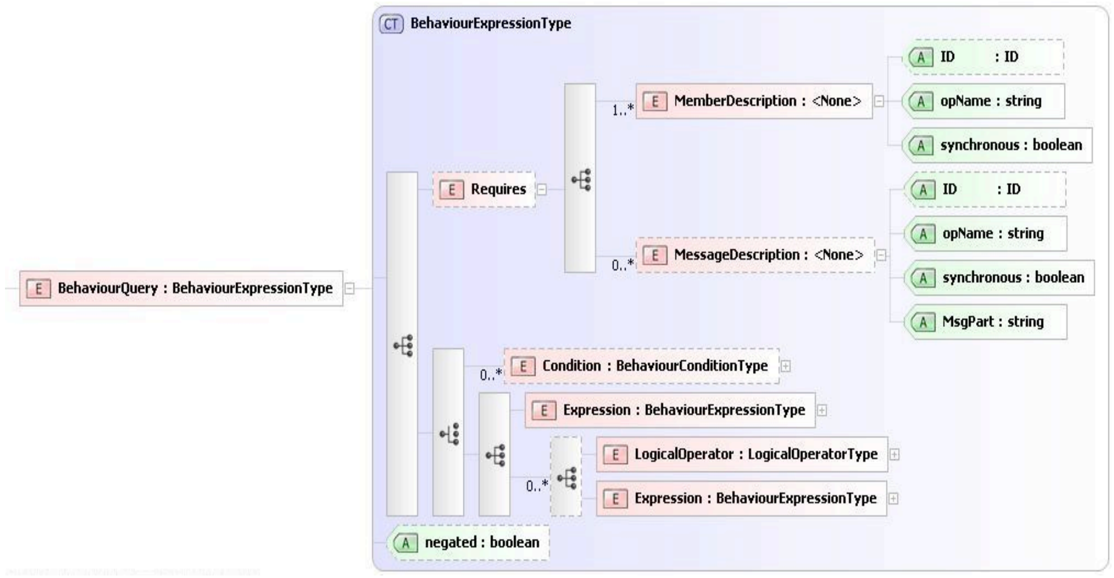
As shown in Figure 4, Q1 is a *dynamic type query* of *push mode*, as specified by element *Parameter* in the query. The structural sub-query is composed of the WSDL specification of *GPS* service. For simplicity, we only show a placeholder for the WSDL in Figure 4. The behavioural sub-query is specified as content of element *<tnsb:BehaviouralQuery>*, and the constraint sub-query is specified as content of element *<tnsb:CosntraintQuery>*.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Created with Liquid XML Studio 1.0.8.0 (http://www.liquid-technologies.com) -->
<tns:ServiceQuery xmlns:tns="http://gredia.eu/schema/SerDiQueL" xmlns:csql="http://gredia.eu/schema/Constraint_SQL"
   xmlns:tnsb="http://gredia.eu/schema/Behavour_SQL"  xmlns:tnsc="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   queryID="UUID:550e8400-e29b-41d4-a716-446655440000" name="Query1">

   <tns:Parameter name="mode" value="PUSH" />
   <tns:Parameter name="type" value="dynamic" />
   <tns:Parameter name="threshold" value="1.0" />

   <!-- Structural sub-query -->
   <tns:StructuralQuery> <!-- WSDL of the GPS service -- ></tns:StructuralQuery>

   <!-- Behavioural sub-query -->
    <tnsb:BehaviouralQuery>
      <tnsb:Requires>
        <tnsb:MemberDescription ID="login" opName="GPSService3PortType.login" synchronous="true" />
        <tnsb:MemberDescription ID="payment" opName="GPSService3PortType.makePayment" synchronous="true" />
        <tnsb:MemberDescription ID="location" opName="GPSService3PortType.getLocation" synchronous="true" />
        <tnsb:MemberDescription ID="logout" opName="GPSService3PortType.logout" synchronous="true"/>
      </tnsb:Requires>

      <tnsb:Expression>
            <tnsb:Condition><tnsb:GuaranteedMember IDREF="login" /></tnsb:Condition>
       </tnsb:Expression>
       <tnsb:LogicalOperator operator="AND" />
       <tnsb:Expression>
            <tnsb:Condition>
                        <tnsb:Sequence ID="pay">
                              <tnsb:Member IDREF="payment" />
                              <tnsb:Member IDREF="location" />
                              <tnsb:Member IDREF="logout" />
                        </tnsb:Sequence>
            </tnsb:Condition>
            <tnsb:Condition>
                        <tnsb:OccursBefore immediate="false" guaranteed="false">
                                    <tnsb:Member1 IDREF="login" />
                                    <tnsb:Member2 IDREF="payment" />
                        </tnsb:OccursBefore>
            </tnsb:Condition>
         </tnsb:Expression>
    </tnsb:BehaviouralQuery>

       <!-- Constraint sub-queries -->
      <csql:ConstraintQuery name="CQ1" contextual="true" type="SOFT" scope=" MONITORING">
```

```
            <csql:LogicalExpression>
                <csql:Condition relation="LESS-THAN-EQUAL-TO">
                    <csql:Operand1>
                         <csql:ContextOperand serviceOperationName="getLocation" serviceID="2009.005">
                              <csql:ContextCategory relation="EQUAL-TO">
                                   <csql:Category1>
                                       <csql:Document location="http://eg.org/CoDAMoS_Extended.xml"  type="ONTOLOGY"/>
                                   </csql:Category1>
                                   <csql:Category2>
                                                 <csql:Constant type="STRING">RELATIVE_TIME</csql:Constant>
                                    </csql:Category2>
                              </csql:ContextCategory>
                          </csql:ContextOperand >
                      </csql:Operand1>
                      <csql:Operand2>
                           <csql:Constant type="NUMERICAL">10</csql:Constant>
                      </csql:Operand2>
                  </csql:Condition>
          </csql:LogicalExpression>
        </csql:ConstraintQuery>

</tns:ServiceQuery>
```

Fig. 4 Example of a query in SerDiQueL

In SerDiQueL, a behavioural sub-query is described in terms of (a) a single (possibly negated) condition or a conjunction of conditions, (b) a sequence of expressions separated by logical operators, or (c) *requires* elements, as shown in the XML schema in Figure 5.



CT BehaviourExpressionType
E BehaviourQuery : BehaviourExpressionType
E Requires
1..*
E MemberDescription : <None>
A ID : ID
A opName : string
A synchronous : boolean
0..*
E MessageDescription : <None>
A ID : ID
A opName : string
A synchronous : boolean
A MsgPart : string
0..*
E Condition : BehaviourConditionType
E Expression : BehaviourExpressionType
0..*
E LogicalOperator : LogicalOperatorType
E Expression : BehaviourExpressionType
A negated : boolean

Fig. 5 XML Schema for behavioural criteria

Requires elements define one or more service operations that need to exist in service specifications, represented as members by the elements *MemberDescription* and *MessageDescription*. These member elements are used in various conditions and expressions of a query. A *member* element has three attributes, namely (a) *ID*, indicating a unique identifier for the member within a query; (b) *opName*, specifying the name of an operation described in the structural sub-query, (for the case of dynamic service discovery, this attribute may also contain the port type for this operation for the WSDL description in the structural sub-query); and (c) *synchronous*, a boolean attribute indicating if the service operation needs to be executed in a synchronous or asynchronous mode in the service. A *Message* description has the attribute *MsgPart* in addition to the above three attributes. The attribute *MsgPart* refers to a specific part of a message in the operation.

A condition is defined as *GuaranteedMember*, *OccursBefore*, *OccursAfter*, *Sequence*, or *Loop* elements, as shown in Figure 6. A *GuaranteedMember* represents a member element (i.e., service operation) that needs to occur in all possible traces of execution in a service. This element is defined by attribute *IDREF* that references requires, sequence, or loop elements. The *OccursBefore* and *OccursAfter* elements represent the order of occurrence of two member elements (Member1 and Member2). They have two boolean attributes, namely (a) *immediate*, specifying if the two members occur in sequence or if there can be other member elements in between them, and (b) *guaranteed*, specifying if the two members need to occur in all possible traces of execution in a service. A *Sequence* element defines two or more members that must occur in a service in the order represented in the sequence. It has an identifier attribute that can be used by the *GuaranteedMember*, *OccursBefore*, *OccursAfter*, *Sequence*, and *Loop* elements. A *Loop* element specifies a sequence of members that are executed several times if certain conditions are satisfied. It has a unique identifier (attribute *ID*) and is defined as a statement (element *Body*) that references other identified elements.

Expressions are defined as a sequence of requires elements, conjunctions of conditions, or other nested expressions connected by logical operators *AND* and *OR*.

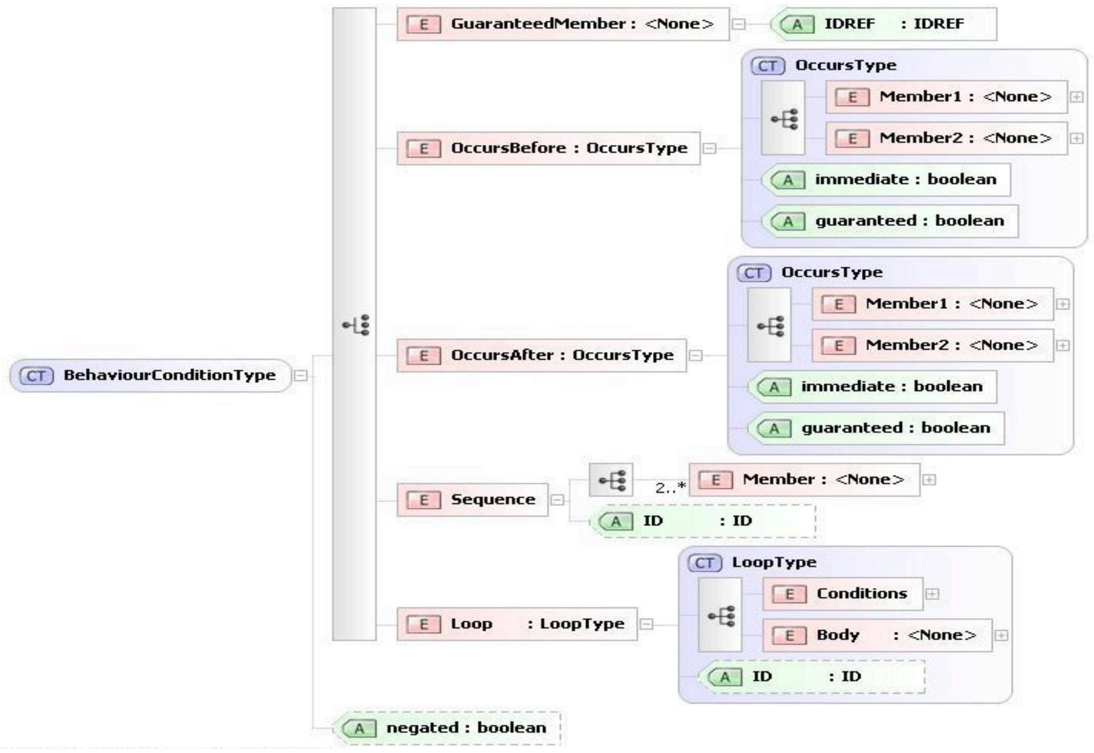


Fig. 6 XML Schema for behavioral criteria

As shown in the example in Figure 4, the behavioural sub-query (see element *<tnsb:BehaviouralQuery>* ... *</tnsb:BehaviouralQuery>*) includes *Requires* elements expressing the requirement for the existence of the following operations in any replacement service:

- *login(userID:string, password:string):boolean*
- *makePayment(accountId:string, amount:double):boolean*
- *getLocation():Location*
- *logout((userID:string):boolean*

In addition, as shown in Figure 4:
(a) operation *login* is defined as a *GuaranteedMember* element given that the user of the GPS service needs to be authenticated (i.e., *login* operation needs to occur in all possible paths of execution in the service);
(b) the operations *makePayment, getLocation* and *logout* need to be executed in this order and, therefore, they are defined in a *Sequence* element;
(c) the operation *login* should be executed before the sequence of operations in (b) specified in element *OccursBefore*.

In SerDiQueL a contextual sub-query is specified in terms of a single logical expression, or a conjunction/disjunction of two or more logical expressions, combined by logical operators AND and OR, or a negated logical expression. A logical expression is defined as a condition, or logical combination of conditions, over elements or attributes of service specifications (for non-contextual constraints) or over context aspects of service operations (for contextual constraints).

A condition can be negated and is defined as a relational operation (*equalTo*, *notEqualTo*, *lessThan*, *greaterThan*, *lessThanEqualTo*, *greaterThanEqualTo*, *notEqualTo*) between two operands (*operand1* and *operand2*). These operands can be non-contextual operands, contextual operands, constants, or arithmetic expressions.

As shown in Figure 7 a non-contextual operand (element *NonContextOperand*) has two attributes, namely (a) *facetName*, specifying the name of the service specification and (b) *facetType*, specifying the type of the service specifications to which the constraint will be evaluated. The operand contains an XPath expression indicating elements and attributes in the service specification referenced in *facetName* attribute.

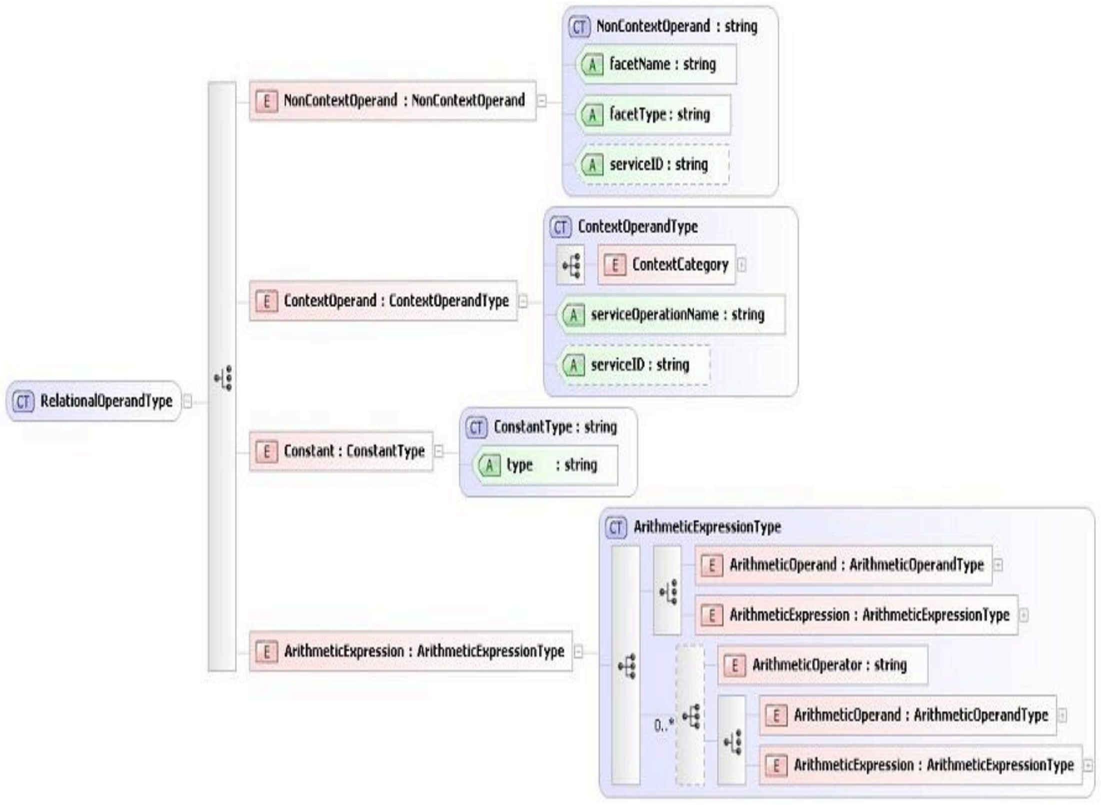

RelationalOperandType
NonContextOperand : NonContextOperand
NonContextOperand : string
facetName : string
facetType : string
serviceID : string
ContextOperand : ContextOperandType
ContextOperandType
ContextCategory
serviceOperationName : string
serviceID : string
Constant : ConstantType
ConstantType : string
type : string
ArithmeticExpression : ArithmeticExpressionType
ArithmeticExpressionType
ArithmeticOperand : ArithmeticOperandType
ArithmeticExpression : ArithmeticExpressionType
ArithmeticOperator : string
0..*
ArithmeticOperand : ArithmeticOperandType
ArithmeticExpression : ArithmeticExpressionType

Fig. 7 XML schema for relational operand in constraint sub-query

A contextual operand (element *ContextOperand*) specifies operations that will provide context information at runtime. More specifically, a contextual operand describes the semantic category of context operations instead of the signature of the operation represented by sub-element ContextCategory. This is due to the fact that context operations may have different signatures across different services. A contextual operand is defined by (a) attribute *serviceOperationName*, specifying the name of the service operation associated with the contextual operand, and (b) attribute *serviceID*, specifying the identifier of a service that provides the operation. The value of attribute *serviceID* is specified when the context operand provides the specification of a context operation of a known service. This is normally the case when the context operation is associated with a service-based application for which the value of a context aspect of the application needs to be dynamically identified during the evaluation of a query (e.g., location of a mobile device application). In this case, attribute *serviceID* referes to the service-based application itself. Otherwise, the value of *serviceID* is specified as “any” (see Figure 7).

A *ContextCategory* element represents the semantic category of an operation, instead of its actual signature. It is defined as a relation between two categories (*Category1* and *Category2*). These categories can be either a reference to a document or a constant. A context category is evaluated against context facets of candidate services. This evaluation verifies if a candidate service has a context operation with semantic category that satisfies the categories specified in a query.

Arithmetic expressions define computations over the values of elements or attributes in service specification or context information. They are defined as a sequence of arithmetic operands or other nested arithmetic expressions connected by arithmetic operators. The arithmetic operators are: addition (*plus*), subtraction (*minus*), multiplication (*multiply*), and division (*divide*) operators. A *function* supports the execution of a complex computation over a series of arguments.

In the example shown in Figure 4, the constraint sub-query CQ1 (see element <*tnsb:ConstraintQuery*> ... </*tnsb:ConstraintQuery*>) is a soft contextual constraint concerned with the time to get response from the *GPS* service. This constraint specifies that any candidate service needs to have a context operation associated with operation *getLocation()* classified in the category RELATIVE_TIME in the ontology http://eg.org/CoDAMoS_Extended.xml, and the result of executing this operation has to be less than 10 seconds for this service to be considered.

3.2 Translation Patterns for Behavioural Criteria

Behavioural conditions involving `Requires` *Elements*

In the case of an asynchronous operation, a *MemberDescription*, or a *MessageDescription* is transformed into an atomic EC formula consisting of a Happens predicate that signifies the occurrence of an operation *O*. For example, consider the following *MemberDescription*:

SerDiQueL Element	EC Representation
<bsql:MemberDescription ID="M " opName="O" synchronous="false"/>	**Happens**(ic:O(_ID,_X),t1, $\Re$(t1,t1))

In this example, the predicate ***Happens****(ic:O(_ID,_X.a),t1, $\Re$(t1,t1))* signifies the occurrence of the operation *O*. It should be noted that in the EC representation, the variable *_ID* takes as value a unique identifier that represents the exact instance of the occurrence of *O*, and the variable *_X* takes the value of the input variable *X* (if any) of *O*.

In the case of a synchronous operation *O,* a *MemberDescription*, or a *MessageDescription* is transformed into a conjunctive EC formula consisting of a *Happens* predicate that signifies the occurrence of the operation *O*, and a *Happens* predicate that signifies the response from *O*. For example, consider the following *MemberDescription*:

SerDiQueL Element	EC Representation
<bsql:MemberDescription ID="M " opName="O" synchronous="true"/>	**Happens**(ic:O(_ID,_X),t1, $\Re$(t1,t1)) => (∃t2) **Happens**(ir:O(_ID, _Y),t2, $\Re$(t1,t2))

In this example, the predicate ***Happens****(ic:O(_ID,_X.a),t1, $\Re$(t1,t1))* signifies the occurrence of the operation *O*, and the predicate ***Happens****(ir:P:O(_ID),t2, $\Re$(t1,t2))* signifies the response from *O.* It should be noted that in this EC representation the variable *_ID* takes as value a unique identifier that represents the exact instance of the occurrence of *O,* the variable *_X* takes the value of the input variable *X* (if any) of *O,* and the variable *_Y* takes the value of the output variable *X* (if any) of *O*.

Behavioural conditions involving `OccursBefore` *elements*

A behavioural condition in SerDiQueL that involves an *OccursBefore* element specifies the order of occurrence of two member elements (Member1 and Member2), where a member could be a *MemberDescription*, or a *MessageDescription*, or another behavioural condition.

A behavioural condition that involves an *OccursBefore* element with a "*false*" value for the attribute *immediate*, can be transformed into EC according to the following pattern:

SerDiQueL Element	EC Representation

<bsql:OccursBefore immediate="false" guaranteed="true"> <bsql:Member1 IDREF="M1"/> <bsql:Member2 IDREF="M2"/> </bsql:OccursBefore>	**EC**(M1,[]) $\wedge$ **EC**(M2,[]) $\wedge$ max_t(M1) < min_t(M2)

In the above pattern[2]

- *EC(M, [t_1,…,t_n])* denotes the EC (sub)formulas that a member element is transformed to;
- *$min_t(M)$* represents the time of the earliest predicate in the EC representation of member M, and *$max_t(M)$* represents the time of the latest predicate in the EC representation of member M.

A behavioural condition that involves *OccursBefore* with "*true*" value of the attribute *immediate*, can be transformed to EC according to the following pattern:

SerDiQueL Element	EC Representation
<bsql:OccursBefore immediate="true" guaranteed="true"> <bsql:Member1 IDREF="M1"/> <bsql:Member2 IDREF="M2"/> </bsql:OccursBefore>	**EC**(M1,[]) $\wedge$ **EC**(M2,[]) $\wedge$ max_t(M1) < min_t(M2) => ¬ **Happens**(ANY(), t2, R(max_t(M1), max_t(M2)))

In the above pattern the predicate ¬ ***Happens****(ANY(), t2, R(max_t(M1), max_t(M2)))* signifies that no operation should occur between the time of the latest predicate in the EC representation of member M1 and the time of the latest predicate in the EC representation of member M2.

Behavioural conditions involving OccursAfter

A behavioural condition that involves *OccursAfter* with "*false*" value of the attribute *immediate,* can be transformed into EC according to the following pattern:

SerDiQueL Element	EC Representation
<bsql:OccursAfter immediate="false" guaranteed="true"> <bsql:Member1 IDREF="M1"/> <bsql:Member2 IDREF="M2"/> </bsql:OccursAfter>	**EC**(M2,[]) $\wedge$ **EC**(M1,[]) $\wedge$ max_t(M2) < min_t(M1)

A behavioural condition that involves *OccursAfter* with "*true*" value of the attribute *immediate*, can be transformed into EC according to the following pattern:

SerDiQueL Element	EC Representation
<bsql:OccursAfter immediate="true" guaranteed="true"> <bsql:Member1 IDREF="M1"/> <bsql:Member2 IDREF="M2"/> </bsql:OccursAfter>	**EC**(M2,[]) $\wedge$ **EC**(M1,[]) $\wedge$ max_t(M2) < min_t(M1) => ¬ ∃t **Happens**(ANY(), t, R(max_t(M2), max_t(M1)))

In the above pattern the expression ¬ ***∃t Happens****(ANY(), t, R(max_t(M2), max_t(M1)))* signifies that no operation should occur between the time of the latest predicate in the EC representation of member M2 and the time of the latest predicate in the EC representation of member M1.

Behavioural conditions involving **`Sequence`** *elements*

[2] *EC(M,[]),$min_t(M)$* and *$max_t(M)$* have the same semantic throughout this paper.

A behavioural condition that involves a *sequence* can be transformed into EC according to the following pattern:

SerDiQueL Element	EC Representation
`<bsql:Sequence ID="S1">`	$\mathbf{EC}(M1,[]) \wedge \mathbf{EC}(M2,[]) \wedge \ldots \wedge \mathbf{EC}(Mn,[])$
`<bsql:Member IDREF="M1"/>`	$max_t(M1) < min_t(M2) \wedge \ldots \wedge max_t(Mn\text{-}1) < min_t(Mn)$
`<bsql:Member IDREF="M2"/>`	
… …… …..	
`<bsql:Member IDREF="Mn"/>`	
`</bsql:Sequence>`	

Behavioural conditions involving* Loop *elements

A behavioural condition in SerDiQueL that involves a *Loop* element specifies a sequence of members that are executed several times if certain conditions are satisfied, where conditions are specified in terms of *MessageDescriptions*. A behavioural condition that involves *Loop* element can be transformed into EC according to the following pattern:

SerDiQueL Element	EC Representation
`<bsql:Loop ID="L1">`	$\mathbf{EC}(M1,[]) \wedge \mathbf{EC}(M2,[]) \wedge \mathbf{Rel}(V_{M1}, V_{M2}) \Rightarrow \mathbf{EC}(M3,[])$
`<bsql:Conditions>`	
`<bsql:Condition relation="Rel">`	
`<bsql:Operand1>`	
`<bsql:Variable IDREF="M1"/>`	
`</bsql:Operand1>`	
`<bsql:Operand2>`	
`<bsql:Variable IDREF="M2"/>`	
`</bsql:Operand2>`	
`</bsql:Condition>`	
`</bsql:Conditions>`	
`<bsql:Body IDREF="M3"/>`	
`</bsql:Loop>`	

In the above EC presentation, V_M refers to the variable of a *MessageDescription* M, identified by the XPATH expression in M and $\mathbf{\textit{Rel}}(V_{M1}, V_{M2})$ signifies a relational predicate over the variables V_{M1} and V_{M2}.

Example of behavioural condition translated into EC

Table 1 shows an example of *SerDiQueL* behavioural condition translated into EC applying the patterns discussed above. This example shows the EC representation of the behavioural condition in the *SerDiQueL* query in Figure 4, which specifies that the operation *login* should be executed before the sequence of operations *makePayment, getLocation* and *acknowledge*.

Table 1. Example of SerDiQueL behavioural condition translated into EC

SerDiQueL	`<tnsb:BehaviourQuery>`
	`<tnsb:Requires>`
	`<tnsb:MemberDescription ID="login" opName="GPSService3PortType.login" synchronous="true" />`
	`<tnsb:MemberDescription ID="payment" opName="GPSService3PortType.makePayment" synchronous="true" />`
	`<tnsb:MemberDescription ID="location" opName="GPSService3PortType.getLocation" synchronous="true" />`
	`<tnsb:MemberDescription ID="acknowledge" opName="GPSService3PortType.acknowledge" synchronous="true"/>`
	`</tnsb:Requires>`
	`<tnsb:Expression>`

	`<tnsb:Condition><tnsb:GuaranteedMember IDREF="login" /></tnsb:Condition>`
	`</tnsb:Expression>`
	`<tnsb:LogicalOperator operator="AND" />`
	`<tnsb:Expression>`
	`<tnsb:Condition>`
	`<tnsb:Sequence ID="pay">`
	`<tnsb:Member IDREF="payment" />`
	`<tnsb:Member IDREF="location" />`
	`<tnsb:Member IDREF="acknowledge" />`
	`</tnsb:Sequence>`
	`</tnsb:Condition>`
	`<tnsb:Condition>`
	`<tnsb:OccursBefore immediate="false" guaranteed="false">`
	`<tnsb:Member1 IDREF="login" />`
	`<tnsb:Member2 IDREF="pay" />`
	`</tnsb:OccursBefore>`
	`</tnsb:Condition>`
	`</tnsb:Expression>`
	`</tnsb:BehaviourQuery>`

EC	(∀t2, t3, t4:Time, ∃ t1: Time) **Happens**(ic:makePayment(ID, accounted, amount),t2,R(t2,t2)) ^ **Happens**(ic:getLocation(ID),t3,R(t2,t3)) ^ **Happens**(ic:acknowledge(ID, val),t4,R(t3,t4)) ⇒ **Happens**(ic:login(ID, userID, password),t1,R(t1,t1)) ^ t1 < t2

3.3 Translation Patterns for Contextual Constraints

A contextual constraint can be transformed into EC using the following pattern.

SerDiQueL Element	EC Representation

`<csql:ConstraintQuery name="CQ" contextual="true" type="SOFT">`
`<csql:LogicalExpression>`
`<csql:Condition relation="REL">`
`<csql:Operand1>`
`<csql:ContextOperand serviceOperationName=”sOp” serviceID=”sId”>`
`<csql:ContextCategory relation=”EQUAL-TO”>`
`<csql:Category1>`
`<csql:Document location=”loc” type=”ONTOLOGY”/>`
`</csql:Category1>`
`<csql:Category2>`
`<csql:Constant type="STRING">CATEGORY</csql:Constant>`
`</csql:Category2>`
`</csql:ContextCategory>`
`</csql: ContextOperand >`
`</csql:Operand1>`
`<csql:Operand2>`
`<csql:Constant type="STRING">VAL</csql:Constant>`
`</csql:Operand2>`
`</csql:Condition>`
`</csql:LogicalExpression>`
`</csql:ConstraintQuery>`
	Happens(ic:cOp(_ID), t1, R(t1, t1)) ^ **Happens**(ir:cOp(_ID, _V_c), t2, R(t1,t2)) ⇒ **Rel**(_V_c, VAL)

In the above pattern

- *cOp* signifies the context operation identified by the semantic category specified in the ontology expressed in the *ContextCategory* element in the context constraint. The first **Happens** predicate signifies the occurrence of the context operation, while the second **Happens** predicate signifies the response from the context operation
- V_c signifies the return value from the context operation
- ***Rel***(V_c, *VAL)* signifies a relational predicate involving the return value of the context operation.

Example of Contextual Constraints translated into EC

Table 2 shows an example of *SerDiQueL* contextual constraints translated into applying the pattern discussed above. This example shows the EC representation of the contextual constraint in the *SerDiQueL* query in Figure 4, which specifies that response time of the *getLocation* operation returned by its context operation should be less than 10 seconds. In the EC representation the operation *getReponseTime* is the context operation that retunrs the response time of operation *getLocation*.

Table 2: Example of SerDiQueL contextual condition translated into EC

SerDiQueL	<csql:ConstraintQuery name="CQ1" contextual="true" type="SOFT" scope=" MONITORING"> <csql:LogicalExpression> <csql:Condition relation="LESS-THAN"> <csql:Operand1> <csql:ContextOperand serviceOperationName="getLocation" serviceID="2009.005"> <csql:ContextCategory relation="EQUAL-TO"> <csql:Category1> <csql:Document location="http://eg.org/CoDAMoS_Extended.xml" type="ONTOLOGY"/> </csql:Category1> <csql:Category2> <csql:Constant type="STRING">RELATIVE_TIME</csql:Constant> </csql:Category2> </csql:ContextCategory> </csql:ContextOperand > </csql:Operand1> <csql:Operand2> <csql:Constant type="NUMERICAL">10</csql:Constant> </csql:Operand2> </csql:Condition> </csql:LogicalExpression> </csql:ConstraintQuery>
EC	**Happens**(ic:getResponseTime(ID), t1, R(t1, t1)) ^ **Happens**(ir:getResponseTime(ID, rv), t2, R(t1,t2)) ⇒ rv < 10

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 invoked 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 [9] and by an event receiver. The external service registry uses eXist [3] database. Communication with the registry is through the use of Remote Method Invocation (RMI).

To evaluate MoRSeD, we have performed a set of experiments to measure and analyse the performance of both pull and push modes of query execution with queries incorporating structural, behavioral, non-contextual, and contextual conditions. In the evaluation we compared the performance of query executions for two cases, namely:

Case(1) - evaluation of the runtime service discovery framework without using the monitor component. In this case we assume that the behavioural and contextual specifications of the services are available in the service registry. We assume the behavioural specifications expressed in BPEL and the contextual specifications expressed in XML format

Case(2) - evaluation of the MoRSeD framework using the monitor component to support evaluation of behavioural criteria in service discovery queries against candidate services.

4.1 Experimental Setup

In the experiments, we used a registry with 150 services with 750 operations in total. In case (1), the registry had a total of 600 facets with structural, behavioural, quality, and context facets, while in case (2) the registry had a total of 450 facets with structural, quality, and context facets. The services used in the experiments were concerned with the GPS service domain of the "*Route Planner*" scenario (see Section 1). The evaluation of the framework's performance was incremental considering registries with 50, 100 and 150 services each time. The incremental evaluation was adopted in order to analyse how the increase in the number of services affects the query execution time. The time taken to execute each query for different registry sizes was calculated as the average across five executions of the query using a Pentium machine of 2.33 GHz with 3.23 GB RAM.

Table 3: Types of queries used in the experiments

Q1	Structural
Q2	Structural and behavioural
Q3	Structural, behavioural and soft non-contextual constraint
Q4	Structural, behavioural, soft non-contextual constraint and contextual constraint

In the experiments, we measured the time taken for executing four different queries drawn from the "*Route Planner*" scenario. The queries included different types of criteria, as summarized in Table 3. For each different type of criteria we used the same weight value of 1. More specifically, we used variants of the query described in Figure 4 without the hard constraint. The reason for not using hard constraints in the experiments was because such constraints could filter out services before the ranking stage during query execution and, therefore, artificially reduce the query execution time.

The query used in the experiment had one extra soft non-contextual and one extra contextual constraint from the query in Figure 4. These constraints are shown in Figure 8.

As shown in Figure 8, the constraint sub-query (C1) is a soft non-contextual constraint representing the fact that the service to be identified to replace the GPS service needs to be available 24 hours a day. This constraint has a weight of 0.5 and is represented by the conditions that verify if the opening time hours specified in the facet QoS has a minimum value of 00:00 and a maximum value of 24:00. This is specified by a conjunction of two LogicalExpression elements with their respective XPath expression contents and constant sub-elements.

The second constraint sub-query (C2) is a soft contextual constraint concerned with the time to process the payment to use GPS service. This constraint specifies that any candidate service needs to have a context operation associated with operation *makePayment* classified in the category GREDIA_RELATIVE_TIME in ontology http://eg.org/CoDAMoS_Extended.xml, and the result of executing this operation has to be equal to SECONDS-5 for this service to be considered.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Created with Liquid XML Studio 1.0.8.0 (http://www.liquid-technologies.com) -->
<tns:ServiceQuery xmlns:tns="http://gredia.eu/schema/SerDiQueL" xmlns:csql="http://gredia.eu/schema/Constraint_SQL"
   xmlns:tnsb="http://gredia.eu/schema/Behavour_SQL"  xmlns:tnsc="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   queryID="UUID:550e8400-e29b-41d4-a716-446655440000" name="Query1">

   <tns:Parameter name="mode" value="PUSH" />
   <tns:Parameter name="type" value="dynamic" />
   <tns:Parameter name="threshold" value="1.0" />
```

```
  <!-- Structural sub-query -->
  <tns:StructuralQuery> <!-- WSDL of the GPS service -- ></tns:StructuralQuery>

  <!-- Behavioural sub-query -->
  <tnsb:BehaviourQuery> ….. ….. …. </tnsb:BehaviourQuery>

  <!-- Constraints sub-queries -->
  <tnsa:ConstraintQuery name="C1" type="SOFT" contextual="false" weight="0.5">
    <tnsa:LogicalExpression>
      <tnsa:Condition relation="EQUAL-TO">
        <tnsa:Operand1>
          <tnsa:NonContextOperand facetName="QoS" facetType="QoS">
              //QoSCharacteristic[Name="Availability"]/Metrics/Metric[Name="OpenTime"][Unit="Hours"]/MinValue
          </tnsa:NonContextOperand>
        </tnsa:Operand1>
        <tnsa:Operand2>
          <tnsa:Constant type="STRING">00:00</tnsa:Constant>
        </tnsa:Operand2>
      </tnsa:Condition>
      <tnsa:LogicalOperator>AND</tnsa:LogicalOperator>
      <tnsa:LogicalExpression>
        <tnsa:Condition relation="EQUAL-TO">
          <tnsa:Operand1>
            <tnsa:NonContextOperand facetName="QoS" facetType="QoS">
              //QoSCharacteristic[Name="Availability"]/Metrics/Metric[Name="OpenTime"][Unit="Hours"]/MaxValue
            </tnsa:NonContextOperand>
          </tnsa:Operand1>
          <tnsa:Operand2>
             <tnsa:Constant type="STRING">24:00</tnsa:Constant>
          </tnsa:Operand2>
         </tnsa:Condition>
      </tnsa:LogicalExpression>
     </tnsa:LogicalExpression>
  </tnsa:ConstraintQuery>

  <tnsa:ConstraintQuery name="C2" contextual="true" type="SOFT" weight="0.5">
      <tnsa:LogicalExpression>
        <tnsa:Condition relation="LESS-THAN-EQUAL-TO">
          <tnsa:Operand1>
            <tnsa:ContextOperand serviceID="7021.0051" serviceOperationName="makePayment">
              <tnsa:ContextCategory relation="EQUAL-TO">
                <tnsa:Category1>
                   <tnsa:Document location="http://eg.org/CoDAMoS_Extended.xml" type="ONTOLOGY" />
                </tnsa:Category1>
                <tnsa:Category2>
                   <tnsa:Constant type="STRING">GREDIA_RELATIVE_TIME</tnsa:Constant>
                </tnsa:Category2>
              </tnsa:ContextCategory>
             </tnsa:ContextOperand>
          </tnsa:Operand1>
          <tnsa:Operand2>
              <tnsa:Constant type="STRING">SECONDS-5</tnsa:Constant>
          </tnsa:Operand2>
        </tnsa:Condition>
      </tnsa:LogicalExpression>
   </tnsa:ConstraintQuery>

</tns:ServiceQuery>
```

Fig. 8. Example of the constraint query used in the evaluation specified in SerDiQueL

4.2 Performance Results

Table 4 presents the execution times in milliseconds of queries Q1 to Q4 in the pull mode of query execution and the average time required for retrieving services from the registry, for the different sizes of service registries for both case (1) and case (2). Table 5 presents a breakdown of the total query execution time into the time that was needed to: (a) retrieve services from the registry, (b) execute structural matching, (c) execute behavioural matching, (d) execute soft non-contextual matching, and (e) execute contextual matching in both cases in the experiments.

It should also be noted that in the results presented in Tables 4 and 5, all the *n* services that were included in the registries of different sizes *n*, were evaluated against all the criteria that were included in each query. This means that in no case the evaluation of any of the criteria in a query was performed against a number of services that was smaller than the given registry size *n*.

Table 4. Experiment results for each query in pull mode of execution (msec)

Number of Services	**50**		**100**		**150**	
	Case 1	**Case 2**	**Case 1**	**Case 2**	**Case 1**	**Case 2**
Registry Retrieval	17867	17941	34747	34532	51577	51257
Q1	1653	1695	3295	3281	4924	4869
Q2	15208	52346	29520	103855	44106	155937
Q3	15384	52512	29836	104161	44571	156384
Q4	24822	61922	45979	120164	67997	180825

As shown in Tables 4 and 5, the time taken to retrieve services from the registry was significantly larger than the time taken to execute the different types of matchings. This is because the eXist database [3] that was used to implement the registry offers a low data retrieval performance. Although the implementation of the service registry is not the focus of our work, the use of a proactive push mode of query execution presented in this paper, alleviated this problem given that replacement services are selected in parallel to the execution of an application from an up-to-date set of candidate services, as discussed below and shown in Table 6. Moreover, except in the case of changes in the context of an application environment, the set of candidate services has a reduced number of services when compared to an entire service registry.

Table 5. Experiment results for different matching criteria in pull mode of execution (msec)

Number of Services	**50**		**100**		**150**	
	Case 1	**Case 2**	**Case 1**	**Case 2**	**Case 1**	**Case 2**
Registry Retrieval	17867	17941	34747	34532	51577	51257
Structural	1653	1695	3295	3281	4924	4869
Behavioural	13555	50651	26225	100574	39182	151068
Non-Contextual	177	166	316	306	465	447
Contextual	9437	9410	16143	16003	23426	24442
Total	42689	79863	80727	154696	119573	232082

As shown in Tables 4 and 5, the matching time for all the different queries increased linearly with the addition of more services in the registry. Furthermore, the execution time for different types of matching criteria also increased linearly with the number of services in the registry in all cases.

The experiment also showed that the times taken to perform behavioural matching in both cases were substantially higher than the times taken for each of the other matchings. This is because in case (1), in the behavioural matching, a path representing the behavioural part of a query needs to be evaluated for all the paths in the state machine that represents a service, and this process had to consider all the possible combinations of mappings between query and service operations. In case (2), the time to perform behavioural matching was larger than the time required to perform the same matching in case (1). This is because in case (2) the monitor generates runtime tests for each candidate service in the registry and verifies the behavioural properties against the generated events. To generate these tests, however, the monitor must computes all the possible sequences of operations that are specified in the WSDL of the candidate service and invoke the operations in the exact order of their appearance in each sequence. Furthermore, the monitor has to capture the SOAP messages exchanged between the service and its client in each test and transform these messages into EC events which are checked against the EC monitoring formulas that represent the behavioural conditions of the query. The increase of the time to perform behavioural matching was observed in queries Q2, Q3, and Q4 since these queries have behavioural constraints. It should be appreciated, however, that this increase in the query execution time is justified by the need to support behavioral matching even when there are no behavioural specifications for services in registries, something that is often the case as indicated by existing public service registries (e.g., SEEKDA).

Furthermore, as shown in Table 5, the time that was taken to evaluate non-contextual constraints was smaller than the time taken for each of the other matching criteria. Another observation that should be noted in connection to this time is that it was also significantly lower than the time taken for evaluating contextual constraints. This was because in non-contextual constraint matching, the non-contextual condition in a query is evaluated against facets in the registry by comparing elements retrieved by evaluating XPath expressions. In the case of contextual matching, however, the computation is more expensive as it requires the invocation of context operations at runtime in order to obtain the context values for evaluating the context conditions in queries. The results for the contextual matching are similar for both case (1) and case (2). This is due to the fact that in both cases, operations in context services are invoked to provide contextual information that needs to be evaluated against the contextual requests.

Table 6 presents the results of executing query Q4 in push mode for cases (1) and (2). The table shows the time needed to: (a) prepare the set of candidate services for a subscribed query at the initial stage in the process and (b) identify a service for replacing a service S in the service-based application due to (i) unavailability of S, (ii) availability of a new service, or (iii) changes in the service bound to the application. We have executed the query five different times for each of situations (i), (ii) and (iii), for each registry size (i.e. 50, 100 and 150 services). In each run, we have simulated the events concerned with the relevant cases. Table 6 presents the average time across all the runs.

As shown in Table 6, the times to identify a service due to situations (i), (ii), and (iii) are very small when compared to the time to identify a service in pull mode of query execution. The initial time required for building the set of candidate services for a subscribed service and query in the push mode of query execution, however, is comparable to the time needed for executing a query in pull mode in both cases (1) and (2). It should be appreciated, however, that in the case of push mode, the initial phase for building the set of candidate services is performed only once for a subscribed service and query, and in parallel to the execution of the application, and the real time needed for identifying a service due to situations (i), (ii) and (ii) are the ones shown in the last three rows of Table 6.

Furthermore, one should consider the longer term cost of the two modes of query execution. More specifically, assuming that the service associated with query Q4 becomes unavailable several times, in the pull mode of query execution the total cost of service discovery required to identify replacement services using Q4, will be in average 42689 milliseconds for each time (for case (1)) and 79863 milliseconds (for case (2)), with a service registry of 50 services; 80727 milliseconds for each time (for case (1)) and 154696 milliseconds (for case (2)), with a service registry of 100 services; and 119573 milliseconds for each time (for case (1)) and 232082 milliseconds (for case (2)), with a service registry of 150 services. In contrast, in the push mode of query execution, the respective total average times will be 27 milliseconds (for case (1)) and 25 milliseconds (for case (2)), for each time. Similar discrepancies exist for the cases in which a new service becomes available (average times of 798 and 1467 milliseconds for cases (1) and (2), respectively, for each time), or there is a change in a service (average times of 828 and 1479 milliseconds for cases (1) and (2), respectively, for each time). Moreover, in the push modes of query execution, the above activities will be executed in parallel to the execution of the application.

The results in Table 6 show that the time to identify a service due to unavailability (situation (i)) of a service is smaller than the time to identify a service due to changes in a service (situation (iii)) or the time to evaluate a new service that becomes available (situation (ii)), in both cases (1) and (2). This is due to the fact that situations (ii) and (iii) require the execution of the matching process between the service and the query in order to calculate their respective distance, while in situation (i) a replacement service is taken from the set of candidate services. Moreover, the average times to identify a replacement service due to unavailability of a participating service are very similar for both cases (1) and (2). However, the times for the situations concerned with the availability of a *new service* (ii) and *change in a service* (iii) in case (2) are substantially larger than the times for these situations in case (1). This is because, in both cases, in situation (i) a replacement service is taken from the set of candidate services, while in situations (ii) and (iii) it is necessary to match a service against the query and, in case (2), the behavioural matching is executed by the use of the monitor component that causes increase of the time.

Table 6: Times to execute query Q4 in push mode of execution (in milliseconds)

			50	**100**	**150**
Prepare Candidate Services	**Registry Retrieval**	**Case 1**	17999	37185	52684
		Case 2	18249	34951	51571
	Structural	**Case 1**	1109	3734	5547
		Case 2	2187	3859	5406
	Behavioural	**Case 1**	13906	26967	39107
		Case 2	52200	102318	152857
	Non-Contextual	**Case 1**	187	343	484
		Case 2	203	344	500
	Contextual	**Case 1**	9327	16405	23952
		Case 2	9608	16216	24138
	Total	**Case 1**	42576	84712	121899
		Case 2	82526	157767	234551
Identification			**Average**		
	Unavailability (i)	**Case 1**	27		
		Case 2	25		

of Replacement Service	**New service (ii)**	**Case 1**	798
		Case 2	1467
	Service change (iii)	**Case 1**	828
		Case 2	1479

We should note that Table 6 presents no results related to changes in the context of the application environment. In this case, a new query must be created and evaluated against all the services in the registry. Therefore, the time to identify a service to replace an existing service due to change in the context of an application environment is equivalent to the time to execute a query in pull mode.

Overall, the results of our experiments have demonstrated that our framework has good performance and that the use of a proactive (push mode) of service discovery provides a considerable gain in the time required for identifying replacement services at runtime. In addition, the decrease in the performance caused by the use of the monitor component is justifiable when it is not possible to guarantee the existence of behavioural service specifications in service registries. It should be noted, however, that the purpose of using the monitor component is not to show that this component is better, but to show that the monitor could be an alternative for performing service discovery in the absence of behavioral and contextual characteristics of the services in the registry, as explained in Section 1.

5 Related Work

In this section we review several approaches that are related to the work described in this paper. More specifically, we review works in the topics of (i) runtime service discovery and (iii) runtime monitoring of service based systems.

The use of semantic matchmaking approaches based on logic reasoning has been advocated in [10][18][19][21][22][23][34][36][37][38][39][40]. These approaches do not consider dynamic service discovery in pull and push modes of query execution. Distributed architecture has been exploited in [34][36][37][38][39][40] to avoid bottleneck or single point failure during service discovery process. Most of these approaches assume service descriptions expressed in OWL-S [41] or RDF [42] and queries expressed using similar semantic information that is used to describe services. A software agent based service discovery and execution architecture is presented in [33]. In this architecture software agents monitor the actions of their human counterparts to develop a user profile containing text values and context information. The user profile is used to search open repositories of web services based on syntactic matching.

Other approaches for service discovery consider graph transformation rules [18][20], or behavioural matching [15][17][24][28]. The work in [20] is limited since it cannot account for changes in the order or names of the parameters, a limitation that is not present in our approach. The approach in [17] proposes the use of (abstract) behavioural models of services to increase the precision of the discovery process. Similarly, in [28], the authors use service behaviour signatures to improve service discovery. The works in [16] and [29], describe functional and quality cross cutting concerns of components and services as aspects and discovery is based on a formal analysis and validation of these descriptions. In [28] a query language based on first-order logic that focuses on properties of behaviour signatures is used to support the discovery process. The work in [24] advocates the use of behavioural specifications represented as BPEL for service discovery for resolving ambiguities between requests and services and use a tree alignment algorithm to identify matching between request and services. However, the above approaches have not been used to support service discovery in a proactive way during the execution of service-based systems, as out approach does.

The work in [30] proposes QoS-based selection of services. In [21], the authors present a goal-based model for service discovery that considers re-use of pre-defined goals, discovery of relevant abstract services described in terms of capabilities, and contracting of concrete services to fulfil requesting goals. Our work differs from these works since it includes other criteria for service selection in a pro-active way.

Several approaches have also been proposed to support context awareness in service discovery [12][14][31][33][35][43]. In [14], context information is represented by key-value pairs attached to the edges of a graph representing service classifications. Unlike our framework, 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. A similar approach is described in [35], where context and QoS queries are bundled together. In [12][43] queries, services, and context information are expressed in ontologies. Context information in [12] can also be used as an implicit input to a service that is not explicitly provided by the user (e.g. user location). In [33], context is treated as the description of the environment in which a user performs his/her daily routines. This context information is extracted by continually monitoring users' action and used to predict what services and/or information to present to the user in the future. The work in [31] locates components based on context-aware browsing. In this approach, the interaction of software developers with the development environment is monitored and candidate components that match the development context based on signature matching are identified and presented to developers for browsing. Unlike our approach, the above context-aware approaches support simple conditions regarding context information in service discovery, do not fully integrate context with behavioural criteria in service discovery, and have limited applicability since they depend on the use of specific ontologies for the expression of context conditions.

Some query languages have been proposed to support web services discovery [11][25][26][27][32]. In [11] the authors propose BP-QL a visual query language for business processes expressed in BPEL. The behavioural part of the query language used in SeDiQueL also supports querying BPEL specifications. However, our work differs from BPQL since it supports the specification of structural, quality, and contextual conditions in the query, and the behavioural conditions can be matched against the execution of the services. The query language proposed in [27] is used to support composition of services based on user's goals. NaLIX [32], which is a language that was developed to allow querying XML databases based on natural language, has also been adapted to cater for service discovery. In [25], the authors propose USQL (Unified Service Query language), an XML-based language to represent syntactic, semantic, and quality of service search criteria. The query language used in our framework is more complete, since it accounts for the representation of behavioural aspects of the application being developed and services to be discovered, as well as context characteristics of services and application environments. An extension of USQL that incorporates behavioural based on UML sequence diagrams has been proposed in [26]. The behavioural sub-query of SerDiQueL is not only restricted to the representation of sequence of operations, but it allows for the representation of other types of behavioural aspects.

Overall, most of the proposed approaches support service discovery for specific types of service criteria in a reactive way and only in pull mode of query execution. Unlike them, our framework supports proactive dynamic service discovery based on a comprehensive set of service and application properties including structural, functional, quality, and contextual properties. It also provides pull and push service discovery mechanisms, optimising service replacement during the execution of an application. Furthermore, our approach provides an expressive query language allowing the specification of a wide spectrum of constraints for the required services and does not require the existence of behavioural and contextual service specifications, as in our previous work [78][79][80].

Run-time monitoring has been the focus of research in the context of different areas including requirements engineering [50][51][52], program verification [53][54][55][56], service centric systems [57][60][47], and context aware systems [62][64][65].

Work in the area of monitoring service-centric systems has focused on the development of standards and languages for specifying monitorable properties and methods for monitoring these properties [57][4][60]. Runtime monitoring has also focused on monitoring service level agreements (SLAs) [58][59]. In [47] a framework is presented to allow non-intrusive adaptation of partner services within BPEL process without any down time of the overall system. In this approach a BPEL process is monitored according to certain QoS criteria and existing partner services may be replaced (in case a partner fails to satisfy QoS criteria) based on various replacement strategies. The replacement service can be syntactically or semantically equivalent to the interface used in BPEL. Formal verification techniques are used to verify the runtime behavioural correctness of service centric systems in [88][89][90][91]. In [88][89], safety and liveness properties of service centric systems are expressed using a subset of UML sequence diagram. These diagrams are transformed into automata applying some formal translation patterns. During the execution of service centric system the messages exchanged between the participating services are captured and used to update the states of the automata to verify the correctness of the execution. In [90][91] a formal model of a web service is constructed using a variant of finite state machine and test cases are generated from this formal model. Generated inputs are fed to the web service to verify that the implementation of the web service conforms to the formal model. The approaches described in [87][92] apply aspect oriented programming for runtime monitoring of service centric systems. In [87], monitorable properties of conversational web services (i.e. stateful services) are expressed in algebraic specifications. A mapping between the operations of a conversational web service and the operations in the corresponding algebraic specification is defined. An evaluator holding the runtime representation of the algebraic specification is dynamically attached to a service execution engine and at runtime it observes the execution of the web service and evaluates whether the corresponding algebraic specification is preserved. Streaming XML evaluation technique is used in [93] for runtime verification of web service choreographies. Choreography constraints are expressed in Linear Temporal Logic (LTL) and then the constraints are translated into XQuery expressions applying some transformation patterns. These XQuery expressions are verified against the runtime XML messages exchanged among the web service using standard XML streaming engine.

In context-based monitoring, a set of rules defining the properties that should be monitored to detect changes of the context are specified [61][62][63]. Some formal [62][63] or semi formal [61] languages are used to specify the properties to be monitored. The monitoring techniques in this field facilitate a wide range of stakeholders for different purposes: they are exploited to help the application developers to design the system that will adapt the user interface based on context changes [64][65][66][67][68][69]; they may help the service provider to better understand the user's required quality of service and improve the delivered QoS [70][71]. Context information can be measured at different level of abstractions, for example low-level context information can be directly captured from the environment using sensors, input devices, and high level context information can be inferred from low level context information and other information sources e.g. browsing user profile [65][71][72][73]. System run-time events (i.e. context information) are matched against the specified properties to detect a change in the context. Run-time events or context information are obtained either from sensors [74], by polling system parameters (e.g. battery level in mobile phone or available bandwidth) [75][76] or user input [74]. Given the distributed nature of context-aware applications, context-based monitoring is mostly implemented as distributed architecture with middleware support [74][75][76][77]. In this setting, a component in the middleware acts as a coordinator that collects context information from distributed sources and forwards the context information to the specific application/monitor that performs the reasoning using context information.

Most of the monitoring approaches discussed above perform monitoring by weaving code that implements the required checks inside the code of the system that is being monitored or the service centric system execution environment. However in our monitoring approach monitoring is carried out by a computational entity that is external to the system that is being monitored, is carried out in parallel with the operation of this system and does not intervene with this operation in any form. Moreover most of the discussed monitoring approaches use some form of linear temporal logic (LTL) or state machines to specify the monitorable properties. Hence, the monitoring conditions that they can support are only relative (e.g. an operation must be executed prior to another an operation) and cannot involve absolute time conditions (e.g., an operation invocation must produce a response within N milliseconds) and/or time boundaries (e.g., an operation cannot be performed before 8am or after 10.00pm). But the specification of temporal constraints with specific time boundaries are essential in specifying and verifying temporal aspects of the execution of computer programs [95], and therefore service based systems. In the monitoring approach of MoRSeD, we specify monitorable properties in Event Calculus which has an explicit time structure allowing the specification of complex quantitative temporal conditions, such as conditions about the exact time that can elapse between events and conditions regarding the time range within which events are expected to occur.

6 Conclusions and Future Work

In this paper we present a monitor-based runtime service discovery framework that supports the identification of services based on different characteristics of the service including structural, behavioural and contextual characteristics. We establish the necessity of considering different characteristics of the service to provide better precision when identifying services to replace existing services during runtime. In the proposed framework, service discovery queries are specified in an XML-based query language, called SerDiQueL and it allows to express combination of various characteristics of the service based systems such as structural, behavioural, quality and contextual conditions. The framework allows the discovery of services that have multi faceted descriptions including service interface, behaviour, quality and context descriptions. The query execution is based on the computation of distances between query and services specifications. Our service discovery framework requires at least the existence of structural description in the service registry and ensures the verification of behavioural and contextual characteristics of services even when there are no available behavioural specifications and up to date contextual values of different aspects of the services in the registry. The proposed runtime service discovery framework supports identification of services based on service discovery queries in both classic pull mode and proactive push mode of query execution. In classic pull mode of query execution, as found in most of the approaches in the literature, a service discovery is triggered only after the need for a new service arises. In such cases, pull mode service discovery needs to wait until the occurrence of a problem in an existing service that would lead to the execution of a query and identification of a better replacement service. The whole process may take considerable time to complete and affect the performance of the service based system. In addition to the classic pull mode query our proposed framework supports proactive push mode service discovery where query execution is performed in parallel to the execution of the service based system based using pre-subscribed queries. These queries are associated with specific service binding points in the service based system and aim to maintain up-to-date sets of candidate replacement services for these binding points.

In absence of behavioural specification of a service in the service registry, the verification of behavioural characteristics of the service is performed by the monitor component of our framework. The monitor deploys a service client for each service that needs to be monitored and the service client invokes the services and generates runtime events. The monitor verifies the satisfiability of the behavioural characteristics of services represented in the service discover query against the runtime events. However, the current implementation of the framework can not guarantee the check for the satisfiability of behavioural properties with respect to all possible workflow patterns that may appear in a service based system. Currently we are extending the framework to ensure the satisfiability check of the behavioural characteristics of the services with respect to all possible workflow patterns.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under Grant Agreement 215483 (S-Cube).

References

1. SECSE Project. http://secse.eng.it.
2. WSDL. http://www.w3.org/TR/wsdl
3. eXist. http://exist.sourceforge.net
4. K. Mahbub, G. Spanoudakis. "A framework for Requirements Monitoring of Service Based Systems", 2nd International Conference on Service Oriented Computing (ICSOC 2004), pp 84 – 93, November 2004.
5. M. Shanahan. "The event calculus explained", In M. J. Wooldridge and M. Veloso, editors, Articial Intelligence Today, Vol. 1600 of LNCS, pages 409--430. Springer, 1999
6. K. Mahbub, G. Spanoudakis. "Run-time Monitoring of Requirements for Systems Composed of Web- Services: Initial Implementation and Evaluation Experience", IEEE International Conference on Web Services (ICWS'05), pp. 257-265, 2005.
7. "Web Services Business Process Execution Language Version 2.0", OASIS Standard, 11 April 2007
8. Axis2, http://ws.apache.org/axis2/
9. WS-Eventing. http://www/w3/org/Submission/WS-Eventing
10. R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint Driven Web Service Composition in METEOR-S, Int. Conf. on Services Comp. 2004.
11. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. 32nd International Conference on Very Large Data Bases, VLDB, Korea, September (2006).
12. F. Bormann, et al, Towards Context-Aware Service Discovery: A Case Study for a new Advice of Charge Service", 14th IST Mobile and Wireless Communications Summit, June 2005.
13. L. Choonhwa and S. Helal. Context Attributes: An Approach to Enable Context-awareness for Service Discovery, 2003 Symp. on App. & the Internet.

14. S. Cuddy, M. Katchabaw, and H. Lutfiyya. Context-Aware Service Selection Based on Dynamic and Static Service Attributes.IEEE Int. Conf. on Wireless and Mobile Computing, Networking and Comm., 2005.
15. D. Grirori, J.C. Corrales, and M.Bouzeghoub. Behavioral Matching for Service Retrieval, International Conference on Web Services, ICWS 2006, USA, September 2006.
16. J. Grundy and G. Ding. Automatic Validation of Deployed J2EE Components Using Aspects. IEEE 16th International Conference on Automated Software Engineering, USA, November 2001.
17. R.J. Hall and A. Zisman. Behavioral Models as Service Descriptions, Int. Conf. on Service Oriented Computing, USA, 2004
18. J.H. Hausmann, R. Heckel and M. Lohman. Model-based Discovery of Web Services, Int. Conf. on Web Services, 2004.
19. W. Hoschek. The Web Service Discovery Architecture, IEEE/ACM Supercomputing Conf., USA, 2002.
20. U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel. Automatic Location of Services, European Semantic Web Conference, 2005.
21. M. Klein and A. Bernstein. Toward High-Precision Service Retrieval. IEEE Internet Computing, 30-36, January 2004.
22. M. Klusch, B. Fries, and K. Sycara. Automated Semantic Web Service Discovery with OWLS-MX, Int. Conf. on Autonomous Agents and Multiagent Systems, 2006.
23. L. Li and I. Horrock. A Software Framework for Matchmaking based on Semantic Web Technology, WWW Conf. Work. on Eservices and the Semantic Web, 2003.
24. R. Mikhaiel and E. Stroulia. “Interface- and Usage-aware Service Discovery”, 4th International Conference on Service Oriented Computing (ICSOC), December 2006.
25. M. Pantazoglou, A. Tsalgatidou, and G. Athanasopoulos. Discovering Web Services in JXTA Peer-to-Peer Services in a Unified Manner. 4th International Conference on Service Oriented Computing (ICSOC), December (2006).
26. M. Pantazoglou, A. Tsalgatidou, and G. Spanoudakis, G.: Behavior- aware, Unified Service Discovery. In Proceedings of the Service-Oriented Computing: a look at the inside Workshop, SOC@Inside'07, Austria, September, 2007.
27. M. Papazoglou, M. Aiello, M. Pistore, J. Yang. XSRL: A Request Language for web services, http://citeseer.ist.psu.edu/575968.html
28. Z. Shen and J. Su. Web Service Discovery based on Behavior Signatures. Int. Conf. on Service Computing , SCC, July 2005.
29. S. Singh, J. Grundy, J. Hosking, J. Sun. An Architecture for Developing Aspect-Oriented Web Services, 3rd European Conf. in Web Services, 2005.
30. X. Wang, T. Vitvar,, T. Kerrigan, and I. Toma . “A QoS-Aware Selection Model for Semantic Web Services”, 4th International Conference on Service Oriented Computing, ICSOC, USA, 2006
31. Y. Ye and G. Fischer. Context-Aware Browsing of Large Component Repositories. IEEE 16th International Conference on Automated Software Engineering, ASE, USA, November 2001.
32. L.Y. Yunyao, H. Yanh, and H. Jagadish,. NaLIX: an Interactive Natural Language Interface for Querying XML, SIGMOD 2005,Baltimore, June (2005).
33. M.B. Blake, D.R. Kahan, and M.F. Nowlan, “Context-Aware Agents for User-Oriented Web Services Discovery and Execution” Special Issue on Context-Based Web Services, Distributed and Parallel Databases, Vol. 21, No. 1, pp 39-58, February 2007
34. Lynch, D., Keeney, J., Lewis, D., O'Sullivan, D, "A Proactive approach to Semantically Oriented Service Discovery", Proceedings of the Second Workshop on Innovations in Web Infrastructure (IWI 2006), Co-located with the 15th International World-Wide Web Conference, Edinburgh, Scotland. May 2006.
35. I. Braun and A. Strunk, G. Stoyanova and B. Buder, "ConQo - A Context- And QoS-Aware Service Discovery", IADIS; Proceedings of WWW/Internet; 2008
36. K. Arabshian and H. Schulzrinne, Distributed Context-aware Agent Architecture for Global Service Discovery, The Second International Workshop on Semantic Web Technology For Ubiquitous and Mobile Applications (SWUMA'06), Trentino, Italy, August 2006
37. K. Arabshian and H. Schulzrinne, An Ontology-based Hierarchical Peer-to-Peer Global Service Discovery System, Journal of Ubiquitous Computing and Intelligence Volume 1, Number 2, pp 133-144, December 2007

38. K. Arabshian, C. Dickmann, H. Schulzrinne, Service Composition in an Ontology-based Global Service Discovery System, Columbia University Technical Report CUCS-033-07, New York, NY, September 2007
39. K. Arabshian and H. Schulzrinne, Combining Ontology Queries with Key Word Search in Service Discovery, ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach California, November 2007
40. R. Romeikat, B. Bauer, "Towards Semantically-Enhanced Distributed Service Discovery", Proceedings of the Second International Conference on Internet and Web Applications (ICIW 2007), Le Morne, Mauritius, May 2007
41. OWL-S 1.0 Release. http://www.daml.org/services/owls/1.0/.
42. World Wide Web Consortium, W3C (2004): Resource Description Framework, W3C Recommendation 10 February 2004, http://www.w3.org/RDF/
43. F. Klan, "Context-aware service discovery, selection and usage", 18th GI-Workshop on the Foundations of Databases, June 2006
44. D. Ardagna and B. Pernici, "Adaptive Service Composition in Flexible Processes",IEEE Transactions on Software Engineering, Volume 33 , Issue 6 (June 2007)
45. L. Baresi, E. Di Nitto, C. Ghezzi, S. Guinea "A framework for the deployment of adaptable web service compositions", Service Oriented Computing and Applications 1(1): 75-91 (2007)
46. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, M. Shan, "Adaptive and Dynamic Service Composition in eFlow", Software rechnology laboratory, HPL-2000-39, March 2000
47. O. Moser, F. Rosenberg, S. Dustdar: "Non-intrusive monitoring and service adaptation for WS-BPEL". WWW 2008: 815-824
48. O. Moser, F. Rosenberg, S. Dustdar: "VieDAME - flexible and robust BPEL processes through monitoring and adaptation". ICSE Companion 2008: 917-918
49. M. B. Juric, B. Mathew, and P. Sarang. "Business Process Execution Language for Web Services: An Architect and Developer's Guide to Orchestrating Web Services", Packt Publishing, second edition, 2006.
50. M. Feather, S. Fickas. "Requirements Monitoring in Dynamic Environments". Proc. of Int. Conf. on Requirements Engineering, 1995
51. M. Feather, et al. "Reconciling System Requirements and Runtime Behaviour". Proc. of 9th Int. Work. on Software Specification & Design, 1998.
52. W. Robinson. "Monitoring Software Requirements using Instrumented Code". In Proc. of the Hawaii Int. Conf. on Systems Sciences, 2002.
53. Chen, F. and Rosu, G, (2003). "Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation". In Electronic Notes in Theoretical Computer Science 89 No. 2, Published by Elsevier Science B.V.
54. Havelund, K. and Roşu, G. (2004). "An Overview of the Runtime Verification Tool Java PathExplorer", Form. Methods Syst. Des. 24, pp.189-215.
55. Dingwall-Smith A., Finkelstein A. "From Requirements to Monitors by Way of Aspects". Proc. of 1st Int. Conf. on Aspect-Oriented Software Development, 2002
56. Capra L., et al. "Reflective middleware solutions for context-aware applications", LNCS 2192, 2001
57. Baresi, L. and Guinea, S. (2005). "Dynamo: Dynamic Monitoring of WS-BPEL Processes", ICSOC 05, 3rd International Conference On Service Oriented Computing, Amsterdam, The Netherlands
58. Ghezzi C., Guinea S. (2007), "Runtime Monitoring in Service Oriented Architectures", In Test and Analysis of Web Services, (eds) Baresi L. & di Nitto E., Springer, 237-264, 2007.
59. Mahbub K., and Spanoudakis G., (2007) "Monitoring WS-Agreements: An Event Calculus Based Approach" Springer monograph on Test and Analysis of Web Services, , (eds) L.Baresi, E. diNitto, Springer Verlang, 2007
60. Qin Li, (2007) "A Dynamic Verification Platform for BPEL Environments" MSc. Thesis, Department of Electrical & Computer Engineering, University of Alberta May 29, 2007
61. V. Talwar, C. Shankar, S. Rafaeli, D. Milojicic, S. Iyer, K. Farkas, and Y. Chen. Adaptive monitoring: Automated change management for monitoring systems. In Proceedings of the 13th Workshop of the HP OpenView University Association (HP-OVUA 2006), pages 21–24, 2006.
62. M. Salifu, Y. Yu, and B.Nuseibeh. Analysing monitoring and switching requirements using constraint satisfiability. In Technical Report- ISSN 1744-1986; Department of Computing; Faculty of Maths, Computing and Technology, UK, 2008.

63. M. Salifu, Y. Yu, and B. Nuseibeh. Specifying monitoring and switching problems in context. In 15th IEEE International Requirements Engineering Conference, 2007.
64. M-A. Hariri, D. Tabary, S. Lepreux, and C. Kolski. Context aware business adaptation toward user interface adaptation. In Communications of SIWN, pages 46–52. Springer Verlag, 2008.
65. K. Goslar, S. Buchholz, Alexander Schill, and H. Vogler. A multidimensional approach to context awareness. In International Institute of Informatics and Systemics; Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI2003), 2003.
66. L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. In In IEEE Transactions of Software Engineering Journal (TSE). November 2003, 2003.
67. A. Seffah, P. Forbrig, and H. Javahery. Multi-devices "multiple" user interfaces: development models and research opportunities. In Journal of Systems and Software 73, pages 287–300, 2004.
68. J. Eisenstein, J. V, and A. Puerta. Adapting to mobile contexts with user-interface modeling. In Proc. of 3 rd IEEE Workshop on Mobile Computing Systems and Applications WMCSA, 2000.
69. J. E. Bardram. The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. In In Hans Gellersen, Roy Want, and Albrecht Schmidt, editors, Proceedings of the 3rd International Conference on Pervasive Computing, Lecture Notes in Computer Science, Munich, Germany. Springer Verlag, 2005.
70. K.E. Wac. Towards qos-awareness of context-aware mobile applications and services. In In: Proceedings of the On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE (OTM2005), 31 Oct - 4 Nov 2005, Agia Napa, Cyprus. pp. 751-760. Lecture Notes in Computer Science 3760, pages 751–760. Springer Verlag, 2005.
71. M. Baldauf and S. Dustdar anf F. Rosenberg. A survey on context-aware systems. In International Journal of Ad Hoc and Ubiquitous Computing, pages 263–277, 2007.
72. J.-Z. Sun and J. Sauvola. Towards a conceptual model for context-aware adaptive services. In Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies, pages 27–29, 2003.
73. C. Anagnostopoulos, A. Tsounis, and S. Hadjiefthymiades. Context awareness in mobile computing environments: A survey. In Mobile eConference, 2004.
74. Newberger A. and Dey A. Designer support for context monitoring and control. In Intel Research, 2003.
75. C. Bettini, D. Maggiorini, and D. Riboni. Distributed context monitoring for continuous mobile services. In John Krogstie, Karlheinz Kautz, David Allen (Eds.): Mobile Information Systems II: IFIP Working Conference on Mobile Information Systems (MOBIS), pages 123–137. Springer, 2005.
76. C. Bettini, D. Maggiorini, and D. Riboni. Distributed context monitoring for the adaptation of continuous services. In World Wide Web Journal (WWWJ), Special issue on Multichannel Adaptive Information Systems on the World Wide Web. Springer, 2007.
77. P. Bratskas, N. Paspallis, and G. A. Papadopoulos. An evaluation of the state of the art in contextaware architectures. In Sixteenth International Conference on Information Systems Development (ISD 2007). Springer Verlag, 2007.
78. A. Zisman, G. Spanoudakis, and J. Dooley. "Proactive Runtime Service Discovery", IEEE 2008 International Service Computing Conference (SCC '08), Hawaii, July 2008.
79. A. Zisman, G. Spanoudakis, and J. Dooley. "A Framework for Dynamic Service Discovery", IEEE Int. Conference on Automated Software Engineering, ASE, Italy, September, 2008.
80. A. Zisman, G. Spanoudakis, J. Dooley. A Query Language for Service Discovery, 4th International Conference on Software and Data Technologies - ICSOFT 2009, Bulgaria, July 2009.
81. G. Spanoudakis, K. Mahbub, and A. Zisman, "A Platform for Context Aware Runtime Web Service Discovery", IEEE International Conference on Web Services (ICWS), July 9-13, 2007, Salt Lake City, Utah, USA
82. Cordella L.P., et al., An Improved Algorithm for Matching Large Graphs, 3rd IAPR-TC15 Work. on Graph-based Representations, 2001
83. Morato J., Marzal M. A., Llorens J., and Moreiro J., 2004. "WordNet Application", Proceedings of GWC 2004. The Second Global Wordnet Conf. 2004, Brno, Czech Republic.
84. A. Kozlenkov, G. Spanoudakis, A. Zisman, V. Fasoulas, F. Sanchez. Architecture-driven Service Discovery for Service Centric Systems, Electronic Government: Concepts, Methodologies, Tools,

and Applications, Chapter 2.24 (version of Journal below), (ed) Ari-Veikko Anttiroiko, Information Science Reference, 978-1-59904-947-2, pp 811-842, 2008.

85. A.Kozlenkov, G.Spanoudakis, A.Zisman, V. Fasoulas, F.Sanchez. Architecture-driven Service Discovery for Service Centric Systems, International Journal of Web Services Research, special issue on Service Engineering, 4(2), April-June 2007.
86. M. J. Duftler, N. K. Mukhi, Aleksander Slominski,and Sanjiva Weerawarana. "Web Services Invocation Framework (WSIF)". In Proceedings of the OOPSLA Workshop on Object–Oriented Web Services, October 2001.
87. D. Bianculli and C. Ghezzi. Monitoring Conversational Web Services. In IWSOSWE' 07, 2007.
88. Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, "Runtime monitoring of web service conversations", Proceedings of the 2007 conference of the center for advanced studies on Collaborative research, Richmond Hill, Ontario, Canada , 2007
89. J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, J. Waterhouse, "Runtime Monitoring of Web Service Conversations," IEEE Transactions on Services Computing, 29 Jun. 2009. IEEE computer Society Digital Library. IEEE Computer Society,
90. D. Dranidis, D. Kourtesis, E. Ramollari, Formal Verification of Web Service Behavioural Conformance through Testing, in Proc. of the 3rd South-East European Workshop on Formal Methods, November 2007
91. D. Dranidis, E. Ramollari, D. Kourtesis, Run-time Verification of Behavioural Conformance for Conversational Web Services, European Conference on Web Services, Eindhoven, November 2009.
92. S. Halle and R. Villemaire, "Runtime Monitoring of Message-Based Workflows with Data", 2008 12th International IEEE Enterprise Distributed Object Computing Conference.
93. S. Hallé, R. Villemaire, "Runtime monitoring of web service choreographies using streaming XML", Proceedings of the 2009 ACM symposium on Applied Computing, Honolulu, Hawaii.
94. K. Mahbub and A. Zisman, "Replacement Policies for Service-Based Systems", 2nd International Workshop on Service Monitoring, Adaptation and Beyond (MONA+), Collocated with ICSOC/ServiceWave, Stockholm, Swedeen, November 23-24, 2009.
95. Lamport, L., What Good is Temporal Logic". Information Processing 83:657-668, 1983.