Designing and Adapting Service-based Systems: A Service Discovery Framework[1]

George Spanoudakis and Andrea Zisman

Abstract.
This chapter describes a service discovery framework that has been developed within the EU 6th Framework projects SeCSE and Gredia. The framework supports design of service-based systems based on existing services and adaptation of service based systems during their execution due to different situations. It assumes services described from different perspectives and uses complex service discovery queries specified in a XML-based language that we have developed. The work is illustrated with the Cell Phone Operator case study.

1. Introduction

Service-based systems are defined as software systems that are composed of services but may also use legacy code and/or software components to provide their required functionality. The design of service-based systems has been recognized as an important topic of research in which it is necessary to have methodologies, techniques, and tools to support the development of such systems. Service integrators, developers, and providers have collaborated to support not only the development, but also the deployment and consumption of service-based systems. The deployment and support for adaptation of service-based systems during execution time has been recognized as necessary for these systems to continue to operate.

George Spanoudakis
City University London, Northampton Square, London, EC1V 0HB, UK, e-mail: g.spanoudakis@soi.city.ac.uk

Andrea Zisman
City University London, Northampton Square, London, EC1V 0HB, UK, e-mail: a.zisman@soi.city.ac.uk

As an example, consider the Cell Phone Operator (CPO) case study being used in this book. In this example the CPO system is composed of several services that support different functional and non-functional aspects of the system (e.g., SMS service, voice service, email service, pay-per-view movie service, mobile phone number portability service, cost and time to use the various services). During the design of this CPO service-based system, it is necessary to be able to identify available services that can be used to support the functional and non-functional aspects of the system, and to develop design models of the system based on the characteristics of existing services. Once the CPO is deployed, it may be necessary to replace a service during execution time of the system. For example, consider a customer who moves countries for a while (change of context), and assume that the time to retrieve a movie using the current pay-per-view movie service participating in the system becomes slow due to the new location of the customer. In this case, it is necessary to identify a service that can replace the current pay-per-view movie service and conforms to the requirements of the system.

In this chapter we present a service discovery framework supporting: (a) design of service-based systems based on existing services, and (b) adaptation of service-based systems during their execution due to (i) unavailability or malfunctioning of the services they deploy, (ii) changes in the context of services they deploy or the service-based system environment, and/or (iii) emergence of new services that are superior to the services already deployed in a service-based system.

The work underpinning the framework described in this chapter has been developed within the EU 6th Framework projects SeCSE [33] and Gredia [13]. Different parts and aspects of the framework have been published in several research papers [24,25,37,38,39,40,54,55,57,58,59]. In this chapter, we present the latest unified version of the framework and demonstrate how it can be applied to support service-based system design and execution time adaptation illustrated by the case study of cell phone operators used in this book.

The design of service-based systems in the framework is based on an iterative service discovery process in which system designers can, whilst developing system design models for a service based system, specify service discovery queries representing functional and quality characteristics of services required for them, and use them to locate services that could be used in the system. Once identified and, subject to their approval by the designers, such services can be linked to the system and used as remote components by it when the system comes to operation. When designers decide to use a discovered service, its model is also automatically integrated into the design model of the system and thus generating a new version of the model. The new version of the design model may be used in further iterations to specify other service requests, identify further candidate services, and possibly interconnect them to the design as well. During this process, it is also possible to realize that certain parts of the system cannot be fulfilled by available services and, therefore, make alternative design decisions for the developing system. Examples of these decisions are concerned with the use of existing legacy code or components that could be statically linked to the system, or the implementation of new software code. This design process terminates either when new serv-

ice requests derived from new versions of the models cannot identify services that match the requests or at the discretion of the designer of the system.

Execution time adaptations of service-based systems are assisted by a pro-active service discovery process in which services are identified in parallel to the execution of the system using pre-subscribed complex queries and services. The queries are identified and specified also prior to system deployment and are executed at runtime under specific conditions that make necessary the adaptation of the system. The queries locate alternative services for the ones already used by the system, which are no longer appropriate. As it is the case in the design of service-based systems, execution time service discovery can express combinations of structural, behavioural and quality conditions that should be satisfied by candidate services. In addition, they can also express context conditions (i.e., parametric conditions about characteristics of the system, its environment and its participating services that are, or could be deployed by the system, which can change frequently and dynamically at execution time). Furthermore, the execution time discovery queries supported by our framework can be executed in both pull and push modes. The former mode provides a reactive response to a runtime problem that makes the need for system adaptation necessary. The latter mode provides a means of pro-active and continuous discovery process that runs in parallel with the system aiming to identify appropriate substitute services for the ones already used by the system when the need for replacing services suddenly arises.

The support for service-based system design has been developed in the SeCSE project [33] to address challenges identified by industrial partners in the areas of telecommunications, automotive, and software in the project. The support for adaptation of service-based system has been developed in the GREDIA project [13] to address challenges identified by industrial partners in the areas of media and banking. These challenges point out the need to:

(i) Extract service discovery queries from design models of service-based systems specifying the functionality and quality properties of such systems;
(ii) Generate service discovery queries from characteristics of services that have already been deployed in systems, but may need to be replaced;
(iii) Provide a query language to support both the expression of arbitrary logical combinations of prioritised functional, non-functional, and contextual properties criteria for the required services, and similarity-based queries of the form "find a service that is similar to service X";
(iv) Match efficiently service discovery queries against service specifications and return services that may have varying degrees of match with the queries;
(v) Assist system designers to select services for a service-based system in cases where the discovery process identifies more than one candidate service satisfying a query or services that do not satisfy a query entirely;
(vi) Integrate discovered services into an iterative design process in which service-based systems design models may be re-formulated following the discovery of services;
(vii) Support pro-active dynamic service discovery during execution time of a service-based system.

The framework assumes services described from different perspectives by a set

of XML-based facets. These facets include (i) textual facets describing general information of the services in an XML format, (ii) structural facets describing operations of services with their data types using WSDL [53], (iii) behavioural facets describing behavioural models of services in BPEL4WS [7], (iv) quality of service facets describing non-functional aspects of services, and (v) context facets describing quality aspects of a service that change dynamically. The identification of services based on distinct aspects provides a more accurate match between queries and services and the consequent discovery of services with the required characteristics, as opposed to techniques that are based only on keywords or interface aspects (e.g., WOOGLE [52] and UDDI [43]), which provide less precise match. The discovery techniques that are used in the framework to assist with both the design and adaptation of service-based systems are based on the computation of distances between queries and the different types of service specifications.

The remainder of this chapter is structured as follows. Section 2 describes an overview of the framework to support design and adaptation of service-based systems. Section 3 presents the query language used in the framework. Section 4 describes the matching process. Section 5 discusses the advantages, lessons learned, and limitations of the work. Section 6 discusses related work on service discovery and service-based system adaptation. Finally, section 7 provides concluding remarks and discussion of future work. The material presented in the chapter is illustrated with the cell phone operator case study.

2. Overview of the Framework

As discussed in Section 1, the framework supports design and adaptation of service-based systems. Fig. 1 shows the overall architecture of our framework. As shown in the figure, the main components of the framework are: (a) *service requestor*, (b) *query processor*, and (c) *service registry intermediary*. The framework uses external service registries and is invoked by an external client application. In order to support execution time adaptation of service-based systems, the framework uses special servers and listeners to allow notification of changes in services and application environment. The external client applications support the creation of service requests to be executed for both design and execution time adaptation. These service requests may contain structural, behavioural, quality, and contextual characteristics.

The *service requestor* receives a service request from a client application. In the case of adaptation of service-based systems, the service requestor also receives context information about the services participating in a service-based system and application environment. The service requestor prepares service queries to be evaluated, organises the results of a query, and returns these results to the client application. To support adaptation, it also manages push query execution mode subscriptions, receives information from listeners about services that become available or about changes to existing services.

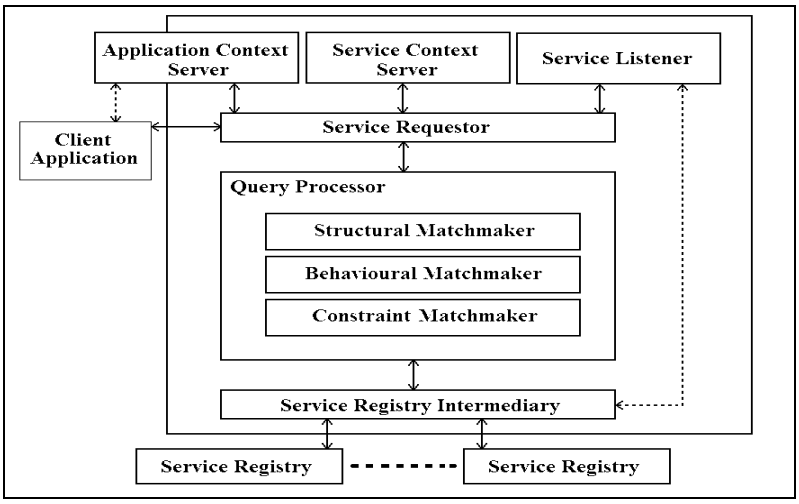

Fig. 1: Architecture overview of the framework

The *query processor* is responsible to parse the different parts of a query and evaluate these parts against service specifications in the various service registries. As shown in the figure, the query processor is formed by three sub-components, namely (i) structural, (ii) behavioural, and (iii) constraint matchmakers. Each of these sub-components is responsible to evaluate a different part of a query.

The *service registry intermediary* supports the use of different service registries and the discovery of services stored in different types of registries. It provides an interface to access services from various registries. The framework allows accessing services from registries organized as faceted structure, as proposed in the SeCSE project [33]. More specifically, in the registries, a service is specified by a set of XML-based facets, namely (i) textual facets, (ii) structural, (iii) behavioural, (iv) quality of service, and (v) context facets.

To support adaptation of service-based systems, the framework uses *service* and *application context servers*, and *service listeners*. The service and application context servers allow the acquisition of context information about the services and the application environment, respectively. Both context servers accept subscriptions for specific types of context information from the service requester and send updates when changes in the context of services and the application occur. The service listener is responsible to send to the service requestor notifications about new services that become available, or about changes in the descriptions of existing services. This information is extracted from external service registries through polling. The notifications are based on subscriptions for specific types of information that the service requester has made to the service listener.

The framework assumes constraints in a query to be *contextual* or *non-contextual*. A contextual constraint is concerned with information that changes dynamically during the operation of the service-based system and/or the services that the system deploys, while non-contextual constraint is concerned with static information related to structural, behavioural, and quality aspects of the services

and systems. The non-contextual constraints can be *hard* or *soft*. A hard constraint must be satisfied by all discovered services for a query and are used to filter services that do not comply with them. A soft constraint does not need to be satisfied by all discovered services, but are used to rank candidate services for a query. The contextual constraints are used in the case of adaptation of service-based systems.

The design process supported by the framework is iterative. The process uses structural and behavioural design models of service-based systems (called SySM and SyBM, respectively) to support discovery of services that can fulfill the models. The identified services are used to reformulate the design models and trigger new service discovery iterations. The behavioural models describe interactions between operations of a service-based system that can be provided by web services, legacy systems, or software components, while the structural models specify the types of the parameters of operations in the behavioural models. In the framework, the structural and behavioural design models are UML class and sequence diagrams, respectively.

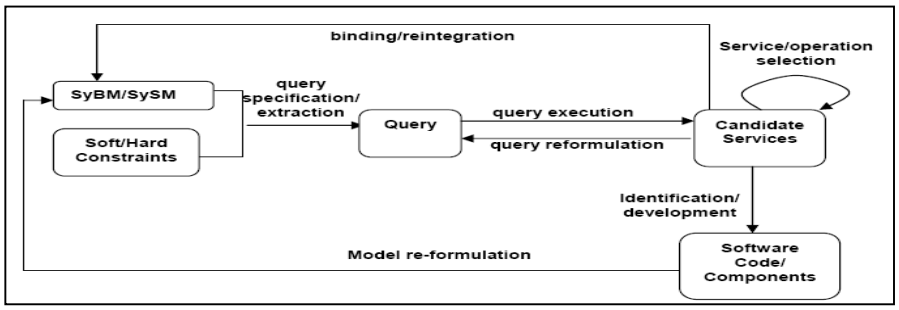

Fig. 2: Design Process

Fig. 2 presents an overview of the iterative design process of the framework. As shown in the figure, queries are specified in reference to the sequence diagrams in SyBM and the classes and interfaces in SySM, and may include additional constraints about the required services. These queries are generated by any client application that is able to produce service discovery queries expressed as UML 2.0 models represented in XMI (e.g., any CASE tool that supports UML 2.0 and representation of the models in XMI). Details about the queries are presented in Section 3. The queries are passed to the service requestor component to be executed (see Fig. 1). The candidate services identified after the execution of queries by the query processor can be bound to the SySM and SyBM models by the designers of the system. When this happens, SySM and SyBM are re-formulated (e.g. by adding message data types and operations of identified services) and their new versions can be used to specify further queries for discovering additional services for other parts of the system. Queries may also be reformulated and re-executed when the identified services are not adequate. The process can be terminated by the system designer at any time, when all the required services have been discovered, or when it is clear that further queries would not be able to identify services that have a better match with the current design models. Queries may also include hard and soft constraints expressed in an XML based language that we have developed (See Section 3).

The execution time adaptation process supported by the framework allows services to be identified based on both pull and push modes of query execution. The pull mode of query execution is performed to identify services (a) that are initially bound to a service-based system and their replacement candidate services, (b) as a first step in the push mode of query execution, (c) due to changes in the context of an application environment, or (d) when a client application requests a service to be discovered. The push mode of query execution is performed when the application is running and a service needs to be replaced due to any of cases (i)-(iii) described in Section 1. For the push mode of query execution, the framework assumes a pro-active approach in which services are identified in parallel to the execution of a service-based system based on subscriptions of application environment, services, and queries associated with these services, so that replacement services can be identified, when notification of changes in services and application environments are pushed to listeners. These notifications are supported by service and application context servers and service listeners (see Fig. 1).

For both design and adaptation of service-based systems, the service discovery technique is based on matching between a query and services executed in a two-phase process. The first phase consists of a *filtering* phase and the second phase consists of a *ranking* phase. In the filtering phase, hard constraints in a query are evaluated against service specifications and candidate services that comply with these constraints are identified. In the ranking phase, candidate services identified in the filtering phase are matched against structural, behavioural, and soft constraints in a query based on the computation of distances.

During design of service-based systems, the ranking phase returns *n*-best services for a query (*n* can be either specified in a query or is equal to ten by default). The designer selects from these services the ones to be used in the system. During adaptation of service-based systems, the ranking phase returns the best service for a query that is used to replace a service in the service-based system. The computation of the distances differs during the design and adaptation phases. Details of the computations and their differences are described in Section 4.

3. Query Language

In order to support service discovery queries in the framework, we have developed an XML-based language named SerDiQueL (Service Discovery Query Language [59]). It allows the specification of structural, behavioural, quality, and contextual characteristics of services to be identified or of systems being developed.

Fig. 3 presents the overall XML schema of SerDiQueL. As shown in the figure, a query specified in the language (ServiceQuery) has three elements representing structural, behavioural, and constraint sub-queries. The division of a query into these three sub-queries is to (i) allow the representation of these three types of information, and (ii) support the representation of queries with arbitrary combinations of these types of information. A *ServiceQuery* element also has a unique identifier, a name, and one or more elements describing different parameters for a query. A parameter element is defined by a name and a value. Examples of parameters that can be used in a query are: (a) name of the query, (b) type of the

query (e.g., static, in the case of design of service-based systems or dynamic, in the case of adaptation of service-based systems), (c) mode of execution (push or pull), (d) author of the query, and (e) number of services to be returned by a query.

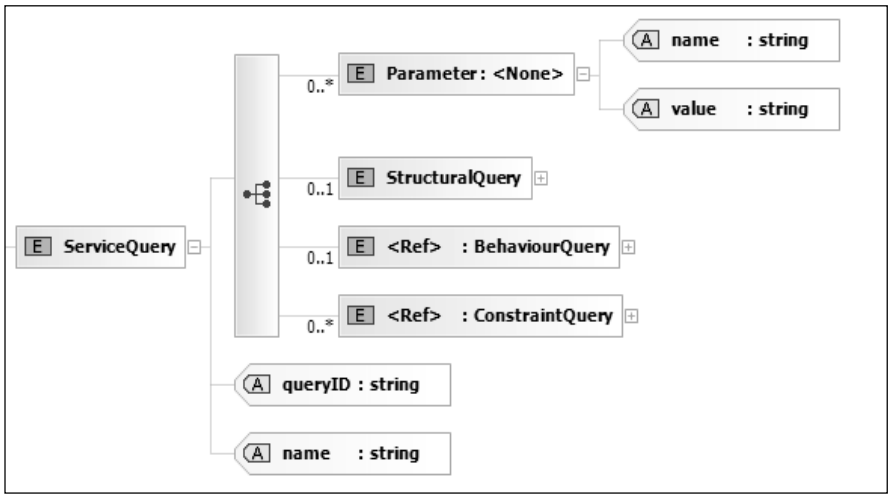

Fig. 3: Overview schema of SerDiQueL

3.1. Structural Sub-query

The structural sub-query describes structural aspects of (i) a service-based system being developed or (ii) a service participating in a running service-based system that needs to be replaced.

The structural sub-queries for case (i) use elements in SySM and SyBM design models of the system (see Section 2) together with a UML 2.0 profile [25] that we have developed, and it is represented as XMI documents. The profile defines a set of stereotypes for different types of UML elements such as messages in sequence diagrams, or operations and classes defining the types of arguments in the messages in the class diagrams. For example, messages in a sequence diagram may be stereotyped as: (1) *query messages* representing service operations needed in identified services; (2) *context messages* representing additional constraints for the query messages (e.g. if a context message has a parameter p1 with the same name as a parameter p2 of a query message, then the type of p1 should be taken as the type of p2); (3) *bound messages* representing concrete service operations that have been discovered in previous query executions.

In order to express a query for a service-based system being developed, system designers select an interaction from the SyBM model of the system and specify the messages in the interaction that need to be realized by service operations that are to be discovered (query messages). The designers can also specify the context messages to impose additional constraints in a query. All messages in an interaction that are not stereotyped are treated as messages irrelevant to the discovery

process. Based on the selected interactions and messages, structural sub-queries for a service-based system under development are automatically generated from the class (SySM) and sequence (SyBM) diagrams.

The description of structural aspects of a service-based system is based on design models of these systems and supports the representation of operations being searched in different services together with the representation of the input and output parameters of these operations and their respective data types. This is important to assist with the matching of structural aspects of the systems with structural aspects of available services. Moreover, it supports the development of a service-based system based on the characteristics of available services instead of on requirements that may never be able to be fulfilled by existing services.

As an example of the specification of a query for case (i), consider part of the behavioural and structural design models of a Cell Phone Operator (CPO) system from the case study used in the book shown in Figures 4 and 5, respectively. As shown in Fig. 4, the system allows the provision of SMS, voice, and email messages; request for movies; and payment of these various services by a user of the CPO. Consider a designer of this service-based system that wants to identify a service that can provide request and payment of movies. In this case, a designer wants to find service operations that can provide implementations of the messages

payMovie(phone:PhoneNumber, quantity:Integer):Boolean and

retrieveMovie(phone:PhoneNumber, info:MovieInfo):MovieStream

as specified in the diagram shown in Fig. 4. The designer creates a query as a copy of the sequence diagram and attaches the query message stereotype to these messages (<<asd_query_message>>). The classes representing the data types of the parameters of the two query messages, and all the classes that are directly or transitively related to them, are automatically identified and put together to formulate the structural part of the query. These classes are shown in Fig. **5**.

The structural sub-queries for case (ii) are represented by the WSDL specification of the service to be replaced. In this case, SerDiQueL supports a complete representation of the structural aspects of a service to be identified as interface descriptions. In the framework, structural sub-queries for a service that needs to be replaced during execution time are automatically generated based on the notification that a service became malfunctioning, unavailable, or there were changes in the characteristics of the service or in the context of the application environment.

As an example, consider the CPO service-based system described above. Assume service S_{Movie} that was found during design time of the system and bound to the system. Consider S_{Movie} with operations

payment(phone:Number, serviceType:String, amount:Integer):Boolean and

getMovie(phone:Number, title:String, director:String, *language:String):Movie*

Suppose a user of the CPO service-based system has moved from London to Shanghai for two months. Consider that the time to retrieve movies using service S_{Movie} from Shanghai is very slow and, therefore, due to change of context, another service that provides movies via cell phones with a better performance needs to be

identified and replaced in the service-based system. In this case, the structural sub-query is the WSDL specification of S_{Movie}.

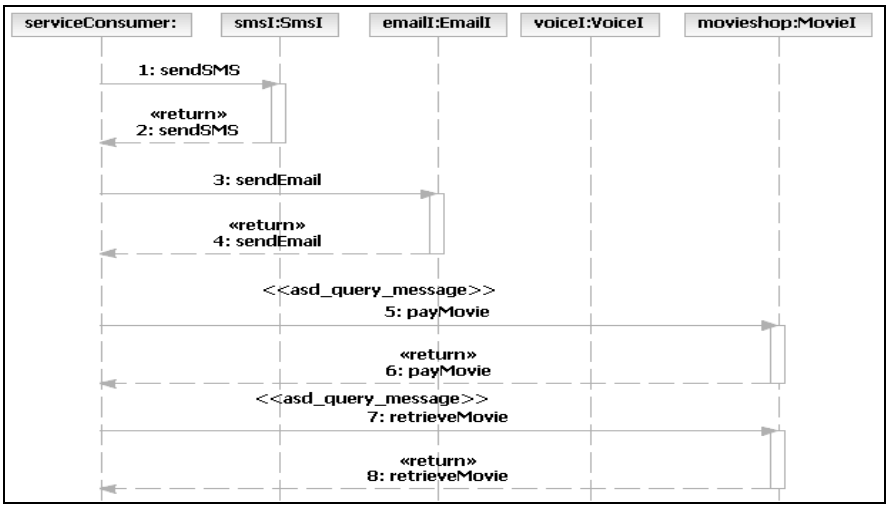

Fig. 4: Behavioural model of CPO service-based system

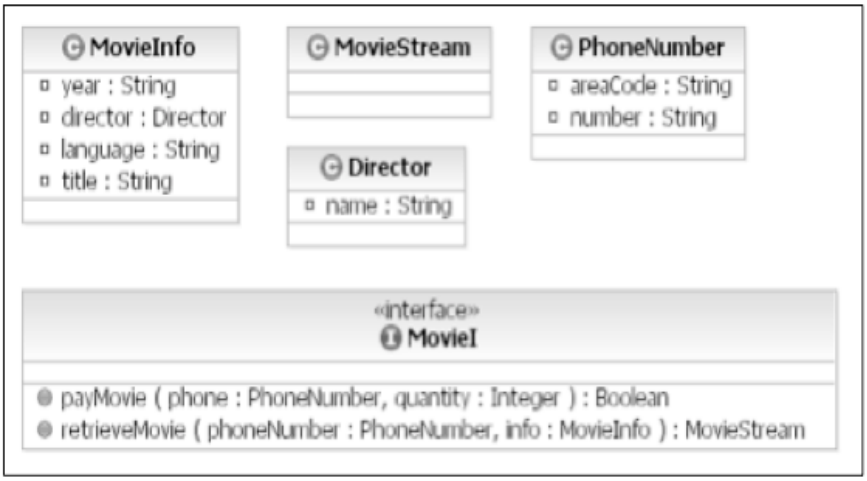

Fig. 5: Structural model of CPO service-based system

The reasons for using UML models enhanced with our UML profile specified in XMI to represent structural sub-queries during design of service-based systems are because (a) UML is the de facto standard for designing software systems and can effectively support the design of service-based systems [10,12], and (b) UML has the expressive power to represent the design models of service-based systems since it can represent modeling of software services, legacy code and software components in a

system. The use of WSDL to represent structural sub-queries in the case of execution time adaptation of service-based systems is due to its wide acceptance as a service interface description language. In addition, during execution time of service-based systems, any replacement service that might be identified for an existing service in a system needs to conform to the interface of the existing service.

3.2. Behavioural Sub-query

The behavioural sub-queries need to allow the specification of the (1) existence of a required functionality, or a sequence of required functionalities, in a service specification; (2) order in which the required functionalities should be executed by a service; (3) dependencies between functionalities (e.g. the functionality realized by an operation always requires the existence of the functionality of another operation); (4) pre-conditions; and (5) loops concerning execution of certain functionalities.

Fig. 6 shows a graphical representation of the SerDiQueL's XML schema for specifying behavioural sub-queries. As shown in the figure, a behavioural sub-query is defined as (a) a single condition, a negated condition, or a conjunction of conditions, or (b) a sequence of expressions separated by logical operators. A behavioural sub-query also specifies *requires* elements.

Requires elements are used to describe the service operations that need to exist in service specifications. Every query must describe one or more required service operations, represented by *MemberDescription* elements in the query (*MemberDescription* elements can be used in various conditions and expressions in 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 a query message (design phase) or of an operation (adaptation phase) described in the structural sub-query; and (c) *synchronous*, indicating if the service operation needs to be executed in a synchronous or asynchronous mode. The parameters and respective data types of the operations are specified in the structural sub-queries.

The existence of *requires* elements in service specifications is verified as an initial step during the execution of a behavioural sub-query rather than during the evaluation of the conditions and expressions of the query that uses these elements. This optimizes the query execution process as there is no need to evaluate any condition or expression that refers to a non-existing *requires* element.

A condition is defined as a *GuaranteedMember, OccursBefore*, *OccursAfter*, *Sequence*, or *Loop* element. A *GuaranteedMember* represents a member element (e.g., a service operation) that needs to occur in all possible traces of execution in a service. This element references requires, sequence, or loop elements. *OccursBefore* and *OccursAfter* elements represent the order of occurrence of two member elements (e.g., *Member1* and *Member2*). Note that in some cases we may require *OccursBefore(m1,m2)* whilst in other cases we may require *OccursAfter(m1,m2)*, or even need to differentiate an *OccurBefore* condition by attributes such as immediate. Hence both *OccursBefore* and *OccursAfter* elements are needed. Furthermore, they have two boolean attributes, namely: (a) attribute immediate, speci-

fying if two members need to occur in direct sequence or if there can be other member elements in between them, and (b) attribute 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 that is executed several times.

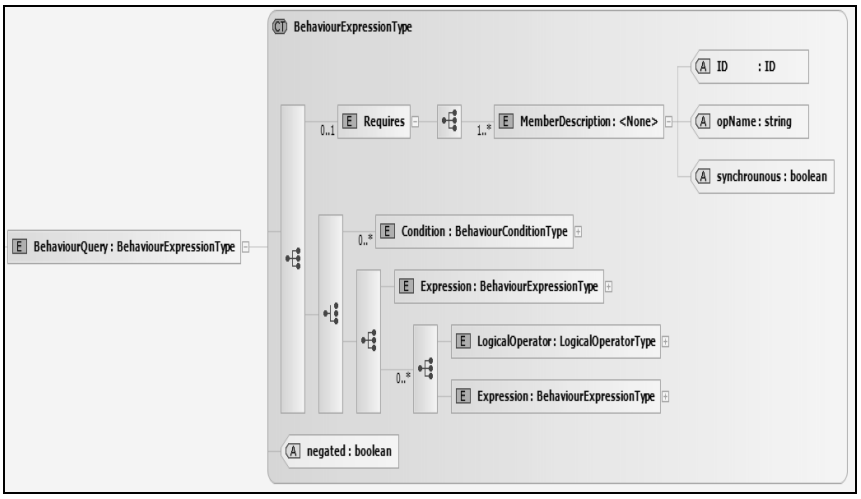

Fig. 6: XML Schema for behavioural sub-query

```
<tnsb:BehaviourQuery>
  <tnsb:Requires>
        <tnsb:MemberDescription ID="pay" opName="payMovie" syn-
chrounous="true”/>
        <tnsb:MemberDescription ID="retrieve" opName="retrieveMovie"
synchrounous="true" />
  </tnsb:Requires>
  <tnsb:Expression>
    <tnsb:Condition>
      <tnsb:OccursBefore immediate'"false"
                                    guaranteed="false">
        <tnsb:Member1 IDREF="pay" />
        <tnsb:Member2 IDREF="retrieve" />
      </tnsb:OccursBefore>
    </tnsb:Condition>
  </tnsb:Expression>
</tnsb:BehaviourQuery>
```

Fig. 7: Example of behavioural sub-query in SerDiQueL

In behavioural sub-queries, expressions are defined as sequences of *requires* elements, conjunctions or disjunctions of conditions, or nested expressions connected by logical operators AND and OR. The definition of requires elements within an expression (E1) enables the specification of queries in which the non-existence of requires elements in a service should not invalidate its selection, if

other expressions in the sub-query that are disjointed with expression E1 (i.e., expressions connected to E1 by logical operator OR) are satisfied by the service.

During the design of service-based systems, the interaction and sequence of messages selected by the system designers from the SyBM model (stereotyped query messages and context messages) are represented in SerDiQueL's behavioural sub-query. In order to illustrate, consider the messages *payMovie(phone:PhoneNumber,quantity:Integer):Boolean and retrieveMovie(phone:PhoneNumber, info:MovieInfo):MovieStream* from the example in Section 3.1. Fig. 7 shows the description of this behavioural sub-query in SerDiQueL. As shown in the figure, the *Requires* elements specify the requirement for the existence of operations *payMovie* and *retrieveMovie,* and the *OccursBefore* element defines the order of these two operations. Similar and more complex behavioural sub-queries can be specified for the execution time adaptation of service-based systems. These cases are not shown here due to space limitations.

3.3. Constraint Sub-query

A constraint sub-query describes different types of additional conditions, which must be fulfilled by a service-based system or by its participating services. These additional conditions may include (a) quality aspects, (b) contextual aspects, or (c) extra structural and behavioural aspects that cannot be represented in the structural and behavioural sub-queries. Examples of these additional conditions are the specification of the time or cost to execute a certain operation in a service, the receiver of a message, or the provider of a service.

As described in Section 2, a constraint can be classified as *contextual* or *non-contextual*. The non-contextual constraints in a sub-query can be evaluated against any type of service specification (facet) in the service registries. The contextual constraints are evaluated against *context facets.* These context facets are associated with services and describe context information of the operations in these services. Context information is specified as context operations that are executed at run-time. The framework assumes the existence of context services that provide context information (see details in Section 4).

A constraint sub-query is defined as a single logical expression, a negated logical expression, or a conjunction or disjunction of two or more logical expressions, combined by logical operators. A constraint sub-query has four attributes, namely (a) *name*, specifying a description of the constraint; (b) *type*, indicating whether the constraint is hard or soft; (c) *weight*, specifying a weight in the range of [0.0, 1.0]; and (d) *contextual*, a boolean attribute indicating whether the constraint is contextual or non-contextual. The weight is used to represent prioritisations of the parameters in a query for soft constraints. If the value of the attribute *contextual* is *true*, the query may contain *ContextOperand* elements.

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, which can be non-contextual, contextual, constants, or arithmetic expressions.

A non-context operand (see element *NonContextOperand*) has two attributes: (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. The constraints can be specified against any element or attribute of a facet in the registries.

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 system for which the value of a context aspect of the system needs to be dynamically identified during the evaluation of a query (e.g., location of a mobile device application). In this case, attribute *serviceID* refers to the service-based system itself. Otherwise, the value of *serviceID* is specified as "any".

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 document category (element *Document*) has an attribute type indicating if the document is an ontology or a context facet, and contains an XPath expression referencing elements in the document. In the case of an ontology document, an attribute with the URL indicating the location of the ontology that describes the context operation is used. The language can support different ontologies for describing context operation categories since it does not make any assumption of the structure and meaning of the ontologies used, apart from the fact that the ontologies need to be described in XML. A context category in a query 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 in a query.

Arithmetic expressions define computations over the values of elements or attributes in service specifications 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*, *subtraction*, *multiplication*, and *division* operators. The operands can be contextual, non-contextual, constants, or functions. A function supports the execution of a complex computation over a series of arguments. The results of these computations are numerical values that can be used as an operand in an arithmetic expression.

```
<tnsa:ConstraintQuery name="C1" contextual="true"
                      type="SOFT" weight="0.5">
 <tnsa:LogicalExpression>
  <tnsa:Condition relation="LESS-THAN-EQUAL-TO">
   <tnsa:Operand1>
    <tnsa:ContextOperand   serviceOperationName="getMovie"
                           serviceID="any">
     <tnsa:ContextCategory relation="EQUAL-TO">
       <tnsa:Category1>
        <tnsa:Document location="http://eg.org/CoDAMoS_Extended.xml"
             type="ONTOLOGY">string(/owl:Class/@rdf:ID)
        </tnsa:Document></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-60</tnsa:Constant>
     </tnsa:Operand2>
     </tnsa:Condition></tnsa:LogicalExpression>
  <LogicalOperator>AND </LogicalOperator>
  <tnsa:LogicalExpression>
   <tnsa:Condition relation=" EQUAL-TO">
    <tnsa:Operand1>
     <tnsa:ContextOperand serviceOperationName="getMovie"
                          serviceID="any">
      <tnsa:ContextCategory relation="EQUAL-TO">
        <tnsa:Category1>
         <tnsa:Document location="http://eg.org/CoDAMoS_Extended.xml"
            type="ONTOLOGY">string(/owl:Class/@rdf:ID)
         </tnsa:Document></tnsa:Category1>
        <tnsa:Category2>
          <tnsa:Constant type="STRING"> GREDIA_LOCATION
          </tnsa:Constant> </tnsa:Category2>
      </tnsa:ContextCategory></tnsa:ContextOperand></tnsa:Operand1>
     <tnsa:Operand2>
       <tnsa:Constant type="STRING">Shanghai
       </tnsa:Constant></tnsa:Operand2>
  </tnsa:Condition></tnsa:LogicalExpression>
</tnsa:ConstraintQuery>
```

Fig. 8: Example of constraint sub-query

In order to illustrate, consider the replacement of service S_{Movie} in the CPO service-based system. Assume a contextual constraint specifying that the time to receive a requested movie while in Shanghai should not be more than 60 seconds. Fig. 8 shows this constraint in SerDiQueL. The constraint specifies that any candidate service that can retrieve movies while the location is Shanghai (i.e., services that match operation *getMovie()* in S_{Movie}) needs to have (a) a context operation classified in the category GREDIA_RELATIVE_TIME in ontology http://eg.org/CoDAMos_Extended.xml with the result of executing this operation being less than or equal to SECONDS-60, and (b) a context operation classified in the category GREDIA_LOCATION in the ontology with the result of executing this operation being equal to Shanghai, for this service to be accepted. Other con-

textual and non-contextual constraints can be described in SerDiQueL during design or adaptation of service-based systems.

4. Query Execution and Matching Process

In both design and adaptation of service-based systems, matchings between queries and services are executed by the query processor (see Fig. 1) in a two-phase process. In the first phase, the query processor searches service registries in order to identify services that satisfy the hard constraints of a query based on exact matchings (*filtering* phase). In the second phase, candidate services identified in the filtering phase are matched against the structural, behavioural, and constraints sub-queries, and the best candidate services for the query are identified (*ranking* phase).

The ranking phase is executed based on the computation of partial distances, namely *structural*, *behavioural*, soft *non-contextual*, and *contextual* distances when applicable (i.e., execution time adaptation). The partial distances computed between services and a query are aggregated into an overall distance which is then used to select the best services for a query.

The *structural* matching between a query and a service is performed by comparing (i) the signatures of query messages in the structural model of a service-based system against the signatures of the operations of WSDL specifications of candidate services, during the design of service-based systems; or (ii) the signature of the operations in the WSDL specification of a service that needs to be replaced in a service-based system against the signature of the operations of WSDL specifications of candidate services, during adaptation of service-based systems. In both cases, the structural matching is based on the comparison of graphs representing the data types of the parameters of the operations and the linguistic distances of the names of operations and parameters.

The *behavioural* matching between a query and a service is performed by comparing the behavioural specification of the services and the behavioural sub-query. In this case, the behavioural specifications of the service and the behavioural sub-query are converted into state machine models and distances between these state machines are calculated based on similarities of these state machines.

The soft constraint matching (*contextual* and *non-contextual*) between a query and a service is performed by analysing the conditions in the constraint part of a query against service specifications.

There may be some differences in the execution process of a query. These differences are due to the lack of hard, behavioural, and soft contextual and non-contextual constraints in a query, or any combinations of these constraints[2]. In cases where there are no hard constraints in a query, the filtering phase is not executed and partial distances are calculated for all the services in registries. Also if

[2] Note that during the design of service-based systems, it is not possible to consider contextual constraints in a query.

there are no behavioral or soft constraints in a query, the computation of the relevant partial distances is bypassed and the overall distance is computed by using only the partial distances of the types of constraints specified in a query. Note that structural constraints are always present in a query and, therefore, distances based on these constraints are always calculated. This is because during design of service-based systems the signatures of the types of operations to be found in services need to be specified, while during adaptation of service-based systems it is expected to have at least a WSDL description of a service to be replaced.

Other differences in the execution process of a query exist in the case when a query is to be performed to support design or adaptation of a service-based system. This difference is mainly concerned with the structural and behavioural matching processes. More specifically, during the design of a service-based system, the structural and behavioural matching processes are flexible allowing the identification of services whose structure and behaviour characteristics have different degrees of similarity to those of a required service, and behaviour matchings with alternative or missing mappings between a required service and an existing service. The flexibility and alternative/missing mappings contribute to the reformulation of the design models of the service-based system under development and to the design of service-based systems based on characteristics of existing services. However, during adaptation of a service-based system, the structural and behavioural matching process requires matches with services that can be used to substitute services in a system without disturbing the rest of the system. Therefore, in this case, for structural matching, it is necessary to guarantee that the input information for invoking the service that needs to be replaced in the system (S) cover the input information needed by the candidate service (S') and that the information produced by S' covers the information expected from S. For behavioural matching, the order of the different functionalities to be executed by a service needs to be preserved. Furthermore, during execution time adaptation of service-based systems, the execution process of a query also differs for push or pulls modes. Details of the ways of executing a query are presented below.

4.1 Query Execution for Design of Service-based Systems

The execution of specified queries to identify services during the design of a service-based system includes the filtering and ranking phases discussed above. During filtering phase, query execution is based on checking the satisfiability of the hard constraints specified as part of the query by the different services that exist in various service registries.

During ranking phase, the execution of queries is based on finding the best possible 1-1 mapping between the service operations required by a query and the operations provided by different services returned by the filtering phase or in the service registries. The search for the best possible match is treated as an instance of the assignment problem, i.e., for each of the alternative total 1-1 mappings (M) of required operations (RO) on to service operations (SO), a total aggregate distance is calculated from distances between the individual (RO, SO) pairs that constitute M (in such pairs RO is a required query operation and SO is an operation

offered by some service). The mapping that has the minimum aggregate distance is selected as the final outcome of the process. In this process, the distance between a pair of required and service operations (RO, SO) is computed as weighted sum of three partial distances between RO and SO, namely the structural, behavioural and soft non-contextual constraint distances. Fig. 9 presents the structural (d_{STR}), behavioural (d_{BEH}), and soft constraint (d_{SOFTC}) distances.

The structural distance between a required and a service operation is computed by considering a linguistic distance between the names of these operations, and distances between their input and output parameters. The linguistic distance between two operation names (see function d_{LING} in Fig. 9) is computed as the ratio of tokens in the names of the two operations, for which there is no token in the other operation having a semantic relation with it in WordNet, or being identical to it (the tokenization that precedes the computation of this distance assumes that capital letters within operation names indicate the start of new tokens).

The distance between the input (output) parameters of two operations is based on finding the best possible morphism between the structures of the data types of these parameters. The computation of this morphism is based on graphs representing the input (output) parameters of the relevant operations. These graphs are formulated by a special starting node with outgoing edges which are labeled by the names of the input (output) parameters of the relevant operation and pointing to nodes representing the types of these parameters. Furthermore, for each of the input (output) parameter types T, the graph includes an edge starting from the node representing T and ending at a node representing the type of the attribute. These edges are labeled with the name of the relevant attributes whilst the nodes of the graph are labeled with the names of the relevant data types. If the type of an attribute is not a primitive one, the same construction process is followed until attributes with primitive data types are reached.

Following the construction of the graph, the structural distance between two parameter sets is computed according to the distance d_{STR} defined in Fig. 9. This distance is computed by finding the morphism between the edges of the graphs representing the input(output) parameters of the two operations which have the minimum aggregate distance. The latter distance is computed as the linguistic distance between the names of the two edges under comparison and the names of their source and destination nodes.

As an example of computing the signature distance between two operations, consider the operation *payMovie(phone:PhoneNumber, quantity:Integer):Boolean* in Fig. 4 and the operation *payment(phoneNumber: String, serviceType: String, amount:Integer): Boolean*. The linguistic distance between the names of these two operations will be d_{LING}*("payMovie", "payment")) = 1/3 = 0.33*[3] since the names of the two operations are tokenized into the sets {"pay", "movie"} and {"payment"} and the token "movie" in the first of these sets has no semantic relation with any of the tokens in the second set whilst the tokens "pay" and "payment"

[3] In the computation of signature distances we assume that $w_1=w_2=w_3=1$.

OPERATION DISTANCE:

$d(RO,SO) = w_S * d_{STR}(RO, SO) + w_B * d_{BEH}(RO, SO)) + w_{SC} * d_{SOFTC}(RO, SO)$

STRUCTURAL DISTANCE:

$d_{STR}(RO,SO) = w_N * d_{LING}(name(RO), name(SO)) + w_I * d_{PAR}(In(RO),In(SO)) + w_O * d_{PAR}(Out(RO), Out(SO))$

where

- In(O) (Out(O)) is the set of the input (output) parameters of O

LINGUISTIC DISTANCE:

$d_{LING}(S1, S2) = N1 + N2 / N$

where

- N1 (N2) is the number of tokens in S1 (S2) which have no common synonym with a token in S2 (S1)
- N total number of tokens of S1 and S2

PARAMETER DISTANCE:

$d_{PS}(P1,P2) = MIN_{M \in Morphisms(Edges(P1), Edges(P2))} \{(\sum_{(e1,e2) \in m} d_E(e_1,e_2) + \#Edges(P1)\text{-non-in-M} + \#Edges(P2)\text{-non-in-M}) / max(\#Edges(P2), \#Edges(P1))\}$

where

- Edges(P1) (Edges(P2)) is the graph formulated to represent the data types of the parameters in P1 (P2)
- #Edges(Pi) is the number of edges in the graph to represent the data types of the parameters in Pi
- Morhisms(Edges(P1), Edges(P2)) is the set of all the possible morphisms between the edges of the graph representing P1 and the edges of the graph representing P2 that covers the graph with the fewer edges
- $d_E(e_1,e_2) = w_1.d_{LING}(name(e1), name(e2) + w_2.d_{LING}(name(sourceNode(e1)), name(sourceNode(e2))) + w_3.d_{LING}(name(destNode(e1),destNode(e2)) /(w_1 + w_2 + w_3)$

BEHAVIOURAL DISTANCE:

$d_{BEH}(RO, SO, k) = 1$ *if* transitions$(SM_{SO})=\varnothing$ *or* transitions$(SM_{RO})=\varnothing$

$d_{BEH}(RO, SO, k) = MIN_{M \in Morphs^{(sK)}(SM_{RO}, SM_{SO})} (\sum_{(t,t') \in M} d_{SIG}(operation(t),operation(t')) + \#transitions(SM_{RO})\text{-non-in-M} + \#transitions(SM_{QO})\text{-non-in-M}) /(MAX(length(SM_{RO}),length(SM_{QO}))$ if transitions$(SM_{SO})\neq\varnothing$ and transitions$(SM_{RO}) \neq\varnothing$

where

- SM_{QO} is the state machine formulated by the behavioural conditions of the behavioural conditions of the query containing RO
- SM_{RO} is the state machine of the service offering the operation RO
- $Morphs^{(sK)}(SM_{RO}, SM_{QO})$) is the set of all the possible 1-1 mappings between the transitions of two paths p and q in SM_{RO} and SM_{QO} that preserve the ordering of the transitions within these paths (i.e. , for all transitions t_i and t_j in p such that $t_i \prec_p t_j$ it also holds that $m(t_i) \prec_q m(t_j)$) and leave up to K transitions in p or q without counterparts
- $Length(SM_{RQ})(length(SM_{QO}))$ is the length of the longest path of transitions in $SM_{RQ}(SM_{QO})$

SOFT CONSTRAINT DISTANCE:

$d_{SOFTC}(RO, SO) = \sum_{C \in SOFT\text{-}CONSTRAINTS(RO)} w_C \times not\text{-}satisfied(C) / \sum_{C \in SOFT\text{-}CONSTRAINTS(RO)} w_C$

where

- SOFT-CONSTRAINTS(RO) is the set of soft constraints in the query which apply to RO
- w_C is a weight expressing the significance of the constraint C for RO ($w_i >0$),
- not-satisfied(C) = 1 if the constraint C_i is not satisfied by S_o, or 0 otherwise.

Fig. 9: Distance functions used in execution of design service discovery queries

have a semantic relation with each other (as "payment is the noun of the verb "pay"). Furthermore, the distance between the input parameters and the output parameters of these operations is $d_{PS}(In(payMovie),In(payment))=(2+0.11)/4=0.527$ and $d_{PS}(Out(payMovie), Out(payment))=0/4=0$. These distances are computed on

the basis of the graphs representing the structures of the relevant parameters, which are shown in Fig. 10.

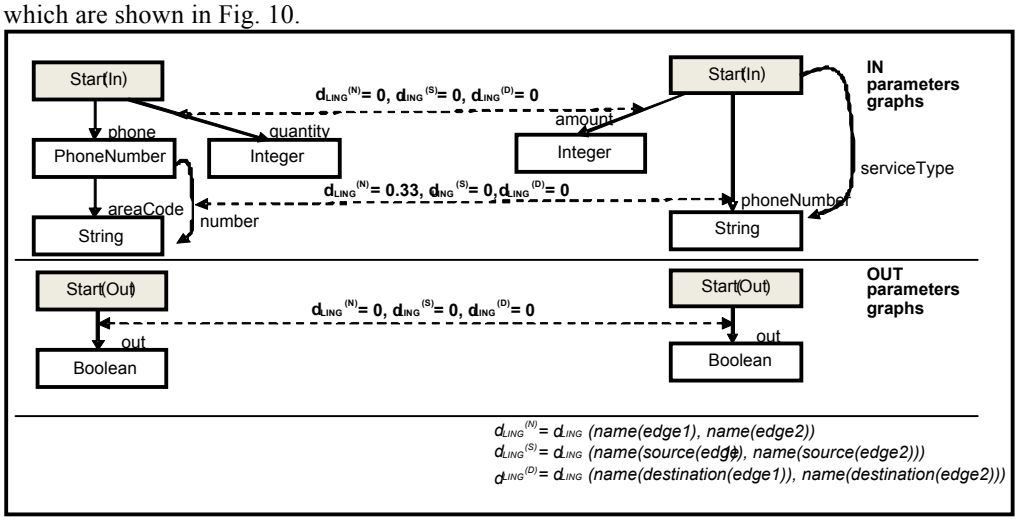

Fig. 10: In/Out Parameter graphs for PayMovie and Payment operations

More specifically, the distance between the input parameters of the two operations is 0.527 as there are two edges of the input graph of *payMovie* that have not been mapped onto any edges of *payment* (i.e., *phone* and *areaCode*), one edge of the input graph of *payment* that has not been mapped onto any edges of *payMovie* (i.e., *serviceType*) and two pairs of edges which have been mapped (i.e., *quantity* onto *amount* and *number* onto *phoneNumber*). The reason for mapping the edges *quantity* and *number* of the input parameter graph of *payMovie* onto the edges *amount* and *phoneNumber* of the input parameter graph of payment is because any alternative mapping would result in higher aggregate distance between the two graphs. The distance between the edge *amount* of *payment* and the edge *phone* of *payMovie*, for example, would be 0.75 as the names of the two edges as well as the names of their destination nodes do not have any semantic relation and therefore the linguistic distances between them are equal to one in both cases. Similarly the distance between the output parameters of the operations *payMovie* and *payment* is equal to zero as the two operations have the same output type.

The behavioural distance between a required query operation and a service operation (see function $d_{BEH}(RO,SO, k)$ in Fig. 9) is computed by matching a state machine representing the behavior of the interface that defines the operation in a query and the state machine of the service that provides a candidate operation for the query. The state machine of the interface that defines the required operation in the query is generated automatically from the query itself. The details of this generation are beyond the scope of this chapter and are given in [40]. It should be noted, however, that this state machine always has a single path. Given these two state machines, the behavioural distance between two operations is computed by finding the best possible match between the single path *p* of the state machine of

the query operation and the different paths of the state machine of the service (SM_{SO}). In the search for this match the individual transitions of *p* are mapped onto transitions of all different paths *q* of SM_{SO}. This mapping is constrained to preserve the order of the transitions in *p* and SM_{SO}, i.e., for all transitions t_i and t_j in *p* such that $t_i \prec_p t_j$ if t_i and t_j are mapped onto $m(t_i)$ and $m(t_j)$ of a path *q* in SM_{SO} respectively it should also hold that $m(t_i) \prec_q m(t_j)$[4]. Furthermore the possible mappings *m* between p and different paths *q* in SM_{SO} are constrained to leave up to *k* transitions of *p* and *q* without a counterpart. Hence, the parameter *k* takes an integer value that controls the flexibility of the behavioural matching process. Given all the possible mappings *m* between *p* and SM_{SO} that satisfy these constraints, $d_{BEH}(RO, SO,k)$ is computed by finding the mapping *m'* that has the minimum aggregate transition distance. This distance is computed as the ratio of the sum of the signature distances between the operations that label the pairs of mapped transitions and the number of the non-mapped transitions of *p* and *q*, over the length of the longest of the two paths *p* and *q*.

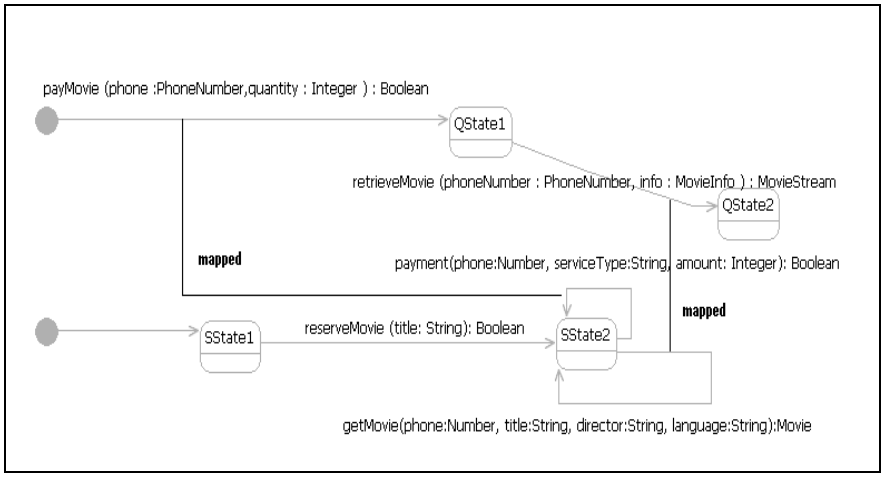

Fig. 11: State machines for *payMovie* and *payment* operations

As an example of the computation of $d_{BEH}(RO, SO,k)$ consider the state machines for the query and service in Fig. 11. Given these state machines the path morphism with the the lower aggregate distance between the two machines is the one that maps the path *(payMovie→ retrieveMovie)* of the state machine of the query onto the path *(payment →getMovie)* of the state machine of the service. The behavioural distance resulting from this mapping would be:

$$d_{BEH}(RO,SO, 0) = (d_{STR}(payMovie,payment) + d_{STR}(retrieveMovie,getMovie)) / 2$$

[4] $\prec_P$ is a relation indicating the order of transitions in path *P* and $\prec_Q$ is a relation indicating the order of transitions in an SM_{SO} path *Q*.

$$= ((d_{LING}(name(payMovie),name(payment)) + d_{STR}(In(payMovie),In(payment)) + d_{STR}(Out(payMovie),Out(payment)))/3) + (d_{LING}(name(retrieveMovie),name(getMovie) + d_{STR}(In(retrieveMovie),In(getMovie)) + d_{SIG}(Out(retrieveMovie),Out(getMovie)))/3))/2$$
$$= ((0.33 + 0.66 + 0)/3 + (0 + 0.744 + 0.33)/3) /2 = 0.344$$

It should be noted that other mappings between paths of these state machines are either not valid as they do not preserve the order of transitions (e.g., the mapping of *(payMovie→ retrieveMovie)* onto *(getMovie →payment)*) or would not result in a minimum distance (e.g., the mapping *(payMovie→ retrieveMovie)* onto *(reserveMovie →payment)*).

4.2 Query Execution for Adaptation of Service-based Systems

During adaptation of service-based systems, the framework can execute queries in both pull and push modes. In the pull mode of query execution, the query requestor invokes the query processor to execute a query. The query processor executes the query and maintains services whose distance from the query does not exceed a specific threshold. The set of maintained services is sorted in ascending distance order and returned to the client application for further action.

In the push mode of query execution, the client application subscribes to the framework the services it deploys and a query for each of these services. Based on a subscribed query Q for a service S in the service-based system, the framework retrieves a set of possible candidate services that could replace S (if necessary). These candidate services are initially identified by executing the query as in the pull mode. To allow a pro-active service discovery process, the framework maintains an up-to-date version of the set of candidate services for a service S as changes in the descriptions and context of the services and/or the environment of the application are notified to the framework through the context servers and service listeners (see Fig. 1). The up-to-date set of candidate services is maintained in parallel to the execution of a service-based system and includes only services whose overall distance from the query subscribed for S does not exceed a given threshold. The services are sorted in ascending distance order in the set.

In the framework, the replacement of *S* in a service-based system may not take place right after modifications occur in the set of candidate services for S. This is because an immediate replacement might be inappropriate as, for example, in cases where service *S* is executing some transactions on behalf of the application, at the time when a new better service is found. The decision to stop the execution of the application in order to replace a service for which a better alternative service has been found is based on *replacement policies*. Details about the replacement policies used in the framework can be found in [26].

The distance threshold for a candidate service is specified in a query by the value of the query element Parameter, as discussed in Section 3. When a threshold is not specified, the framework uses a default value of 0.5.

The push mode of query execution covers four different cases. These are the cases where: (a) a service S in the system becomes malfunctioning or unavailable (Case A); (b) there are changes in the structure, functionality, quality or context of any service in the set of candidate services for S or in S (Case B); (c) there are changes in the context of the service-based system environment (Case C); or (d) new services become available or existing services have their characteristics modified (Case D). In the following, we discuss the push execution mode for each case.

Case A: In this case, service S is replaced by the first service in the set of candidate services. By virtue of the process of maintaining this set, the first service in the set is guaranteed to have the smallest distance to query Q associated with S. Following the replacement, S is removed from Set_S.

Case B: In this case, service S can be either a service in the service-based system or a service in the set of candidate services for a service in the system. Service S is evaluated against query Q to verify if it still matches the query. The new overall distance between Q and S is calculated. If S was a candidate replacement service and the distance between S and Q is below the threshold distance, S remains in the set of candidate services. The position of S in the set of candidate services may change, however. If S becomes the best replacement service in the set, S will replace a service in the system when the replacement policy permits. Otherwise, if the distance between S and Q is above the threshold, S is removed from the set of candidate services. If S is a service currently deployed by the service-based system, but is no longer the best option based on its new distance with Q, S is replaced by the first service in the set of candidate services, when the replacement policy permits the change.

Case C: In this case, a value in a context constraint in query Q is modified and a new query Q' needs to be created to reflect the new context value. The service S associated with Q that is currently bound to the system needs to be evaluated against the new context constraint in Q'. If S does not match the new query Q', the services in the set of candidate services are evaluated against Q' and a new set of candidate services may be generated. This is necessary for identifying a service S' that best matches Q' and bind it to the system, as soon as possible, so that the system can continue its execution, while the framework tries to find new services that match Q' in the service registries. Following the use of S', the framework will do an exhaustive search in registries (pull mode) to update the set of candidate services based on Q'. The same search will be executed when there are no services in the current set of candidate services that match Q'. After updating the set of candidate services, if there is a service that is better than S', this service will replace S' subject to the replacement policy.

Case D: In this case new services may appear in registries for the first time, or the descriptions of existing services in the registries that did not initially match a query Q change. After being notified of a new service S, or updated service descriptions for S, the framework evaluates S against each subscribed query Q for each service deployed in the service-based application. Depending on the distance value between S and the queries, S may be included in a set of candidate services for a service in the system or replace a service in the application, depending on the replacement policy for this service.

In any of the above cases for push mode, or in the case of pull mode of query execution, the ranking phase of the matching process between a service and a query is executed in three substages. In the first of these substages, the structural and behavioural parts of a query are evaluated against candidate services and a structural-behavioural partial distance between each of the services and the query is computed. In the second substage, the soft non-contextual constraints of a query are evaluated against the set of candidate services and a soft non-contextual partial distance is computed for each candidate service. Finally, in the third substage, the contextual constraints of a query are evaluated against the candidate services and a contextual partial distance is computed for each candidate service. At the end, the partial distances computed for each service are aggregated into an overall distance between each service and the query and the service for which this distance is below a threshold are maintained. Fig. 12 presents the overall distance and partial structural-behavioural distance. The soft contextual and non-contextual partial constraint distances are the same as the soft constraint distance shown in Fig. 9.

The structural evaluation of a query against services is executed by comparing operations in the structural sub-query with operations in structural specifications of services expressed in WSDL based on the comparison of graphs of the data types of the parameters of the operations and linguistic distances of the names of the operations and parameters. The graphs of the data types of the parameters are constructed as presented in Section 4.1. The matching process uses a variant of the *VF2* algorithm for detecting graph morphisms that we have previously developed for linear composition of static service discovery [55].

More specifically, a query operation Qop having an input parameter data type graph ITG_{Qop} and an output parameter data type graph OTG_{Qop} matches a service operation Sop having an input parameter data type graph ITG_{Sop} and an output parameter data type graph OTG_{Sop}, if ITG_{Sop} is a sub-graph of ITG_{Qop} and OTG_{Qop} is a sub-graph of OTG_{Sop}. In other words, a candidate service operation Sop matches a query operation Qop, if the data types of its input parameters are super-types of the input parameters of the query operation, and the data types of its output parameters are subtypes of the output parameters of the query operation. This is necessary for adaptation to guarantee that the input information assumed for invoking Qop will cover the input information needed by Sop, and the output information produced by Sop will cover the output information expected from Qop. After computing the structural distance for each pair of query and service operations in a query Q and a service S, the framework identifies all the possible mappings between the operations in Q and operations in S in which each operation in Q is mapped onto a single operation in S.

For each of these mappings, the framework computes the behavioural distance between the mapped service and query operations based on the comparisons of paths representing the behavioural sub-query and behavioural service specification. More specifically, the behaviour matching is executed by (i) transforming behavioural service specifications into state machines, (ii) extracting all the possible paths from the generated state machine, (iii) transforming the behavioural sub-query into paths, and (iv) verifying if the path representing the behavioural sub-query can be matched against a path of the state machine of a service.

When a path representing the behavioural sub-query can be matched with a path in the state machine of a service, the behavioural distance for each pair of mappings of query and service operations in these paths is set to zero. Otherwise, the behavioural distance for each pair of mappings of query and service operations in these paths is set to one. After computing the structural and behavioural distances for all pairs of query and service operations in all possible operation mappings, the framework selects the mapping that has the minimal value for all the pairs divided by the number of operations in the query.

OVERALL DISTANCE:

$OD(Q, S) = (d_{STR\_BEH}(Q, S) + d_{NCON}(Q, S) + d_{CON}(Q, S)) / N$

where N is the number of computed partial distances

STRUCTURAL_BEHAVIOUR DISTANCE:

$d_{STR\_BEH}(Q,S) = Min\ (SUM\ (d_{SB}(Q_{opi}, S_{opj}) / n)$

where

- $d_{SB}(Q_{opi}, S_{opj}) = (d_S(Q_{opi}, S_{opj}) + d_B(Q_{opi}, S_{opj})) / 2;\ 1 <= I <= n;\ 1 <= j <= m;$
- n is the number of operations in Q;
- m is the number of operations in S;
- $d_S(Q_{opi}, S_{opj})$ is the structural distance between an operation in Q and an operation in S
- $d_B(Q_{opi}, S_{opj})$ is the behavioural distance between an operation in Q and an operation in S

$d_S(Q_{opi}, S_{opj}) = (d_{LING}(Q_{opi}, S_{opj}) + d_{IN}(Q_{opi}, S_{opj}) + d_{OUT}(Q_{opi}, S_{opj})) / 3$

where

- $d_{LING}(Q_{opi}, S_{opj})$ is calculated as in Fig. 9
- $d_{IN}(Q_{opi}, S_{opj})$ = #UnMap_Edges(IN_Graph) / #Edges(IN_Graph)
- $d_{OUT}(Q_{opi}, S_{opj})$ = #UnMap_Edges(OUT_Graph) / #Edges(OUT_Graph)

$d_S(Q_{opi}, S_{opj}) = 0$ if path in the state machine of a behavioural sub-query can be mapped to a path in the stet machine of behavioural specification;

$d_S(Q_{opi}, S_{opj}) = 1$ if path in the state machine of a behavioural sub-query cannot be mapped to a path in the state machine of the behavioural specification;

Fig. 12: Distance functions used in execution of service discovery queries

As in the case of design of service-based systems, the evaluation of soft (non-contextual) constraints is executed by assessing constraint expressions in the constraint sub-queries against service specification facets. This evaluation takes place by retrieving the values of the XPath expressions from service specification facets and assessing arithmetic, relational and logical expressions that define the constraint using these values. The result of this evaluation is a binary value indicating whether a specific constraint is satisfied (0) or not (1). Based on the evaluation of individual constraints, a weighted soft (non-contextual) constraint partial distance is calculated (see soft constraint distance shown in Fig. 9).

The evaluation of contextual constraints is based on the work described in [39]. Context constraints are evaluated against context facets of candidate services. This evaluation is concerned with the runtime execution of context operations defined in the constraint sub-query and the comparison of the results of the execution of these operations with the value specified in the constraint. In the framework, context information is provided by context operations, which are as-

sociated with service operations and executed at runtime. Our work assumes the existence of context services that provide context information. These context services provide context operations that are dynamically executed in order to generate context values associated with the context operations. A service operation may have one or more context operations. A context operation may be related to one or more service operations, or a whole service-based application.

The context operations associated with a service are specified in context facets represented in XML format. A context facet specifies the service to which the facet is associated and the context operations available for this service based on semantic categories defined in terms of ontologies. In the current version of the framework we use an extended version of the CODAMOS ontology [8], as shown in the example of Fig. 8. However, the approach does not impose any restriction on the form of ontology used to describe semantic categories, as long as the ontology is specified in XML.

		Op1			Op2			Op3			D_{SB}
		d_S	d_B	d_{SB}	d_S	d_B	d_{SB}	d_S	d_B	d_{SB}	
C1	payment	0	1	.5							.7
	getMovie	.8	1	.9							
C2	payment	0	0	0							.0
	getMovie				0	0	0				
C3	payment	0	0	0							.13
	getMovie							.53	0	.26	
C4	payment				.8	1	.9				.9
	getMovie	.8	1	.9							
C5	payment				.8	1	.9				.7
	getMovie				0	1	.5				
C6	payment				.8	1	.9				.58
	getMovie							.53	1	.26	
C7	payment							.8	1	.9	.9
	getMovie	.8	1	.9							
C8	payment							.8	0	.4	.7
	getMovie				0	0	0				
C9	payment							.8	1	.9	.83
	getMovie							.53	1	.76	

Fig. 13: Structural and behavioural distances for all mapping combinations

The evaluation of a contextual constraint results in a binary value indicating whether the constraint is satisfied (0) or not (1) and the computation of a weighted contextual constraint partial distance between a query and a service.

To illustrate the query execution process and the computation of distance for execution time adaptation, consider the example in which a user of CPO service-based system has moved temporarily to Shanghai. In this case, due to change in the location (context) of the application's environment (Case C above), service S_{Movie} does not match the new context constraint and a service that matches the constraints need to be identified. Suppose that Q1 is a query describing the service that needs to be identified to replace S_{Movie} with the following characteristics:

(a) the structural sub-query is the WSDL description of S_{Movie} with operations *payment(phone:Number, serviceType:String, amount:Integer):Boolean* and *getMovie(phone:Number, title:String, director:String*, *language:String):Movie*
(b) the behavioural sub-query states that operation payment() needs to be executed before operation getMovie(), similar to the query in Fig. 7;
(c) the constraint sub-query as described in Fig. 8.

Consider $S_{MovieNew}$ a service in the set of candidate services for S_{Movie} that matches the contextual constraint of Q1 and has overall distance with Q1 below the expected threshold. Assume $S_{MovieNew}$ with the operations below and a state machine in which *Op1* is executed before *Op3* and *Op3* is executed before *Op2* (*Op1 → OP3 → Op2)*.

Op1: payment(phone:Number, serviceType:String, amount:Integer):Boolean
Op2:getMovie(phone:Number,title:String,director:String,language:String):Movie
Op3:listRelatedMovies(phone:Number,title:String,director:String,language:String):String

Fig. 13 shows the structural (d_S) and behavioural (d_B) distances for all possible combinations of mappings of pairs of operations in Q1 (d_{SB}) and $S_{MovieNew}$ and the structural_behavioural distance (D_{SB}) for each combination. As shown in this figure, the behavioural distances for the mappings in combinations 2, 3, and 8 are zero, since in these combinations the mappings of the query and service operations guarantee the order specified by the behavioural condition in the query. More specifically, in combination 2, the query operation payment is mapped to service operation Op1, the query operation *getMovie* is mapped to service operation Op2, and Op1 occurs before Op2 in the state machine of the service. Similar situations occur in (a) combination 3, in which payment is mapped to Op1, *getMovie* is mapped to Op3, and Op1 occurs before Op3 in the state machine of the service; and (b) combination 8, in which payment is mapped to Op3, *getMovie* is mapped to Op2, and Op3 occurs before Op2 in the state machine of the service. In all other combinations, the behavioural distances are set to 1 since the service operations mapped to payment and *getMovie* query operations do not preserve the order specified by the behavioural condition of the query. For example, in combinations C4, C7, and C6, payment is mapped to a service operation that occurs after the service operation to which query operation *getMovie* is mapped. In this example the best mapping is the one corresponding to combination 2.

5. Discussion

The framework that we have presented in the preceding sections provides a common basis for supporting both design time and execution time adaptation of service-based systems based on service discovery.

In particular, the discovery of services during the design of service based systems needs to be compatible with established system design specification languages and processes. This means that it should enable the specification of service discovery queries in ways that are conceptually close to design specification languages in order to make it easy to specify the discovery queries during the design phase of the software development life cycle and derive discovery conditions for

the services to be discovered from design models. Furthermore, it should be possible to create representations of the discovered services in the same language that has been used to specify the design model which has driven their selection, and integrate the representations of the services that the designers decide to use consistently into these models.

It should also be noted that, from a matching point of view, the discovery process should be able to offer varying degrees of flexibility as such degrees might be appropriate and required at different stages of the system design process depending on the maturity of the ongoing design model. In early stages of the system design process, for instance, it is very likely to require a high degree of matching flexibility in discovery in order to ensure that no services which could be potentially useful for a system are missed due to strict matching. Later in the design process, however, when the design model of a system is more likely to have taken a rather elaborate and stable form, the degree of flexibility in matching may need to be reduced in order to ensure that the key assumptions of the design model and the constraints that it defines for the system are preserved by any of the services that can be located through the discovery process. In general, the degree of matching flexibility correlates negatively with the maturity of the design model: the more mature the design model less flexibility is required and vice versa.

The discovery framework that we have presented in this chapter addresses these requirements as it supports the graphical specification of service discovery queries in reference to design models of service based systems expressed in UML, the automatic expansion of these queries with parts of the specification of the design model which are relevant to the required services, the transformation of the expanded queries into queries that are expressed in a common executable query language, the execution of the queries, and the transformation of the descriptions of the located services back into UML. The discovery framework provides a range of features that enable designers to control the flexibility of the matching process including the abilities: (a) to distinguish between hard and soft constraints that should be satisfied by the located services and allow the generation of results that satisfy the latter type of constraints only partially, (b) to define the weight that each soft constraint should have in the matching process, (c) to opt between different types of structural matching between the types used in the operation signatures in queries and services (e.g., strict subgraph matching vs. detection of non overall structure preserving morphisms between these types), and (d) to opt between different types of matching for query and service behavioural models.

Our framework could provide more comprehensive support for service discovery during system design subject to certain enhancements. The first of these enhancements is support for verifying that the design model generated from integrating the descriptions of the discovered services with the original design model used in the discovery process satisfies certain properties (e.g., avoidance of deadlocks). The results of verification analysis could be used to decide which of the discovered services should be integrated into the system. The verification process could also be used in order to support the specification of discovery queries in subsequent iterations. If the conjunction of a service model and a design model is found to violate a property (e.g. avoidance of deadlocks), and the violation can be attrib-

uted to certain elements in the model of the service, a subsequent query could be formulated with conditions that would filter out services having these elements.

The discovery of services during the execution time adaptation of service-based systems needs to support different situations that may trigger the need for adapting the systems. In this respect, the framework that we described in this chapter provides the required support for (i) unavailability of malfunctioning of a service, (ii) changes in the characteristics of a service, (iii) changes in the context of a service or the application's environment, and (iv) availability of a service that is superior to one being used in the system. Moreover, the language to allow for the specification of queries should be able to express complex and different types of constraints. Services are described from different perspectives and, therefore, in order to guarantee a better precision during the identification of a service to replace a service on a running service-based system, it is necessary to consider combinations of these different perspectives.

Although our experience has demonstrated that the discovery of services based on different perspectives increases its precision, it is not possible to guarantee that services will always be described in terms of their structural, behavioural, and quality characteristics. Therefore, it is necessary to provide ways to infer the behaviour of services based on monitoring of these services.

Also for execution time adaptation of service-based systems another critical factor is the performance of the service discovery process. This factor is often neglected in the literature. Our measure for addressing this factor is the pro-active discovery approach. In this approach, which is realized by our framework, services are continually identified in parallel to the execution of the system and pointers to those of them that can replace existing system services are maintained so as to be able to rebind to them immediately if the need arises. Based on this pro-active approach, our framework provides considerable reduction in the time required for identifying replacement services (see [56] for experimental results).

Some limitations of the adaptation process provided by the framework include the need to support changes in service-based systems that are not only concerned with the replacement of a service by another service, but that consider (a) replacement of a service by a composition of services, (b) replacement of a group of services by a single service or another composition of service, (c) changes in other parts of a service-based system workflow (e.g., variables, conditions, loops). Adaptation of service-based system should consider other aspects that may require adaptation apart from (i)-(iv) above such as dynamic evolution and changes of business activities that underpin and need to be supported by the service-based system, as well as dynamic evolution of user requirements and demands.

6. Related Work

Two areas of research related to the work presented in this chapter, namely service discovery and service-based system adaptation, are reviewed below.

Service discovery has been a main strand of research in service oriented computing. The different approaches to service discovery can be classified based on the *core algorithmic aspects* of the search process that they use into (i) keyword,

(ii) model, and (iii) semantics based approaches. Some approaches support also context based service discovery.

Keyword based service discovery approaches specify service requests as sets of keywords and are implemented predominantly as part of service search engines. Examples of such engines include Seekda [34] and Strikeiron [41]. Seekda offers a search facility that looks into textual and interface (i.e. WSDL) descriptions of services. Seekda allows also the application of predefined filters on discovered service results (e.g country or service provider). Both these search engines provide browsing facilities for discovery and continually updated fixed types of searches based on criteria such as the frequency of service usage. Keyword based searches are also deployed in active service registries such as AWSR [42]. AWSR searches for RSS data feeds providing information about services in different web-sites, combines them together, and offers them in new "syndicated" data feeds. Similarly, text-based requirements specifications are matched with textual descriptions of services to drive the discovery process in [49]. The latter approach uses also term disambiguation based on WordNet Overall, keyword based approaches are easy to implement, conform to the paradigm of information retrieval over the Internet, and are more natural and easier to use from an end user perspective. However, they tend to have low precision and recall and are unable to support complex querying conditions regarding interface, behaviour, or quality of services.

In model based service discovery approaches, queries and services are described in some form of structured models without using ontologies and semantic matchmaking techniques. In [14], for example, service discovery is based on the use of behavioural service models represented in WSCL [54] and matching these models with graphs representing users requirements based on graph matching. The work in [44] proposes QoS-based selection. A goal-based model that considers re-use of predefined goals, discovery of relevant abstract services described in terms of capabilities, and contracting of concrete services to fulfill requesting goals has been proposed in [21]. Other approaches deploy graph transformation rules [17,20], or behavioural matching [14,16,27,35]. The approach in [16] uses abstract behavioural models of services. In [15] and [36], functional and quality crosscutting concerns of components and services are specified as aspects, and discovery is based on a formal analysis and validation of aspect based descriptions. The work in The use of behavioural specifications expressed in BPEL and a tree-alignment algorithm to identify query-service matchings is used in [27].

The approach described in [24] supports the specification of service discovery queries using system design models expressed in UML and uses graph matching techniques. In [30], the authors propose USQL, an XML-based language to represent syntactic, semantic, and quality of service search criteria. An extension of USQL that incorporates behavioural models expressed as UML sequence diagrams has been proposed in [31]. An XML based service query language is also used in [54,37] for service discovery. The queries expressed in this language can cover service interface, behaviour and QoS characteristics and can be specified by system developers to discover replacement services for systems at runtime. The query language proposed in [32] is used to support composition of services based on user goals. In [5] the authors propose BP-QL a visual query language for business processes.

Some of the model based approaches are hybrid as they deploy both keyword-based lexical matching and model matching. WSDL-M2 [23] uses lexical matching to calculate linguistic similarities between concepts, structural matching to evaluate the overall similarity between composite concepts, and combines vector-space model techniques with synonyms and semantic relations based on WordNet. The work in [51] combines WordNet-based techniques and structure matching for service discovery. The work in [45] uses four similarity assessment methods for service matching.

Model based approaches enable the specification of semantically richer queries and can produce results of higher precision than keyword based approaches. These advantages arise when services that have descriptions in the form of structured models assumed by the approaches are available. However, this is not always the case. Therefore, these approaches are often non applicable especially in situations where normal users are searching for services to support them, rather than to build some "system" out of them. On the other hand model based approaches work better for system designers who might have, complete or partial, system design models providing the basis for specifying model based service discovery queries.

The semantic based approaches assume services described in terms of structured models that may incorporate logical conditions expressing behavioural and quality service properties, annotated by ontologies. These ontologies are used to signify the semantics of services, providing a basis for detecting semantic similarities between services as well as between services and queries. Several techniques have been developed using semantic approaches including [1,17,19,21,22]. METEOR-S [1], adopts a constraint driven service discovery approach in which service requests are integrated into the composition process of a service-based system. In [17], service discovery is based on matching requests specified in a variant of Description Logic (DL). The work in [21] supports explicit and implicit service semantics and uses logic based approximate matching and Information Retrieval (IR) techniques. In [35] a query language based on first-order logic that focuses on properties of service behaviour signatures specified in OWL is used to support the discovery process. Approaches for service discovery based on service capabilities have been proposed in [28,50]. The work in [50] uses DAML-S to describe service capabilities, while in [28] services are described in OWL. In [50] service requests are matched against service advertisement. The approach considers four degrees of matching, namely exact, plugin, subsumes, and fails. The work in [28] reduces these four degrees of matching to three degrees (exact, inclusive, and weak), and considers discovery of pervasive services based on context and QoS characteristics. The semantic service discovery category includes also many approaches that have been developed to support context aware service discovery such as [6,48].

Semantic service discovery approaches are normally expected to be more precise than their model and keyword based counterparts. This expectation, however, is not plausible unless ontologies are accurate and consistent. Generally, there is lack of appropriate service ontologies and semantic based approaches are limited to those cases where there are adequately described services. This limitation is also due to the fact that semantic approaches are more difficult to use, as they require the specification of some complex logic-based queries. Similarly, these approaches require substantial investment from service providers to provide complex

service models, annotate them in reference to existing ontologies, and maintain service descriptions when ontologies evolve.

Context oriented service discovery approaches are found in [5,6,9,39,48]. These approaches may use matching techniques similar to those described above, but differ on the automatic generation of discovery queries from service deployment context. In [9], context information is represented by key-value pairs attached to the edges of a graph representing service classifications. This approach does not integrate context information with behavioural and quality matching, and context information is stored explicitly in a service repository that must be updated following context changes. In [6] queries, services, and context information are expressed in ontologies. The work in [5] focuses on user context information. The approach in [48] 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. The above approaches support the use of context conditions in service discovery but do not fully integrate such conditions with behavioural criteria. They also have limited applicability since they depend on the use of specific ontologies for the expression of context conditions. An approach that uses other discovery criteria (service interface and behaviour) without assuming the use of a particular ontology is presented in [39]. Context oriented service discovery approaches are useful when the criteria for service discovery are related to the deployment context of the required service and their main strength arises from their ability to construct automatically queries with context discovery conditions and execute them in a seamless way. The drawback of such approaches is that they often do not take into account other discovery conditions and if they do these conditions need to be specified manually.

Recently, some approaches that support **adaptation of service-based systems** started to appear. The dynamic binding approach described in [4] provides binding and reconfiguration rules to support evolution of service compositions during runtime. The work supports four types of rules, namely: (a) rule to discover services based on defined constraints and preferences and to bind the best identified service; (b) rule to bind to a service from a list of candidate services defined during design time of the composition; (c) rule to delete or modify a binding to point to another service; and (d) rule to stop the execution of the composition.

A self-healing approach for service compositions based on monitoring rules and reaction strategies is proposed in [3]. Examples of reaction strategies, i.e., actions taken by the system when monitoring expressions are not verified, include the re-execution of the same service invocation, the selection of a new service, the creation of a composition of services that can execute the behaviour of the faulty interaction or changes in the way that the process is monitored (i.e., less strict). Another self-healing approach is found in the PAWS framework [2], which also uses monitoring and recovery actions. Examples of PAWS recovery actions include retrying/redoing the process task, substituting the problematic service by a candidate service, or executing a compensation action.

The VieDAME framework [29] uses an aspect-oriented approach to allow adaptation of service-based systems for certain QoS criteria based on alternative

services. A service participating in the system can be marked as replaceable to indicate that alternative services can be invoked instead of the original one, when necessary. Policies are used to indicate alternative services. In [18], the authors propose PROSA, a pro-active adaptation approach based on online testing.

7. Conclusions and Future Work

In this chapter we presented a framework to support design and execution time adaptation of service-based systems based on service discovery. The work presented in this chapter unifies the results of two EU F6 projects in the area of service-oriented computing, namely SeCSE [33] and Gredia [13] projects. The work has been developed based on the challenges identified by industrial partners in both projects in the areas of telecommunications, automotive, software, media, and banking. Different characteristics of predecessors of the unified framework focusing on design and execution time service discovery have been discussed in earlier publications (see Section 1). This chapter, however, presents the unified version of the framework for the first time, and uses the Cell Phone Operator system case study of this book to illustrate how the framework works, hence making it easier for the reader to appreciate how it compares with alternative approaches.

Several evaluations of different aspects of the framework have been conducted and reported elsewhere (e.g. in [58,40]). These evaluations measure the framework in terms of the recall and precision that it achieves in service discovery as well as the efficiency of the service discovery processes that it implements both at design time and at execution time. Overall, as we have reported in the cited papers, the results of these evaluations have been very positive.

Currently, we are extending the framework to address some of the points that we discuss in Section 5. In particular, we are extending the framework to support: (a) discovery based on behavioural service operation composition, (b) the verification of design models after introducing into them models of discovered services, (c) pro-active negotiation of service level agreements as part of the execution time service discovery process, and (d) other forms of adaptation of service-based systems triggered by changes in business activities and user desires.

References

1. R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint Driven Web Service Composition in METEOR-S, Int. Conf. on Services Comp. 2004.
2. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A Framework for Executing Adaptive Web-Service Processes. IEEE Software, 24 (6), (2007).
3. Baresi, L., Ghezzi, C., Guinea, S.: Towards Self-Healing Compositions of Services. Studies in Computational Intelligence, v. 42, Springer (2007).
4. Baresi, L., Di Nitto, E., Ghezzi, C., Guinea, S.: A Framework for the Deployment of Adaptable Web Service Compositions. Service Oriented Computing and Applications Journal (to appear).
5. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. 32nd International Conference on Very Large Data Bases, VLDB, Korea, September (2006).
6. 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, 2005.
7. BPEL4WS.http://www128.ibm.com/developerworks/library/specification/ws-bpel/
8. CoDAMoS. www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/CoDAMoS/ontology/
9. 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.
10. Deubler M., Meisinger M., Kruger I. Modelling Crosscutting Services with UML Sequence Diagrams. 8th Int. Conf. on Model Driven Engineering Languages and Systems, 2005
11. J. Dooley, A. Zisman, G. Spanoudakis. Runtime Service Discovery for Grid Applications. Book chapter in Grid Technology for Maximizing Collaborative Decision Management and Support: Advancing Effective Virtual Organizations, 2009.
12. Gardner T. UML Modelling of Automated Business Processes with a Mapping to BPEL4WS. *In 2nd European Workshop on OO and Web Services (ecoop), 2004*.
13. GREDIA. www.gredia.eu.
14. D. Grirori, J.C. Corrales, and M.Bouzeghoub. Behavioral Matching for Service Retrieval, International Conference on Web Services, ICWS 2006, USA, September 2006.
15. J. Grundy and G. Ding. Automatic Validation of Deployed J2EE Components Using Aspects. IEEE 16th International Conference on Automated Software Engineering, USA, 2001.
16. R.J. Hall and A. Zisman. Behavioral Models as Service Descriptions, Int. Conf. on Service Oriented Computing, 2004
17. J.H. Hausmann, R. Heckel and M. Lohman. Model-based Discovery of Web Services, Int. Conf. on Web Services, 2004.
18. Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M.: A Framework for Proactive Self-Adaptation of Service-based Applications Based on Online Testing, 1st Eur. Conf. Towards a Service-Based Internet, ServiceWave, LNCS 5377, 2008.
19. W. Hoschek. The Web Service Discovery Architecture, IEEE/ACM Supercomputing Conf., 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, 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. Kokash N., van den Heuvel W.J., D'Andrea V. Leveraging Web Services Discovery with Customizable Hybrid Matching, Int. Conf. on Web Services, ICWS 2006, 2006.
24. A. Kozlenkov V. Fasoulas F. Sanchez G. Spanoudakis A. Zisman. A Framework for Architecture-driven Service Discovery, International Workshop on Service Oriented Software Engineering (IW-SOSE'06), ICSE, China, 2006.
25. Kozlenkov A., Spanoudakis G., Zisman A., Fasoulas F., Sanchez F. Architecture-driven Service Discovery for Service Centric Systems, International Journal of Web Services Research, special issue on Service Engineering,, 4(2):81-112, 2007
26. K Mahbub and A. Zisman, Replacement Policies for Service-based Systems, 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+), Stockholm, November 2009.
27. R. Mikhaiel and E. Stroulia, Interface- and Usage-aware Service Discovery, 4th Int. Conf. on Service Oriented Computing (ICSOC), 2006.
28. B. Mokhtar S., Preuveneers D., Georgantas N., Issarny V., and Berbers Y. EASY: Efficient semantic Service discovery in pervasive computing environments with QoS and context support. Journal of Systems and Software 81: 785-808, 2008.
29. O.Moser, F. Rosenberg, S. Dustdar, Non-Intrusive Monitoring and Service Adaptation for WS-BPEL, 17th Int. World Wide Web Conference, 2008.
30. 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), 2006.

31. 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.
32. M. Papazoglou, M. Aiello, M. Pistore, J. Yang. XSRL: A Request Language for web services, http://citeseer.ist.psu.edu/575968.html
33. SeCSE. Secse.eng.it/pls/secse/ecolnet.home.
34. Seekda . [Online]. Available at: http://seekda.com/
35. Z. Shen and J. Su. Web Service Discovery based on Behavior Signatures. IEEE Int. Conf. on Service Computing, 2005.
36. S. Singh, J. Grundy, J. Hosking, J. Sun. An Architecture for Developing Aspect-Oriented Web Services, 3rd European Conf. in Web Services, 2005.
37. G. Spanoudakis G., A. Zisman, A. Kozlenkov: A Service Discovery Framework for Service Centric Systems, 2005 IEEE Conference on Services Computing (SCC 2005), 2005.
38. G. Spanoudakis and A.Zisman. UML-based Service Discovery Tool, 21st IEEE International Conference on Automated Software Engineering Conference, ASE, Japan, 2006.
39. G. Spanoudakis, K. Mahbub, and A. Zisman. A Platform for Context-Aware Run-time Service Discovery, IEEE Int. Conf. on Web Services (ICWS 2007), USA, 2007
40. G. Spanoudakis and A. Zisman. Discovering Services during Service-based System Design using UML, IEEE Transactions of Software Engineering (to appear).
41. Strikeiron, [Online]. Available: http://strikeiron.com/
42. M. Treiber and S. Dustdar, Active web service registries, *IEEE Internet Computing*, 11(5): 66–71, 2007.
43. UDDI. www.uddi.org.
44. X. Wang, T. Vitvar, T. Kerrigan, and I. Toma, "A QoS-Aware Selection Model for Semantic Web Services", 4th Int. Conf. on Service Oriented Computing, ICSOC, USA, 2006
45. J. Wu and Z. Wu "Similarity-based Web Service Matchmaking". IEEE Int. Conf. on Services Computing, SCC, 2005.
46. WSDL. http://www.w3.org/TR/wsdl
47. XQuery. http://www.w3.org/TR/xquery/
48. Y. Ye and G. Fischer. Context-Aware Browsing of Large Component Repositories. IEEE 16th Int. Conf. on Automated Software Engineering, ASE, USA, 2001.
49. K. Zachos, N.A.M Maiden, S.Jones and X.Zhu, 'Discovering Web Services To Specify More Complete System Requirements', 19th Conf. on Advanced Information System Engineering, (CAiSE), 2007.
50. Paolucci M., Kawamura T., Payne T.R., and Sycara K. "Semantic Matching of Web Services Capabilities". Int. Semantic Web Conference, Italy, 2002.
51. Wang Y. and Stroulia E. "Semantic Structure Matching for assessing Web-Service Similarity", 1st Int. Conf. on Service Oriented Compusting, 2003.
52. WOOGLE. http://www.gujian.net/woogle/
53. WSDL. http://www.w3.org/TR/wsdl..
54. WSCL. Web Services conversation language. http://www.w3.org/TR/wscl10
55. A. Zisman, K. Mahbub, and G. Spanoudakis. A Service Discovery Framework based on Linear Composition, 2007 IEEE International Conference on Services Computing (SCC 2007), USA, July 2007.
56. Zisman A., Spanoudakis G., Dooley J.: A Framework for Dynamic Service Discovery, 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008.
57. A. Zisman, J. Dooley, G. Spanoudakis. Proactive Runtime Service Discovery, IEEE 2008 International Service Computing Conference (SCC '08), Hawaii, 2008.
58. A. Zisman and G. Spanoudakis. UML-based Service Discovery Framework, 4th International Conference on Service Oriented Computing, ICSOC, Chicago, 2006.
59. A. Zisman, G. Spanoudakis, J. Dooley. A Query Language for Service Discovery, 4th International Conference on Software and Data Technologies - ICSOFT, Bulgaria, 2009.