

A Framework for Dynamic Service Discovery

Andrea Zisman

Department of Computing
City University

London, EC1V 0HB, UK
a.zisman@soi.city.ac.uk

George Spanoudakis

Department of Computing
City University

London, EC1V 0HB, UK
gespan@soi.city.ac.uk

James Dooley

Department of Computing
City University

London, EC1V 0HB, UK
James.Dooley.1@soi.city.ac.uk

Abstract - Service discovery has been recognised as an important activity for service-based systems. In this paper we describe a framework for dynamic service discovery that supports the identification of service during the execution time of service-based systems. In the framework, services are identified based on structural, behavioural, quality, and contextual characteristics of a system represented in query languages. The framework supports both pull and push modes of query execution. In the approach, a service is identified based on the computation of distances between a query and a candidate service. A prototype tool has been implemented in order to illustrate and evaluate the framework. The paper also describes the results of a set of experiments that we have conducted to evaluate the work.

I. INTRODUCTION

Service-based systems, in which software systems are constructed as compositions of services, have become an important paradigm for the development of software systems. In such setting, dynamic service discovery has been recognised as an important activity. More specifically, it is necessary to identify services that can replace services participating in service-based systems during execution time of such systems, in order to allow these systems to continue to operate.

There are many situations in which it is necessary to replace services during service-based system execution. Examples of these situations are: (a) unavailability, or malfunctioning of services participating in the system, (b) changes in the structure, functionality, quality, or context of services participating in the system, (c) changes in the context of the system that uses a service, or (d) availability of a “better” service due to the provision of a new service or changes in the characteristics of an existing service. Here by “better” service we mean a service that can fulfil the structural, functional, quality, and contextual properties of a system in a superior way than existing services. Moreover, “context” signifies information about the operational environment of a system or of a service that may change dynamically (e.g., location, server workloads, active network connections, availability of network resources).

To illustrate the above situations, consider a service-based application that allows users to purchase goods on-the-go from a mobile phone. One of the main requirements of the users of this application is to be able to purchase their goods in a fast and efficient way. More specifically, the application offers services that allow users to search for a certain item on the mobile device, display information about the item including its picture on the mobile device, compare prices of a selected item,

calculate the cost of delivering the item, allow an item to be paid by transferring money from the user’s bank account after checking for the account’s balance, and inform the user of the account’s new balance after an item has been purchased.

Suppose a user of this application wants to purchase an item. However, after informing the system of the item that the user wants to buy, the application stops and information about the item is not displayed for the user (*case a*). In this case, the application needs to search for a new service to support the tasks of searching for and displaying information of an item. After the new service is identified and added into the application, the user selects an item to be purchased and the application continues by comparing prices of the item from different suppliers. Unfortunately, at this stage, there are many users in the system and the time spent to compare prices of the item drops considerable (*case b*). The application tries to identify a service that can perform the above tasks with an acceptable execution time that does not compromise the application. A new service is added into the application and after the user places the order of the cheapest item, the cost of delivery is calculated. After this, the value of the item and delivery cost are debited from the user’s bank account and the new balance is displayed for the user. After some hours, the user decides to purchase another item to be delivered later on in the day. At this time, the user is in a part of the city in which there is no wifi network connection and it is necessary to use a GPRS network connection. However, given that GPRS connection has a lower performance than wifi connection, the *search* service should not return a picture of the item to be purchased (*case c*). In this case, a *search* service that does not return a picture of the item needs to be identified and added to the application. In the next day, a new service that allows payments for an item to be executed not only by debiting the value of an item from a user’s bank account, but also supports credit card payments becomes available (*case d*). The application adds this service and at the next time that the user attempts to purchase an item, the system offers the user the option of paying by using a credit card.

Many approaches have been proposed to support service discovery in general [2][13][15][16][18], and dynamic service discovery, in particular [6][7][8][26][31]. However, these approaches support dynamic service discovery based on certain characteristics of the system using pull mode of query execution.

Unlike the above approaches, in this paper, we propose a dynamic service discovery framework that

allows the identification of services based on various characteristics of the system such as structural, behavioural, quality, and contextual aspects represented in query languages. Moreover, our framework allows the discovery of services that are described from different perspectives namely interface (WSDL[32]), behavioural (BPEL[4]), quality (XML-based format), and context (XML-based format) perspectives acquired from specific context services.

The framework supports service discovery based on both pull and push query execution modes in order to address the situations illustrated above. The pull mode of query execution is performed by searching service registries when a service-based application is deployed, in order to assist with the identification of services that are initially bound to the application and identification of candidate replacement services for the bound services. The push mode of query execution is performed during execution time of an application to support cases (a) to (d) above, based on subscribed services and queries and up-to-date sets of candidate replacement services. The query execution is based on the computation of distances between query and services.

The remainder of this paper is structured as follows. In Section II we present an overview of the framework with its main components and the languages that it deploys to represent queries. In Section III we describe the service discovery process supported by the framework. In Section IV we discuss implementation and evaluation aspects of our work. In Section V we give an account of related work. Finally, in Section VI we discuss concluding remarks and future work.

II. DYNAMIC SERVICE DISCOVERY FRAMEWORK

A. Framework Overview

Fig. 1 shows the overall architecture of our dynamic service discovery framework with its main components. As shown in the figure, the main components of the framework are: (i) service requester, (ii) service matchmaker, (iii) service listener, (iv) service context server, (v) application context server, and (vi) service registry intermediary.

The *service requester* orchestrates the functionality offered by the other components in the framework. It (a) receives a service request from a client application as well as context information about the services and application environment, (b) prepares service queries to be evaluated, (c) organises the results of a query and returns these results to a client application, (d) manages push query execution mode subscriptions, (e) receives information from listeners about services that become available or changes to existing services, (f) invokes the other components to execute a query.

In the framework, a query may contain different criteria, namely: (i) *structural*, specified by service models describing the interface of a required service; (ii) *behavioural*, represented by part of a behavioural model of the workflow of an application within which a required service is deployed, describing the functionality of the required service; and (iii) *constraints*, specifying extra conditions for the service to be discovered. These extra conditions may be concerned with structural, functional,

or quality aspects of a service to be discovered that cannot be represented by interface or behavioural model descriptions used in the framework. For example, 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.

The constraints in a query can be *contextual* or *non-contextual*. A contextual constraint is concerned with information that changes dynamically during the operation of the service-based application and/or the services that the system deploys, while a non-contextual constraint is concerned with static information. 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. The soft constraints do not need to be satisfied by all discovered services, but are used to rank candidate services for a query.

In the framework, the contextual constraints are evaluated as soft given that it is possible to have changes in the context of the application environment or services while evaluating the constraints. For example, in the scenario described in Section I, it is possible that while the application is trying to identify a service that supports GPRS connection, the user changes his location to an area of the city that only supports Bluetooth connection.

The *service matchmaker* is responsible to parse the different criteria of a query and evaluate these criteria against service specifications in the various service registries. The evaluation of a query is executed in a four-stage process. In the first stage, hard non-contextual constraints of a query are matched against services in the registries returning a set of candidate services that are compliant with these constraints.

In the second stage, the structural and behavioural models of a query are evaluated against the structural and behavioural specifications of services returned by the first stage in the process, or services in the registries, in the absence of hard non-contextual constraints in a query. The structural evaluation is based on the matching of the names and parameters of the operations of a required service in a query and the operations of a service. The matching of the parameters takes into consideration the data types of the parameters by using a variant of the VF2 sub-graph isomorphism algorithm [29] that we have developed. The behavioural evaluation is based on the comparison of state machines representing the behavioural models of services and queries (see Subsection III.A).

In the third stage, the soft non-contextual constraints of a query are evaluated against specifications of services returned by the previous stage generating a new set of candidate services. In the fourth stage, the context characteristics of the candidate services are matched against the context constraints in a query.

In order for a query to be executed it needs to contain at least structural criteria. When a query does not have behavioural criteria and/or constraints, the candidate services are not evaluated against the criteria and the respective stages in the process are not executed.

The *service registry intermediary* supports the use of different service registries and the discovery of services stored in these registries by providing an interface to access services from various registries. The framework supports services from registries that are based on a faceted structure developed in SeCSE project [28]. In this structure, a service is specified by a set of *facets* representing its different aspects such as (i) *structural facets* describing the operations of a services with their data types using WSDL[32], (ii) *behavioural facets* describing behavioural models of services in BPEL[4], (iii) *quality of service facets* describing quality aspects of services represented in XML-based schema, and (iv) *context facets* describing the types of context information for a service and its location represented in XML-based ontologies.

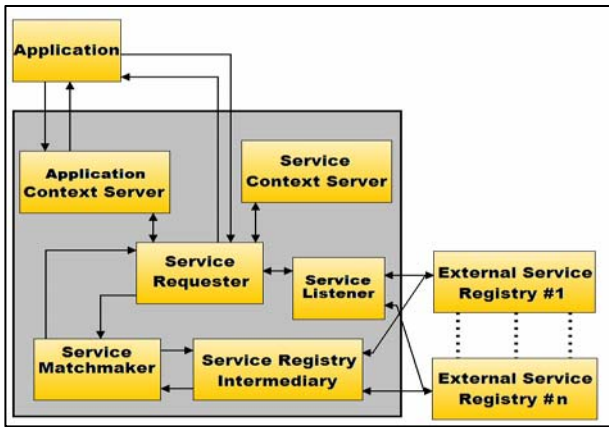


Figure 1: Architecture of the framework

The *service context server* and the *application context server listener* allow context information of the services and application environment, respectively, to be disseminated. Both context servers enable context information to be subscribed. In this case, the server sends updates to the service requester whenever changes of context occur in the services or application.

The *service listener* notifies to service requesters information about a new service that becomes available, or information about changes in the characteristics of a service. These notifications are based on subscriptions.

B. Service Discovery Queries

To represent the various criteria for discovery in a query we use different languages. More specifically, the structural criteria of a service to be replaced in an application are described by the WSDL specification of the service to be replaced. The behavioural criteria are represented by part of the behavioural model specified in BPEL of the workflow of the application in which the service to be replaced is being used. Further constraints about the service to be discovered are specified in an XML-based language called *Constraint SQL* (Constraint Service Query Language) presented in the following. Given their wide use, we do not provide descriptions of WSDL and BPEL.

Constraint SQL. Fig. 2 shows a graphical representation of part of the XML schema for specifying constraints. As shown in the figure, a constraint 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 AND and OR.

A logical expression is defined as an atomic condition, or logical combination of atomic conditions, over elements or attributes of service specifications (for non-contextual constraints) or over context aspects of service operations (for contextual constraints).

A constraint query has three attributes, namely (a) *weight*, specifying a weight in the range of [0.0, 1.0]; (b) *type*, indicating whether the constraint is *contextual*, *hard non-contextual* or *soft non-contextual*; and (c) *name*, specifying a description of the constraint. The weight is used to represent the prioritisations of the parameters in a query for contextual and soft non-contextual constraints.

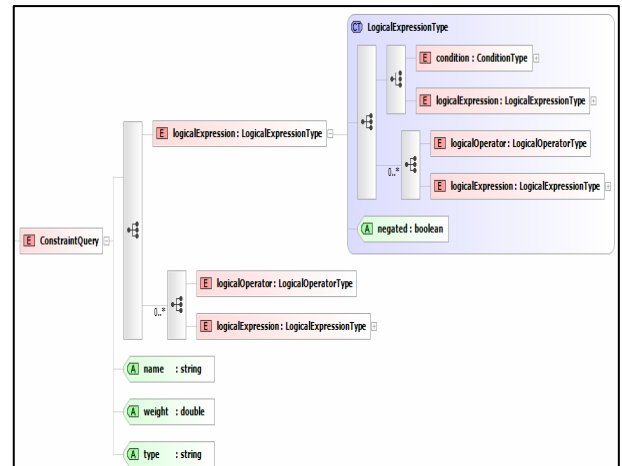


Figure 2: XML schema for *Constraint SQL* language

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

Arithmetic expressions define computations over the values of elements or attributes in service specification or context information. They are defined as a sequence of arithmetic operands or other nested arithmetic expressions connected by arithmetic operators. The arithmetic operators supported by the language are: *addition* (plus), *subtraction* (minus), *multiplication* (multiply), and *division* (divide) operators. The arithmetic operands can be contextual operands, non-contextual operands, 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. A function has a name and a sequence of one or more arguments. Each of these arguments may be contextual operands, non-contextual operands, constants, or arithmetic expressions.

A non-contextual operand is defined as a single XPath expression, or a sequence of two or more XPath expressions, combined by logical operators *AND* and *OR*. An XPath expression references a service specification facet against which the XPath will be evaluated, and the element or attribute in the facet. Therefore, the constraints can be specified against any element or attribute of any facet in the registries.

A contextual operand 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. This is due to the fact that context operations may have different signatures across different services. A contextual operand is defined by declaring (a) the name of the service operation associated with the contextual operand and (b) the contextual category of the operation.

```
<ConstraintQuery name="Query_1"
  weight="0.5" type="soft non-contextual">
<logicalExpression>
<condition negated="false">
<equalTo>
<operand1> <non-contextualOperand>
<xpathExpression>
<facet><name>QoS</name>
<type>QoS</type></facet>
<xpath> //QoSCharacteristic[Name="speed"]
/Metrics/Metric[Name="ResultReturn"]/
[Unit="seconds"]/MaxValue </xpath>
</xpathExpression>
</non-contextualOperand></operand1>
<operand2><constant>
<value>5</value>
<type>NUMERICAL</type></constant>
</operand2></equalTo></condition>
</logicalExpression>
</ConstraintQuery>
```

Figure 3.A: Example of a soft non-contextual constraint

```
<ConstraintQuery name="Query_2"
  weight="0.5" type="contextual">
<logicalExpression>
<condition negated="false">
<equalTo>
<operand1>
<contextOperand serviceName="transferAmount">
<contextCategory> <xpathExpression>
<equalTo> <qualifiedXPath>
<ontology>http://localhost:8082/ontology/
CoDAMoS_Extended.xml</ontology>
<xpath>string(/owl:Class/@rdf:ID)</xpath>
</qualifiedXPath>
<constant dataType="STRING">GREDIAAvailability
</constant> </equalTo></xpathExpression>
</contextCategory> </contextOperand></operand1>
<operand2>
<constant dataType="STRING">Hours-24:Days-7
</constant> </operand2></equalTo></condition>
</logicalExpression>
</ConstraintQuery>
```

Figure 3.B: Example of a contextual constraint

A contextual category is defined by a logical expression of conditions over the description of the category. These conditions are defined by relations over an XPath expression, or a sequence of XPath expressions, or constant values. An XPath expression includes a reference to an XML document and an element or attribute to be evaluated against this document. The document in an XPath expression can be the context facet of a service in the service registry or an ontology that this context facet uses in order to describe the category of context service operations. The constraint language can support different ontologies for describing context and contextual operations since it does not make any assumption of the structure and meaning of the ontologies used to describe semantic categories, apart from the fact that they need to be described in XML.

A contextual condition is evaluated by checking if the candidate service has a contextual operation with a semantic category that satisfies the conditions of the operation category in the query.

Fig. 3.A shows an example of a soft non-contextual constraint and Fig. 3.B shows an example of a contextual constraint for the scenario in Section I. The constraint in Fig. 3.A refers to the time that it should take to make a payment for an item by transferring money from the user's bank account, which should not be more than five seconds. The constraint in Fig.3.B refers to the availability of the service that offers transferring of money, which should be 24 hours and seven days a week. This constraint specifies that any candidate service needs to have a context operation classified under context category GREDIAAvailability in the ontology, and the result of executing this operation has to be equal to (Hours:24; Days:7) for this service to be accepted.

The development and use of a new constraint query language in our framework has followed a consideration of possible alternative languages for this purpose, including XQuery[34], OCL[24], and RuleML[27]. However, XQuery is not adequate to support service discovery since it returns documents instead of mappings of service operations or paths of service operations, it is too complex, and it does not allow for the representation of quality and contextual characteristics. In the case of OCL, the use of this language would require the description of services and context operations to be represented in UML. This, however, would be restrictive in an open environment where service discovery should accommodate different types of service registries and service representation standards which are not always object oriented (e.g. BPEL, non functional service specification facets). Finally, RuleML is also not appropriate for representing contextual categories and requires the specifications of actions.

III. DYNAMIC SERVICE DISCOVERY PROCESS

In the framework, the dynamic service discovery process executes queries in both pull and push modes. More specifically, the pull mode of query execution is performed to identify services (i) that are initially bound to an application and their candidate services, (ii) as a pre-step for the push mode of query execution, and (iii) due to changes in the context of an application environment.

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 (a)-(d) described in Section I. The framework adopts a proactive discovery process in which services to replace participating services in an application are identified in parallel to the execution of the application, based on subscribed services and queries. These subscriptions support asynchronous events to be pushed into listener components used in the framework. We describe below the matching and the query execution processes used in the framework to replace services during application execution time for cases (a)-(d) described in Section I.

A. Matching Process

The matching process between a query and services is executed by the *service matchmaker* based on invocations

received from the *service requester* in a four-stage process, as described in Subsection II.A.

The different criteria in a query are matched against service specifications based on distance measures between a query and services. More specifically, the hard non-contextual constraints of a query are used to filter services that comply with these constraints, when such constraints are defined in a query. The other criteria in a query contributes to the computation of the overall distance of a service S with respect to a query Q . This overall distance is specified as:

$$OD(Q, S) = (D_{Structural}(Q, S) + D_{Behavioural}(Q, S) + D_{Soft_non-context}(Q, S) + D_{Context}(Q, S)) / 4$$

The *structural distance* ($D_{Structural}(Q, S)$) between a service S and the structural criteria of a query Q is calculated based on the matching of the signature of the operations in the structural model of a service to be replaced and the signature of the operations in a service in the registry. The matching of query and service operations 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 the operations and parameters. This matching process uses a variant of the VF2 [29] subgraph algorithm that we have developed.

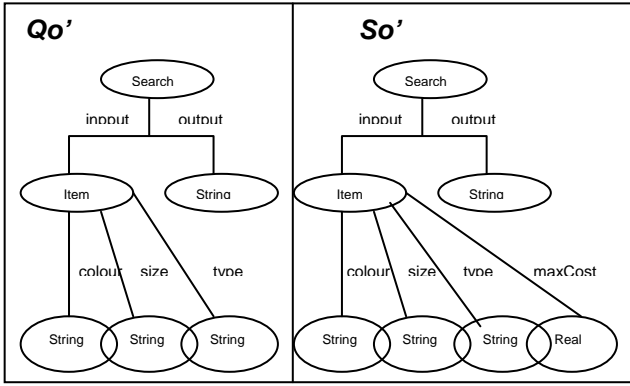


Figure 4: Example of data type graphs

More specifically, a query operation Qo with data type graphs of its input parameters ITG_{Qo} and its output parameters OTG_{Qo} matches a service operation So with data type graphs of its input parameters ITG_{So} and its output parameters OTG_{So} , if ITG_{So} is a sub-graph of ITG_{Qo} and OTG_{Qo} is a sub-graph of OTG_{So} . In other words, Qo matches So , if the data types of the input (output) parameters of a candidate service operation are super-types (sub-types) of the input (output) parameters of the query operation.

Consider m_{in} the subgraph morphism from ITG_{So} to ITG_{Qo} and m_{out} the subgraph morphism from OTG_{Qo} to OTG_{So} , the distance between Qo and So is given by:

$$D_{Structural}(Qo, So) = (D_{Linguistic}(Qo, So) + D_{input}(Qo, So) + D_{output}(Qo, So)) / 3$$

where:

- $D_{Linguistic}(Qo, So)$ is the sum of the linguistic distance of the names of the operations and the parameters based on WordNet lexicon [22];
- $D_{input}(Qo, So) = |\text{NotMapEdges}(ITG_{Qo})| / |\text{Edges}(ITG_{Qo})|$
- $D_{output}(Qo, So) = |\text{NotMapEdges}(OTG_{So})| / |\text{Edges}(OTG_{So})|$

- $\text{NotMapEdges}(ITG_{Qo}) = \{e \mid e \in \text{Edges}(ITG_{Qo}) \text{ and } \neg \exists (x, y) \in m_{in}: y=e \text{ and } x \in \text{Edges}(ITG_{So})\}$
- $\text{NotMapEdges}(OTG_{So}) = \{e \mid e \in \text{Edges}(OTG_{So}) \text{ and } \neg \exists (x, y) \in m_{out}: y=e \text{ and } x \in \text{Edges}(OTG_{Qo})\}$

In order to illustrate, consider Qo' : $\text{Search}(item:Item):string$ a query operation, with input parameter $item$ of composite data type $Item$ formed by primitive types $colour:string$, $type:string$, and $size:string$; and So' : $\text{SearchGoods}(item:Item):string$, a service operation with input parameter $item$ of composite data type $Item$ formed by primitive types $colour:string$, $type:string$, $size:string$, and $maxCost:string$. Fig. 4 shows an example of the data type graphs for the input and output parameters of query operation Qo' : $\text{Search}(item:Item):string$ and service operation So' : $\text{SearchGoods}(item:Item):string$. The structural distance between Qo' and So' is:

$D_{Structural}(Qo', So') = (0.5 + 0.25 + 0) / 3 = 0.25$, with

$D_{Linguistic}(Qo', So') = 0.5$ (the distance between words Search and SearchGoods ; and $item$ and $item$)

$D_{input}(Qo', So') = 1/4 = 0.25$ (since $maxCost$ edge in the graph of So' has no matching edge in the graph of Qo')

$D_{output}(Qo', So') = 0$ (since the output parameters in Qo' and So' are of the same type string)

The *behavioural distance* ($D_{Behavioural}(Q, S)$) between a service S and the behavioural criteria of a query Q (part of the workflow application concerned with a service that needs to be replaced specified in BPEL) is calculated based on the comparison of the state machines representing behavioural models of service and query. More specifically, the behavioural distance is calculated by verifying if a path p of the state machine of Q (SM_Q) is admissible to the state machine of S (SM_S). A path p of SM_Q is admissible into SM_S if it can be transformed into a path q of SM_S . The transformation of a path p into a path q is based on the path transformation work proposed by some of the authors in [20]. The transformation is generated by comparing all transitions t_i ($1 \leq i \leq |p|$) in p and the transition t_j ($1 \leq j \leq |q|$) in q . More specifically, each transition t_i in p is traversed and checked if it can be:

(i) *mapped* onto the next transition t_j in q : t_i can be mapped onto t_j if (a) t_i is triggered by the dispatch of an invocation of an operation o_x in the required service (a $S.send(o_x)$ event) and t_j is a transition triggered by the receipt of an o_x invocation (i.e. a $receive(o_x)$ event), or (b) t_i is a transition triggered by the invocation of an operation o_x in the query workflow (i.e., a $S.receive(o_x)$ event) and t_j is a transition triggered by an event $Q.send(o_x)$, or (c) both t_i and t_j are transitions that happen automatically (also known as “tau” transitions);

(ii) *ignored*: t_i is ignored if it is an internal transition, i.e. a transition triggered by an internal condition check, an assignment of a value to an internal variable of the query workflow, an invocation of an operation in a third party service, a receipt of the results of an operation invoked in a third party service, an invocation of an operation by a third party service, or a reply to an operation invoked by a third party service; or

(iii) *mapped* onto the final transition t_n of a sub-path of transitions t_1, \dots, t_n ($n \geq 1$) in q where t_1, \dots, t_{n-1} are internal transitions of q that can be ignored subject to the conditions of (ii) above.

When none of cases (i)-(iii) applies for a transition t_i in path p , the attempt to map p onto a path q of service S fails and another path of S is considered. If there is no path q in S to which p can be mapped, the matching between the query and the service fails. The matching process may result into a set of alternative mappings of the paths of the state machine of the query onto the paths of the state machine of the service. This set is subsequently used for calculating an aggregated distance between the query and the service which is used to rank candidate services with respect to their behavioural equivalence to the query. The function that is used to compute such distances is defined as follows:

$$D_{Behavioural}(Q, S) = D(SM_Q, SM_S) = \sum_{m \in Path_Mapp(Q, S)} MappDistance(m) / |Path_Mapp(Q, S)|$$

$$MappDistance(m) = \sum_{(t, t') \in m} D_{Structural}(op(t), op(t')) / |m|$$

where:

- $Path_Mapp(Q, S)$ is the set of all the complete mappings of the paths p in SM_Q to the paths in SM_S
- $Op(t)$ ($Op(t')$) is the operation associated with the send or receive transition t (t')
- $D_{Structural}(o1, o2)$ is the structural distance between operations $o1$ and $o2$

In order to illustrate consider, the state machines of the behavioural model of query Q' and the state machine of the behavioural specification of service S' , as shown in Fig. 5. In this case, transitions *transfer(amount:real, from:string, to:string):boolean* and *updateAccount(account:string, amount:real):boolean* in the state machine of Q' are mapped to the transitions with the same name in the state machine of S' . Given that S' has an extra transition *getBalance(account:string):real*, the behavioural distance between Q' and S' is:

$$D_{Behavioural}(Q', S') = D(SM_{Q'}, SM_{S'}) = 1 / 3 = 0.34$$

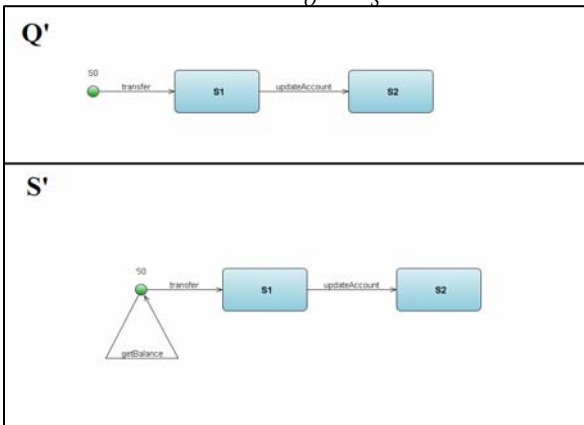


Figure 5: State machines for query Q' and service S'

The *soft non-contextual constraint* distance ($D_{Soft_non-context}(Q, S)$) is calculated based on the weights associated with all soft non-contextual constraints in a query Q . The function that is used to compute such distance is:

$$D_{Soft_non-context}(Q, S) = \sum w_i * D(C_i) / \sum w_i \quad \text{where,}$$

- C_i is a soft non-contextual constraint in Q ($1 \leq i \leq n$); n is the number of soft non-contextual constraints in Q
- w_i is the weight associated with a constraint C_i in Q
- $D(C_i) = 0$, when C_i is satisfied by service S
- $D(C_i) = 1$, when C_i is not satisfied by service S

The *contextual constraint* distance ($D_{Context}(Q, S)$) is calculated based on the weights associated with all contextual constraints in a query and is given by:

$$D_{Context}(Q, S) = \sum w_i * D(CC_i) / \sum w_i \quad \text{where,}$$

- CC_i is a contextual constraint in Q ($1 \leq i \leq n$); n is the number of contextual constraints in Q
- w_i is the weight associated with a constraint CC_i in Q
- $D(CC_i) = 0$, when CC_i is satisfied by service S
- $D(CC_i) = 1$, when CC_i is not satisfied by service S

B. Service Discovery Execution

The dynamic service discovery execution in the framework, requires an application to subscribe its environment and participating services, together with queries associated with these services, so that replacement services can be identified, when notification of changes in these services and application environments are pushed to the listeners. In the following, we describe the service discovery process for replacing a service in an application when (a) a service becomes malfunctioning or unavailable, (b) there are changes in the structure, functionality, quality, or context aspects of a subscribed service (i.e., changes in the respective facets); (c) there are changes in the context of an application environment, or (d) new services become available or existing services have their characteristics modified.

For cases (b), (c), and (d) above, it is possible that the replacement of a service in an application should not be executed immediately after the discovery process is triggered and a replacement service is identified. For these cases, it may be better to give continuity to the execution of an application with its current service, although this may not be the best service at the moment, instead of stopping the application in the middle of its execution to replace a service. For example, in the case of the scenario in Section I, if a new service that supports credit card payment becomes available when the application is debiting the user's bank account, it is better to wait to replace this service in the application, instead of risking charging the user twice.

In order to avoid stopping the application during its execution to replace a service when this is not desirable, the approach uses *replacement policies* associated with the subscribed services. These policies specify if the service should be replaced immediately when necessary or after the application terminates.

In order to follow the description of the service discovery execution process, consider S a service participating in application AP that may need to be replaced; Q a query associated with S ; Set_S the set of candidate services for Q , including S , ranked in ascending order of the distances between the services and query Q . The candidate services in Set_S are services that match the criteria of query Q and have a distance with Q that is

less or equal to a certain threshold distance. Assume that S, AP, Q, and Set_S have been subscribed.

The subscription of the services in Set_S is to guarantee that the set of candidate services for S is maintained up-to-date with respect to query Q and application AP. More specifically, in the case of changes in the facets of a service S' in Set_S (i.e., changes in the characteristics of S'), the new version of S' will be checked against Q and S' will be maintained or removed from Set_S, if S' matches Q or not, respectively. In the case of changes in the environment of AP, the services in Set_S will be checked against these changes. In the case of a new service that becomes available, or an existing service that has been modified, and matches Q, this service will be included in Set_S.

Case (a): *Service S becomes malfunctioning or unavailable.*

In this case, service S is replaced by a service S' in Set_S which has the smaller distance with Q from all the services in Set_S. Service S is removed from Set_S and unsubscribed.

```

Service_Facet_Change (S', F, Q, Dmax, RP)
//S': subscribed service
//F: new facets for S'
//Q: query
//RP: Replacement Policy
//Dmax: maximum distance for services in Set_S
Compute new distance Dnew between Q and S'
If ((F do not match Q) or (Dnew > Dmax))
    Remove S' from Set_S;
    Unsubscribe S'
Else
    Sort Set_S with Dnew
End If
If (S' in AP)
    If (S' not equal to Set_S[1])
        If (S' is not in Set_S)
            Substitute S' by Set_S[1] in AP
        Else
            Substitute S' by Set_S[1] in AP depending on RP
        End If
    End If
Else
    If (S' equal to Set_S[1])
        Substitute service S in AP by S' depending on RP
    End If
End If
End Service_Facet_Change

```

Figure 6: Algorithm for changes in a subscribed service

Case (b): *Changes in the structural, functional, quality, or context characteristics of a subscribed service S'*

Consider S' a service participating in AP (S' = S) or another service in Set_S (S' ≠ S). In this case new versions of the structural, functional, quality, or context facets for S' are included in the service registry. Fig. 6 shows the algorithm for this case.

As shown in Fig. 6, a new overall distance between Q and S' is calculated. In addition, the new versions of the changed facets need to be evaluated against query Q to verify if these facets match the query, or if the current version of service S' can still be a candidate service for Q (i.e., the new distance between service S' and query Q is below the threshold distance).

In the case in which service S' is the service in the application (S' = S), but not a candidate service for Q anymore, a new service from the set of candidate services for Q (Set_S) needs to replace S' so that the application can continue. If S' is still a possible candidate service for Q, but not the first element in the set (i.e., S' does not have the smallest distance with Q when compared to the other elements in the set), the first element in the set may replace S' in the application, depending on the replacement policy.

In the situation in which S' is a service in the set of candidates, but not the current service in the application, and S' has the smallest distance with Q from all the elements in the set (S' is the first element in the set), S' may replace service S in the application, depending on the replacement policy.

Case (c): *Changes in the context of an application environment.*

In this case the context constraint of query Q is modified and a new query Q' is created with new context constraint. Fig. 7 shows the algorithm for this case. As shown in the figure, service S needs to be evaluated against the new context constraint in Q'. In the case that S does not match Q', the services in Set_S also need to be evaluated against Q' and a new set of candidate services (Set_S') may be generated.

```

Application_Context_Change (S, Set_S, Q')
//S: subscribed service in AP
//Set_S: set of candidate services for Q
//Q': new query with new application context constraint
//Set_S': set of candidate services for Q'
If (S does not match Q')
    Unsubscribe S
    Rebuild Set_S w.r.t. Q' and generate Set_S'
    If (Set_S' is not empty)
        Substitute S by Set_S'[1]
    End If
End If
Pull Q'
Rebuild Set_S' w.r.t. Q'
If (service in AP is not equal to Set_S'[1])
    Substitute service in AP by Set_S'[1]
End If
End Application_Context_Change

```

Figure 7: Algorithm for change in the application context

This is in order to try to identify a service S' in Set_S that can fulfil Q' (although S' may not be the best service for Q' of all available services in the registry) so that the application can continue its execution while trying to find new services that match Q' from the service registries. The use of a service in the application that may not be the best match to query Q' is acceptable since context constraint is used for ranking the service with respect to a query and not as a filtering constraint.

If the new set of candidate services is not empty, the first element in the set is used to replace S in the application to allow it to continue. In any case, query Q' is executed against the services in the registries (pull mode of execution) in order to build a set of candidate services for the new query Q' and replace the service in the application by the best service identified for query Q'.

Case (d): *A new service (S') becomes available or an existing service (S) has its characteristics modified.*

In this case, for services that have its characteristics modified we are referring to services in the registry that are not subscribed services (i.e., services that are not participating in the application or are not candidate services for the queries associated with the application - we have discussed these services in Case (b)).

The new available or modified service in the registry (S') needs to be evaluated against query Q and, depending on the result, S' may replace S in the application. Fig. 8 shows the algorithm for this case.

As shown in Fig. 8, it is possible that the new available service is better than service S, but the application may be in the middle of its execution and the immediate replacement of service S is not desirable.

```

New_Modified_Service (S, S', Q, Dmax, RP)
//S: subscribed service in AP
//S': new available or modified service
//Q: query
//Dmax: maximum distance for services in Set_S
//RP: replacement policy for S
  Compute distance Dnew between Q and S'
  If (Dnew <= Dmax)
    Add S' to Set_S
    Subscribe S'
    Sort Set_S with Dnew
    If S' == Set_S[1]
      Substitute S by S' in AP following RP
    End If
  End If
End New_Modified_Service

```

Figure 8: Algorithm for new available or modified service

IV. IMPLEMENTATION ASPECTS AND EVALUATION

A prototype tool of the framework has been implemented in Java. The tool is exposed as a web service using Apache Axis2 [1], allowing its deployment by any client that can produce service requests in the format required by the framework. Moreover, the specification of service subscriptions for the push mode of query execution is assisted by WS-Eventing [33] model and by an event receiver on the application to support the delivery of push mode notifications. The external service registry uses eXist database [10] and communications with the registry is through the use of Remote Method Invocation (RMI).

To evaluate the performance of the service discovery process, we have executed a set of experiments. Our evaluation focuses on the performance of executing query matching with structural, behavioural, non-contextual, and contextual conditions. More specifically, we have evaluated the time for executing five queries with the following criteria:

- Q1: structural only;
- Q2: structural and behavioural;
- Q3: structural, behavioural, and soft non-contextual;
- Q4: structural, behavioural, and contextual;
- Q5: structural, behavioural, soft non-contextual, and contextual.

The queries used in the experiment were related to the on-the-go mobile purchase application described in Section I. These queries were specified to identify candidate services to replace the service that allows an

item to be paid by transferring money from the user's bank account after checking for account's balance.

The structural criteria of queries Q1 to Q5 are described in the WSDL specification of the service to be replaced, the behavioural criteria are represented in BPEL, and the non-contextual and contextual constraints are represented in *Constraint_SQL*. Fig. 9 shows the operations of the structural parts of the queries with their respective parameters. The non-contextual constraint used in the queries is concerned with the time to pay for an item, as described in Fig. 3.A. The contextual constraint is concerned with the availability of the service as described in Fig. 3.B. Due to lack of space we do not present here the WSDL and BPEL specifications used in the queries, but they can be found in [9].

We have executed queries Q1 to Q5 against a service registry with 80 services, 320 service facets (each service has a structural, a behavioural, a quality, and a contextual facet), and 400 operations. The services in the registry were related to online retailing, search engines, banking, and travel planning.

The performance for each query was evaluated incrementally for 20, 40, 60, and 80 services in order to analyse how the increase in the number of services affects the execution time. The time taken for each query in each registry increment was calculated as the average of five executions of the query using a Pentium 3.00 GHz with 1GB RAM machine.

```

credit(accountId:string, amount:double):balance
transferAmount(fromAccountId:int, toAccountID:int,
amount:double):boolean
debit(accountId:string, amount:double):balance
getBalance(accountID:string):balance

```

Figure 9: Operations in the queries

TABLE I. EXPERIMENTS RESULTS IN SECONDS

# of Services	Registry Retrieval	Q1	Q2	Q3	Q4	Q5
20	29.36	9.78	11.16	11.25	12.36	12.45
40	48.70	12.60	19.70	19.81	20.90	21.03
60	63.93	15.00	27.65	27.76	29.01	29.12
80	87.26	17.84	35.72	35.81	37.03	37.12

Table 1 shows this time in seconds for the five queries and different numbers of services, as well as the average time required for retrieving the services from the registry. Fig. 10 shows these results in a graph. As shown in Table 1, the time for matching queries increases linearly with the addition of more services in the registry. The results in Table 1 and Fig. 10 demonstrate that there is an increase in the time to execute structural and behavioural matching (Q2) when compared to the time spent to execute structural matching only (Q1).

Moreover, the experiment also shows that the performance in queries Q3 and Q4 is similar to the performance in query Q2, with query Q4 being slightly higher than query Q3. This is because, the matching of soft non-contextual and contextual constraints is executed in the set of services returned after the matching of structural and behavioural constraints, which has a smaller number of services when compared to the structural and behavioural matching that are evaluated against the services in the registries. More specifically,

out of the 80 services used in the experiments, only five services matched the structural and behavioural criteria used in the queries and were, therefore, evaluated against soft non-contextual and contextual constraints.

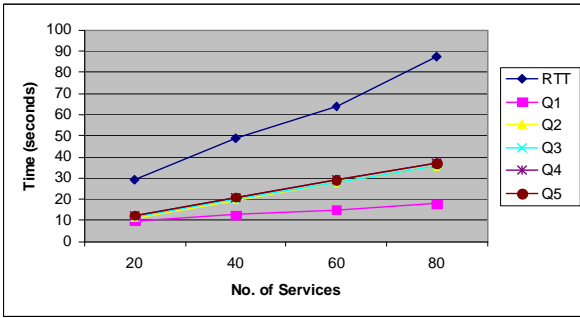


Figure 10: Average performance for queries Q1 – Q5

Furthermore, the higher performance for query Q4 when compared to query Q3 is because contextual matching requires evaluation of contextual operations at runtime before checking the contextual facet of a service against a query. A comparison of the time in seconds to execute (i) structural and behavioural, (ii) soft non-contextual, and (iii) contextual criteria in a query for a single service, without considering the time to retrieve services from the registry, is shown in Fig. 11.

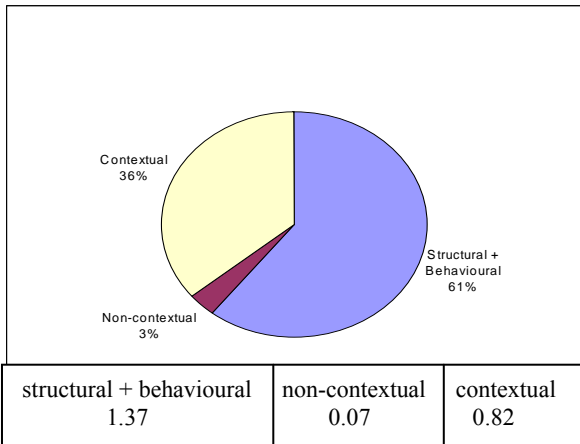


Figure 11: Results for a single service in seconds

The experiment also demonstrated that the time required to retrieve services from the registry is much higher than the time for executing query matching. Thus, the proactive push mode of query execution in which replacement services are identified in parallel to the execution of an application provides a substantial reduction in the time required for identifying services at runtime. Moreover, as described in Subsection III.B, except in situations where there is change in the context of an application, a replacement service is identified by evaluating queries against up-to-date sets of limited number of candidate services rather than entire registries.

Overall, the results of our experiment have demonstrated a good performance for query execution which when combined with the push mode of execution can provide an efficient service discovery process.

V. RELATED WORK

Several approaches have been proposed to support service discovery. We present below an account of some of these approaches.

Service discovery is supported by semantic matchmaking approaches based on logic reasoning over terminological concept relations defined by ontologies [2][14][16][18][19][21]. The METEOR-S [2] system adopts a constraint driven service discovery approach in which service requests are integrated into the composition process of a service-based system. In [13] service discovery is based on matching requests specified in a variant of Description Logic (DL). The work in [18] supports explicit and implicit service semantics and uses logic based approximate matching and IR techniques.

Hausmann et al. [12] specify requests and services using graph transformation rules. The approach in [16] focuses on operation signature checking based on string matching, but it is limited since it cannot account for changes in the order or names of the parameters. The approach in [12] advocates the use of (abstract) behavioural models of services to increase the precision of the discovery process. Similarly, in [30], the authors use service behaviour signatures to improve service discovery. In [23] and [25], the discovery of software components is also based on behavioural specifications of the components being sought. The works in [11] and [31], describe functional and quality cross cutting concerns of components and services as aspects and discovery is based on a formal analysis and validation of these descriptions.

Several approaches have also been proposed to support context awareness in service discovery [3][5][6][7][8][17][26]. In [8], 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 [5] queries, services, and context information are expressed in ontologies. Context information in this approach can also be used as an implicit input to a service that is not explicitly provided by the user (e.g. user location). The approach in [3] focuses on user context information (e.g. location and time) and uses it to discover the most appropriate network operator before making phone calls. Other approaches focusing on discovery protocols in mobile computing also support context [6][26].

The work in [35] 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 context-aware approaches support simple conditions regarding context information in service discovery, do not fully integrate context with behavioural criteria in service discovery, and have limited applicability since they depend on the use of specific ontologies for the expression of context conditions.

Overall, most of the proposed approaches support service discovery for only specific types of service criteria and only in pull mode. Unlike them, our framework supports dynamic service discovery based on a comprehensive set of service and application properties including structural, functional, quality, and contextual

properties. It also provides both pull and push service discovery mechanisms, optimising service replacement during the execution of an application. Furthermore, we provide an expressive query language allowing the specification of a wide spectrum of constraints for the required services and uses fine grain quantification of similarities between queries and services based on distance measures.

VI. CONCLUSION AND FINAL REMARKS

In this paper we have presented a framework for dynamic service discovery that allows the identification of services described from different perspectives and based on different characteristics such as structural, functional, quality, and contextual aspects, which are represented in complex query languages.

The intended users of this framework are the developers of service-based applications. These developers are responsible for writing the application code as well as for specifying the queries to be executed in order to discover replacement services when the services used in the application fail to satisfy certain criteria at runtime. The part of the queries that needs to be manually specified by developers includes only the contextual and non-contextual constraints for required services, since the behavioural and structural parts of the queries can be automatically extracted from the specification of the workflow of the application and interface specifications of the services that are deployed in the application. Service discovery queries could also be modified at runtime by either developers or system administrators of the application provided that the latter have a good understanding of the application requirements, design, and functionality.

The presented framework supports both pull and push modes of query execution and the matching of queries with services is based on the computation of distances. Furthermore, a prototype implementation of the framework has been developed and used to evaluate the framework in terms of performance with positive results.

We are currently investigating ways of defining replacement policies and conducting experiments to measure the frequency of changes of different contextual aspects of the services in order to identify ways of optimising the push mode of query execution.

ACKNOWLEDGEMENT

The work reported in this paper has been funded by the European Commission under the Information Society Technologies Programme as part of the project GREDIA (contract FP6-34363).

REFERENCES

- [1] Apache. <http://ws.apache.org/axis2/>.
- [2] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint Driven Web Service Composition in METEOR-S, Int. Conf. on Services Comp. 2004.
- [3] F. Bormann, et al, Towards Context-Aware Service Discovery: A Case Study for a new Advice of Charge Service”, 14th IST Mobile and Wireless Communications Summit, June 2005.
- [4] BPEL4WS. <http://www128.ibm.com/developerworks/library/specification/ws-bpel/>
- [5] T. Broens, et al. Context-aware, ontology-based, service discovery. LNCS 3295, 72-83.
- [6] L. Choonhwa and S. Helal. Context Attributes: An Approach to Enable Context-awareness for Service Discovery, 2003 Symp. on App. & the Internet.
- [7] 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.
- [8] C. Doukeridis, N. Loutas, and M. Vazirgiannis. A System Architecture for Context-Aware Service Discovery. Electr. Notes Theor. Comput. Sci. 146(1): 101-116 (2006)
- [9] DSD. Dynamic Service Discovery Framework Project, http://www soi.city.ac.uk/~zisman/DSD_Project
- [10] eXist. <http://exist.sourceforge.net>
- [11] J. Grundy and G. Ding. Automatic Validation of Deployed J2EE Components Using Aspects. IEEE 16th International Conference on Automated Software Engineering, ASE, USA, November 2001.
- [12] R.J. Hall and A. Zisman. Behavioral Models as Service Descriptions, Int. Conf. on Service Oriented Computing, ICSOC 2004, New York.
- [13] J.H. Hausmann, R. Heckel and M. Lohman. Model-based Discovery of Web Services, Int. Conf. on Web Services, 2004.
- [14] I. Horrocks, P.F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The Making of A Web Ontology Language, J. of Web Semantics, 1(1), 7-26, 2003.
- [15] W. Hoschek. The Web Service Discovery Architecture, IEEE/ACM Supercomputing Conf., USA, 2002.
- [16] U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel. Automatic Location of Services, European Semantic Web Conference, 2005.
- [17] M. Khedr and A. Karmouch. Enhancing Service Discovery with Context Information, ITS'02, 2002.
- [18] M. Klein and A. Bernstein. Toward High-Precision Service Retrieval. IEEE Internet Computing, 30-36, January 2004.
- [19] M. Klusch, B. Fries, and K. Sycara. Automated Semantic Web Service Discovery with OWLS-MX, Int. Conf. on Autonomous Agents and Multiagent Systems, 2006.
- [20] A. Kozlenkov and A. Zisman. Discovering, Recording, and Handling Inconsistencies in Software Specifications, Int. Journal of Computer and Information Science, 5(2), 89-108, June, 2004.
- [21] L. Li and I. Horrocks. A Software Framework for Matchmaking based on Semantic Web Technology, WWW Conf. Work. on E-Services and the Semantic Web, 2003.
- [22] J. Morato, M.A. Marzal, J. Llorens, and J. Moreira. WordNet Application, The Second Global Wordnet Conference, 2004.
- [23] H. Niu and Y. Park. An Execution-based Retrieval of Object-Oriented Components. Proceedings of the 37th ACM Southeast Regional Conference, 1999.
- [24] OCL. <http://www.omg.org/docs/ptc/03-10-14.pdf>
- [25] Y. Park. Software Retrieval by Samples Using Concept Analysis. *J. Syst. Softw.* 54(2):179-183, November 2000.
- [26] P.Pawar and A. Tokmakoff. Ontology-based Context-aware service discovery for pervasive environments, IEEE Int. Work. On Service Integration in Pervasive Environment, June, 2006
- [27] RuleML. <http://www.ruleml.org/>
- [28] SECSE Project. <http://secse.eng.it>
- [29] SECSE A2.D8, Platform for Architecture Service Discovery V2.0: Specification, April 2006
- [30] Z. Shen and J. Su. Web Service Discovery based on Behavior Signatures. Int. Conf. on Service Computing, SCC, July 2005.
- [31] S. Singh, J. Grundy, J. Hosking, J. Sun. An Architecture for Developing Aspect-Oriented Web Services, 3rd European Conf. in Web Services, 2005.
- [32] WSDL. <http://www.w3.org/TR/wsdl>
- [33] WS-Eventing. <http://www.w3.org/Submission/WS-Eventing>
- [34] XQuery. <http://www.w3.org/TR/xquery/>
- [35] Y. Ye and G. Fischer. Context-Aware Browsing of Large Component Repositories. IEEE 16th International Conference on Automated Software Engineering, ASE, USA, November 2001.