ASSERT4SoA
Advanced Security Service cERTificate for SOA
CP - STREP - Grant No. 257351

ASSERTS Aware Service Query
Language and Discovery Engine

Deliverable D.2.1
Rev. 1.0

Legal Notice
All information included in this document is subject to change without notice. The Members of the Assert4Soa Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the Assert4Soa Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.
ASSERT4SoA

D.2.1 - ASSERTs Aware Service Query Language and Discovery Engine
### Project Title
ASSERT4SoA – Advanced Security Service Certificate for SOA

### Grant Number
257351

### Project Type
CP - STREP

<table>
<thead>
<tr>
<th>Title of Deliverable</th>
<th>ASSERTS AWARE SERVICE QUERY LANGUAGE AND DISCOVERY ENGINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtitle of Deliverable</td>
<td>ASSERTS AWARE SERVICE QUERY LANGUAGE AND DISCOVERY ENGINE</td>
</tr>
<tr>
<td>Deliverable Number</td>
<td>D2.1</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>PP</td>
</tr>
<tr>
<td>Contractual Delivery Date</td>
<td>September 30, 2011</td>
</tr>
<tr>
<td>Actual Delivery Date</td>
<td></td>
</tr>
<tr>
<td>Contributing WPs</td>
<td>WP2</td>
</tr>
<tr>
<td>Editor(s)</td>
<td>George Spanoudakis</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Khaled Mahbub, Luca Pino, Howard Foster (City), Antonio Maña, Gimena Pujol (UMA)</td>
</tr>
<tr>
<td>Reviewer(s)</td>
<td>Valentina Di Giacomo (ENG), Antonino Sabetta and Samuel P. Kaluvuri (SAP)</td>
</tr>
</tbody>
</table>
### Document Review

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Reviewer</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>25/9/2011</td>
<td>0.6</td>
<td>Valentina Di Giacomo</td>
<td>........</td>
</tr>
<tr>
<td>26/9/2011</td>
<td>0.6</td>
<td>Antonino Sabetta and Samuel P. Kaluvuri</td>
<td>See inline comments + review summary</td>
</tr>
</tbody>
</table>
Executive Summary

This report is part of the deliverable D02.1 and describes a query language (A-SerDiQueL) that we have developed in ASSERT4SOA to support dynamic service discovery based on a set of discovery criteria that relate not only to structural, behavioural and quality properties of the sought services but also criteria regarding security properties of services and ASSERTs (i.e. certificates) that may have been issued to assure such properties. The report describes also the design and implementation of a discovery engine that we have developed to support this query language and give examples of using it. The initial prototype implementation of the discovery engine has also been released as part of deliverable D02.1, to enable a testing of the features of the engine. The final implementation of the engine will be delivered as part of the integrated ASSERT4SOA framework (i.e., in deliverables D06.2 and D06.3).
ASSERT4SOA
## Contents

1. Introduction .......................................................................................................................... 12  
   1.1 Goals and scope .................................................................................................................. 12  
   1.2 Terminology and conventions .......................................................................................... 12  
   1.3 Main technical achievements ........................................................................................... 12  
   1.4 Deliverable structure ......................................................................................................... 13  

2. Overview of state of the art in service discovery ............................................................... 14  
   2.1 Introduction ....................................................................................................................... 14  
   2.2 Single service discovery approaches ............................................................................... 14  
   2.3 Discovery of Service Compositions .................................................................................. 18  
   2.4 Support for security ......................................................................................................... 23  
      2.4.1 Security of discovery process ...................................................................................... 24  
      2.4.2 Security aware service discovery ............................................................................... 25  
      2.4.3 Security aware service composition .......................................................................... 27  
      2.4.4 Securing interactions with services ........................................................................... 29  
      2.4.5 Summary .................................................................................................................... 32  

3. A-SerDiQueL: The ASSERTs-aware version of SerDiQueL ............................................... 33  
   3.1 Introduction ....................................................................................................................... 33  
   3.2 Overview SerDiQueL .......................................................................................................... 33  
      3.2.1 Overall query structure .............................................................................................. 33  
      3.2.2 Query parameters ........................................................................................................ 34  
      3.2.3 Structural discovery queries ...................................................................................... 35  
      3.2.4 Behavioural discovery queries .................................................................................... 36  
      3.2.5 Query Constraints ....................................................................................................... 39  
   3.3 A-SerDiQueL ....................................................................................................................... 45  
      3.3.1 Extensions ................................................................................................................... 45  
      3.3.2 Example queries .......................................................................................................... 52  

4. A-SerDiQueL Query Engine ............................................................................................... 55  
   4.1 Overall discovery process ............................................................................................... 55  
   4.2 Architecture ....................................................................................................................... 57  
   4.3 Programmatic interface of query engine .......................................................................... 59  
   4.4 Detailed design .................................................................................................................. 61  
   4.5 Main functional features ................................................................................................. 64
4.6 Security mechanisms .............................................................. 68
4.7 Installation instructions and usage examples .................................. 70
  4.7.1 Installation ................................................................. 70
  4.7.2 Usage Guide ............................................................... 72
4.8 Limitations and known problems ................................................. 76
5. Conclusions ............................................................................. 78
  5.1 Main conclusions .................................................................. 78
  5.2 Future work ......................................................................... 79
6. Bibliography ............................................................................. 80
A. A-SerDiQueL Schema ................................................................. 86
B. Example query ......................................................................... 99
Chapter 1

Introduction

1.1 Goals and scope

The purpose of this report is to define the query language for the expression of ASSERTs aware service discovery query and describe the implementation of the prototype of the discovery (aka query) engine that has been developed by ASSERT4SOA in order to realise this language.

The specification of the query language has been based on requirements regarding the forms of dynamic service discovery that should be supported by the ASSERT4SOA framework, as specified in [80]. The design and the implementation of the query engine supporting this language has also been informed by the specification of the overall architecture of the ASSERT4SOA framework which is being developed in the work package WP6 of the project and existed in draft form at the time of the initial implementation of the engine.

The D2.1 deliverable, in addition to this report, includes a prototype implementation of the “query engine” that has been specified in this report.

1.2 Terminology and conventions

This report assumes the terminology introduced in the requirements specification document of ASSERT4SOA [80] and the general vocabulary specification of the project [81]. Hence, any term that is used in this report assumes the meaning defined in [80][81], if an identical term appears in the latter documents.

1.3 Main technical achievements

The main technical achievements described in this report are
the definition of a query language (A-SerDiQueL), to enable the expression and consideration of discovery criteria related to security certificates (ASSERTs) and security properties in dynamic service discovery, and

• the development of a discovery engine to support this query language.

1.4 Deliverable structure

The rest of this deliverable is structured as follows. In Chapter 2, we provide an overview of the state of the art in service discovery and explain the choice of SerDiQueL as the basis of the query language that we have developed for the Assert4SOA framework. In Chapter 3, we provide an overview of SerDiQueL and define the extension of it that will be used in Assert4SOA, namely A-SerDiQueL. In Chapter 4, we describe the design and implementation of the query engine that realises A-SerDiQueL. Finally, in Chapter 5, we provide concluding remarks and identify potential directions for future work.
Chapter 2

Overview of state of the art in service discovery

2.1 Introduction

This chapter presents an overview of methods, tools and techniques supporting service discovery and outlines the main reasons that have led to the development of the ASSERT4SoA approach to address this activity.

2.2 Single service discovery approaches

In this section we provide an overview of single service discovery techniques, i.e., techniques that support the discovery of a service for a service based system without attempting to formulate complete or partial service workflows (aka service compositions). They merely attempt to identify a single service that can fit within a system based on given criteria that this service needs to satisfy. In many cases, these criteria may express conditions that are necessary for the new service to fit within an existing service workflow (composition). Also, the discovered service may be a composite service itself. None of these cases, however, is treated as discovery of service compositions in the context of this document as the discovery process does not attempt to create a new composition. Techniques supporting the discovery of service compositions are overviewed in Section 2.3 below.

The techniques that we overview are classified into groups depending on the main characteristics of the algorithmic approach deployed for service discovery. According to this criterion, techniques are grouped into:

Semantic approaches – These are techniques that assume descriptions of services that have been expressed in an ontology or annotated with links to
ontological descriptions. Such techniques make use of the ontological descriptions during the matching process in order to improve the precision and completeness of the discovery process.

Text based discovery – These are techniques that make use of information retrieval techniques. In this group, discovery criteria are expressed as keywords which are subsequently matched with textual or structural descriptions of services. Typically, such techniques are deployed for early and design time service discovery.

Graph matching techniques – These are techniques that make use of different types of graph matching techniques (e.g. weighted bipartite graph matching, graph transformations, etc.) without relying on any form of ontology or formal reasoning of semantic service descriptions.

A summary of representative techniques in each of the above categories is provided below.

Discovery based on text matching

Keyword-based retrieval underpins some service registries available on the Internet (e.g. Seekda [60] and servicefinder [61]). These approaches also enable discovery through service categories and the use of tagged service descriptions. Text based service discovery is easy to use, due to the simplicity in the expression of the discovery queries. It is also useful in static service discovery, where the developers of SBA are usually concerned with finding a service that fits their requirements or the requirements of an application being designed. However, it cannot offer the matching precision that is required in dynamic service discovery that is executed to support automatic service replacement in applications. This is because in the analysis and design stages of service based applications, it is often useful to identify even services that do not match perfectly with what is required as a means of exploring alternative solutions and considering alternative designs and implementation paths for the application. At runtime, however, when the design of the overall application and its coordination logic have been fixed, the imprecision that typically characterizes keyword-based techniques is not acceptable, as decisions about replacing the partner services of a system with alternatives identified during the discovery process, in many cases, need to be taken in an automated manner.

Semantic service discovery

Semantic service discovery techniques constitute a significant approach to service discovery that is based on explicit representations of the semantics of services and logic reasoning techniques that analyse these representations. There has been a vast number of techniques that realise the semantic service discovery approach, including [49][50].
A system realising the semantic approach is OWLS-MX [49]. OWLS-MX uses logic based approximate matching and information retrieval techniques.

A semantic approach has also been advocated in [50], where a service discovery prototype that uses a Description Logic reasoner to match service discovery requests with ontology based service descriptions expressed in DAML-S.

Despite some experimental evidence showing acceptable precision and recall over competitors, however, the semantic approaches do not appear to be adequate for dynamic service discovery. This is because the ontological matches do not necessarily coincide with behavioural and interface service matching at the level required for dynamic service discovery. It should also be pointed out that, to the best of our knowledge, none of the semantic approaches supports push mode query execution.

**Graph matching techniques**

Other approaches for service discovery consider graph transformation rules [47], or behavioural matching [44],[51],[62]. The work in [47] is limited since it cannot account for changes in the order or names of the parameters. In [62], the authors use service behaviour signatures to improve service discovery. In AOWS [45],[63] the functional and quality characteristics of components and services are described as aspects and discovery is based on a formal analysis and validation of these descriptions. The work in [51] advocates the use of behavioural specifications represented as BPEL for service discovery for resolving ambiguities between requests and services and uses a tree-alignment algorithm to identify matchings between request and services.

Graph matching underpins also the runtime service discovery system [72],[76] that supports SerDiQueL, the query language that we have selected to extend in order to address the ASSERTs-aware service discovery required for ASSERT4SoA. This system uses graph morphism detection algorithms to match service interfaces and graph search algorithms to identify the compatibility of behavioural discovery criteria with behavioural service description models expressed in BPEL.

**Context awareness**

Several approaches have also been proposed to support context awareness in service discovery [35],[38],[48],[57],[68]. In [38], 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. Furthermore, the context information is stored explicitly in a service repository that must be updated following context changes. In [34] queries, services and context information are expressed in ontologies. The approach in [35] focuses on user context information (e.g. location and time) and uses it to discover the most appropriate network operator before making phone calls. The work in [68] locates components based
on context-aware 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.

Query languages for service discovery

Query languages, other than SerDiQueL, have also been proposed to support services discovery [34][55][56][69]. BP-QL [34] is a visual query language for BPEL. The query language proposed in [56] is used to support composition of services based on user’s goals. NaLIX [69], a language for querying XML databases based on natural language, has also been applied to service discovery.

USQL (Unified Service Query language) [55] is an XML-based language enabling discovery based on syntactic, semantic, and quality of service search criteria. USQL does not support the specification of behavioural criteria for the services to be discovered, as well as context characteristics of services and application environments.

In [62] a query language based on first-order logic that focuses on properties of behaviour signatures is used to support the discovery process.

Lately some works [82][83][84] used SPARQL [85] as a query language for the semantic discovery of web services. More generally, SPARQL is an RDF query language and it is a W3C Recommendation.

Summary

In summary, most of the proposed approaches support service discovery based on limited sets of service criteria and in reactive (pull) mode of query execution. Unlike them, SerDiQueL supports proactive dynamic service discovery based on a comprehensive set of service and application criteria including structural, functional, quality, and contextual characteristics. It also supports pull and push service discovery, resulting in more efficient service replacement during the execution of an application.

Furthermore, SerDiQueL uses fine grain quantification of similarities between queries and services based on distance measures and assumes service descriptions based on industry accepted standards (WSDL, BPEL).

Due to these reasons, SerDiQuel became a reasonable choice as a basis for developing support for handling security related criteria and ASSERTs as part of service discovery.
### Table 2.1: Summary of the single service discovery approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Algorithm</th>
<th>Service description language</th>
<th>QoS support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seekda [60]</td>
<td>Keyword-based</td>
<td>WSDL</td>
<td>No</td>
</tr>
<tr>
<td>Servicefinder [61]</td>
<td>Keyword-based</td>
<td>WSDL</td>
<td>No</td>
</tr>
<tr>
<td>OWLS-MX [49]</td>
<td>Semantic (logic-based and IR)</td>
<td>OWL-S</td>
<td>No</td>
</tr>
<tr>
<td>L. Li et al. [50]</td>
<td>Semantic (logic-based)</td>
<td>DAML-S</td>
<td>No</td>
</tr>
<tr>
<td>R. Mikhaiel et al. [51]</td>
<td>Graph-based (tree alignment)</td>
<td>BPEL</td>
<td>No</td>
</tr>
<tr>
<td>Z. Shen et al. [62]</td>
<td>Graph-based (RE-tree [86])</td>
<td>Behaviour signatures*</td>
<td>No</td>
</tr>
<tr>
<td>AWOS [45][63]</td>
<td>Graph-based</td>
<td>AOWSDL [87]</td>
<td>Yes</td>
</tr>
<tr>
<td>SerDiQueL [72][76]</td>
<td>Graph-based (morphism detection)</td>
<td>WSDL, BPEL and XMLs</td>
<td>Yes</td>
</tr>
<tr>
<td>S. Cuddy et al. [38]</td>
<td>Context graph-based</td>
<td>Not explained</td>
<td>No</td>
</tr>
<tr>
<td>C. Beeri et al. [34]</td>
<td>Context graph-based</td>
<td>BPEL</td>
<td>No</td>
</tr>
<tr>
<td>F. Bormann et al. [35]</td>
<td>Context</td>
<td>n/a – not on WS</td>
<td>No</td>
</tr>
<tr>
<td>Y. Ye et al. [68]</td>
<td>Context</td>
<td>n/a – not on WS</td>
<td>No</td>
</tr>
</tbody>
</table>

2.3 Discovery of Service Compositions

If the basic single service discovery fails to find the requested functionality, there is another way a discovery platform can try to fulfil the request: to compose an ad-hoc service on the fly by discovering and combining some web services that provide the different parts of the functionality.

This additional step in the discovery process can be realized with the aid of different approaches emanating from several areas of research (theoretical computer science, automated reasoning, algorithms and complexity, semantic and ontology, optimization of networks, etc.). This wide range of possibilities offers a lot of outcomes that can satisfy different types of discovery (static or dynamic, with human intervention or not), based mostly on which parts of the composition process can be automated. In particular we categorize the works in this section in two main groups:

- The ones that automate the translation of the service query into workflows/plans and the binding to concrete services, providing ways to fully automate the building of a service composition
- The ones that focus on automating the service discovery, adaptation and binding, when the workflow is already available.

---

1 n/a means “not applicable”. The service description languages marked with an * are new languages just introduced in the respective works.
Automated building of service composition

One of the objectives of the introduction of SOA was the need of dynamicity in applications nowadays: some common requests in the software development are fault-tolerance and adaptability to changes of the requirements or the environment.

The approach of automating the phase of building a new service composition is a step in the direction of dynamicity and it also answers to problems of complexity, response-time and scalability of a manual approach.

The problem is typically to construct a workflow or a plan that can satisfy the requirements and associate to it the appropriate services, given a set of web services’ descriptions.

An early work on this matter is SAHARA [1][2], a framework to compose services in a Wide-Area network, where the approach is not specific on web-services, rather it composes more generic data operators. The composition path is built by running the shortest path algorithm on the graph of the operator space. They propose to build domain-specific graphs and to cache popular results to limit the size of the graphs, but the solution isn’t scalable anyway in a more general context without the notion of local and wide-area paths. Furthermore the data operator point of view is a little restricting, not allowing for example a service to just retrieve information or to compose data from/to different services.

Another work from the same period is SWORD [3], a toolkit for efficient service composition. In this work a web service is represented as a rule, expressing that given certain inputs, the service will provide a particular output. These rules are expressed using Entity-Relationship assertions and are elaborated through a rule-based Expert System to generate plans, given the preconditions and postconditions of the requirement. They allow only simple queries, by not allowing arbitrary joins (like “find all pairs of movies with the same director”) and not providing arithmetic/function symbols, to maintain an efficient and simple model.

Most recent works, however, prefer to use standard languages to describe web services (and composition requests), in particular OWL-S (and DAML-S). The reasons are mostly business-related and include: (a) in this way developers don’t need to learn further (logic) languages, (b) it simplifies the process of integrating an existent service discovery platform and (c) to avoid the error-prone (manual) process of converting the service descriptions in another language.

A framework for the automated service composition is described in [4] and it uses the services’ DAML-S description (DAML-S is a predecessor of OWL-S). In particular, the approach of this work is to try to find a single service corresponding to the high-level goal requested by the user, in case this step fails then a repository of abstract workflows is interrogated. Only if also this
other step fails the framework tries to build a new composition, by chaining services through their input-output and precondition-effect (the matching of IOPEs is provided by a specific component that admits the composition of the I/O data from different services, allowing the creation of more complex compositions).

In [5][6] the DAML-S Service Profile of each service is converted in extralogical axioms of propositional Linear Logic. The service composition request is then specified as a Linear Logic sequent and the system uses a theorem prover to check if the request can be satisfy by a composition of web services. If a composition is possible, then a process calculus representation of it is generated from the proof and it is possible to request a flow model (DAML-S Service Profile or BPEL4WS). Non-functional properties, like security and QoS ones, are taken in consideration as well as the functional ones, thanks to inference rules.

CoSMoS [7][8] is a semantic-based model for services and compositions that is slightly different from OWL-S since it allows also semantic annotation of operations “concepts” (in addition of I/O) and these concepts aren’t data types. In this context they introduce SeGSeC: a service composition mechanism that supports CoSMoS (i.e. semantic annotations). In this work the services must be described in CoSMoS/WSDL and the service request can be written in natural language: the tool then translates it into a CoSMoS semantic graph representation. The composition starts with the discovery of the service for the initial concept in the request (the one that provides the goal output) and then goes on by finding the services that provide the inputs for the initial service, using also the semantic information. At the end of the composition process the workflow is checked to guarantee that it satisfies the semantic request; otherwise the tool tries to find other compositions.

One of the most recent works in the field is DynamiCoS [9][10], a framework for dynamic service composition that supports requests in natural language (but also in a formal language based on OWL) and functional and non-functional properties. The first step of the composition in this framework is the service discovery, based on semantic concepts. The semantic connections between the I/O of the discovered services are stored in a Casual Link Matrix (CLM); so then the composition is built starting from the requested output searching backwards for compatible services through the CLM.

The framework itself doesn’t include the service discovery component and the necessary interpreters to convert the service descriptions in the internal formalism (they claim that the approach can be applied with OWL-S Service Profile, WSMO Capability Model or SA-WSDL specification).
Automated service discovery in service compositions

Several works just focus on automating only a part of the whole discovery of service composition process. In particular an interesting topic in the service discovery area is, given a workflow, to find the most suitable services for the involved tasks or, in case no perfect matches are available, to adapt the workflow to consider the services that behave in a very similar way to the requested task. The “most suitable” unit of measure is usually the semantic correlation, but there are some works focusing on reaching the best QoS (after a first selection of services is done).

The earliest work on this subject is eFlow [11]: a platform for composition of e-services (the concept of e-service shares several characteristics with the web services). This platform offers means of describing the workflow of the services through the GUI (by defining flow graphs) or through a simple composition language (an XML language called CSDL: Composite Service Definition Language) that allows dynamic discovery of services or dynamic selection and instantiation (with possibilities of multiple instantiations) of services from a list. The discovery is obtained by executing generic XQL queries on the repository of the service descriptions: the platform allows whatever format for the descriptions as long as it’s based on XML. The obtained dynamic composition, however, isn’t guaranteed to be correct: the framework is built just to compose but it doesn’t make use of any verification tool.

Table 2.2: Summary of the automated building of service composition approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Algorithm</th>
<th>Service description language</th>
<th>Allowed WF patterns</th>
<th>QoS support</th>
<th>Security support</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAHARA [1][2]</td>
<td>Graph-based</td>
<td>n/a - not only on WS</td>
<td>Exclusive choice</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SWORD [3]</td>
<td>Logic-based (rule-based system)</td>
<td>Language* based on Entity Relationship model</td>
<td>Parallel split</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>S. Majithia et al. [4]</td>
<td>Backward chain of I/O</td>
<td>DAML-S</td>
<td>-</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>J. Rao [5][6]</td>
<td>Logic-based (theorem proving)</td>
<td>DAML-S</td>
<td>Exclusive choice and parallel split</td>
<td>Yes</td>
<td>As service goals and constraints</td>
</tr>
<tr>
<td>CoSMoS/SeGS eC [7][8]</td>
<td>Semantic graph-based</td>
<td>CoSMoS/WSDL* (semantic extension of WSDL)</td>
<td>Exclusive choice</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DynamiCoS [9][10]</td>
<td>Semantic graph-based</td>
<td>Internal language* (needs interpreters)</td>
<td>-</td>
<td>Yes / not explained</td>
<td>Yes / not explained</td>
</tr>
</tbody>
</table>

2 n/a means “not applicable”. The service description languages marked with an * are new languages just introduced in the respective works. In the allowed workflow patterns the sequence pattern isn’t listed, since it’s the basic one that every work provides.
An example of work that extensively uses semantic is [12], an ontology-based framework for the automatic composition. The desired workflow, with semantic annotations, is described through a language called CSSL (Composite Service Specification Language); then syntactic, semantic and qualitative composability rules are used to select the services for the composition. In particular the service WSDLs must be augmented with semantic properties from the DAML+OIL ontology presented in the paper. An interesting feature from this work is the introduction of three measures for the selection between the different resulting compositions, called ranking, relevance and completeness (in particular the first two measures are calculated on the basis of stored templates).

Another work on automatic composition based on ontologies is [13]. In this work the service request is defined with TWFO (Transactional WorkFlow Ontology), an ontology used to describe workflows with transaction support. The main difference with other works is that the registry must also contain the workflow of the described web services (expressed in TWFO). Then, after the candidate services are found through the DAML-S registry, the system tries to compose the workflow of each service in the requested workflow (called Master Workflow). The work doesn’t go too much into details on the discovery process and it doesn’t cope with the selection of a single composition at the end of the composition process.

Regarding the automatic service composition based on QoS criteria, it should be noted that this kind of approach needs, in addition to the workflow of the composite service, the list of the compatible services for each task as input. So, since a list of services has been already discovered, the matter to solve is reduced to just aggregate the different QoS data to find the best composition. A work in this area is [14], that uses some of the workflow patterns from [15] to define aggregation functions for QoS criteria. Basically they use the patterns to do a stepwise graph reduction, and for every step the aggregate value of the QoS criteria is calculated.

A niche work on QoS composition is SpiderNet [16][17]: in this work is described a framework for QoS assurance and load balancing of multimedia service compositions. The input of their tool is the composition of functionalities (a function graph) and a QoS requirement vector. Then the service composition is done through a bounded composition probing protocol: at each step a probe is sent from the actual service node to the most promising of its neighbours, to look for the next functions. Each node provides as a result the list of the service components that implement the desired functions and the statistical QoS (the assumption is that the nodes are cooperative and trustworthy). The results of this interesting work, however, are restricted to multimedia services (or few other exceptions) because the composition process is slow and focuses to provide long-living services. Furthermore, the algorithm is based on probing the network, so it’s supposed to be used only on bounded networks.
A more complete approach is given in METEOR-S [18][33], where the semantic and the QoS approaches are combined in a single automatic service composition framework. METEOR-S is more broadly a framework for the complete life-cycle of semantic web services, the particular component that deals with service composition is called MWSCF (METEOR-S Web Service Composition Framework).

The definition of the desired workflow is made through a specific GUI tool, where the user (service designer) should also associate each activity to a discovery URL. Then the framework ranks the services on two dimensions: the semantic matching and the QoS criteria matching. The service designer can specify the weights of each criteria to have control on the service selection process. Sadly, the framework isn’t able to automatically generate an executable but it needs some user intervention for the data binding.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Matching approach</th>
<th>Input format (workflow)</th>
<th>Service description</th>
<th>QoS support</th>
<th>Security support</th>
</tr>
</thead>
<tbody>
<tr>
<td>eFlow [11]</td>
<td>n/a</td>
<td>CSDL and XQL</td>
<td>XML</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>B. Medjahed et al. [12]</td>
<td>Syntactic and semantic logic-based</td>
<td>CSSL</td>
<td>WSDL with semantic in DAML+OIL</td>
<td>Just fees (extensible)</td>
<td>Privacy and encryption</td>
</tr>
<tr>
<td>J. Korhonen et al. [13]</td>
<td>Ontology-based reasoning</td>
<td>TWFO</td>
<td>DAML-S and TWFO</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>M. C. Jaeger et al. [14]</td>
<td>QoS aggregation (minimize function)</td>
<td>Workflow (+candidate services)</td>
<td>n/a</td>
<td>Yes</td>
<td>Encryption</td>
</tr>
<tr>
<td>SpiderNet [16][17]</td>
<td>Network probing</td>
<td>Function graph and QoS req. / not explained</td>
<td>WSDL / WSEL (QoS)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>METEOR-S [18][33]</td>
<td>Semantic and QoS ranking</td>
<td>BPEL-like, generated through a GUI</td>
<td>WSDL (with semantics) and WSEL (QoS)</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2.3*: Summary of the automated service discovery in service composition approaches.

2.4 Support for security

Supporting security in both service discovery and composition focuses on a number of related areas of security policy properties. This section describes the start-of-the-art literature and research in these areas. In particular we describe the start-of-the-art work in the area of security for the service discovery process, the awareness of service security properties for service discovery and service composition which is guided by security policies and their properties.

* n/a means “not applicable”. All the workflow languages in the Input format column are new languages just introduced in the respective works.
2.4.1 Security of discovery process

Security of the discovery process focuses on the mechanisms to secure the process and to assure that the process is not interfered or modified during the discovery steps of service selection. Most of the related work focuses on the protocols used during the discovery process and the architecture required for such protocols.

In [19] the authors describe a protocol called Splendor which is designed to enable a secure, private and location aware (mobile) service discovery mechanism. This includes security aspects of mutual authentication, service authorization, confidentiality, integrity, and non-repudiation. The authors’ component architecture describes a number of communicating components (illustrated in Figure 2.1) to uphold these security properties including; clients and directories, services and proxies, proxies and directories, clients and proxies, and clients and services. However, the actual security of the service discovery is mainly achieved by having service discovery proxies and centralized “trustworthy” third-party servers. Security policies are applied and changed at the proxy’s side, thus no policy synchronization is necessary at the service provider’s side. The main assumption in this work is that the proxies are already trusted and secure, thus the discovery process is effectively guarded from modifications through interception of messages.

![Splendor Architecture](image)

Figure 2.1: The Splendor Architecture for Secure Service Discovery Protocol

In [20] the authors describe a Secure Pervasive Discovery Protocol (SPDP) which aims at providing a solution for enumerating the services available in single hop ad-hoc networks. SPDP is a similar approach to Splendor but focuses on a simplified architecture (two components representing user and service agents), mainly to reduce the interactions required between network devices and thus save energy in ad-hoc networks (a goal of SPDP). SPDP also offers authentication, authorization, data integrity, confidentiality, and non-repudiation but through an anarchy Public Key Infrastructure (PKI). The authors claim that the benefit of their approach over others is that it is distributed (rather than relying on centralized trustworthy servers). To achieve this SPDP relies on the service providers providing their own
certificates as a Certificate Authority (CA). SPDP also specifically integrates a security model to uphold the security properties of the protocol and process.

2.4.2 Security aware service discovery

Assessing the security of a service in service discovery is also an important and related work for security aware service discovery. Existing research has focused largely on two sub-areas in this problem; the first being service discovery driven by some specific security or privacy properties and second, the trust and reputation aware service discovery mechanisms.

First, for service discovery driven by specific security properties, in [21] the authors describe an approach to Web Service discovery based on privacy preferences. The preferences are specified as part of privacy policies (architecturally placed with service descriptions in a central service repository). The privacy descriptions consist of a vocabulary for properties including terms for disclosure, openness and anonymity. The process of applying the privacy-aware policy for the web services is accomplished in several stages (illustrated in Figure 2.2). First, a client sends their preferences to a discovery agent. Then, a correspondence will be established between the user's interests and the web service privacy policies. Finally, the degree of user confidence to the privacy-aware policies on services is evaluated and a selection is made based upon the confidence levels obtained. The confidence values are evaluated by comparison of normalized preferences (for common measurements) and a weighting algorithm used to calculate the degree of privacy-aware confidence for services.

![Figure 2.2: A Process for Privacy Preferences Service Discovery and Selection](image)

Similarly, the work presented in [22] uses policies described in extended service descriptions for authorization and privacy for semantic web services. The descriptions are proposed ontologies to annotate OWL-S input and output parameters with respect to their security characteristics, including encryption.
and digital signatures. Several extensions to OWL-S are proposed in the form of objects. First an information object which itself is extended to support either encrypted information or signed information. The distinguishing differences between the sub-classes are the specific cryptographic algorithm that was used for signing or encrypting the specified object data. The approach also adds a series of policy types to OWL-S including a PrivacyPolicy, ConfidentialityPolicy and AuthorizationPolicy. The authors describe a design-time “best service selection” process based upon the use of the reasoning on policies within a capability-based Rei matching engine, called the MatchMaker. They also discuss how this may be used for run-time compliance checking, but allude to the difficulties of trusting what providers offer as descriptions and what the services they provide actually undertake in execution.

Second, on service discovery which includes awareness of trust and reputation, in [23] the authors describe an approach for dynamic service discovery considering reputation. They propose a reputation and endorsement system (RES) agency to facilitate the tasks of the service clients, along with Web Service Agent Proxies (WSAP) for each service client which communicate with other agents, accept requests from their users, and are autonomous. One particular feature of this approach is that WSAPs can monitor the activities and usages of the service by the client and help in future usages. A WSAP can also offer and obtain advice from other WSAPs. Specifically, a WSAP can also directly communicate with other WSAPs that participate in the RES agencies thereby allowing it to better realize the choices of its client. The distributed trust system in the approach consists of a set of principles, i.e., the parties involved either as service provider or requester. The principals engage with each other over a set of services. A rating of a service is a vector of attribute values. The reputation of a service aggregates the ratings of that service by other principals. An endorsement of a service by a principal is modelled as a Boolean scalar and a time limit on the validity of the endorsement. A typical usage scenario of this approach is illustrated in Figure 2.3. The authors also discuss some limitations of the approach, such as facing the challenges of fake reputation entries by impostor agents. Also there is little to stop a WSAP from unfairly setting bad ratings for a service that competes with the owner's own service and thus compromise the integrity of the reputation data.

Addressing some of the issues discussed in the previous work on reputation in services discovery, the authors in [24] propose a threat model focusing on the possible attacks to a reputation management system and a trust model that defines the trust of a web service using trust and penalty vectors. The threat model covers malicious behaviour, collusion and complex discovery attacks. Malicious behaviour includes providing different services to those which are advertised and providing false information about other providers and their services.
Collusion is discussed as a result of service usage, whereby consumer feedback may be low but due to a high-level of usage may indicate that the service provided is of a high (useful) nature. Also, providers may collaborate to enhance trust value by repeatedly boosting each other’s service quality values. Further more complex strategies may also involve service providers pretending to be good for a certain amount of time and garner reputation from consumers. Then the malicious service providers exploit this reputation to deceive consumers that request its services. The authors trust model is represented by both trust and penalty vectors whilst an overall trust value of a peer is evaluated by subtracting the penalty vector from the trust vector. Trust values are recorded for experience, authenticity and reputation. Penalty values are recorded for inconsistency (of feedback values) and misusage (behavioural fluctuation). The authors provide some experiments using a trust framework on some 4000 peers in a service system varying the effect of introducing trust and penalty values for service providers. The proposed trust model and framework provide a useful reference to experiment with trust and reputation properties.

2.4.3 Security aware service composition

Verification of security properties on service composition

Among the works focusing on security in service composition particular relevance has been given to verification, through model checking, of already existent compositions’ security. The service composition can be checked for flaws at design time or in a later stage of development, usually after concrete
services are associated with each task. To perform the check the composition is
modelled with formal languages and the requirements are expressed as
properties on the model.

The design time verification is applied on a specification of the system. To
encourage the use of this kind of verification, the language of the required
specification is conventionally a common language of the Software Engineering
area, usually UML.

Works meant for design time verification of security properties, like [25] and
[26], usually support the system definition in UML (or similar tools), since it is
a common language used in the Software Engineering area, encouraging in this
way the use of this kind of approach. In particular [26] has an unusual
approach with respect to the normal verification since they add the concept of
patterns. Basically, the first step in their approach is to express security design
patterns (i.e. design patterns of best practices to achieve some security goal) in
UML sequence diagram. These patterns are then converted into the formal
language CCS [27] through some rules. The model checking of the security
properties can be done, in the end, on compositions of these security patterns,
to verify if the security properties are preserved.

A more general work in the verification of security properties in service
composition is in [28][29]. They introduce a calculus (a typed extension of λ-
calculus) to describe and compose services. In particular their language can be
used to describe a model and check the security-related activities (access
events, e.g. writing a file, opening a socket connection) of a service composition.
The main remark on this work is that there’s no description of the modelling
phase, leaving to the reader the burden of planning how to convert the services
into their language.

Security aware service composition discovery

Another point of view regarding security in service composition is to obtain the
guarantee that a composition respects some security policies directly from the
discovery process, when an automatic composition of services approach is used.

A work that falls into this category is [30], where planning techniques are used
to compose workflows compliant with some lattice-based access control models
(e.g. multi-level secure systems). The focus is on how to find efficient
algorithms for workflow planning, even though in the limited case of sequences
of operators.

In [31] the authors describe an approach to security conscious web service
composition through the declaration of security constraints required on service
provision and of the constraints declared by service providers. Security
constraints are declared in the SAML [32] assertions. Examples are provided
for both authentication and authorisation assertions although a common
security ontology is not provided. The architecture of using the constraints specified is based upon a Web Service brokering model. A Secure WS-Broker (SWS-Broker) is used to manage service requests and sets of security constraints, and generate a secure workflow for service composition (based upon some pre-defined workflow patterns). Services are composed and selected based upon the request and the constraints specified. The approach also provides an implementation of the broker consisting of a workflow modeller, service locator, security matchmaker and WS-BPEL generator. The security matchmaker builds a tree structure of the path of security considerations (from the constraints applied to the workflow) and analyses the possible composition paths and security constraints from discovered services. WS-Agreement nodes are also generated as part of service message structures to express the constraints applied.

Furthermore, as mentioned in Table 2.2 and Table 2.3, some of the works in Section 2.3 allow the expression of few security properties in the request as well as non-functional properties. It should be noted that many of these approaches focus just on one or two types of security properties and the integration between high-level and low-level specifications (i.e. associations between a protocol name and a more general property) of the security properties isn’t taken into account. Generally, security properties are checked only against single services in the composition, not giving information on the overall security of the composition.

2.4.4 Securing interactions with services

Securing the exchange of messages (as interactions) between services requires both access control and message transport security. Since service-orientation is built upon a message-oriented model, the state-of-the-art in messaging is also considered. Here we outline some of the work that focuses on these security aspects for service interactions.

Access control

Access control to services has traditionally been associated with access to other system resources either through local system user authentication or a distributed access control server. More recently, access control mechanisms have also been supported by more flexible security policies and an architecture to support the decision making of access to resources using these policies. Standards organisations have also provided consortium agreed notations for policy languages around access control. One such language is XACML [73] is an OASIS standard that describes both a policy language implemented in XML and an access control decision request/response language implemented in XML. The policy language details general access control requirements, and has standard extension points for defining new functions, data types, combining logic, etc. The usage model for XACML is based upon a Policy Enforcement Point (PEP), which is responsible for protecting access to one or more resources. When a resource access (such as a service operation) is attempted, the PEP sends a description of the
ASSERT4Soa

attempted access to a Policy Decision Point (PDP) in the form of an authorization decision request. The PDP evaluates this request against its available policies and attributes and produces an authorization decision that is returned to the PEP. The PEP is responsible for enforcing the decision for access control purposes. Some of the short-comings of XACML are discussed in [74], where the authors highlight that XACML does not provide mechanisms to identify individual security actions and conflicts cannot be resolved across policies. Their approach leverages elements of the Semantic Web to represent role-based policies for accessing service operations. Their framework also supports entities using the policies to delegate rights, with restrictions attached, in a distributed style to other entities.

From a broader organisational perspective, the authors in [75] describe an approach to access control for cross-organisational (enterprise) service compositions. The authors argue that the main criteria for a flexible access control mechanism is to support interoperability in security infrastructure the security and access control models used by the respective service providers are independent from each other, and that dictating a common access control model is a non-suitable approach in a federation; hence domain specific access control models might be necessary. Their framework (illustrated in Figure 2.4) enables interaction security through tokens requested from the service provider and request handling by the provider against security policies. Tokens can be held by the requestor and are valid for a certain period of time.

![Figure 2.4: 2LAC Security Architecture for Access Control of Web Services](image)

**Encryption of Service Interactions**

Message encryption for services relies on both payload (the application data content of the message) and encryption on the network transport layer. The earliest form of computerized message encryption used a symmetric or secret key. All parties who wished to encrypt or decrypt data had to have access to the same key. An encryption algorithm called DES (later superseded by triple DES) became the standard. However, the key needs to be distributed to all the
assert4soa

parties wishing to encrypt and decrypt data. If the key is intercepted, a hostile party could decrypt the information. More recently, a hybrid approach of symmetric and public and private key encryption is used, since it doesn't require key distribution. However, it is much slower to actually encrypt data using most asymmetric mechanisms. More fine-grained service message encryption has been discussed in [75], where the authors describe an approach based upon Identity and Attribute based encryption. Identity based encryption (IBE) makes it possible for a user to encrypt a message or a document using the identity of its intended recipient as a public encryption key, and without the need for the public key certificate of that recipient. In a variant to this technique, an identity is a set of attributes instead of the name of an entity. Assuming each attribute has precise semantics and that all communicating parties share a common understanding regarding these attributes and their values, the identity based encryption mechanism can be used in an "attribute-based encryption" (ABE) fashion. Attributes of service interactions can be encrypted using a unique identifier of the user that requested the service. In order to avoid carrying extra information, the endpoint reference information contained in the ReplyTo of the message's header, is used as the identifier of the user. The authors discuss the strengths of the approach is the fine-grained encryption of sensitive information (rather than who payload) however, the weaknesses are largely around time to semantically reason on each attribute and filter those that need to be encrypted for interaction.

For encryption of the network transport the Transport Layer Security (TLS) is a protocol that ensures privacy between communicating applications and their users on the Internet. When a server and client communicate, TLS ensures that no third party may eavesdrop or tamper with any message. TLS is the successor to the Secure Sockets Layer (SSL). TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. The TLS Record Protocol provides connection security with some encryption method such as the Data Encryption Standard (DES). The TLS Record Protocol can also be used without encryption. The TLS Handshake Protocol allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before data is exchanged. Network layer encryption may be applied to sections of a network rather than end-to-end; in this case the network layer packets are encapsulated within IP packets. A major advantage of network layer encryption is that it doesn't need normally to be concerned with the details of the transmission medium. A feature of encryption up to and including the network layer is that it is generally transparent to the user. This means that users may be unaware of security breaches, and a single breach could have implications for many users (such as denial of service request). This is not the case for application layer encryption. As with link layer encryption, delays associated with encryption and decryption processes need to be kept to an acceptable level, but hardware-based devices capable of carrying out these processes have become increasingly available.
2.4.5 Summary

The work summarised previously separates the concerns of the support for security in service discovery, compositions and interactions. The current state-of-the-art practices in these areas assume that the specifications for security are specified in particular constraint notations (such as SAML, XACML, etc.) which is then used at various points in compliance checking to assure that services offer appropriate features to uphold constraint properties. Furthermore, a discovery and compliance architecture typically consists of some common components (such as a service registry, a compliance agent and a reasoner on property violations).
Chapter 3

A-SerDiQueL: The ASSERTs-aware version of SerDiQueL

3.1 Introduction

This chapter presents an overview of the Service Discovery Query Language (SerDiQueL) that we have adopted in ASSERT4SOA for the specification of service discovery queries and the amendments that we have introduced to this language in order to support the specification of service discovery criteria regarding ASSERTs.

3.2 Overview SerDiQueL

SerDiQueL is an XML-based language that allows expressing combinations of various characteristics of the services to be identified. More specifically this language supports the specification of structural, behavioural, quality, and contextual characteristics of services to be discovered. SerDiQueL has been developed to support both static and dynamic service discovery, as described in [40][71][76].

3.2.1 Overall query structure

Figure 3.1 shows an overview of XML schema of SerDiQueL (the full XML schema of SerDiQueL is given in Appendix A).

As shown in Figure 3.1, a query specified in SerDiQueL has a unique identifier, a name, one or more elements describing different parameters for a query, and three other elements representing the structural, behavioural, and constraint sub-queries. Each of these elements is described in the following sub sections. We will also use an example query (referred to as Query1 hereafter) to explain
the different elements of SerDiQueL. This example query is specified to identify a replacement for a *Global Positioning Service* (GPS) which is responsible to provide the location of a vehicle after receiving payment for the service.

![Figure 3.1: Overview Schema of SerDiQueL](image)

### 3.2.2 Query parameters

A parameter element is defined by a name and a value. The list of Parameter names that can be used in a query are:

- **(a) type** – This parameter determines whether the query should be used for dynamic service discovery or static service discovery. Acceptable value for this parameter is "dynamic" or "static".
- **(b) mode** – This parameter determines the mode of query execution. Acceptable value for this parameter is "PUSH" or "PULL".
- **(c) author** – This parameter signifies the author of the query.
- **(d) threshold** – This parameter determines the distance threshold for selecting the set of candidate replacement services.

Table 3.1 shows the different parameter and attribute values used in Query1. As shown in the table, the name of the query is *Query1* and it has the unique ID UUID:550e8400. The query is to be executed in push mode and at run time as indicated by the values of the “mode” and “type” parameters, respectively. The maximum acceptable distance of a service specification from the query is 0.5 as indicated by the value of the “threshold” parameter.

```
<ServiceQuery queryID="UUID:550e8400" name="Query1">
  <Parameter name="mode" value="PUSH"/>
  <Parameter name="type" value="dynamic"/>
  <Parameter name="threshold" value="0.5"/>
</ServiceQuery>
```

Table 3.1: Parameters in Query1
3.2.3 Structural discovery queries

The structural part of a query describes criteria regarding the interface of the service that is sought (aka “structural criteria”). This is normally the interface of the partner service that the query is associated with in a service based application. In SerDiQueL, the structural criteria of the sought service are expressed in WSDL (typically a query uses the WSDL specification of the service that is associated with the query and can be replaced by one of the services discovered through the execution of this query). The use of WSDL to represent structural aspects of a query in SerDiQueL is due to its wide use to describe service interfaces.

Table 3.2 shows structural sub query in Query1. As shown in the table, the structural sub-query is composed of the WSDL specification of GPS service. For simplicity, some parts of the WSDL are omitted.

```
<ServiceQuery queryID="UUID:550e8400" name="Query1">
  ....
</ServiceQuery>
<StructuralQuery>
  <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    targetNamespace="http://scube.org/gps">
    <wsdl:types>
      <xs:schema targetNamespace="http://common/xsd">
        <xs:complexType name="Location">
          <xs:sequence>
            <xs:element minOccurs="0" name="latitude" type="xs:int"/>
            <xs:element minOccurs="0" name="longitude" type="xs:int"/>
            <xs:element minOccurs="0" name="name" nillable="true"
              type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:schema>
    </wsdl:types>
    <wsdl:message name="logoutRequest">
      <wsdl:part element="ns:logout" name="parameters" />
    </wsdl:message>
    <wsdl:message name="logoutResponse">
      <wsdl:part element="ns:logoutResponse" name="parameters" />
    </wsdl:message>
    ....
    <wsdl:portType name="GPSService3PortType">
      <wsdl:operation name="logout">
        <wsdl:input message="axis2:logoutRequest" .../>
        <wsdl:output message="axis2:logoutResponse" .../>
      </wsdl:operation>
      <wsdl:operation name="acknowledge"...>
      <wsdl:operation name="getLocation"...>
      <wsdl:operation name="login"...>
      <wsdl:operation name="makePayment"...>
    </wsdl:portType>
  </wsdl:definitions>
</StructuralQuery>
```
3.2.4 Behavioural discovery queries

The behavioural part of a query supports the representation of behavioural aspects of required services (e.g., the order of the execution of operations, whether certain operations should be executed in all possible executions of a service etc.). The specification of the behavioural part of a query is based on temporal logic. SerDiQueL supports the representation of the main temporal logic operators as well as loops. More specifically, in SerDiQueL, a behavioural discovery query allows the description of conditions that verify (a) the existence of a certain functionality, or a sequence of functionalities, in a service specification; (b) the order in which certain functionalities should be executed by a service; (c) dependencies between functionalities; (d) pre-conditions; and (e) any iterations (loops) that may be required in the execution of certain functionalities.

![XML schema for the behavioural discovery query](image)

Behavioural conditions in a SerDiQueL query can be: (a) a single condition, or a negated condition, or a conjunction of conditions, (b) a sequence of expressions separated by logical operators, or (c) requires elements. This shown in the XML schema in Figure 3.2.
A behavioural discovery query also allows for the specification of requires elements. Requires elements define one or more service operations that need to exist in service specifications, represented as members (see element MemberDescription). These member elements are used in various conditions and expressions of a query. A member element has three attributes, namely (a) ID, indicating a unique identifier for the member within a query; (b) opName, specifying the name of an operation described in the structural sub-query, (for the case of dynamic service discovery, this attribute may also contain the port type for this operation for the WSDL description in the structural sub-query); and (c) synchronous, a Boolean attribute indicating if the service operation needs to be executed in a synchronous or asynchronous mode in the service.

A condition is defined as GuaranteedMember, OccursBefore, OccursAfter, Sequence, or Loop elements, as shown in Figure 3.3. A GuaranteedMember represents a member element (i.e., service operation) that needs to occur in all possible traces of execution in a service. This element is defined by attribute IDREF that references requires, sequence, or loop elements. The OccursBefore and OccursAfter elements represent the order of occurrence of two member elements (Member1 and Member2). They have two Boolean attributes, namely (a) immediate, specifying if the two members occur in sequence or if there can be other member elements in between them, and (b) guaranteed, specifying if the two members need to occur in all possible traces of execution in a service. A Sequence element defines two or more members that must occur in a service in the order represented in the sequence. It has an identifier attribute that can be used by the GuaranteedMember, OccursBefore, OccursAfter, Sequence, and Loop elements. A Loop element specifies a sequence of members that are executed several times if certain conditions are satisfied. It has a unique identifier (attribute ID) and is defined as a statement (element Body) that references other identified elements.
Figure 3.3: XML schema for behavioural condition

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

Table 3.3 shows an example of a behavioural sub query that appears in Query1. As shown in the table, the behavioural sub-query includes Requires elements expressing the requirement for the existence of the following operations in any replacement service:

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

In addition, as shown in the table:
(a) the operation `login` is defined as a GuaranteedMember element given that the user of the GPS service needs to be authenticated (i.e., login operation needs to occur in all possible paths of execution in the service);
(b) the operations `makePayment`, `getLocation` and `logout` need to be executed in this order and, therefore, they are defined in a Sequence element;
(c) the operation `login` should be executed before the sequence of operations in (b) specified in element OccursBefore.
<ServiceQuery queryID="UUID:550e8400" name="Query1">
  ....
  <BehaviourQuery>
  <Requires>
    <MemberDescription ID="login"
      opName="GPSService3PortType.login" synchronous="true"/>
    <MemberDescription ID="payment"
      opName="GPSService3PortType.makePayment" synchronous="true"/>
    <MemberDescription ID="location"
      opName="GPSService3PortType.getLocation" synchronous="true"/>
    <MemberDescription ID="logout"
      opName="GPSService3PortType.logout" synchronous="true"/>
  </Requires>
  <Expression><Condition>
    <GuaranteedMember IDREF="login"/>
  </Condition></Expression>
  <LogicalOperator operator="AND"/>
  <Expression><Condition>
    <Sequence ID="pay">
      <Member IDREF="payment"/>
      <Member IDREF="location"/>
      <Member IDREF="logout"/>
    </Sequence>
  </Condition>
  <Condition>
    <OccursBefore immediate="false" guaranteed="false">
      <Member1 IDREF="login"/>
      <Member2 IDREF="pay"/>
    </OccursBefore>
  </Condition>
  </Expression>
  </BehaviourQuery>
  ....
</ServiceQuery>

Table 3.3: Behavioural sub query in Query1

3.2.5 Query Constraints

The constraint sub-query describes different types of extra conditions that need to be fulfilled by a service. These extra conditions may be concerned with quality or context characteristics of the sought service. The quality characteristics of a service include non-behavioural and interface features such as the availability of a service, the performance levels that can be guaranteed for it etc. The context characteristics of the service are also non behavioural and non interface characteristics that change frequently and dynamically and which, therefore, cannot be represented by static elements in the description of a service but have to be obtained by invoking special operations associated with the service known as “context operations”.

D.2.1 - Asserts Aware Service Query Language and Discovery Engine 39/ 109
Depending on whether it refers to context or quality characteristics of a service, a query constraint is classified within a query as contextual or non-contextual. This classification is important since the non-contextual constraints in a sub-query can be evaluated against any type of service specification element (facet) in service registries, whilst contextual constraints are evaluated only against context facets. Furthermore, contextual constraints are the last constraints that are evaluated during the execution of a query to minimise the possibility of a change in the value of the relevant characteristic of the service prior to the completion of the service execution.

Figure 3.4 shows a graphical representation of SerDiQueL’s XML schema for specifying constraints. As shown in the figure, 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 AND and OR.

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. When the value of the contextual attribute is true, the query may contain ContextOperand elements. If the value is false, the query may contain NonContextOperand. The ContextOperand and NonContextOperand are mutually exclusive, i.e where one may occur, the other must not.
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) between two operands (operand1 and operand2). These operands can be non-contextual operands, contextual operands, constants, or arithmetic expressions.

As shown in Figure 3.5, a non-contextual operand (element NonContextOperand) has two attributes, namely (a) facetName, specifying the name of the service specification and (b) facetType, specifying the type of the service specifications to which the constraint will be evaluated. The operand contains an XPath expression indicating elements and attributes in the service specification referenced in facetName attribute. Therefore, the constraints can be specified against any element or attribute of any facet in the registries.

Figure 3.5: XML schema for relational operand

A contextual operand (element ContextOperand) specifies operations that will provide context information at runtime. More specifically, a contextual operand describes the semantic category of context operations instead of the signature of the operation represented by sub-element ContextCategory. This is due to the fact that context operations may have different signatures across different services. A contextual operand is defined by (a) attribute serviceOperationName, specifying the name of the service operation associated with the contextual operand, and (b) attribute serviceID, specifying the identifier of a service that provides the operation. The value of attribute serviceID is specified when the context operand provides the specification of a
context operation of a known service. This is normally the case when the context operation is associated with a service-based application for which the value of a context aspect of the application needs to be dynamically identified during the evaluation of a query (e.g., location of a mobile device application). In this case, attribute serviceID refers to the service-based application itself. Otherwise, the value of serviceID is specified as “any” (see Figure 3.5).

A ContextCategory element represents the semantic category of an operation, instead of its actual signature. As shown in Figure 3.6, a ContextCategory 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 specified in a query.

Arithmetic expressions define computations over the values of elements or attributes in service specification or context information. They are defined as a sequence of arithmetic operands or other nested arithmetic expressions connected by arithmetic operators. The arithmetic operators are: addition (plus), subtraction (minus), multiplication (multiply), and division (divide) operators. The operands can be contextual operands, non-contextual operands, constants, or functions, as shown in Figure 3.7.

![Figure 3.6: XML schema for semantic category](image)
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 also a contextual operand, a non-contextual operand, a constant, or an arithmetic expression. The currently supported functions are MIN and MAX, which choose the minimum or maximum value of those expressions supplied.

Table 3.4 shows behavioural sub query in Query1. As shown in the table, the first constraint sub-query (C1) is a soft non-contextual constraint specifies that the service to be identified needs to be available 12 hours a day. This constraint has a weight of 0.5 and is represented by the conditions that verify if the opening time hours specified in the facet QoS has a minimum value of 00:00 and a maximum value of 12:00. This is specified by a conjunction of two LogicalExpression elements with their respective XPath expression contents and constant sub-elements.

The second constraint sub-query (C2) is a soft contextual constraint concerned with the time to get response from the GPS service. This constraint specifies that any candidate service needs to have a context operation associated with operation getLocation() classified in the category RELATIVE_TIME in the ontology http://eg.org/CoDAMoS_Extended.xml, and the result of executing this operation has to be less than 10 seconds for this service to be considered.

```xml
<ServiceQuery queryID="UUID:550e8400" name="Query1">
   ....
   <ConstraintQuery name="C1" type="SOFT" contextual="false" weight="0.5">
      <LogicalExpression>
         <Condition relation="EQUAL-TO">
            <Operand1>
               <NonContextOperand facetName="QoS" facetType="QoS">
                  //QoSCharacteristic[Name='Availability']/Metrics/Metric[Name='OpenTime'][Unit='Hours']/MinValue
               </NonContextOperand>
            </Operand1>
            <Operand2>
               <Constant type="STRING">00:00</Constant>
            </Operand2>
         </Condition>
      </LogicalExpression>
   </ConstraintQuery>
</ServiceQuery>
```
Table 3.4: Constraint sub queries in Query1
3.3 A-SerDiQueL

3.3.1 Extensions

In this section, we summarize the differences of the new version of SerDiQueL, called A-SerDiQueL, from the previous one. The queries written in the old language aren't compatible with the new schema of A-SerDiQueL, but the discovery engine that we have developed to support A-SerDiQueL is able to support both versions and so it is backward compatible.

Figure 3.8 presents the overall XML Schema of A-SerDiQueL. The main structure of a query is unchanged, i.e., a query is still partitioned into structural, behavioural and constraints parts and includes a specification of parameters that determine general aspects of its execution (e.g. execution in PUSH or PULL mode).

In A-SerDiQueL queries, however, in addition to the above types of subqueries we can have conditions related to security certificates expressed as ASSERTs (AssertQuery element). Furthermore, the set of query parameters has been extended, the way of expressing these parameters has been changed, and there is an additional query element that determines how the services identified following the execution of the query should be ordered (OrderBy element).

![Figure 3.8: Overview schema of A-SerDiQueL](image)

### Parameters

The main reason of the need for a new way of expressing query parameters in A-SerDiQueL is because it is necessary in order to be able to validate the correctness of the specification of the parameter values of a query during parsing and to support the specification of structured parameters. In the previous version of SerDiQueL the parameters were elements with a name and a value attributes, as shown in the example in Table 3.5.

---
Hence, since the possible parameters and values were not listed in the schema of SerDiQueL, the XML parser of the earlier version of the language could not validate the parameter values statically, postponing the check until the query evaluation phase and preventing any check at the client side (i.e., the client of the discovery engine).

To address these limitations, in A-SerDiQueL the parameters are specific sub-elements of the element Parameters.

This allows the validation of the parameters and their values through the XML parser and the expression of structured parameters. More specifically, as shown in Figure 3.9, the possible parameters in A-SerDiQueL query are:

- **Mode**: like the parameter from the previous SerDiQueL, it is required and its possible values are PUSH and PULL
- **Type**: like the parameter from the previous SerDiQueL, it is required and its possible values are dynamic and static
- **Threshold**: like the parameter from the previous SerDiQueL, it’s not required and its possible values range between 0 and 1 (0 excluded) with two fractional digits. The default value, if left out, is 0.5

![Figure 3.9: XML Schema for the parameters of the query](image)

- **Composition**: New optional parameter that is used to express whether the discovery engine should search for composition of services that satisfy the query if a single service isn’t found. If the parameter isn’t set the tool should take it as false. The additional optional attribute patternsRef is used to set the source of the composition patterns and its possible values are URIs.
- **Registry**: New optional element that acts as a container for parameters that refer to the security of the interactions between the discovery engine.

```xml
<Parameter name="mode" value="PULL" />
<Parameter name="type" value="dynamic" />
<Parameter name="threshold" value="1.0" />
```

Table 3.5: Example of SerDiQueL parameters
engine and the external service registries that this engine should contact during the service discovery process. The parameters that can be specified within this container element are:

- The parameters `QueryConfidentiality` and `ResultsConfidentiality`. These two parameters allow to express whether the transmission of any query criteria that will need to be transmitted to service registries during the discovery process must preserve confidentiality (`QueryConfidentiality`) and if the results to be transmitted back from a registry to the discovery engine during the query execution process are also required to be confidential (`ResultsConfidentiality`). More in detail this means that the transmission of those data (query and/or results) should be confidential, i.e. only the discovery engine and the external registries can have the knowledge of the data. The two parameters take Boolean values indicating whether confidentiality is required (when the relevant parameter value is set to true) or not (when the relevant parameter value is set to false). If the values of these parameters aren’t set the discovery engine should assume false as the default value. The additional optional attribute `excludeNonconforming` is a Boolean attribute that is used to determine whether registries which do not conform to the confidentiality conditions of the query should be contacted or otherwise during the query execution process. The value true indicates that non-conforming registries should be excluded. The default value of this attribute is true.

- The parameter `Authentication` allows to express if the registries must be authenticated. If the parameter isn’t set the tool should take it as false.

An example of the specification of query parameters in A-SerDiQueL is given in Table 3.6. In particular the parameters Mode, Type and Threshold tell the system that the query is to be executed in pull mode and at run time, with a maximum acceptable distance for a service specification from the query of 0.5. In case no single services are discovered, the query allows service composition through the patterns that can be found at http://assert4soa.eu/patterns, thanks to `Composition` parameter.

Any communication with the registries, the query and the returning results, should be confidential and authenticated, thanks to the parameters in the `Registry` element. In particular if a registry isn’t compliant with the query confidentiality and the authentication is excluded from the search; the same thing doesn’t apply for the confidentiality of the results since the attribute `excludeNonconforming` of the element `ResultsConfidentiality` is set to false.

---

4 The parameters `QueryConfidentiality` and `ResultsConfidentiality` are relevant only if a part of the query is supported and sent to the external service registries (see also the description of `externalExecution` at later in this section)
The new element AssertQuery (Figure 3.10) is used to define constraints regarding the certificate and the security properties to be satisfied.

This element has a similar specification to ConstraintQuery elements. Its main differences from the latter elements are:

- The ASSERT constraints are always non-contextual, so the element doesn’t need a contextual attribute.
- In the AssertQuery, it is possible to specify if the matchmaker to be used should be a specific one, based on the certificate type, or the generic one. This is obtained through the optional attribute matchmakers: the possible values are basic, if only the generic matchmaker must be used, specialized, if the specific ASSERT-related matchmaker must be used (resulting in an error if the matchmaker isn’t available) or available if the best matchmaker available is used (if a specialized matchmaker isn’t available then the basic one is used, this is the default).
- In the LogicalExpression of an ASSERT constraint is possible to specify a security property instead of a condition on the certificate. This is obtained through the element AssertProperty.
- Finally the element AssertOperand is used instead of the ContextualOperand and NonContextOperand to define a RelationalOperand on the ASSERT.
The following example shows the specification of certificate related constraints in a query. In particular, the constraint sub-query is a soft constraint specifying that the service to be discovered needs to have an ASSERT where the name of the issuer is Fraunhofer_SIT.
Execution

The definition of all the different types of constraints in a A-SeRDiQueL query has been extended to allow the provider of the query to express whether the execution (matching) of the relevant constraint should be done by the discovery engine internally or the constraint should be transformed into the query language of an external registry and be executed there (if supported). As shown in Figure 3.11, this is allowed by the introduction of a new optional Boolean attribute `externalExecution` in the element `ConstraintQuery`. The default value of this attribute (i.e., `false`) indicates that constraint should be checked by the discovery engine after the description of a service has been fetched from an external registry. The use of the attribute `externalExecution` enables a form of control that has implications for performance and security. More specifically, the local execution of constraints in the discovery school prevents the exposure of security related discovery criteria to external registries if necessary. However, as a consequence of preventing such execution a larger set of service descriptions, which might not eventually fit with the criteria of a query, may have to be transmitted from external registries to the discovery engine.

---

5 Note that this control is different from the controls regarding the confidentiality of transmission of data to and from the registry since even if the transmission of discovery criteria to a registry is confidential, at the registry side the criteria will have to be decrypted in order to enable matching.
The example of the following table demonstrates the use of the external execution attribute. More specifically, in the shown example this attribute is appropriate since the criteria is just looking for the starting point of availability of the service, so no secure information is transmitted to the registries.

```xml
<ConstraintQuery name="C2" type="SOFT" contextual="false" weight="0.5" externalExecution="true">
  <LogicalExpression>
    <Condition relation="EQUAL-TO">
      <Operand1>
        <NonContextOperand facetName="QoS" facetType="QoS">
          //QoSCharacteristic[Name="Availability"]/Metrics/Metric[Name="OpenTime"][Unit="Hours"]/MinValue
        </NonContextOperand>
      </Operand1>
      <Operand2>
        <Constant type="STRING">00:00</Constant>
      </Operand2>
    </Condition>
  </LogicalExpression>
</ConstraintQuery>
```

Table 3.8: Example of A-SerDiQueL constraint using the externalExecution attribute

In the example in Table 3.7, instead, the parameter externalExecution could be not appropriate since we could want to keep private the fact that we are looking only for particular trusted certificates.

As stated before, the general parameters QueryConfidentiality and ResultsConfidentiality are relevant only if at least one constraint is required to be executed externally. This means that, depending on these attributes, the matching of a constraint can require:
Table 3.9: Summary of the possible execution modes of the constraints

Constraints preferences

To be able to sort the results by some soft constraints, all the ConstraintQuery and the AssertQuery must be named with a unique ID (another difference compared to the previous SerDiQueL, where the name was a simple string that could be repeated). Then it is possible to define an OrderBy element where specify the reference of the constraints in the required order in its sub-elements ConstraintRef (Figure 3.8).

In particular in the following example the query expresses that the order of the results should be firstly decided by soft constraint C1, and then C2 (the soft constraint are described in Table 3.4).

Table 3.10: Example of an ordering preference

3.3.2 Example queries

Table 3.11 shows an example query expressed in A-SerDiQueL. As introduced in Section 3.2.1 this example query is specified to identify a replacement for a Global Positioning Service (GPS) which is responsible to provide the location of a vehicle after receiving payment for the service. The example query contains one structural sub query, one behavioural sub query, one non-contextual constraint (C1), one contextual constraint (C2), and two soft ASSERT queries (A1 and A2). The parameters, structural sub query, behavioural sub query, and constraints (C1 and C2) of this example query have been explained in sections 3.2.2, 3.2.3, 3.2.4 and 3.2.5 respectively. In this section we explain the ASSERT sub queries. The complete query in XML is presented in Appendix B.

The first ASSERT constraint sub-query (A1) is a soft constraint specifies that the service to be identified needs to have an ASSERT where the name of the ASSERT issuer is Fraunhofer_SIT.

The second ASSERT constraint sub-query (A2) is a soft constraint specifies that the service to be identified needs to have an ASSERT where the type of the ASSERT should be M.
It should be noted that the example query does not contain any ASSERT constraint sub-query related to the ASSERT property. This is because the schema to specify ASSERT properties in Assert4Soa project is scheduled to be finalized later in the project.

```xml
<?xml version="1.0" encoding="utf-8"?>
<tns:ServiceQuery xmlns:tns="http://assert4soa.eu/schema/SerDiQueL-v2"
  xmlns:par="http://assert4soa.eu/schema/Parameters"
  xmlns:tnsd="http://assert4soa.eu/schema/Assert_SQL"
  xmlns:tnsa="http://assert4soa.eu/schema/Constraint_SQL"
  xmlns:tnsb="http://gredia.eu/schema/Behavour_SQL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://assert4soa.eu/schema/SerDiQueL-v2 ./SerDiQueL-v2.xsd"
  queryID="UUID:550e8400-e29b-41d4-a716-446655440000" name="Query1">
  <par:Parameters>
    ...
  </par:Parameters>

  <!-- Structural sub-query -->
  <tns:StructuralQuery>...
  </tns:StructuralQuery>

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

  <!-- Constraints sub-queries -->
  <tnsa:ConstraintQuery name="C1" type="SOFT"
    contextual="false" weight="0.5" externalExecution="true">
    ...
  </tnsa:ConstraintQuery>

  <tnsa:ConstraintQuery name="C2" contextual="true"
    type="SOFT" weight="0.5">
    ...
  </tnsa:ConstraintQuery>

  <tnsd:AssertQuery name="A1" type="SOFT">
    <tnsd:LogicalExpression>
      <tnsd:Condition relation="EQUAL-TO">
        <tnsd:Operand1>
          <tnsd:AssertOperand facetName="Assert"
            facetType="Assert"/>
        </tnsd:Operand1>
        <tnsd:Operand2>
          <tnsd:Constant type="STRING">
            Fraunhofer_SIT
          </tnsd:Constant>
        </tnsd:Operand2>
      </tnsd:Condition>
    </tnsd:LogicalExpression>
  </tnsd:AssertQuery>

  <tnsd:AssertQuery name="A2" type="HARD" externalExecution="true">
    ...
  </tnsd:AssertQuery>
</tns:ServiceQuery>
```
Table 3.11: The listing of the example query Query1
Chapter 4

A-SerDiQueL Query Engine

4.1 Overall discovery process

The overall discovery process envisaged for the ASSERT4SOA framework is shown in the activity diagram of Figure 4.1.

As pointed out earlier, the ASSERT4SOA framework focuses on support for service discovery at runtime and, therefore, the process shown in Figure 4.1 refers to discovery whilst a service-based application is in operation. Also, it covers – at a high, non-algorithmic level – not only the discovery of single services but also the discovery of service compositions. This is necessary for clarifying the relation between the two alternative forms of discovery that the fully developed version of the ASSERT4SOA framework will support. It should be
noted, however, that service composition is not supported by the discovery engine that is the main focus of this report.

As shown in Figure 4.1, the discovery process starts when the Assert4Soa Discovery Engine receives a query that is to be used for discovering possible alternative services for one of the partner services of a service based application. This query is expressed in A-SerDiQueL. Following the parsing of the query, the parts of it which refer to ASSERTs or, equivalently, the security related discovery criteria of the query (referred to as “Ce query” in the figure) are separated from the parts referring to other functional and quality discovery criteria (referred to as N query, i.e., normal query). This distinction is necessary as the part of the query that refers to ASSERT related discovery criteria is used in order to identify the composition patterns that could be applied in identifying service compositions that can ensure these criteria (see the activity IdentifyCPatterns in the process).

Subsequently the N part of the query and the composition patterns are either sent to the discovery engine for an one-off execution if the execution mode of the query is PULL or are subscribed to it for multiple executions if the execution mode of the query is PUSH. In PUSH mode, multiple executions may be triggered by changes in the descriptions of services already identified as possible matches with a query or due to the emergence of new services that can fit with the query. In both the PUSH and the PULL mode of query execution, the discovery engine executes the received query at least once and returns any services and service compositions that match the discovery criteria of the query (see the activity Execute N-Query/Cpatterns). Any services and/or service compositions that are found to match with the discovery criteria of the query (and the composition patterns in the case of compositions) at this stage are used to update a Candidate Service Set. This set is used as a cache of replacement services for the partner service that was associated with the query in the first place and any subsequent service replacement request will retrieve the first service from this set. In the case of candidate service compositions we envisage that the Assert4Soa framework will generate a virtual service pointer that will be used to represent the composition identified by the framework and can be used by client applications to make use of the composition through the framework. Note, however, that this is a matter that will be resolved later in the development of the framework.

It should also be noted that following the initial formation of the Candidate Service Set there is no guarantee that its member services satisfy indeed the security criteria of the query. Hence the initial formation of the Candidate Service Set is followed by checking that the ASSERT related criteria of the query are satisfied by the candidate services/service compositions in it (see the activity Check Certificates of Service Compositions in RS) and ordering the elements that satisfy these criteria in descending order of the degree of match that they have with these criteria (see the activity OrderRS wrt Ce). Following these two stages in the overall process, any candidate service/service
composition that does satisfy the ASSERT related criteria is removed from the Candidate Service Set and a new discovery process is initiated in order to try to identify further services that could meet first the non security criteria of the query and then the security related criteria.

Certain parts of the overall discovery process described in Figure 4.1 can be also triggered by events other than the request for the execution of a query. More specifically,

- service replacement requests lead to the selection of the first service in the Candidate Service Set in order to bind and use it in the service based application,
- the publication of new certificates (ASSERTs) for one of the services in the candidate service set will trigger the re-evaluation of the ASSERT related criteria for a candidate set that has been built for a query executed in PUSH mode and the possible re-ordering of this set, and
- changes in the descriptions of services in the deployed service registry(ies) or the publication of new services in them can lead to the execution of the non ASSERT related parts of queries executed in the PUSH mode in the first place and potentially the re-execution of the ASSERT related parts if the candidate services set of the query would need to be altered given the results produced by the non ASSERT related part of the query.

The realisation of the activities of the discovery process shown in Figure 4.1 is based on the discovery engine that we describe in the following.

4.2 Architecture

The overall architecture of the ASSERT4SoA framework is shown in Figure 4.2. As shown in this figure the Discovery Engine is composed of four components (shown in the dotted rectangle), namely: (i) Query Handler, (ii) Discovery Manager, (iii) Matchmaking Subsystem and (iv) Composition Manager.

The Query Handler is responsible to receive the complex query through the EntryPoint. It parses and checks the validity of the received query, and passes it to the service discovery manager to execute the query (see the activity Extract N/Ce query and the transition “Exists Non Executed N Query” in the process of Figure 4.1). It also manages the subscription of queries (see Section 4.4 for detail). The interface provided by the Query Handler is specified in the next section. The Discovery Engine supports query execution in pull and push mode. In pull mode of query execution the query handler the discovery manager to locate a list of replacement services for service specified in the query. The

---

6 The reason for re-attempting to find services/service compositions meeting the criteria of the query is because there is a possibility that new services might have been published in the service registry meanwhile.
push mode of query execution is performed when a service based system (SBS) is running and a service needs to be replaced due to (a) unavailability, or malfunctioning of services participating in the SBS, (b) changes in the structure, functionality, quality, context or security certificate of services participating in the SBS, (c) changes in the context of the SBS 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. In either mode of query execution the Query Handler maintains a buffer of the best N ranked services returned by the Discover Manager (see Candidate Service Set in the process of Figure 4.1). If the number of services in the buffer falls below a specific threshold, the Query Handler instructs the Discovery Manager to re-executes the query.

Figure 4.2: Overall Architecture of ASSERT4SOA Framework

The Discovery Manager orchestrates the functionality offered by the other components involved in the service discovery process. This manager receives complex queries from the Query Handler. It also contacts the registry abstraction layer to retrieve the available services to be considered for matchmaking and passes the query along with the retrieved service descriptions to the Matchmaking Subsystem to execute the queries. If the matchmaking subsystem does not return any candidate service, the Discovery Manager builds new queries based on the patterns of service composition (if composition is allowed by the query) and sends the new queries to the composition manager to find composite services.
The Matchmaking Subsystem is responsible to evaluate the different criteria of a query (e.g. structural, behavioural, contextual, QoS and security related criteria) against a list of services and to compute the distance between a query and candidate. More specifically, it executes a complex query by delegating the execution of different query parts to appropriate and specialised match makers (e.g. structural matcher, behavioural matcher etc.), synthesises the results returned by these match makers, and ranks these results in descending order of the degree of their fit with the overall query.

The Composition Manager is responsible to receive a query along with composition patterns and build up compositions of services by applying the composition patterns that relate to the discovery query (i.e. could be used given the security discovery criteria specified in the query). However it should be noted that the Composition Manager component is still in preliminary design phase and the current implementation of the discovery engine does not support discovery of composite service. An extended version of the discover engine that supports composite service discovery will be delivered in month 18th of the project.

4.3 Programmatic interface of query engine

Figure 4.3 shows the interfaces exposed by different components of the discovery engine.

The functionality of the Query Handler component is exposed by the interface IQueryHandler. This interface has the following method:

- **sendRequest**: This method accepts the string representation of an XML query (parameter query in Figure 4.3) and an URL (parameter callBackURL in Figure 4.3) that signifies where to send the results of PUSH mode query execution. In case of PULL mode query execution the parameter callBackURL is ignored by the method. This method a list of service specifications that match with the query (parameter CandidateService[] in Figure 4.3). Sequence diagrams presented in Figure 4.8 and Figure 4.9 describe the sequence of activities performed by this method.

- **handleEvent**: This method accepts an event that signifies (a) unavailability, or malfunctioning of a service in the registry, (b) changes in the structure, functionality, quality, context or security certificate of a service in the registry, or (c) availability of a new service.

The functionality of the Discovery Manager component is exposed by the interface IDiscoveryManager. This interface has the following methods:
• **discoverServices**: This method accepts an object representation of a pull mode query (parameter `query` in Figure 4.3) and returns a list of service specifications that match with the query (parameter `CandidateService[]` in Figure 4.3). Sequence diagram presented in Figure 4.10 describes the sequence of activities performed by this method.

• **discoverSingleService**: This method accepts an object representation of a query (parameter `query` in Figure 4.3) and a service ID. This method returns the service if it matches with the query. This method is invoked by the **Query Handler** in case of push mode query execution. Sequence diagram presented in Figure 4.11 describes the sequence of activities performed by this method.

The functionality of the **Matchmaking Subsystem** component is exposed by the interface **IMasterMatchMaker**. This interface has the following methods:

• **matchServices** : This method accepts object representation of a query (parameter `query` in Figure 4.3) and a list of service specifications (parameter `services` in Figure 4.3). This method returns a list of service specifications that match with the query (parameter `CandidateService[]` in Figure 4.3). Sequence diagram presented in Figure 4.12 describes the sequence of activities performed by this method in detail.

---

**Figure 4.3: Interfaces of Discovery Engine components**
The functionality of Composition Manager component is exposed by the interface ICompositionManager. This interface has the following methods,

- `doComposition`: This method accepts object representation of a query (parameter `query` in Figure 4.3) and composition patterns (parameter `patterns` in Figure 4.3). This method returns a list of candidate services that satisfy the query (parameter `CandidateService[]` in Figure 4.3).

### 4.4 Detailed design

Figure 4.4 shows the classes that implement the functionality of the Query Handler component. These classes are: `QueryHandler`, `QueryParser`, and `PubSubEngine`.

The `QueryHandler` class implements the interface `IQueryHandler`.

The class `QueryParser` parses XML queries by delegating different parts of the query to different types of parsers to parse the query. In particular, `AssertQueryParser` parses the ASSERT related constraints, `BehaviouralQueryParser` parses the behavioural discovery query and `ConstraintQueryParser` parses the QoS constraints in the XML query. Following parsing, `QueryParser` produces the object representation (presented by the `Query` class) of the query. Different specialized parsers are used to parse different parts of the query in order to support extensibility of the query parser.

The `PubSubEngine` class facilitates the management of the query subscriptions. The method `subscribeQuery` of this class accepts a query and returns the unique subscription ID of the query. This class is also responsible to notify the `QueryHandler` with information about a new service that becomes available or about changes in the specification of existing services in the registry.
Figure 4.4: Classes for Query Handler Component

Figure 4.5 shows the classes that implement the functionality of the Discovery Manager component. The DiscoveryManager class implements the interface IDiscoveryManager.

The RegistryAdapter class of the Discovery Manager component retrieves the service descriptions from the service registry and produces an object representation of each service (these representations are constructed as instances of the CandidateService class). The RegistryAdapter class offers its functionality through two methods. The retrieveServices method retrieves all the available services in the registry and the retrieveServiceByIds accepts a list of IDs retrieves only those services from the registry that match the specified IDs.
Figure 4.5: Classes for DiscoveryManager Component

Figure 4.6: Classes for Matchmaking Subsystem Component

The MasterMatchMaker class delegates different classes to match different parts of the query with the service specification. The class MatchMaker is an abstract class with an abstract method doMatchMaking. This method accepts an object representation of a query and a list of service specifications. The method doMatchMaking returns a list of service specifications that match with the query. All the specific matchmakers that may be used in the framework must implement this abstract class in a way that realises the matching needed by the specific part of the query delegated to them. For example, the StructuralMatchMaker class matches the structural part of a query with the structural specifications of services, BehaviouralMatchMaker class matches the behavioural part of a query with the behavioural specifications of services, and ConstraintMatchMaker class matches the constraint query with the respective service specifications. CommonAssertMatchMaker matches the ASSERT queries.
that are related to common modules for all types of certificates. Specific type ASSERT match maker class (e.g. ASSERT-E-MatchMaker, ASSERT-O-MatchMaker, ASSERT-M-MatchMaker) matches the ASSERT queries that apply to specific type of ASSERT.

Figure 4.7 shows the main classes that implement the functionality of the Composition Manager component. The CompositionManager class implements the interface ICompositionManager.

![Diagram](image)

Figure 4.7: Main Classes for Composition Manager Component

### 4.5 Main functional features

The sequence diagram in Figure 4.8 shows the typical interaction of different classes with the Query Handler in pull mode.
In step 1, the QueryHandler receives the string representation of the XML query from the EntryPoint. In step 2, the QueryHandler checks the validity of the submitted query by invoking QueryParser. If the submitted query is valid then the QueryHandler sends the query to the Discovery Manager to find service that matches the query (step 3).

Figure 4.9 shows the typical interaction of different classes with the Query Handler in push mode.

In step 1, the QueryHandler receives the string representation of the XML query from the EntryPoint. In step 2, the QueryHandler checks the validity of the submitted query by invoking QueryParser. If the submitted query is valid
then the QueryHandler creates a subscription for the query by invoking Pub-Sub Engine (step 3) and sends the query to the Discovery Manager to find service that matches the query (step 4). Steps 5 and 6 shows typical interactions that may arise in push made query execution due to (a) unavailability, or malfunctioning of a service in the registry, (b) changes in the structure, functionality, quality, context or security certificate of a service in the registry, or (c) availability of a new service. More specifically, in step 5 the QueryHandler receives an event that signifies any of the cases listed above. In step 6, QueryHandler invokes the DiscoveryManager to find if the relevant service satisfies the query. If the service returned by DiscoveryManager appears to be the best alternative service then the QueryHandler may notify the NotificationListener associated with the EntryPoint about this new best service.

Figure 4.10 shows the interaction of the Discovery Manager component with different classes in pull mode of query execution.

In step 1 the DiscoveryManager receives a query from the QueryHandler. In step 2, the DiscoveryManager invokes the RegistryAdapter to retrieve the available service descriptions from the registry. In step 3, the DiscoveryManager passes the retrieved service descriptions and the discovery query to the MasterMatchMaker in order to identify the services that match the criteria described in the query. If the MasterMatchMaker fails to identify any service (i.e. no service matches the query), then in step 4, the DiscoveryManager invokes the CompositionManager to identify composition of services that matches the query. If a composition is found then the DiscoveryManager invokes the MasterMatchMaker to verify the services in the composition against the query.
Figure 4.11 shows the interaction of the Discovery Manager component with different classes in push mode of query execution.

In step 1 the DiscoveryManager receives the query and the ID of a service that should be matched with the query. In step 2, the discovery manager retrieves the service from the registry signified by the service ID in step 1. In step 3, invokes the MasterMatchMaker to find if the relevant service satisfies the query.

The sequence diagram in Figure 4.12 shows the interaction of Matchmaking Subsystem component with different classes.

In step 1, the MasterMatchMaker receives a query and a list of service specifications from the DiscoveryManager. In step 2, the MasterMatchMaker invokes the ConstraintMatchMaker to filter out the services that fail to satisfy the hard constraints in the query. In step 3, the MasterMatchMaker invokes the CommonAssertMatchMaker to filter out the services that fail to satisfy the common hard ASSERT constraints in the query. In step 4, the MasterMatchMaker invokes the StructuralMatchMaker to identify the services that satisfy the structural criteria in the query. If the query contains any behavioural constraint then in step 5, the MasterMatchMaker invokes the BehaviouralMatchMaker to identify the services that satisfy the behavioural criteria in the query. If the query contains any soft constraint then in step 6, the MasterMatchMaker invokes the ConstraintMatchMaker to identify the services that satisfy the soft constraints in the query. In step 7, the MasterMatchMaker invokes the CommonAssertMatchMaker to identify the services that satisfy the common soft ASSERT constraints in the query. In the next steps, the MasterMatchMaker invokes type specific match makers to identify services that satisfy the type specific ASSERT queries. For simplicity
in the above diagram we only show that the MasterMatchMaker invokes the ASSERT-M-MatchMaker.

Figure 4.12: Typical interaction of different classes with Matchmaking Subsystem component.

4.6 Security mechanisms

The discovery process is at the heart of the whole Assert4Soa framework. It is therefore essential to provide adequate security to this process, since a failure in it may affect the security of the whole Assert4Soa scheme.

As stated in D6.1 [79] the main threat that the Assert4Soa framework faces is that malicious users can force the selection of a service that is not the best with regards to the needs of the client. This is indeed closely related to the discovery process. When a client requests a service it sends a query to the Assert4Soa framework, the framework has to provide a reply indicating the services that are available and in particular, those that fulfill the security requirements specified in the query. If the process is not correct or fair, the client might be prevented to choose the service that best fitted its needs. Likewise, the process is subject to other threats like those related to the privacy of the information contained in queries, etc.

The main security requirements of the discovery process and the proposed security measures are summarized below. Both, requirements and proposed solutions will be revised later in the project, when the advances in the definition of the Assert4Soa framework and its components are detailed enough to ensure that the proposed solutions are (i) appropriate for the requirements; and (ii) implementable in practice, respecting different constraints (efficiency, client acceptance, complexity of management, etc.).
**Requirement:** Clients can trust that responses to queries include all fitting services.

**Assumptions:** Clients trust the ASSERT4SOA framework. ASSERT4SOA framework instances have means (such as X.509 digital certificates) to establish their identity.

**Proposed Solution:** Responses are authenticated, either by being digitally signed by the framework, or by being sent using an authenticated communication channel.

---

**Requirement:** Clients can trust that responses to queries include all fitting services.

**Assumptions:** Clients trust the ASSERT4SOA framework. Clients have means to analyse ASSERTs.

**Proposed Solution:** Responses to queries contain either the ASSERTs, or more likely, information that allows clients to retrieve the ASSERTs, associated with the services included in the response.

---

**Requirement:** Query responses are verifiable.

**Assumptions:** Clients trust the ASSERT4SOA framework. The different instances of the ASSERT4SOA framework have means (such as X.509 digital certificates) to establish their identity.

**Proposed Solution:** Queries (or part of them) can be encrypted for the ASSERT4SOA framework, or are sent using a confidential communication channel.

**Note:** This is not specific to ASSERT4SOA, but to all service frameworks.

---

**Requirement:** Privacy of information included in the queries is ensured.

**Assumptions:** Clients trust the ASSERT4SOA framework. Clients have means to analyse ASSERTs.

**Proposed Solution:** Queries are authenticated, either by being digitally signed by the client, or by being sent using an authenticated communication channel.

---

**Requirement:** Reasoning mechanisms are verifiable.

**Assumptions:** Clients do not completely trust the ASSERT4SOA framework.
Clients have means to analyse ASSERTs.

**Proposed Solution:** Responses to queries contain information about the reasoning used to select the services included in the response or the Orchestration Patterns applied.

**Requirement:** Reasoning mechanisms are customizable.

**Assumptions:** Clients need additional control over the selection process

**Proposed Solution:** Queries contain information to customize the reasoning mechanisms used to produce the response to the query, or the Orchestration Patterns that may be used to fulfil the query.

### 4.7 Installation instructions and usage examples

An initial prototype of the A-SerDiQueL discovery engine that was described in Sections 4.2-4.7 has been developed in the first year of the project and been made available internally in the consortium for testing purposes. This engine is based on the initial version of the SerDiQueL engine [72] and extends it to support the extensions of A-SerDiQueL.

The initial prototype implementation of the engine is to be integrated in the overall ASSERT4SOA framework that is to be delivered as part of the deliverables D06.2 and D06.3. Prior to this integration we also expect to perform additional testing of the engine and address some limitations of the current implementation (see Section 4.8).

Nevertheless to enable the use of the current version of the engine, in the following, we provide instructions on how the engine can be installed and used. The current prototype of the engine can be used to execute A-SerDiQueL queries against an eXist service registry and provides a simple GUI for this purpose. The actual implementation of the engine has been made available in the shared SVN repository of the project.

#### 4.7.1 Installation

To install the **ASSERT4SOA Discovery Engine**, first download the distribution zip file “Assert4SOADiscoveryEngine.zip” from the project’s SVN repository, at [https://ra.crema.unimi.it/svn/ASSERT4SOA/WPs/WP2/D2.1](https://ra.crema.unimi.it/svn/ASSERT4SOA/WPs/WP2/D2.1), and unzip the contents of the file to a suitable folder (e.g. C:\Assert4SOADiscoveryEngine). The archive contains five folders which are described below:

- **DiscoveryEngine** – This folder contains the source code and binary distribution of the discovery engine.
DiscoveryEngineClient – This folder contains the source code and binary distribution of a client application to the discovery engine.

ContextOperation – This folder contains the source code and binary distribution of an example context service that will be required to execute the example queries explained in Section 4.7.2.

SeCSERegistry – The current implementation of the discovery engine requires the use of SeCSE registry [59], which facilitates access to multifaceted service descriptions. This folder contains the binary distribution of SeCSE registry.

eXist – This folder contains binary distribution of eXist database [42]. As described above that SeCSE registry facilitates access to multifaceted service descriptions, eXist database is used as back end store for multifaceted service descriptions.

**Dependencies:** Assert4SOA Discovery Engine requires that Java SE6 (or above) be installed and correctly configured (see java.sun.com for download and installation instructions).

**Environment Variables:** Assert4SOA Discovery Engine relies on the following environment variables in order to correctly operate:

- “RSD_HOME”: This variable points to the directory where the Discovery Tool is installed. For example: “C:\Assert4SOADiscoveryEngine\DiscoveryEngine”.
- “SERVICE_LOCATION”: This variable contains the URL location of the SeCSE registry to use. The value of this variable should be: http://localhost:8082/SeCSERegistry/service/SeCSERegistry
- “EXIST_HOME”: This variable points to the eXist database installation directory. For example: “C:\Assert4SOADiscoveryEngine\eXist”
- “WORDNET_DATA”: This variable points to the XML configuration file for WordNet [52] (this is used for lexical analysis in the structural matching process). For example: “C:\Assert4SOADiscoveryEngine\DiscoveryEngine\wordnet.xml”. In addition this file should contain a reference to the location of the wordnet data in the file system. The exact part of the wordnet.xml file that needs to be configured for this is shown in bold in Table 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<jwnl_properties language="en">
  <version publisher="Princeton" number="2.0" language="en"/>
  <dictionary class="net.didion.jwnl.dictionary.FileBackedDictionary">
```

---

D.2.1 - Assert4SOA Aware Service Query Language and Discovery Engine 71/109
4.7.2 Usage Guide

This distribution comes with three example queries. These queries are located in "C:\Assert4SOADiscoveryEngine\DiscoveryEngine\examples" folder. Also the eXist registry contained in this distribution has been populated with descriptions of 28 different services that are relevant to the example queries. To execute the example queries follow the steps described below.

1. **Start the Discovery Engine:** Once the discovery tool has been unzipped, and configured, the tool is ready to be used. In order to start the tool, execute the "C:\Assert4SOADiscoveryEngine\DiscoveryEngine\RunDiscoveryEngineWS.bat" file in a command prompt window. This will start the discovery tool as a Web Service and a front end GUI as shown in Figure 4.13 will be displayed.

![Diagram](image-url)

---

### Table 4.1: "Wordnet.xml" file content

......

```xml
<param name="file_manager" value="net.didion.jwnl.dictionary.file_manager.FileManagerImpl">
  <param name="dictionary_path" value="C:\Assert4SOADiscoveryEngine\DiscoveryEngine\wordnetdata"/>
</param>
</dictionary>
<resource class="PrincetonResource"/>
</jwnl_properties>
```

---

D.2.1 - Asserts Aware Service Query Language and Discovery Engine  

72/ 109
2. **Start the SeCSE Registry:** Execute the C:\assert4soa\DiscoveryEngine\SeCSERegistry\bin\startup.bat file in a command prompt window to start the SeCSE registry.

3. **Start the eXist database:** Execute the C:\assert4soa\DiscoveryEngine\eXist\bin\startup.bat file in a command prompt window to start the eXist database.

4. **Start the Context service:** Execute the following command in a command prompt window to start the context service.
   
   C:\assert4soa\DiscoveryEngine\ContextOperation\dist>java -jar ContextOperation.jar

5. **Start the Discovery Engine Client:** Start the client by executing the “C:\assert4soa\DiscoveryEngine\DiscoveryEngineClient\RunDiscoveryEngineClient.bat” file in a command prompt window. The GUI of the client will be displayed as shown in Figure 4.14.

6. **Select Query:** Select an example query from C:\assert4soa\DiscoveryEngine\DiscoveryEngine\examples using the File->Open menu of the client GUI. Once a query is selected, it will be displayed in the client GUI as shown in Figure 4.15. Click on the
doSubmitQuery button in the client GUI to submit the query to the discovery engine.

7. The Discovery Engine GUI will be updated once it receives and executes the query as shown in Figure 4.16. Also the command prompt window that was used to start the discovery engine in step 1 will let you monitor the progress of the query execution as shown in Figure 4.17.

8. When the discovery engine finishes the execution of the query it sends the WSDL of the best mapped service back to the client and the client GUI will display the result as shown in Figure 4.18.

Figure 4.15: ASSERT4SoA Discovery Engine Client GUI with Query
Figure 4.16: ASSERT4SOA Discovery Engine GUI – Query Execution

Figure 4.17: ASSERT4SOA Discovery Engine – Query Execution
4.8 Limitations and known problems

The realization of the discovery engine that has been released as part of the D2.1 deliverable has certain limitations which will be addressed subsequently in the project.

More specifically, the currently released discovery engine does not support the following features of A-SerDiQueL:

- The parameter in A-SerDiQueL queries which specifies whether the query requires confidentiality when transmitted from the framework to the registries (QueryConfidentiality element).

- The parameter in A-SerDiQueL queries which specifies whether the results transmitted from the registry to the framework are required to be confidential (ResultConfidentiality element).

- The parameter in A-SerDiQueL queries which specifies whether the registries must be authenticated (i.e., the Authentication element).
The version of the discovery engine that will be delivered as part of the integrated version of the framework in month 24 will also support the confidentiality of communications with external registries as specified in Assert4SoAL queries (as discussed in Section 3.3.1).

Also, in the implementation of the discovery engine that is released by this deliverable, there is no support for the deployment of special types of matchmakers during the discovery process (see the attribute matchmakers in the element AssertQuery). This feature is supported at the parsing level but during the execution of a query only the generic matchmaker of the discovery engine is used to evaluate AssertQuery elements.

Special matchmakers supporting the matching and ordering of different types of ASSERTs, namely ASSERTs-M, ASSERTs-E and ASSERTs-O are to be developed by the by the work packages WP3, WP4 and WP5 according to the work programme of the project (see deliverables D03.2, D04.2, and D05.4). These matchmakers will be integrated with the discovery engine within the overall Assert4SoA framework and will be released with the integrated version of the framework to enable the deployment of special matching capabilities for special types of certificates instead of the generic ones that are provided by the core implementation of the discovery engine that is released as part of D2.1.

Finally, the current prototype of the discovery engine supports only single service discovery and does realise the discovery of service compositions in response to discovery queries. The extension that will provide this functionality will be provided by the deliverable D2.4 according to the work programme of the project.
Chapter 5

Conclusions

5.1 Main conclusions

In this report, we have provided a specification of A-SerDiQueL, an XML based language for specifying service discovery queries that could be used for identifying and replacing the constituent services of service based applications. The special feature of A-SerDiQueL in contrast with its competitors is the support that it offers for the specification of discovery criteria that relate to security certificates, in particular ASSERTs.

A-SerDiQueL has been developed as an extension of SerDiQueL, a language developed by City University to support static and dynamic service discovery. Despite providing comprehensive support for the specification of structural, behavioural, quality and context discovery criteria, SerDiQueL did not provide any support for the expression of criteria regarding security certificates and/or the security of the discovery process itself. Furthermore, its original design did not allow for the specification of criteria to trigger the building of service compositions as part of the discovery process.

These limitations have been addressed in the design of A-SerDiQueL that we have developed in ASSERT4SoA. They have also been partly addressed in the implementation of the discovery engine that we developed to support A-SerDiQueL. In particular, the implementation of the discovery engine that has been released as part of the deliverable D2.1 does not support the identification of service compositions in response to service discovery queries. This is because the implementation of the relevant component of the discovery engine and the overall ASSERT4SoA framework is to be developed and released later as part of the deliverable D2.4 according to the work programme of the project.
5.2 Future work

The specification of A-SerDiQueL and the discovery engine supporting it provides partial support for the discovery of services that fulfill conditions regarding ASSERTs.

At the query language definition level, A-SerDiQueL provides comprehensive support for addressing the service discovery requirements that have been identified in the project (see the Deliverable 7.1 [80]). At the implementation level, however, there are some further steps to be taken before reaching the level of support of ASSERT-aware service discovery as envisaged in the description of the project.

More specifically, the implementation provided by the discovery engine released with this deliverable needs to be integrated with:

- special types of match makers for the different types of certificates are to be developed by the work packages WP3, WP4 and WP5,
- the composition manager that will support the identification of compositions in response to discovery queries, and
- the overall ASSERT4SoA framework.

The above forms of integration will take place in WP6 and be released in two separate consecutive versions as part of the deliverables D06.2 and D06.3 in months 24 and 36 of the project.
Bibliography


[34] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. 32nd Int. Conf. on Very Large Data Bases. 2006.


ASSERT4SoA


WSDL. http://www.w3.org/TR/wsd1

XQuery. http://www.w3.org/TR/xquery/


OWL. http://www.w3.org/TR/owl-ref/


SPARQL. http://www.w3.org/TR/rdf-sparql-query/

Appendix A

A-SerDiQueL Schema

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http://assert4soa.eu/schema/SerDiQueL-v2"
  xmlns:par="http://assert4soa.eu/schema/Parameters"
  xmlns:bsql="http://gredia.eu/schema/Behaviour_SQL"
  xmlns:asql="http://assert4soa.eu/schema/Assert_SQL"
  xmlns:csql="http://assert4soa.eu/schema/Constraint_SQL"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  elementFormDefault="qualified"
  targetNamespace="http://assert4soa.eu/schema/SerDiQueL-v2"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="2.0">
  <xs:import schemaLocation="./Constraint_SQL.xsd"
    namespace="http://assert4soa.eu/schema/Constraint_SQL" />
  <xs:import schemaLocation="./Behaviour_SQL.xsd"
    namespace="http://gredia.eu/schema/Behaviour_SQL" />
  <xs:import schemaLocation="./Assert_SQL.xsd"
    namespace="http://assert4soa.eu/schema/Assert_SQL" />
  <xs:import schemaLocation="./WSDL.xsd"
    namespace="http://schemas.xmlsoap.org/wsdl/">
  </xs:import>
  <xs:import schemaLocation="./Parameters.xsd"
    namespace="http://assert4soa.eu/schema/Parameters" />
  <xs:element name="ServiceQuery">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="par:Parameters" />
        <xs:element minOccurs="0" name="StructuralQuery">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="wsdl:definitions" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element minOccurs="0" ref="bsql:BehaviourQuery" />
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="csql:ConstraintQuery" />
        <xs:element minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
Table A.1: The main A-SerDiQueL schema

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http://assert4soa.eu/schema/Parameters"
  elementFormDefault="qualified"
  targetNamespace="http://assert4soa.eu/schema/Parameters"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Parameters">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Mode">
          <xs:complexType>
            <xs:attribute name="value" use="required">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="PULL"/>
                  <xs:enumeration value="PUSH"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
        <xs:element name="Type">
          <xs:complexType>
            <xs:attribute name="value" use="required">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="static"/>
                  <xs:enumeration value="dynamic"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
A.2.1 - Assert4Soa Aware Service Query Language and Discovery Engine

Table A.2: The schema for the parameter part
<xs:element name="BehaviourQuery" type="BehaviourExpressionType" />
<xs:complexType name="LogicalOperatorType">
  <xs:attribute name="operator" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="AND" />
        <xs:enumeration value="OR" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
<xs:complexType name="BehaviourExpressionType">
  <xs:sequence>
    <xs:element minOccurs="0" name="Requires">
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" name="MemberDescription">
            <xs:complexType>
              <xs:attribute name="ID" type="xs:ID" use="optional" />
              <xs:attribute name="opName" type="xs:string" use="required" />
              <xs:attribute name="synchronous" type="xs:boolean" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:choice>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="Condition" type="BehaviourConditionType" />
    <xs:sequence>
      <xs:element name="Expression" type="BehaviourExpressionType" />
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="LogicalOperator" type="LogicalOperatorType" />
        <xs:element name="Expression" type="BehaviourExpressionType" />
      </xs:sequence>
    </xs:sequence>
    <xs:choice>
      <xs:attribute name="negated" type="xs:boolean" use="optional" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="BehaviourConditionType">
  <xs:choice>
    <xs:element name="GuaranteedMember">
      <xs:complexType>
        <xs:attribute name="IDREF" type="xs:IDREF" use="required" />
      </xs:complexType>
    </xs:element>
    <xs:element name="OccursBefore">
      <xs:complexType>
        <xs:attribute name="Occurrence" type="xs:integer" use="required" />
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:complexType>
<xs:complexType>
  <xs:complexContent mixed="false">
    <xs:extension 
      xmlns:q1="http://gredia.eu/schema/Behavour_SQL" 
      base="q1:OccursType" />
  </xs:complexContent>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
<xs:element name="OccursAfter">
  <xs:complexType>
    <xs:complexContent mixed="false">
      <xs:extension 
        xmlns:q1="http://gredia.eu/schema/Behavour_SQL" 
        base="q1:OccursType" />
    </xs:complexContent>
  </xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
<xs:element name="Sequence">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="2" maxOccurs="unbounded" name="Member">
        <xs:complexType>
          <xs:attribute name="IDREF" type="xs:IDREF" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="ID" type="xs:ID" use="optional" />
  </xs:complexType>
</xs:element>
<xs:element name="Loop" type="LoopType" />
</xs:choice>
</xs:complexType>
<xs:complexType name="OccursType">
  <xs:sequence>
    <xs:element name="Member1">
      <xs:complexType>
        <xs:attribute name="IDREF" type="xs:IDREF" use="required" />
      </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:element name="Member2">
    <xs:complexType>
      <xs:attribute name="IDREF" type="xs:IDREF" use="required" />
    </xs:complexType>
  </xs:element>
</xs:complexType>
</xs:element>
<xs:element>
  <xs:complexType>
    <xs:attribute name="immediate" type="xs:boolean" use="required" />
    <xs:attribute name="guaranteed" type="xs:boolean" use="required" />
  </xs:complexType>
</xs:element>
<xs:element>
  <xs:complexType>
    <xs:attribute name="negated" type="xs:boolean" use="optional" />
    <xs:complexType name="OccursType">
      <xs:sequence>
        <xs:element name="Member1">
          <xs:complexType>
            <xs:attribute name="IDREF" type="xs:IDREF" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:element name="Member2">
        <xs:complexType>
          <xs:attribute name="IDREF" type="xs:IDREF" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:complexType>
  </xs:element>
</xs:choice>
</xs:complexType>
<xs:complexType name="LoopType">
  <xs:attribute name="immediate" type="xs:boolean" use="required" />
  <xs:attribute name="guaranteed" type="xs:boolean" use="required" />
</xs:complexType>
Table A.3: The schema for the behavioural part of the queries

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns="http://assert4soa.eu/schema/Constraint_SQL"
  elementFormDefault="qualified"
  targetNamespace="http://assert4soa.eu/schema/Constraint_SQL"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.1">
  <xs:element name="ConstraintQuery">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="LogicalExpression" type="LogicalExpressionType"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="LogicalOperator">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="AND"/>
                <xs:enumeration value="OR"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:ID" use="required"/>
        <xs:attribute default="0.5" name="weight" use="optional">
          <xs:simpleType>
            <xs:restriction base="xs:double">
              <xs:minInclusive value="0"/>
              <xs:maxInclusive value="1"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="type" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="HARD"/>
              <xs:enumeration value="SOFT"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute>
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="contextual"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
use="required" /> 
  <xs:attribute name="externalExecution" type="xs:boolean" 
    default="false" />
  </xs:complexType>
</xs:element>
</xs:complexType name="LogicalExpressionType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="Condition" type="ConditionType" />
      <xs:element name="LogicalExpression" type="LogicalExpressionType" />
    </xs:choice>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="LogicalOperator" type="LogicalOperator">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="AND" />
            <xs:enumeration value="OR" />
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="LogicalExpression" type="LogicalExpressionType" />
    </xs:sequence>
  </xs:sequence>
  <xs:attribute default="false" name="negated" type="xs:boolean" />
</xs:complexType>
</xs:complexType name="ConditionType">
  <xs:sequence>
    <xs:element name="Operand1" type="RelationalOperandType" />
    <xs:element name="Operand2" type="RelationalOperandType" />
  </xs:sequence>
  <xs:attribute default="false" name="negated" type="xs:boolean" />
  <xs:attribute name="relation" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="EQUAL-TO" />
        <xs:enumeration value="NOT-EQUAL-TO" />
        <xs:enumeration value="LESS-THAN" />
        <xs:enumeration value="GREATER-THAN" />
        <xs:enumeration value="LESS-THAN-EQUAL-TO" />
        <xs:enumeration value="GREATER-THAN-EQUAL-TO" />
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
</xs:complexType name="RelationalOperandType">
  <xs:choice>
    <xs:element name="NonContextOperand" type="NonContextOperand" />
    <xs:element name="ContextOperand" type="ContextOperandType" />
    <xs:element name="Constant" type="ConstantType" />
    <xs:element name="ArithmeticExpression" type="ArithmeticExpressionType" />
  </xs:choice>
</xs:complexType>
</xs:complexType>
D.2.1 - Asserts Aware Service Query Language and Discovery Engine   93/ 109
<xs:element>
  <xs:element name="Constant" type="ConstantType" />
</xs:choice>
</xs:complexType>
<xs:complexType name="ConstantType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="type" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="NUMERICAL" />
            <xs:enumeration value="STRING" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="ArithmeticExpressionType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="ArithmeticOperand" type="ArithmeticOperandType" />
      <xs:element name="ArithmeticExpression" type="ArithmeticExpressionType" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ArithmeticOperandType">
  <xs:choice>
    <xs:element name="NonContextOperand" type="NonContextOperand" />
    <xs:element name="ContextOperand" type="ContextOperandType" />
    <xs:element name="Constant" type="ConstantType" />
    <xs:element name="Function" />
  </xs:choice>
</xs:complexType>
Table A.4: The schema for the contextual and non-contextual conditions
<xs:simpleType name="LogicalExpressionType">
    <xs:sequence>
        <xs:choice>
            <xs:element name="Condition" type="ConditionType" />
            <xs:element name="AssertProperty" type="AssertPropertyType" />
            <xs:element name="LogicalExpression" type="LogicalExpressionType" />
        </xs:choice>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="LogicalOperator" type="LogicalOperatorType" />
            <xs:restriction base="xs:string">
                <xs:enumeration value="AND" />
                <xs:enumeration value="OR" />
            </xs:restriction>
        </xs:sequence>
    </xs:sequence>
</xs:simpleType>

<xs:complexType name="LogicalExpressionType">
    <xs:sequence>
        <xs:choice>
            <xs:element name="Condition" type="ConditionType" />
            <xs:element name="AssertProperty" type="AssertPropertyType" />
            <xs:element name="LogicalExpression" type="LogicalExpressionType" />
        </xs:choice>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="LogicalOperator" type="LogicalOperatorType" />
            <xs:restriction base="xs:string">
                <xs:enumeration value="EQUAL-TO" />
                <xs:enumeration value="NOT-EQUAL-TO" />
            </xs:restriction>
        </xs:sequence>
    </xs:sequence>
    <xs:attribute default="false" name="negated" type="xs:boolean" />
</xs:complexType>

<xs:complexType name="ConditionType">
    <xs:sequence>
        <xs:element name="Operand1" type="RelationalOperandType" />
        <xs:element name="Operand2" type="RelationalOperandType" />
    </xs:sequence>
    <xs:attribute default="false" name="negated" type="xs:boolean" />
    <xs:attribute name="relation" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:string">
                <xs:enumeration value="EQUAL-TO" />
                <xs:enumeration value="NOT-EQUAL-TO" />
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
\[
\begin{align*}
\text{RelationalOperandType} & \quad \text{Explicit comparison operators:} \\
& \quad \text{LESS-THAN} & \quad \text{GREATER-THAN} \\
& \quad \text{LESS-THAN-EQUAL-TO} & \quad \text{GREATER-THAN-EQUAL-TO} \\
\end{align*}
\]
<xs:enumeration value="MINUS"/>
<xs:enumeration value="MULTIPLY"/>
<xs:enumeration value="DIVIDE"/>
</xs:restriction>
</xs:element>
<xs:choice>
  <xs:element name="ArithmeticOperand" type="ArithmeticOperandType"/>
  <xs:element name="ArithmeticExpression" type="ArithmeticExpressionType"/>
</xs:choice>
</xs:sequence>
</xs:complexType>
<xs:complexType name="ArithmeticOperandType">
  <xs:choice>
    <xs:element name="AssertOperand" type="xs:string"/>
    <xs:element name="Constant" type="ConstantType"/>
    <xs:element name="Function">
      <xs:complexType>
        <xs:sequence maxOccurs="unbounded">
          <xs:element name="ArithmeticExpression" type="ArithmeticExpressionType"/>
        </xs:sequence>
        <xs:attribute name="name">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="MAX"/>
              <xs:enumeration value="MIN"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
  </xs:choice>
</xs:complexType>
<xs:complexType name="AssertPropertyType">
  <xs:sequence>
    <xs:element name="PropertySpec" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>

Table A.5: The schema for the conditions on the ASSERTs
Appendix B

Example query

```xml
<?xml version="1.0" encoding="utf-8"?>
<tns:ServiceQuery xmlns:tns="http://assert4soa.eu/schema/SerDiQueL-v2"
    xmlns:par="http://assert4soa.eu/schema/Parameters"
    xmlns:tns="http://assert4soa.eu/schema/Assert_SQL"
    xmlns:tns="http://assert4soa.eu/schema/Constraint_SQL"
    xmlns:tns="http://gredia.eu/schema/Behaviour_SQL"
    xmlns:tns="http://schemas.xmlsoap.org/wsdl/
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://assert4soa.eu/schema/SerDiQueL-v2 ./SerDiQueL-v2.xsd"
    queryID="UUID:550e8400-e29b-41d4-a716-446655440000" name="Query1">
    <par:Parameters>
        <par:Mode value="PULL" />
        <par:Type value="dynamic" />
        <par:Threshold value="1.0" />
        <par:Composition value="true"
            patternsRef="http://assert4soa.eu/patterns"/>
        <par:Registry>
            <par:QueryConfidentiality value="true" />
            <par:ResultsConfidentiality value="true"
                excludeNonconforming="false" />
            <par:Authentication value="true" />
        </par:Registry>
    </par:Parameters>
    <!-- Structural sub-query -->
    <tns:StructuralQuery>
        <wsdl:definitions xmlns:ax29="http://common/xsd"
            xmlns:axis2="http://scube.org/gps"
            xmlns:ns1="http://org.apache.axis2/xsd"
            xmlns:ns="http://scube.org/xsd/gps"
            xmlns:plnk=
                "http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
            xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
            xmlns:http="http://schemas.xmlsoap.org/wsdl/http"/>
```
<assert4soa>

<xs:schema xmlns:ax210="http://common/xsd"
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  targetNamespace="http://scube.org/xsd/gps">
  <xs:import namespace="http://common/xsd" />
  <xs:element name="acknowledge">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="val" nillable="true" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="acknowledgeResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="return" type="xs:boolean"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="login">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="ID" nillable="true" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="loginResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="return" type="xs:boolean"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="logout">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="ID" nillable="true" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

<xs:import namespace="http://www.w3.org/2001/XMLSchema" />
<xs:schema xmlns="http://schemas.xmlsoap.org/wsdl/soap/">
  targetNamespace="http://scube.org/gps"
</xs:schema>
<xs:documentation>GPSService3</xs:documentation>
</assert4soa>
<assert4soa>

<xsd:element name="logoutResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="0" name="return" type="xsd:boolean"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="makePayment">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="0" name="amount" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="makePaymentResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="0" name="return" type="xsd:boolean"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="getLocationResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element minOccurs="0" name="return" nillable="true" type="ax210:Location"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://common/xsd">
  <xsd:complexType name="Location">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="latitude" type="xsd:int"/>
      <xsd:element minOccurs="0" name="longitude" type="xsd:int"/>
      <xsd:element minOccurs="0" name="name" nillable="true" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
</assert4soa>
<wsdl:input message="axis2:makePaymentRequest" wsaw:Action="urn:makePayment"/>
<wsdl:output message="axis2:makePaymentResponse" wsaw:Action="urn:makePaymentResponse"/>
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="GPSService3Soap11Binding" type="axis2:GPSService3PortType">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="logout">
<soap:operation soapAction="urn:logout" style="document"/>
<wsdl:input>
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="acknowledge">
<soap:operation soapAction="urn:acknowledge" style="document"/>
<wsdl:input>
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="getLocation">
<soap:operation soapAction="urn:getLocation" style="document"/>
<wsdl:input>
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="login">
<soap:operation soapAction="urn:login" style="document"/>
<wsdl:input>
<soap:body use="literal"/>
</wsdl:input>
<wsdl:output>
<soap:body use="literal"/>
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="makePayment">
<soap:operation soapAction="urn:makePayment" style="document"/>
</wsdl:operation>
<soap12:operation soapAction="urn:makePayment"
style="document" />
<wsdl:input>
<soap12:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap12:body use="literal" />
</wsdl:output>
</wsdl:operation>
<wsdl:binding name="GPSService3HttpBinding"
type="axis2:GPSService3PortType">
<http:binding verb="POST" />
<wsdl:operation name="logout">
<http:operation location="GPSService3/logout" />
<wsdl:input>
<mime:content part="logout" type="text/xml" />
</wsdl:input>
<wsdl:output>
<mime:content part="logout" type="text/xml" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="acknowledge">
<http:operation location="GPSService3/acknowledge" />
<wsdl:input>
<mime:content part="acknowledge" type="text/xml" />
</wsdl:input>
<wsdl:output>
<mime:content part="acknowledge" type="text/xml" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="getLocation">
<http:operation location="GPSService3/getLocation" />
<wsdl:input>
<mime:content part="getLocation" type="text/xml" />
</wsdl:input>
<wsdl:output>
<mime:content part="getLocation" type="text/xml" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="login">
<http:operation location="GPSService3/login" />
<wsdl:input>
<mime:content part="login" type="text/xml" />
</wsdl:input>
<wsdl:output>
<mime:content part="login" type="text/xml" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="makePayment">
<http:operation location="GPSService3/makePayment" />
<wsdl:input>
<mime:content part="makePayment" type="text/xml" />
</wsdl:input>
</wsdl:operation>
</wsdl:input>
<wsdl:output>
  <mime:content part="makePayment" type="text/xml" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
</wsdl:service name="GPSService3">
  <wsdl:port binding="axis2:GPSService3Soap11Binding"
    name="GPSService3HttpSoap11Endpoint">
    <soap:address
      location="http://138.40.94.100:8888/axis2/services/GPSService3.GPSService3HttpSoap11Endpoint/" />
  </wsdl:port>
  <wsdl:port binding="axis2:GPSService3Soap12Binding"
    name="GPSService3HttpSoap12Endpoint">
    <soap12:address
      location="http://138.40.94.100:8888/axis2/services/GPSService3.GPSService3HttpSoap12Endpoint/" />
  </wsdl:port>
  <wsdl:port binding="axis2:GPSService3HttpBinding"
    name="GPSService3HttpEndpoint">
    <http:address
      location="http://138.40.94.100:8888/axis2/services/GPSService3.GPSService3HttpEndpoint/" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
</tns:StructuralQuery>

<!-- Behavioural sub-query -->
<tnsb:BehaviourQuery>
  <tnsb:Requires>
    <tnsb:MemberDescription ID="login"
      opName="GPSService3PortType.login" synchronous="true" />
    <tnsb:MemberDescription ID="payment"
      opName="GPSService3PortType.makePayment" synchronous="true" />
    <tnsb:MemberDescription ID="location"
      opName="GPSService3PortType.getLocation" synchronous="true" />
    <tnsb:MemberDescription ID="acknowledge"
      opName="GPSService3PortType.acknowledge" synchronous="true" />
    <tnsb:MemberDescription ID="logout"
      opName="GPSService3PortType.logout" synchronous="true" />
  </tnsb:Requires>
  <tnsb:Expression>
    <tnsb:Condition>
      <tnsb:GuaranteedMember IDREF="login" />
    </tnsb:Condition>
  </tnsb:Expression>
  <tnsb:LogicalOperator operator="AND" />
  <tnsb:Expression>
<tnsb:Condition>
  <tnsb:Sequence ID="pay">
    <tnsb:Member IDREF="payment" />
    <tnsb:Member IDREF="location" />
    <tnsb:Member IDREF="acknowledge" />
  </tnsb:Sequence>
</tnsb:Condition>

<tnsb:Condition>
  <tnsb:OccursBefore immediate="false" guaranteed="false">
    <tnsb:Member1 IDREF="login" />
    <tnsb:Member2 IDREF="pay" />
  </tnsb:OccursBefore>
</tnsb:Condition>
</tnsb:Expression>
</tnsb:BehaviourQuery>

<!-- Constraints sub-queries -->
<tnsa:ConstraintQuery name="C1" type="SOFT" contextual="false" weight="0.5" externalExecution="true">
  <tnsa:LogicalExpression>
    <tnsa:Condition relation="EQUAL-TO">
      <tnsa:Operand1>
        <tnsa:NonContextOperand facetName="QoS" facetType="QoS">
          //QoSCharacteristic[Name="Availability"]/Metrics/Metric[Name="OpenTime"][Unit="Hours"]/MinValue
        </tnsa:NonContextOperand>
      </tnsa:Operand1>
      <tnsa:Operand2>
        <tnsa:Constant type="STRING">00:00</tnsa:Constant>
      </tnsa:Operand2>
    </tnsa:Condition>
    <tnsa:LogicalOperator>AND</tnsa:LogicalOperator>
    <tnsa:LogicalExpression>
      <tnsa:Condition relation="EQUAL-TO">
        <tnsa:Operand1>
          <tnsa:NonContextOperand facetName="QoS" facetType="QoS">
            //QoSCharacteristic[Name="Availability"]/Metrics/Metric[Name="OpenTime"][Unit="Hours"]/MaxValue
          </tnsa:NonContextOperand>
        </tnsa:Operand1>
        <tnsa:Operand2>
          <tnsa:Constant type="STRING">12:00</tnsa:Constant>
        </tnsa:Operand2>
      </tnsa:Condition>
    </tnsa:LogicalExpression>
  </tnsa:LogicalExpression>
</tnsa:ConstraintQuery>

<tnsa:ConstraintQuery name="C2" contextual="true" type="SOFT" weight="0.5">
  <tnsa:LogicalExpression>
  </tnsa:LogicalExpression>
<tnsa:Condition relation="LESS-TIAN-EQUAL-TO">
  <tnsa:Operand1>
    <tnsa:ContextOperand serviceID="7021.001"
      serviceOperationName="makePayment">
      <tnsa:ContextCategory relation="EQUAL-TO">
        <tnsa:Category1>
          <tnsa:Document location="http://eg.org/CoDAMoS_Extended.xml"
            type="ONTeLOGY" />
        </tnsa:Category1>
        <tnsa:Category2>
          <tnsa:Constant type="STRING">
            GREDIA_RELATIVE_TIME
          </tnsa:Constant>
        </tnsa:Category2>
      </tnsa:ContextCategory>
    </tnsa:Operand1>
    <tnsa:Operand2>
      <tnsa:Constant type="STRING">SECONDS-5</tnsa:Constant>
    </tnsa:Operand2>
  </tnsa:Condition>
</tnsa:LogicalExpression>
</tnsa:ConstraintQuery>

<tnsd:AssertQuery name="A1" type="SOFT">
  <tnsd:LogicalExpression>
    <tnsd:Condition relation="EQUAL-TO">
      <tnsd:Operand1>
        <tnsd:AssertOperand facetName="Assert"
          facetType="Assert"/>
        <tnsd:AssertOperand facetName="Assert"
          facetType="Assert"/>
      </tnsd:Operand1>
      <tnsd:Operand2>
        <tnsd:Constant type="STRING">Fraunhofer_SIT</tnsd:Constant>
      </tnsd:Operand2>
    </tnsa:Condition>
  </tnsa:LogicalExpression>
</tnsa:AssertQuery>

<tnsd:AssertQuery name="A2" type="HARD" externalExecution="true">
  <tnsd:LogicalExpression>
    <tnsd:Condition relation="EQUAL-TO">
      <tnsd:Operand1>
        <tnsd:AssertOperand facetName="Assert"
          facetType="Assert"/>
        <tnsd:AssertOperand facetName="Assert"
          facetType="Assert"/>
      </tnsd:Operand1>
      <tnsd:Operand2>
        <tnsd:Constant type="STRING">Fraunhofer_SIT</tnsd:Constant>
      </tnsd:Operand2>
    </tnsa:Condition>
  </tnsa:LogicalExpression>
</tnsa:AssertQuery>
Table B.1: The complete listing of the example query Query1

```
<tnsd:Constant type="STRING">M</tnsd:Constant>
</tnsd:Operand2>
</tnsd:Condition>
</tnsd:LogicalExpression>
</tnsd:AssertQuery>

<tnsd:AssertQuery name="A4" type="SOFT" matchmakers="specialized">
  <tnsd:LogicalExpression>
    <tnsd:AssertProperty>
      <tnsd:PropertySpec>SOMEPROPERTY</tnsd:PropertySpec>
    </tnsd:AssertProperty>
  </tnsd:LogicalExpression>
</tnsd:AssertQuery>
</tns:ServiceQuery>
```