# Report October 1st

## Improving the formalization of the algorithm and Building our architecture

The basic input for the Rhone algorithm is: (1) a query; (2) a list of concrete services.

**Definition 1 (Query):** A query $Q$ is defined as a set of *abstract services*, a set of *constraints*, and a set of *user preferences* in accordance with the grammar:

$$Q(\overline{I}, \overline{O}) := A_1(\overline{I}, \overline{O}), A_2(\overline{I}, \overline{O}), .., A_n(\overline{I}, \overline{O}), C_1, C_2, .., C_m[P_1, P_2, .., P_k]$$

The left side of the definition is called the *head* of the query; and the right side is called the *body*. $\overline{I}$ and $\overline{O}$ are a set of *input* and *output* parameters, respectively. Input parameters that exists in both sides of the definition are called *head variables*. In contrast, input parameters that exists only in the query body are called *local variables*. The abstract services $(A_1, A_2, .., A_n)$ specify a set of abstract functions performed by the query. $C_1, C_2, .., C_m$ are constraints over the *input* and/or *output* parameters. The user preferences (over the services) are signed in $P_1, P_2, .., P_k$. $C$ and $P$ are in the form $x \otimes constant$ such that $\otimes \in \{\geq, \leq, =, \neq, <, >\}$.

Let us suppose the following query example in order to illustrate the definition.

**Example 1:** *The user wants to retrieve the DNA information from patients infected by the disease 'K' using services that have availability higher than 99%, price per call less than 0.2 dollars, and the total cost less then 1 dollar.*

Example 1 is expressed following the Definition 1 as below. The decorations ? and ! are used to specify input and output parameters, respectively.

$$Q(d?, dna!) := A1(d?, p!), A2(p?, dna!), d = \text{“}K\text{”}[availability > 99\%,\ price\ per\ call < 0.2\$,\ total\ cost < 1\$]$$

Analyzing the query, it is possible to note that the parameters “d?” and “dna!” appear in both sides of the definition. Due to that they are *head* variables. On the other hand, “p!” and “p?” are *local* variables considering that they appear only in the body definition. Additionally, note that the local variables “p!” and “p?” have the same name. Intuitively, this fact indicates a dependency between the abstract services which use these variables (in that case A1 and A2).

In the example, A1 and A2 are the abstract services that specify the functions performed by the query. A1 retrieves the patients infected by a given disease. A2 retrieves that DNA information of a patient. The constraint ($d =$ “$K$”) over the input parameter ‘d’ will be further used while executing the query over a database (the where clause). *Availability*, *price per call* and *total cost* are the user preferences over the services.

**Definition 2 (Concrete service):** A concrete service $(S)$ has its form similar to a query:

$$S(\overline{I}, \overline{O}) := A_1(\overline{I}, \overline{O}), A_2(\overline{I}, \overline{O}), .., A_n(\overline{I}, \overline{O})[P_1, P_2, .., P_k]$$

A concrete service $(S)$ is defined as a set of abstract services $(A)$, and by its quality constraints $P$. These quality constraints associated to the service represent the service level agreement exported by the concrete service.

**Example 2:** Considering the query and the abstract services described in the Example 1, the following concrete services are examples in accordance with the Definition 2.

$S1(a?, b!) := A1(a?, b!)[availability > 99\%,\ price\ per\ call = 0.2\$]$
$S2(a?, b!) := A1(a?, b!)[availability > 99\%,\ price\ per\ call = 0.1\$]$
$S3(a?, b!) := A1(a?, b!)[availability > 98\%,\ price\ per\ call = 0.1\$]$
$S4(a?, b!) := A2(a?, b!)[availability > 99.5\%,\ price\ per\ call = 0.1\$]$
$S5(a?, b!) := A2(a?, b!)[availability > 99.7\%,\ price\ per\ call = 0.1\$]$
$S6(a?, b!) := A3(a?, b!)[availability > 99.7\%,\ price\ per\ call = 0.1\$]$
$S7(a?, b!) := A2(a?, c!), A3(c?, b!)[availability > 99.7\%,\ price\ per\ call = 0.1\$]$

Given the query and a list of concrete services as input, the algorithm will try to find concrete services that are *candidates* to be part of the rewriting process. The candidate concrete services are identified while searching for matches between abstract services in $S$ and abstract service in $Q$. The following definitions will guide us while selecting candidate concrete services.

**Definition 3 (abstract service equivalence):** A match between abstract services occurs when a abstract service $A_i$ is equivalent to $A_j$, denoted $A_i = A_j$. Given two abstract services $A_i$ and $A_j$, $A_i = A_j$ iff: (1) $A_i$ and $A_j$ have the same abstract function name; (2) the number of *input* parameters of $A_i$ is equal to $A_j$; and (3) the number of *output* parameters of $A_i$ is equal to $A_j$.

Based on the assumption that the concrete services can express service compositions in which the services involved may be able to change the world. A concrete service (S) is selected as *candidate* to the rewriting process if for each abstract service in $S$ there is an equivalent in $Q$; there is no abstract service in $S$ that does not exist in $Q$; and the quality constrains in $Q$ must be guaranteed in $S$.

**Definition 4 (candidate service):** Given a query $Q$ and a concrete service $S$, $S$ is a *candidate* service iff: (1) $\nexists A_i \; s.t. \; A_i \in S \; and \; A_i \notin Q$; and (2) the quality constraints in $S$ does not violate the user preferences in $Q$.

Considering the query in the Example 1 and the concrete services in the Example 2, it is possible to see that:

- S1 and S3 are not a candidate services because they violate the user preferences.
- S6 is not a candidate service because it does not cover any abstract service in Q.
- S7 is not a candidate service because it covers abstract service A3 that is not present in Q.
- S2, S4 and S5 are candidate services once: all their abstract services have an equivalent in Q; and there is no violation in the user preference.

An important concept in our approach is the *candidate service description* (CSD). A CSD describes how a *candidate* concrete service can be used in the query rewriting process which included mappings between variables, covered abstract services, and quality constrains. Intuitively, a rewriting is a set of *candidate service descriptions* that fully covers the original query, and do not violates the user preferences.

**Definition 5 (candidate service description):** CSD is a complex data structure defined as $\langle S, h, \varphi, G, P \rangle$ where $S$ is a concrete service. $h$ are mappings between terms in the head of $S$ to terms in the body of $S$. $\varphi$ are mapping between terms from the abstract composition to terms in the concrete service definition. $G$ is a set of abstract services covered by $S$. $P$ is a set quality constraints associated to the service $S$.

The CSD for a given service will be created following rules: (1) for all head variables in $S$, there is a mapping for a head variable in $Q$; and (2) if $x$ is an local variable in $S$ mapped to a local variable in $Q$, then $S$ must cover all abstract services in $Q$ which uses $x$ or cover only one abstract service that uses $x$.

**Example 3:** To illustrate the rules above consider the following example. Supposing that we have the query $Q$ and the concrete services $S1$, $S2$, $S3$ and $S4$:

$$Q(d?, birth!, dna!) := A1(d?, p!), A2(p?, birth!), A3(p?, dna!)$$
$$S1(a?, b!) := A1(a?, c!), A2(c?, b!)$$
$$S2(a?, b!) := A3(a?, b!)$$
$$S3(a?, b!) := A1(a?, b!)$$
$$S4(a?, b!) := A2(a?, b!)$$

In the query $Q$ it is possible to note that "p!" is a *local* variable which is used as input ("p?") for the abstract services $A2$ and $A3$. Looking to the concrete service $S1$ no CSD will be created for it because the *local* variable $c!$ is mapped to the local variable $p!$, but $S1$ does not cover all abstract services which expects that variable. On the other hand, CSDs are constructed to the services $S2$, $S3$ and $S4$ once even existing the mapping from a local variable in the concrete service to a local variable in the query, all of them only cover one abstract service which uses that *local* variable. To be more clear about these rules, consider the rewriting below in which the CSDs for the services $S2$, $S3$ and $S4$ are used:

$$Q(d?, birth!, dna!) := S3(d?, p!), S3(p?, birth!), S4(p?, dna!)$$

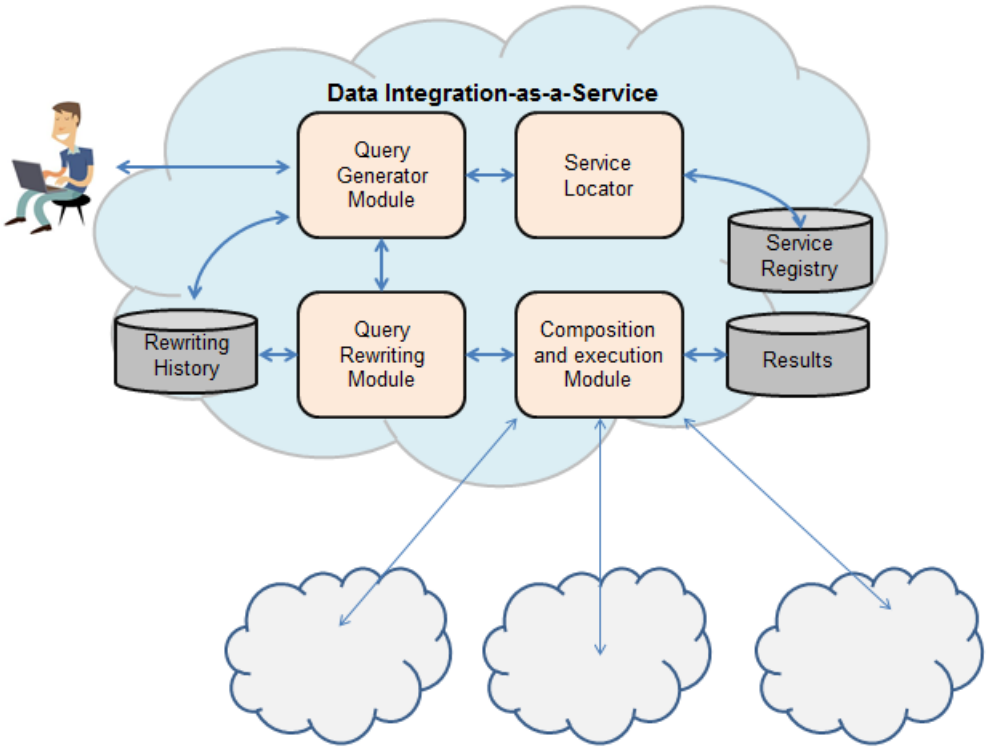
The rewriting above is the only one possible for the query. However, let us suppose that a CSD for $S1$ was created violating the rule number two, consequently the wrong rewriting below would be created:

$$Q(d?, birth!, dna!) := S1(d?, birth!), S4(p?, dna!)$$

The problem here is regarding the *local* variable $p?$ which appears in $S4$, and it apparently should come from $S1$, but we can not guarantee that the same *local* variable internally used in $S1$ is the one expected by $S4$. That is the reason the rule two exists.

**Architecture**

*Query Generator Module* helps the user to create his queries, and to define his preferences. Once we have a query this module will try to find if there are previous rewritings to a equivalent query. If true, *the composition and execution module* can publish and execute the new composition retrieving and integrating the new data. If there are no previous rewritings, the module interacts with the *service locator* in order to identify in our *service registry* services that can answer the query or part of it. The query and the selected services are used in the *Query Rewriting Module* to generate the rewritings for the query. The rewritings are sent to *the composition and execution module* to be published and executed (perhaps we can use BPEL to compose and execute). The integration process could be done in our database or in the database of one related services (somehow we should analyze which one is the best option).


Data Integration-as-a-Service
Query Generator Module
Service Locator
Service Registry
Rewriting History
Query Rewriting Module
Composition and execution Module
Results