Abstract:
This report includes technical documentation regarding the final version of the implementation of the WP3 services (detection, repairing, cleaning), with emphasis on the new features with respect to what was reported in Deliverable D3.2.
<table>
<thead>
<tr>
<th>Ver.</th>
<th>Date</th>
<th>Contributor(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>June 16th, 2015</td>
<td>Giorgos Flouris</td>
<td>Table of contents, initial assignments</td>
</tr>
<tr>
<td>0.3</td>
<td>July 6th, 2015</td>
<td>FORTH</td>
<td>First draft (Sections 1, 2, 3)</td>
</tr>
<tr>
<td>0.5</td>
<td>July 15th, 2015</td>
<td>FORTH</td>
<td>Second draft (Sections 1, 2, 3, 6)</td>
</tr>
<tr>
<td>0.7</td>
<td>July 24th, 2015</td>
<td>All</td>
<td>Third draft (all sections, and appendix)</td>
</tr>
<tr>
<td>0.9</td>
<td>July 28th, 2015</td>
<td>All</td>
<td>Deliverable ready for internal review</td>
</tr>
<tr>
<td>1.0</td>
<td>July 31st, 2015</td>
<td>All</td>
<td>Deliverable complete</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

1 INTRODUCTION ......................................................... 5
  1.1 CHANGE DETECTION SERVICE ...................................... 5
  1.2 MONITORING AND PROPAGATION SERVICE ......................... 6
  1.3 REPAIRING SERVICE .............................................. 6
  1.4 CLEANING SERVICE .............................................. 6

2 PRELIMINARIES .......................................................... 8

3 THE DETECTION SERVICE ............................................... 9
  3.1 FORMAL SPECIFICATIONS .......................................... 9
      3.1.1 FORMAL SEMANTICS OF CHANGES ............................ 9
      3.1.2 ASSOCIATIONS ........................................... 11
      3.1.3 CHANGE DETECTION FOR EVOLUTION ANALYSIS .............. 12
  3.2 TECHNICAL DOCUMENTATION ...................................... 14
      3.2.1 IMPLEMENTATION ASSUMPTIONS AND PREREQUISITES ......... 14
      3.2.2 DEPLOYMENT PROCESS .................................... 15
      3.2.3 EXPOSED WEB SERVICE INTERFACES ....................... 16
      3.2.4 EXPOSED USER INTERFACES ................................ 16
  3.3 D2V: A TOOL FOR CHANGE DETECTION ............................. 17
      3.3.1 CREATING COMPLEX CHANGES .............................. 17
      3.3.2 EDIT/DELETE COMPLEX CHANGES .......................... 20
      3.3.3 VISUALIZATION OF EVOLUTION HISTORY .................. 20
      3.3.4 TECHNICAL DETAILS AND DOCUMENTATION ................ 21

4 THE MONITORING AND PROPAGATION SERVICE ......................... 23
  4.1 SERVICE DESCRIPTION ........................................... 23
  4.2 THE API .......................................................... 23
      4.2.1 CREATE A MONITORING TASK .............................. 23
      4.2.2 MODIFY A MONITORING TASK .............................. 25
      4.2.3 DELETE A MONITORING TASK ............................... 25
      4.2.4 PAUSE A MONITORING TASK ................................ 25
      4.2.5 REQUEST MONITORING TASK CONFIGURATION ............... 25
      4.2.6 LIST MONITORING TASKS ................................... 25
      4.2.7 SUBSCRIBE TO A MONITORING TASK ....................... 25
      4.2.8 UNSUBSCRIBE FROM A MONITORING TASK .................. 26
      4.2.9 PROPAGATE NOTIFICATIONS ................................ 26
  4.3 TECHNICAL DOCUMENTATION ...................................... 26
      4.3.1 IMPLEMENTATION TECHNOLOGY .............................. 26
      4.3.2 PREREQUISITES ........................................... 26
      4.3.3 PROJECT DEPENDENCIES ................................... 26
      4.3.4 DEPLOYMENT AND CONFIGURATION ........................... 27

5 THE CLEANING SERVICE ................................................. 29
  5.1 CLEANING WORKFLOW .............................................. 29
  5.2 INTERACTIVE CLEANING WITH OPENREFINE ......................... 29
  5.3 TECHNICAL DOCUMENTATION ...................................... 30
      5.3.1 PREREQUISITES ........................................... 30
      5.3.2 HOW TO INSTALL ........................................... 30
      5.3.3 HOW TO RUN ................................................ 33
      5.3.4 INTEGRATION OF NEW METRICS .............................. 33

Grant Agreement No. 601043
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DIACHRON Layer Architecture</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Visualization of Completeness and Unambiguity</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>The Ontology of Changes</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Representation of Associations</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Properties File</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>Creating Mark_As_Obsolete (based on templates)</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Creating Mark_As_Obsolete (advanced version)</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>Dataset-centric and version-centric view</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Service logical architecture</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>Configuration example</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>Updated cleaning workflow</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>User interface of Cleaning Module</td>
<td>31</td>
</tr>
<tr>
<td>13</td>
<td>Metric selection interface</td>
<td>31</td>
</tr>
<tr>
<td>14</td>
<td>Cleaning statistics</td>
<td>32</td>
</tr>
<tr>
<td>15</td>
<td>Export of cleaned data set in RDF format</td>
<td>32</td>
</tr>
<tr>
<td>16</td>
<td>Cleaning process workflow</td>
<td>34</td>
</tr>
<tr>
<td>17</td>
<td>Quality Statistics in the Quality Report Ontology</td>
<td>36</td>
</tr>
</tbody>
</table>

# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Configuration Variables</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>Cleaning Methods</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>Ontological Simple Changes Codelist</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>Multidimensional Simple Changes Codelist</td>
<td>42</td>
</tr>
</tbody>
</table>
**1 Introduction**

This is a technical deliverable, in which we provide documentation regarding the final version of the implementation of the DIACHRON services of WP3. In addition, we provide the final version of the specifications of these services, with emphasis on any differences with respect to what was reported in Deliverables D3.1 [5] and D3.2 [6].

Figure 1 shows (enclosed in yellow) the components described in this deliverable, and their interactions within the context of the entire DIACHRON platform. The corresponding services of each component have already been implemented and we have uploaded the source code in the project’s github account for internal use by the partners (https://github.com/diachron). All services will be formally deployed as part of the final integrated DIACHRON environment in Deliverable D6.5 in M30. In the following, we describe the new back-end implementation and optimization of the services we described in the previous deliverables D3.1, D3.2 [5], [6].

**1.1 Change Detection Service**

The change detection service is responsible for identifying the changes that occurred between any two given versions (and led from one version to the other). Said detection occurs after the change has happened, i.e., the system assumes no knowledge on the change process itself, and the only input for performing the detection is...
the content of the two versions to be compared. The implementation of this service abides by the specifications described in previous deliverables (D3.1, D3.2).

The change detection service will be included in the integrated DIACHRON environment, but has also been publicly deployed already under URL http://139.91.183.65:8080 (Apache Tomcat 7.0.54). An interface for the change detection service, as a stand-alone web application (named D2V) has been deployed at http://213.249.38.66:3343/D2V/.

The change detection service is described in Section 3. Emphasis is given to the modifications made in the original design (compared to D3.1 [5]), which was based on the pilots’ feedback and internal discussions; these changes are described in Section 3.

1.2 Monitoring and Propagation Service

The monitoring and propagation service provides publish-subscribe functionality for creating monitoring tasks and notifications. A monitoring task is a process configured by the user in order to notify the user when specific types of events occur on the dataset, e.g., when a new version of the dataset is stored in the DIACHRON archive.

Since with each dataset version the changes with the previous version are also getting stored in the archive, the monitoring task can be further configured to trigger notifications for specific change types. Other users can subscribe to one or more monitoring tasks so that notifications for detected changes are propagated to them.

The design of this service and its API specification have been presented in documents D3.1 [5] and D6.1 [7] and, as described in these documents, the service is heavily dependent upon the archive and the change detection services. In this deliverable, some minor changes are reported and technical information about the implementation and the deployment of the service are provided.

The monitoring and propagation service will be included in the integrated DIACHRON environment. A standalone deployment of the service is not provided since it requires interaction with other platform components in order to provide its functionality.

1.3 Repairing Service

The repairing service deals with the problem of identifying and resolving invalidities in datasets. An invalidity is defined as a violation of a certain constraint associated with the underlying data, such as the requirement for two concepts to be disjoint. The repairing service of DIACHRON considers constraints of logical nature, in particular constraints that can be expressed in the language DL-LiteA [2]. It provides an efficient methodology for identifying invalidities (taking into account DL-LiteA reasoning), as well as for resolving them in a manner that has the least impact (in terms of lost knowledge) to the dataset.

The details on the theoretical background upon which this service is based have been described in Deliverable D3.1 [5], whereas the used algorithms and the technical documentation on the implementation of the service appeared in Deliverable D3.2 [6]. Further details can be found in [3].

Since our original deployment (in M16 – D3.2), we had no new requirements regarding the repairing service. As a result, all our optimizations and improvements were minor internal functionality improvements.

Further, none of the related signatures of the service have changed. Therefore, the descriptions provided in previous deliverables (D3.1 [5], D3.2 [6]) need not be updated, and further details are omitted from this deliverable.

The repairing service will be included in the integrated DIACHRON environment, but has also been publicly deployed already under URL http://139.91.183.65:8080 (Apache Tomcat 7.0.54).

1.4 Cleaning Service

In contrast to the repairing service, which deals with inconsistencies of a logical nature, the cleaning service aims to address a different aspect of data quality: it verifies whether small pieces of information are logically consistent in themselves, whether they are syntactically well-formed, and whether their representation follows good practices of data engineering and data publishing. Since the correction of such problems requires user involvement and often detailed knowledge about the data, the cleaning process cannot be performed fully automatically. The cleaning
service therefore focuses on reporting cleaning suggestions and offers automated cleaning by deleting any data with quality problems. For those more advanced cleaning tasks that inherently require user interaction, we offer a cleaning application, which shares code with the cleaning service.

The main purpose of the cleaning service and the cleaning application is to support the user in the following data cleaning issues: i) automatic identification of inconsistencies/anomalies in the data and ii) generation of suggestions addressing the identified problems. The general design of the cleaning service has appeared in D3.1 [5]. The initial set of features of the cleaning service and the cleaning application have not changed from D3.2 [6]. This deliverable refers back to D3.2 where things remain unchanged; instead, it focuses on the changes, enhancements and improvements made since then.

The cleaning service will be included in the integrated DIACHRON environment, but has also been publicly deployed already under URL http://purl.org/net/diachron/cleaning/service/ (using OpenRefine v2.6-rc1, which uses Jetty 6.1.22).
2 Preliminaries

The following preliminaries are necessary for following the theoretical description behind the change detection service (Section 3.1) and are included for reasons of self-containment.

We consider two disjoint sets \( U, L \), denoting the URIs and literals (we ignore here blank nodes that can be avoided when data are published according to the LOD paradigm); the set \( T = U \times U \times (U \cup L) \) is the set of all RDF triples. A version \( \mathcal{D}_i \) is a set of RDF triples (\( \mathcal{D}_i \subseteq T \)); a dataset \( \mathcal{D} \) is a sequence of versions \( \mathcal{D} = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle \).

SPARQL 1.1 [9, 1] is the official W3C recommended language for querying RDF graphs. The building block of a SPARQL statement is a triple pattern \( t p \) that is like an RDF triple, but may contain variables (prefixed with character \(?\)); variables are taken from an infinite set of variables \( V \), disjoint from the sets \( U, L \), so the set of triple patterns is: \( TP = (U \cup V) \times (U \cup V) \times (U \cup L \cup V) \). SPARQL triple patterns can be combined into graph patterns \( gp \), using operators like join ("."), optional (OPTIONAL) and union (UNION) [1] and may also include conditions (using FILTER). In this work, we are only interested in SELECT SPARQL queries, which are of the form: "SELECT \( v_1, \ldots, v_n \) WHERE \( gp \)". For the precise semantics and further details on the evaluation of SPARQL queries, the reader is referred to [9, 1].
Grant Agreement No. 601043

3 The Detection Service

This section describes the change detection service. Note that we have made several improvements compared to the descriptions appearing in D3.1 [5] and D3.2 [6]. These improvements include the consolidation of the theoretical framework regarding the service, as well as optimizations related to the implementation efficiency of the service. Both types are described below in Sections 3.1, 3.2 respectively. In addition, Section 3.2 contains all the technical details relevant for using the change detection service. Finally, in Section 3.3, we describe a tool, named D2V, for visualizing and analysing the evolution history of a dataset. The tool has been developed by FORTH, as a proof-of-concept for the use of the change detection service and is not pilot-specific, so it is expected to be customized and adapted to the specific needs of each pilot.

3.1 Formal Specifications

Here, we describe the theoretical background behind the change detection service. Even though most of this work has already been described in previous deliverables, we felt it necessary to provide an updated description here, for reasons of self-containment, despite the fact that the improvements made since D3.2 are technical and do not affect the behaviour or semantics of the change detection service. A more detailed, up-to-date account of the formal specification of the change detection service appears in [10], [12].

3.1.1 Formal Semantics of Changes

Language of Changes

We assume a set $L = \{c_1, \ldots, c_n\}$ of changes, which is disjoint from the sets of variables ($V$), URIs ($U$) and literals ($L$). The set $L$ is called a language of changes and is partitioned into the set of simple changes (denoted by $L^s$) and the set of complex changes (denoted by $L^c$). Each change has a certain arity; given a change $c$, a change specification is an expression of the form $c(p_1, \ldots, p_n)$, where $n$ is the arity of $c$, and $p_1, \ldots, p_n \in V$.

The detection semantics of a change specification are determined by the changes that it consumes and the related conditions. Formally:

Definition 1. Given a change specification $c(p_1, \ldots, p_n)$, the detection semantics of $c(p_1, \ldots, p_n)$ is defined as a tuple $⟨δ, φ_{old}, φ_{new}⟩$ where:

- $δ$ determines the consumed changes of $c$ and is defined as:
  - For simple changes, $δ = (δ^+, δ^-)$, where $δ^+$, $δ^-$ are sets of triple patterns (corresponding to the added/deleted triples respectively);
  - For complex changes, $δ$ is a set of change specifications of changes from $L^s$.

- $φ_{old}, φ_{new}$ are graph patterns, called the conditions related to $D_1, D_2$, respectively.

The definition of 1 determines the triple patterns that constitute to the SPARQL that is used for the detection of a change.

Any actual detection will give specific values (URIs or literals) to the variables appearing in a change specification. For example, when change Add_Property_Instance is detected between any given pair of versions, the returned result should specify the subject and object of the instance added to the property; essentially, this corresponds to an association of the three variables (parameters) of Add_Property_Instance to specific URIs/literals. Formally, given a change $c$, a change instantiation is an expression of the form $c(x_1, \ldots, x_n)$, where $n$ is the arity of $c$, and $x_1, \ldots, x_n \in U \cup L$. Note also that detectability is defined differently for simple and complex changes.

Detection Semantics for Simple Changes

For simple changes, a detectable change instantiation corresponds to a certain assignment of the variables in $δ^+, δ^-$, $φ_{old}, φ_{new}$, such that the conditions $⟨φ_{old}, φ_{new}⟩$ are true in the underlying datasets, and the triples in $δ^+, δ^-$ have been added/deleted, respectively, from $D_1$ to get $D_2$:
Definition 2. A change instantiation $c(x_1, \ldots, x_n)$ of a simple change specification $c(p_1, \ldots, p_n)$ is detectable for the pair $D_1, D_2$ iff there is a $\mu \in [\phi_{old}]^{D_1} \cap [\phi_{new}]^{D_2}$ such that for all $tp \in \delta^+$: $\mu(tp) \in D_2 \setminus D_1$ and for all $tp \in \delta^- : \mu(tp) \in D_1 \setminus D_2$ and for all $i$: $\mu(p_i) = x_i$.

Simple changes must satisfy the properties of completeness and unambiguity (see also [5]); this guarantees that the detection process exhibits a sound and deterministic behaviour [8]. Essentially, what we need to show is that each changes that the dataset underwent is properly captured by one, and only one, simple change. Formally:

Definition 3. A detectable change instantiation $c(x_1, \ldots, x_n)$ consumes $t \in D_2 \setminus D_1$ (respectively, $t \in D_1 \setminus D_2$) iff there is a $\mu \in [\phi_{old}]^{D_1} \cap [\phi_{new}]^{D_2}$ and a $tp \in \delta^+$ (respectively, $tp \in \delta^-$) such that $\mu(tp) = t$ and for all $i$: $\mu(p_i) = x_i$.

The concept of consumption represents the fact that low-level changes are “assigned” to simple ones, essentially allowing a grouping (partitioning) of low-level changes into simple ones. To fulfill its purpose, this “partitioning” should be perfect, as dictated by the properties of completeness and unambiguity. Formally:

Definition 4. Consider a set of simple changes $C$. This set is called complete iff for any pair of versions $D_1, D_2$ and for all $t \in (D_2 \setminus D_1) \cup (D_1 \setminus D_2)$, there is an instantiation $c(x_1, \ldots, x_n)$ of some $c \in C$ such that $c(x_1, \ldots, x_n)$ is detectable and consumes $t$.

Definition 5. Consider a set of simple changes $C$. This set is called unambiguous iff for any pair of versions $D_1, D_2$ and for all $t \in (D_2 \setminus D_1) \cup (D_1 \setminus D_2)$, if $c, c' \in C$ and $c(x_1, \ldots, x_n), c'(x_1', \ldots, x_m')$ are detectable and consume $t$, then $c(x_1, \ldots, x_n) = c'(x_1', \ldots, x_m')$.

In a nutshell, completeness guarantees that all low level changes are associated with at least one simple change, thereby making the reported delta complete (i.e., not missing any change); unambiguity guarantees that no race conditions will emerge between simple changes attempting to consume the same low level change (see Figure 2 for a visualization of the notions of completeness and unambiguity). The combination of these two properties guarantees that the delta is produced in a deterministic manner and that it will properly reflect the changes that were actually performed. Regarding the simple changes, $L^s$, used in this work (see [10] and [6]), the following holds:

Proposition 1. The simple changes appearing in $L^s$ [10] satisfy the properties of completeness and unambiguity.
Detection Semantics for Complex Changes  
As complex changes can be freely defined by the user, it would be unrealistic to assume that they will have any quality guarantees, such as completeness or unambiguity. As a consequence, the detection process may lead to non-deterministic consumption of simple changes and conflicts; to avoid this, complex changes are associated with a priority level, which is used to resolve such conflicts.

The detection for complex changes is defined on top of simple ones. A complex change is detectable if its conditions are true for some assignment, while at the same time the corresponding simple changes in it are detectable.

Definition 6. A complex change instantiation \( c(x_1, \ldots, x_n) \) is initially detectable for the pair \( \mathcal{D}_1, \mathcal{D}_2 \) iff there is a \( \mu \in \phi_{old} \cap \phi_{new} \) such that for all \( c'(\mu(p_1), \ldots, \mu(p_n)) \in \delta \) and all \( i \), \( \mu(p_i) = x_i \).

Definition 7. An initially detectable complex change instantiation \( c(x_1, \ldots, x_n) \) consumes a simple change instantiation \( c'(x'_1, \ldots, x'_m) \) iff there is a \( \mu \in \phi_{old} \cap \phi_{new} \) such that for all \( i \), \( \mu(p_i) = x_i \) and \( \mu(p'_i) = x'_i \).

Definition 8. A complex change instantiation \( c(x_1, \ldots, x_n) \) is detectable for the pair \( \mathcal{D}_1, \mathcal{D}_2 \) iff it is initially detectable and for all \( i \), \( \mu(p_i) = x_i \) and \( c, c' \) have at least one consumed simple change instantiation in common.

3.1.2 Associations

Associations are used to describe renamings, i.e., cases where the same real-world entity is described using a different URI in the old and new version. Renamings can happen in classes, properties or individuals, and are captured via an association table that describes said associations. In such an association table, each row represents an association among URIs where in the first column we have the URI value in the old version and in the second column we have the URI value in the new version. The associations could be provided as input by the user or detected automatically via instance matching or schema matching tools [4].

Associations are very important because, as some of our pilots reported, there are cases in which a set of URIs (or even all URIs) may change due to a potential change in the domain. Without associations, in such cases, the change detection process would report a potentially large number of delete/add changes, which would be impractical. On the other hand, reporting a set of associations among old/new values would be much more correct intuitively.

The notion of associations was also reported as mappings in deliverable D3.2 under “Mapping Filters” within the complex change definition as a desired and useful feature. In deliverable D3.2, mappings were represented using the main class Map; as we changed the term into associations, we will now represent mappings using the class Association. In this deliverable, we analyse how we extended the change detection mechanism to support associations.

First of all, we defined a new simple change, called ASSOCIATION which will be used in both the ontological and the multidimensional models. This change consists of two parameters, namely old_value, new_value, and it is instantiated from the instances of class Association. More details about the associations representation are presented in section 3.1.3 and specifically Figure 4. Since we treat associations as simple changes, the only modification we had to do in our infrastructure was to completely remove the Mapping Filters from the definition of Complex Changes.

For example, consider a case in which we have the following URI associations within a version pair:

- \([http://www.orphanet.org/rdfns#pat_id_530\] associated with \([http://www.orpha.net/ORDO/Orphanet_2374]\)
- \([http://www.orphanet.org/rdfns#pat_id_3064\] associated with \([http://www.orpha.net/ORDO/Orphanet_2749]\)
The change detection process considering these associations will create and report the following instances of the simple change ASSOCIATION:

- ASSOCIATION(http://www.orphanet.org/rdfns#pat_id_3064,http://www.orpha.net/ORDO/Orphanet_2749)

Note that the change detection process takes into account such associations, and does not report simple changes which refer to additions/deletions of triples that contain the renamed entities. This restriction applies in pairs of additions/deletions.

For example, consider the following pair of changes:

- DELETE_LABEL(http://www.orphanet.org/rdfns#pat_id_530,"label1")
- ADD_LABEL(http://www.orphanet.org/rdfns#pat_id_3064,"label1")

Neither of these changes will be reported, because they are not considered “proper” changes, but artefacts of the renaming operation. On the other hand, the following pair of changes will be reported:

- DELETE_COMMENT(http://www.orphanet.org/rdfns#pat_id_530,"Comment1")
- ADD_COMMENT(http://www.orphanet.org/rdfns#pat_id_3064,"Comment2")

Note that, in the latter case, the two comments (“Comment1”, “Comment2”) are different; as a result, they are considered actual changes – we indeed assume that the user deliberately changed the comment, after (or before) making the renaming, so the change in the comment should be reported normally. The same would happen, if, e.g., only the first (or only the second) change in the pair above existed; again, we assume that this change happened independently of the renaming, and should be reported normally.

### 3.1.3 Change Detection for Evolution Analysis

#### Representing Detected Changes in the Ontology of Changes

We treat detected changes (i.e., change instantiations) as first-class citizens in order to be able to perform queries analysing the evolution of datasets. Further, we are interested in performing combined queries, in which both the datasets and the changes should be considered to get an answer. To achieve this, the representation of the changes that are detected on the data cannot be separated from the data itself.

For example, consider the following sports-related query: “return all the left backs born before 1980, which were transferred to Athletic Bilbao between versions $D_1$ and $D_2$ and used to play for Real Madrid CF in any version”. Such a query requires access to the changes (to identify all transfers to Athletic Bilbao between the given versions), and to the data (to identify which of those transfers were related to left backs born before 1980); in addition, it requires access to all previous versions (cross-snapshot query) to determine whether any of the potential results (players) used to play for Real Madrid CF in any version.

To answer such queries, the repository should include all versions, as well as their changes. We opt to store the changes in a structured form; their representation should include connections with the actual entities (e.g., teams or players) and the versions that they refer to. This can be achieved by representing changes as RDF entities, with connections to the actual data and versions, so that a detectable change can be associated with the corresponding data entities that it refers to. In fact an RDF entity can be associated with the Diachron:entity class following the generic DIACHRON Model.

In particular, we propose the use of an adequate schema (that we call the ontology of changes) for storing the detected changes, thereby allowing a supervisory look of the detected changes and their association with the entities they refer to in the actual datasets, facilitating the formulation and the answering of queries that refer to both the data and their evolution. A simplified version of the ontology (populated) is shown in Figure 3.
In a nutshell, the schema in our representation describes the change specifications and detection semantics, whereas the detected changes (change instantiations) are classified as instances under this schema. More specifically, at schema level, we introduce one class for each simple and complex change $c \in \mathcal{L}$. Each such class is subsumed by one of the main classes “Simple_Change” or “Complex_Change”, indicating the type of $c$. Each change is also associated with its user-defined name, a number of properties (one per parameter), and the names of these parameters (not shown in Figure 3 to avoid cluttering the image).

For complex changes, we also store information regarding the changes being consumed by each complex change, as well as the SPARQL query used for its detection, which is automatically generated at change definition time; this is done for efficiency, to avoid having to generate this query in every run of the detection process. Note that the information related to complex changes is generated on the fly at change creation time (in contrast to simple changes, which are built in the ontology at design time). All schema information is stored in a dataset-specific named graph (“D/changes/App1/schema”, for a dataset $D$ and a related application App1); this is necessary because each different application may adopt a different set of complex changes.

At the instance level, we introduce one individual for each detectable change instantiation $c(x_1, \ldots , x_n)$ in each pair of versions. This individual is associated with the values of its parameters, which are essentially URIs or literals from the actual dataset versions. This provides the “link” between the change repository and the data, thereby allowing queries involving both the changes and the data. In addition, complex changes are connected with their consumed simple ones. The triples that describe this information are stored in an adequate named graph (e.g., “D/changes/v1-v2”, for the changes detected between $v_1, v_2$ of the dataset $D$). This implementation optimizes the query response for queries that are applied to detected changes over specific version pairs. We remind that in the implementation described in deliverables D3.1, D3.2, all the detected changes were stored in a single named graph and the version pairs were denoted using adequate properties.

Representing Associations in the Ontology of Changes

As mentioned in deliverable D3.2 [6] and Section 3.1.2, another useful feature of the ontology of changes is the storage of associations. Associations are stored as instances classified under the class Association, and include properties pointing to the old and new values in the corresponding association. The association instances are stored along with the detected changes of said version pair (i.e., in the same named graph) in order to record the versions between which they are applicable. For example, Figure 4 shows the representation of the associations of $\{x_1\}$ with $\{x_2\}$ and of $\{y_1\}$ with $\{y_2\}$. 
Change Detection Process and Storage To detect simple and complex changes, we rely on plain SPARQL update queries, which are generated from the information drawn from the detection semantics of the corresponding changes (Definition 1). For simple changes, this information is known at design time, so the query is loaded from a configuration file, whereas for complex changes, the corresponding query is generated once at change-creation time (run-time) and is retrieved from the ontology of changes (see Figure 3). The results of the generated queries determine the change instantiations that are detectable; this information is further processed to determine the actual triples to be inserted in the ontology of changes.

The SPARQL update queries used for detecting a simple change are \texttt{INSERT} queries, which directly update the ontology of changes with the values of the change instantiation; thus, each variable in the change specification, appears in a triple pattern which will be inserted. For example, if a simple change has three parameters, then the SPARQL update query used for its detection will insert three triple patterns in the ontology of changes; one for each parameter. Then, the \texttt{WHERE} clause of the query includes the triple patterns that should (or should not) be found in each of the versions in order for a change instantiation to be detectable; more specifically, the triple patterns in $\delta^+$ must be found in $D_2$ but not in $D_1$, the triple patterns in $\delta^-$ must be found in $D_1$ but not in $D_2$, and the graph patterns in $\phi_{old}, \phi_{new}$ should be applied in $D_1, D_2$, respectively.

The generation of the SPARQL queries for the complex changes follows a similar pattern. The main difference is that complex changes check the existence of simple changes in the ontology of changes, rather than triples in the two versions (as is the case with simple changes detection); therefore, complex changes should be detected after the detection of simple changes and their storage in the ontology. Note also that the considered simple changes should not have been marked as “consumed” by other detectable changes of a higher priority; thus, it is important for queries associated with complex changes to be executed in a particular order, as implied by their priority.

3.2 Technical Documentation

As already stated in deliverable D3.2, the Change Detection module contains the services and methods responsible for the change detection and representation among any given dataset versions. Moreover, there are methods that refer to the definition and management of user defined changes, i.e., complex changes.

3.2.1 Implementation Assumptions and Prerequisites

The assumptions which should be met by the implementation described in this deliverable are more or less the same as the ones described in deliverable D3.2. Again, we consider datasets which are already ingested and \textit{transformed} into the corresponding DIACHRON model in a complete and consistent way. We also remind the importance of the persistence of object URIs across versions which play the role of unique identifiers.

In this deliverable, we introduce an assumption which entails from the fact that in our new implementation, we have separated the detected changes (and change ontologies) per version pair for a more efficient management of changes. As a result, we need to keep the relations among a diachronic dataset with its versions and the corresponding detected changes.
To do so, for each considered dataset we assume we have a dataset URI which may potentially have a label as well. Moreover, we have a changes ontology schema which contains the change definitions for the corresponding dataset.

For example, for the dataset provided by EMBL, we used the URI http://www.ebi.ac.uk/efo/ along with a human understandable label EFO. The schema for this dataset can be found in the named graph with URI http://www.ebi.ac.uk/efo/changes/schema. Next, we assume that the dataset’s versions which are considered by the change detection module will be also connected with the dataset URI within the triple store. Consider a specific version of EMBL whose data are stored within named graph with URI http://www.diachron-fp7.eu/resource/recordset/efo/2.50. We also use a label to identify this dataset version i.e., 2.50. The relation between the dataset and its version is expressed using the triple (http://www.ebi.ac.uk/efo/, rdfs:member, http://www.diachron–fp7.eu/resource/recordset/efo/2.50).

Following a similar approach, we use a base changes ontology URI which is connected (under rdfs:member relation) with all the changes ontologies which are created for this dataset. Continuing the example from EMBL dataset, we have the changes ontology base URI http://www.ebi.ac.uk/efo/changes. Consider the versions v1 http://www.diachron-fp7.eu/resource/recordset/efo/2.49,v2 http://www.diachron-fp7.eu/resource/recordset/efo/2.50. A change detection among versions v1,v2 will store the detected changes within a named graph with URI http://www.ebi.ac.uk/efo/changes/2.49-2.50. The relation between this specific changes ontology URI and the base changes ontology URI is expressed using the triple (http://www.ebi.ac.uk/efo/changes, rdfs:member, http://www.ebi.ac.uk/efo/changes/2.49 – 2.50). We also relate the version URIs with their corresponding changes ontology using the triples:

(http://www.ebi.ac.uk/efo/changes/2.49-2.50 co:old\_version http://www.diachron-fp7.eu/resource/recordset/efo/2.49),


Finally, the infrastructure prerequisites used in the new implementation remain the same as in deliverable D3.2. We keep using Virtuoso RDFStore as a back-end exposed by the SesameAPI and JDBCApi.

3.2.2 Deployment Process

When someone wants to deploy the services of this module on his own application server and infrastructure the first obvious step is the deployment of the WAR file. However, this is not sufficient as there are some configuration parameters which have to be set by the user in order the services to work correctly. These parameters are defined and set in a file which is essentially a properties file. This type of file is usually used as a good-practice way to initialize parameters which remain constant in the whole life-cycle of any application. Based on our new implementation, we slightly changed the properties file as we do not keep the detected changes for every version pair in a single changes ontology. As a result, we do not need the parameter Changes Ontology. Moreover, as the changes ontology schema entails from the dataset URI, we also do not need the parameter Changes Ontology Schema. For completeness, we analyse again all the parameters which are contained in the properties file and a snapshot of such a file is displayed in Figure 5:

Repository IP: The URL of Virtuoso Server which is used for the storage and management of the Ontology of Changes.

Repository Username: The desired username to access Virtuoso Server.

Repository Password: The desired password to access Virtuoso Server.

Repository Port: The port in which Virtuoso Endpoint accepts incoming connections.

Dataset URI: The URI of the dataset whose versions are considered by the change detection module.

Simple Changes Folder: The local folder which contains the SPARQL queries which correspond on the simple changes. Each SPARQL query is stored in a single file and each simple change is detected using a corresponding SPARQL query which updates the ontology of changes accordingly.
Simple_Changes: The simple changes which will be considered from this change detection setting. As a good practice, we decided to provide in this field the file names of the SPARQL query files which belong in the folder defined by the field Simple_Changes_Folder.

Last but not least, the relations among the considered dataset URI and its versions which were discussed before must be stored within the triple store as well. For this purpose we used a dedicated named graph within Virtuoso Triplestore (http://datasets) which keeps not only the relations among a dataset URI and its versions, but also the relations among a changes ontology base URI along with the detected changes per version pair (i.e., changes ontologies).

3.2.3 Exposed Web Service Interfaces

The Change Detection module consists of services which can be used a) for the management of complex changes (e.g., definition, deletion etc.) and b) for the detection and the exploitation of the ontology of changes.

Despite the fact that the back-end implementation changed, the service signatures remain the same as the ones presented in deliverable D3.2. We did not want to cause incompatibilities neither to potential pilot applications who directly use the change detection services nor to the integration layer which consumes them.

However, as it was asked by our pilots, in Appendix A we provide a more detailed description of the service which is used for the definition of a complex change (i.e., method defineCCJSON). In precise we analyse the JSON definition the each complex change along with the accepted values of each field.

3.2.4 Exposed User Interfaces

Section 3.3 describes the exposed user interfaces which were implemented as a proof-of-concept for the change detection module. This interface provides basic functionalities for complex changes management along with the exploitation of the change detection process for a visual analysis of dataset evolution.
3.3 D2V: A Tool for Change Detection

D2V is a tool for assisting users in managing, querying and visualizing the evolution history of dynamic RDF datasets based on the change detection service. Our goal is to provide an adequate user interface, allowing the management (creation, edit, delete) of complex changes, as well as the visualization of the contents of the ontology of changes, to enable the visual analysis of dataset dynamics. The tool can support users with a shared or dedicated personal space. To help first-time users, personal accounts can come as totally blank (no complex changes defined) or with some predefined changes for convenience reasons. We should note that this is a suggestion that could be adapted or taken into account by any data pilot who wants to implement its own interface according to its needs.

3.3.1 Creating Complex Changes

The tool provides two modes for change creation, which allows for a different range of changes to be created, but also require a different level of familiarity of the user with the system.

The first mode is accessible via the “Define complex change” button, and is meant for less experienced users. It allows the creation of a complex change with minimal input, according to certain templates that correspond to interesting changes. For example, various ontological applications (especially in the biological domain) refrain from deleting classes, but use a subsumption relationship to a special class to indicate deprecation. In this case, the user would like to create a complex change (e.g., “Mark_As_Obsolete”) that would differentiate the addition of a subsumption relationship to this special class from the addition of a subsumption relationship to any other class, as the former has special semantics. This can be easily done by clicking on “Define change” - “Addition” - “Add Specialization Of” and then choosing “http://www.geneontology.org/formats/oboInOwl#ObsoleteClass” as the superclass from the respective drop-down menu (see figure 6).

More experienced users could use the “advanced” creation mode, where many different options are allowed, and the range of possible changes to be created is much larger; on the down side, using this interface requires some familiarity with the semantics of the simple changes.

First of all, the user is prompted to fill in the basic mandatory information of each complex change including its name (a descriptive unique name), a priority (float number denoting the priority of the change that solves ambiguity issues), its parameters (an ordered set of parameters, defined through their descriptive names) and the specification of the simple change parameters that they get their values from. Also, a description field to capture human understandable information can be used optionally.

In addition, D2V allows the identification of the simple changes that are associated with the complex one, and the definition of relevant filters. In particular, join filters are meant to join parameters of different simple changes, so that a bag of simple changes related to a given URI can be grouped to form one complex change. Selection filters are filters in the classical sense, where the user can narrow the detectability range of the change in different ways. Finally the user can apply one or more version filters, to express that some extra conditions should or not be claimed in \( D_1 \) or \( D_2 \) of a dataset in order to define a complex change. To model associations, the special simple change ASSOCIATION can be used. For more details on the semantics of the different filters and options, the reader is referred to [11].

As mentioned and described in [6] this semantic information gathered by all fields of the “advanced” creation mode constitutes the required input that the user interface should provide in order to allow the change detection service to manipulate the corresponding change. More precisely the tool generates dynamically the SPARQL query used for each complex change according to the user selections and stores it in ontology as a JSON string (one object per field) that encapsulates the change definition and is exploited by the change detection service.

To define “Mark_As_Obsolete” using the advanced interface, the user after choosing the “Define complex change advanced” button should add the “Add_Superclass” simple change in the list of associated changes, add a selection filter requiring “superclass” to be equal to “http://www.geneontology.org/formats/oboInOwl#ObsoleteClass” and set the single parameter for the complex change to be equal to the “subclass” parameter of the “Add_Superclass” change (see Figure 7).

\[\text{http://obi-ontology.org/page/OBIDeprecationPolicy}\]
Figure 6: Creating Mark_AsObsolete (based on templates).
Figure 7: Creating Mark_As_Obsolete (advanced version).
3.3.2 Edit/Delete Complex Changes

The tool also supports the edit or deletion of a complex change. This can be done simply through the corresponding “Edit complex change” or “Delete complex change” button, which shows the list of currently defined complex changes and allows the user to edit or delete the selected one.

For deletion, the complex change is deleted when the user selects it and presses the “Delete” button. For edit, the user is presented with all the relevant information that defines the specific complex change, allowing him to edit any of the fields. Note that, even if the complex change has been created based on advanced functionality or not (templates), the tool on edit displays the corresponding fields completed in the advanced user interface.

We should mention here that each complex change is identified by its unique name per assigned dataset. Remember that each dataset has been assigned its own versions and schema regarding to the assumptions that have been described in 3.2.1.

It should be also noted that the creation and deletion of complex changes might affect the already stored detected information; therefore, after each creation and deletion, the system will have to partly redo some of the change detections to update the already stored detected changes for the selected dataset.

3.3.3 Visualization of Evolution History

The visualization interface allows the analysis of the evolution history of a dataset in general, as well as the comparison of any two arbitrarily selected dataset versions. The operation mode is selected in the opening option menu that appears after choosing the “Visualize Changes” button.

Let us first consider the mode of evolution history analysis. In this case, the user is first presented with the dataset-centric view, where the entire evolution history along all ingested versions of the chosen dataset is shown. This allows viewing all the changes that were detected between each pair of consequent versions – hovering over the bars shows additional statistics (see Figure 8-top for the evolution history of EFO2).

Below the dataset-centric view, the user is presented with a panel providing a wide variety of different visualization options, allowing more customized analyses.

This interface allows a version-centric or change-centric view, where changes are grouped by version pair, or by change type; the results can be filtered for specific version pairs or change types, and grouped accordingly (Figure 8-bottom).

As in the dataset-centric view, the numbers are relative, but hovering over the graphic gives absolute numbers. The chart type can be changed, allowing pie, column, bar, line and area charts to be used, as well as a tabular form that can be sorted in descending or ascending order by clicking on a column name. These charts are useful for understanding the mix of changes in each/ across version pair(s), as well as the number of appearances for each change across specific version pairs.

For more details on the version-centric and the change-centric views, a click on a chart element (e.g., on a piece of the pie), will show analytical details about the changes being visualized in this element of the chart (namely, all changes appearing in the corresponding version pair, or all detected instantiations of the corresponding change in the selected version pairs). The user can use a radio button to switch to a change-centric view, in order to explore the total number of appearances of each change for the selected version pairs. Again, the results are graphically represented in a customizable manner.

Further, clicking on a term in the latter table, returns another table presenting all the different change instantiations involving this term, turning the view into term-centric; this is useful to understand the evolution of a term over time. The results in this table are filtered according to the chosen change types and versions.

We should note at this point that all options of the panel can be changed to reformulate the query at any time, giving this way the opportunity to produce a variety of statistics on the fly.

If the operation mode is changed to the comparison of a custom pair of versions, the interface and functionality is similar; the main difference is that now the comparison is restricted to the version pair selected by the user in the opening screen. Note that the user can select any pair, even versions that are not consecutive.

http://www.ebi.ac.uk/efo/
A difference in this scenario is that the changes may be detected at run-time; if the changes between the chosen pair of versions has not been computed before, the system calls the change detection algorithm before creating the initial pie chart; otherwise, the information necessary to display the pie chart is retrieved from the database.

3.3.4 Technical Details and Documentation

D2V is a cross-platform application implemented on top of open-source infrastructures and technologies. It was implemented in Java programming language, the user interface was developed using HTML/CSS technologies in combination with JQuery and Java Servlets which were implemented for the communication with the RDF store. For the visualization of results, we used the Google Charts Library \(^3\) which is an open library that aims to create customizable charts which are rendered using HTML5/SVG to provide cross-browser compatibility.

D2V follows the same approach with the change detection module by separating the change definitions from the detected changes per version pair. As a result, for each dataset URI, D2V keeps its change definitions in a separate named graph. Moreover, it also uses separate named graphs to store the detected changes per version pair. Finally, all the considered named graphs are accessed via SPARQL queries.

The tool meets the same assumptions and prerequisites as the ones described in 3.2. Thus, it connects to the DIACHRON archive where ontology information is located for holding changes definitions and detections. The tool deployment in web server is a trivial process that includes the upload of the corresponding war file. We have successfully made the deployment in Tomcat server and GlassFish, but it can be similarly run in any web server that supports Java.

\(^3\)https://developers.google.com/chart

Figure 8: Dataset-centric and version-centric view.
Note that the D2V uses a similar properties file (named config_diachron.properties) like the one described in 3.2.2 which contains configuration parameters (i.e., Virtuoso credentials for DIACHRON archive etc) that should be updated after deployment with the actual information. In fact the properties file has two differences. The first is that the property Dataset_URI is not used because the tool sets this value dynamically accordingly to the user’s selection from the top left-side menu of the tool. The second one is that the new property Dataset URIs used to denote the named graphs that contain the default schema for each dataset. We have uploaded the source code of the tool in the project’s github account for internal use by the partners (https://github.com/diachron/d2v_change_detection_tool).
4 The Monitoring and Propagation Service

This section describes the monitoring and propagation service. This is the first implementation of the service since its dependency on the other core services of the platform and the services of the integration layer led to the decision to include the delivery of this service to this second iteration of the development of the project. 4.1 briefly describes the service exposed functionality while 4.3 includes technical details about the implementation and deployment instructions.

4.1 Service Description

The monitoring and propagation service provides publish-subscribe functionality for creating change monitoring tasks and notifications. A monitoring task is a process configured by the user in order to notify the user when specific type of events occur on the dataset, e.g. a new version of the dataset is stored in the DIACHRON archive.

Since with each dataset version the changes with the previous version are also getting stored in the archive, the monitoring task can be further configured to trigger notifications for specific change type. Other users subscribe to one or more monitoring tasks so that notifications for detected changes are propagated to them.

A single monitoring task for a dataset can serve multiple users. Once it has been created by a user of the platform, other users could also subscribe to it. Monitoring tasks can be reused and thus changes will be propagated to them without having to recreate a monitoring task for the specific dataset. In order to for the module to function the integration layer of the platform must provide the appropriate events to the module. Then the propagation service will match these events to monitoring tasks based on the configuration parameters and subscribers will be appropriately notified. Change notifications will be propagated to users by various ways such as email notifications or ATOM feeds or similar syndication technologies.

The complete design of the service has been presented in deliverable D3.1 [5] and the following figure (Figure 9) depicts the update version of the logical architecture of the service. The main modifications to the original design are in the way that the service parts are placed in the platforms layer and in the way the service interacts with the integration layer components and the users of the system.

The main components of the service architecture are the following:

- the HTTP API, which allow the communication of the service consumers with it,
- the Monitoring and Propagation API, which offers a suite of operations over the service,
- the Mediator, which allows the communication with other DIACHRON components,
- the Monitoring Manager, which coordinates the monitoring tasks,
- the Propagation Manager, which coordinates subscriptions and notification propagation,
- the Persistence Store, which provides storage for monitoring tasks configurations and subscriptions.

4.2 The API

This section describes the HTTP API exposed by the monitoring and propagation service. The user of the service can create and manage monitoring tasks, subscribe and unsubscribe from monitoring tasks and receive change notifications.

4.2.1 Create a Monitoring Task

Description: The user provides the monitoring task configuration, which consists of the following parameters:

- The definition of the monitoring task. This consists of the id of the dataset that is to be monitored and optionally the change types of interest which are expected to be reported when a new version of the dataset is stored in the DIACHRON archive,
Figure 9: Service logical architecture
the monitoring period during which the dataset is to be monitored for changes.

The monitoring task will start running immediately and a monitoring task id will be returned.

Option: If the configuration is invalid, the service responds with an “invalid configuration” message. Note that a given configuration is invalid, when it is identical to an already existing monitoring task configuration, or when a wrong dataset id is given, or other error exist in the definition for the task.

4.2.2 Modify a Monitoring Task

Description: The user provides new definition about a monitoring task. The monitoring and propagation service replies with a “verification” message.

• The new definition of the monitoring task.

Option 1: If new configuration is invalid, the service responds with an “invalid configuration” message.

4.2.3 Delete a Monitoring Task

Description: The user requests a monitoring task configuration, to be deleted. The monitoring and propagation service replies with a “verification” message.

• the monitoring task id that she wants to delete.

Option: If the user is not the one who created or started the monitoring task (may be just a subscriber or not), the service replies with an “unauthorized user” message.

4.2.4 Pause a Monitoring Task

Description: The user requests a monitoring task to stop running for a specific period, and then it resumes. The service returns the time when it paused.

• the monitoring task id that will be paused.

Option 1: If there is no such monitoring task running or being already stopped, the service replies with an “already paused” message.

4.2.5 Request Monitoring Task Configuration

Description: The user requests a monitoring task configuration and the service returns all configuration parameters.

• the monitoring task id that she wants to retrieve its configuration.

4.2.6 List Monitoring Tasks

Description: The user requests all created monitoring tasks to be listed and the service returns all monitoring task ids.

4.2.7 Subscribe to a Monitoring Task

Description: The user makes a subscription request with the following parameters:

• the monitoring task id that she wants to subscribe,

• the subscription period during which she wants to get notifications,

• the notification type she wants to receive (email, ATOM feeds, etc).

The service returns a subscription id.
4.2.8 Unsubscribe from a Monitoring Task

Description: The user makes an unsubscribe request from a monitoring task. The service replies with a “verification” message. The parameters of the request are

- the monitoring task id that she wants to unsubscribe from.

4.2.9 Propagate Notifications

Description: The user receives notifications from a monitoring task that she has subscribed to, in the requested form.

4.3 Technical Documentation

4.3.1 Implementation Technology

The monitoring and propagation service is developed using the Java JDK version 7. Development has been done using the popular Eclipse IDE and Subversion was used as the version control system in order to synchronize the ATHENA team members. Since the Eclipse IDE is used the source code is structured as an Eclipse Maven project. The Apache Maven tool is a software project management and comprehension tool that can manage a project’s build, reporting and documentation from a central piece of information.

The source code along with its documentation (JavaDoc) can be found in github repository site located at https://github.com/diachron

4.3.2 Prerequisites

The monitoring and propagation service employs third-party software and technologies that are available as open-source. Specifically, the following prerequisites have to be installed in order for the service to be deployable:

- **Persistence Store.** The service uses Openlink Virtuoso 7.1 as its persistence store. It utilizes JDBC as the Data Access Layer in order to communicate with Virtuoso, which is done in two levels. The first one uses traditional JDBC data access methods to communicate directly with Virtuoso’s low-level, underlying SQL engine, while the second one is VirtJena, a bridge that makes Jena an interface between Jena’s classes and Virtuoso for RDF models. Deployment of the service requires a working instance of Virtuoso, compatible with past versions back to Virtuoso 6.

- **Application Server.** The application server is responsible for exposing the API functionality over HTTP. While the service has been tested with Apache Tomcat, any compliant J2EE application server or other Servlet Container can be employed.

- **Java Development Kit.** The Java Development Kit is required for the monitoring service to function. For testing purposes JDK 7 has been used, but any version equal to or newer than JDK 7 is adequate.

4.3.3 Project Dependencies

The service’s source code has dependencies in various open source Java libraries. The most important are the following:

- **RDF Model Management.** The Apache Jena Semantic Web Framework is used as a Java framework for storing and processing RDF at runtime in the service. More specifically, it uses Jena as an interface for storing and manipulating RDF data at runtime. Jena has various modular components that provide storage, query engine, endpoint and other functionalities.
**Database Connection.** Connection to and communication with the RDF store is performed via JDBC using the Virtuoso JDBC driver and the Virtuoso Jena RDF Data Provider (VirtJena) library. VirtJena is a storage provider for RDF graphs in the context of the Jena Framework, thus enabling direct querying and processing of graphs stored in Virtuoso using Jena classes. Furthermore, it provides abstractions for reading/writing to/from the Virtuoso store.

**Spring Web MVC Framework.**

The Spring Web MVC Framework is used for the development of the web application. Appropriate controllers have been implemented to invoke the specific functionality, such as creating or listing monitoring tasks.

### 4.3.4 Deployment and Configuration

The deployment process is rather simple as long as the prerequisites software mentioned before is installed. The war that contains the service is deployed in Tomcat application server using the standard procedures described in Tomcat’s documentation web site. As noted above any other compliant servlet container can be used and if so the corresponding deployment instructions must be followed. After the initialization of the application server the HTTP API is available in the application server host path as described in 4.2. A prebuild version (.war file) of the service can be found also in the source code repository at https://github.com/diachron.

In order for the service to function properly it must connect to the Virtuoso Server as described above and thus there are some connection parameters that should be provided to the module. This can be done by including a configuration file inside the war file or in the class path of the application server hosting the module so as to be detectable by the module.

This configuration file must be named “virt-connection.properties” and is structured as standard Java property files. The following table depicts the variables that can be set in this property file.

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtuoso-username</td>
<td>The username of a user of the Virtuoso server</td>
</tr>
<tr>
<td>virtuoso-pwd</td>
<td>The password of the user</td>
</tr>
<tr>
<td>virtuoso-host</td>
<td>The IP or hostname of the server running the Virtuoso server</td>
</tr>
<tr>
<td>virtuoso-port</td>
<td>The port configured in Virtuoso for incoming JDBC connections</td>
</tr>
<tr>
<td>virtuoso-connection-pooling</td>
<td>A boolean flag denoting if a connection pool should be created upon initialization</td>
</tr>
<tr>
<td>virtuoso-initial-pool</td>
<td>If connection pool is enabled then this variable configures the initial number of connections in the pool</td>
</tr>
<tr>
<td>virtuoso-max-pool</td>
<td>If connection pool is enabled then this variable configures the maximum number of connections in the pool</td>
</tr>
</tbody>
</table>

Figure 10 depicts an example of the “virt-connection.properties” configuration file.
5 The Cleaning Service

This section presents the cleaning service, which enables users to identify quality problems in RDF data and to handle these problems either by removing all triples affected by problems (automatic cleaning using a web service) or in an interactive cleaning process using an OpenRefine extension for RDF cleaning.

The great majority of details regarding functionality as well as the architecture of the cleaning module introduced in D3.1 [5] and D3.2 [6], remains unchanged. To keep this document short, we will not present them again. In this deliverable, we describe the changes and extensions that have been made in the recent period.

The following improvements were made:

- Code reuse by integrating the cleaning web service into the OpenRefine extension
- Implementation of a user friendly frontend interface for the cleaning web service
- Automatic creation of an OpenRefine project from the given RDF data file
- Extension of the cleaning report by statistics that summarise information about quality problems identified. Respectively, the QR Ontology introduced in D3.2 [6] has been extended by the corresponding terms.

A detailed description of the particular changes as well as design decisions and implementation details will be presented below.

5.1 Cleaning Workflow

The Cleaning Module described in D3.2 [6] comprises two separate components – an OpenRefine extension to clean data in interactive way, and the cleaning web service for automatical cleaning. To offer the user a single entry point for both ways of cleaning, we have now integrated the web service into the OpenRefine extension without changing the functionalities of the single components. The updated workflow of the cleaning process including both ways of cleaning is presented in Figure 11.

The user starts cleaning by uploading a data file or entering a URL of a dataset, and then selecting the way he would like to clean data – interactive cleaning using OpenRefine or automatic cleaning by web services (Figure 12). Several RDF serialisations, including Turtle and RDF/XML, are supported as input formats. By selecting “OpenRefine”, a corresponding OpenRefine project will be created from the defined data set, and the user will be forwarded to the OpenRefine perspective. The web service provides two different cleaning methods: either the generation of cleaning suggestions, or deleting the problematic triples from the original data set. More details about the different cleaning methods are presented in sections 5.2 and 5.4.

5.2 Interactive Cleaning with OpenRefine

The main steps of the cleaning workflow using OpenRefine, as initially described in D3.2 [6], remain unchanged:

1. Creation of a project and representation of a data set as a “subject, predicate, object” table. In contrast to the previous cleaning module version, the backend automatically creates an OpenRefine project from the input file provided by the user.

2. Identification of quality problems. In this step the user defines a set of metrics according to his needs. The metric selection interface has changed to accommodate the additional quality metrics that have been added: metrics are grouped by quality categories and dimensions (cf. D5.1 [13] for this terminology); category and dimension sub-trees can be expanded and collapsed, or the user can select a category or dimension in whole. Figure 13 shows the updated version.

\[http://openrefine.org/\]
3. Generation of cleaning suggestions and interactive cleaning. To support the user in the cleaning process for the UndefinedClass and UndefinedProperty problems (i.e. using classes and properties for which no definition exists), we provide the user with concrete examples of classes or properties that could solve this problem. Any class or property that was not found in the vocabularies used (e.g. Persona) is compared with respect to the Levenshtein distance metric to all classes/properties available in the vocabularies used by the dataset. The class/property with the most similar name (e.g. Person) is returned as the most probable match.

4. Export of cleaned data. The list of possible output formats has been extended to the most popular formats. The service now supports the Turtle, RDF/XML, N-Triples, and N3 serialization formats. We also implemented a user interface for this step (Figure 15).

5.3 Technical Documentation

5.3.1 Prerequisites

The cleaning module is implemented as a self-contained extension for OpenRefine and has no technical prerequisites except OpenRefine.

5.3.2 How to install

To use the OpenRefine quality extension, you should follow one of the instruction lists given below.

**Build the extension from source code**

The extension can be installed from scratch by building the OpenRefine project together with the quality
extension. We have tested our implementation with OpenRefine v2.6-rc1 Release Candidate 1⁵.

Step 1. Check out the sources of OpenRefine from their GitHub repository.
git clone https://github.com/OpenRefine/OpenRefine

⁵https://github.com/OpenRefine/OpenRefine/releases/tag/v2.6-rc1
Step 2. The extension must be checked out to the “extension” directory within the OpenRefine root directory.

\textit{git clone https://github.com/diachron/quality-extension.git}

Step 3. The two targets for cleaning and building the quality extension must be added to the ant build automation script (build.xml) in the “extension” directory of OpenRefine. The command

\texttt{<ant dir="quality-extension/" target="build"/>}

must be added to the \texttt{build} target, and

\texttt{<ant dir="quality-extension/" target="clean"/>}

to the \texttt{clean} target correspondingly.

Step 4. The \texttt{/refine build} command must be run to build the OpenRefine project together with the quality extension.

**Install the extension with .zip file**

The other way of installing the quality extension requires neither building the quality extension nor the OpenRefine project. The installation follows a few simple steps:

Step 1. Make sure that the OpenRefine project has already been installed on the local machine.

Step 2. Run OpenRefine and browse to the “workspace” directory by clicking the link at the bottom of the project list.
Step 3. Download the quality-extension.zip file from the quality extension repository and extract into the “extension” directory in the workspace directory.

Step 4. Restart OpenRefine.

5.3.3 How to run

OpenRefine with the installed extension can be run with the command ./refine. This will start OpenRefine on localhost:3333. When a specific port and address is desired, use ./refine -i 0.0.0.0 -p 3333.

5.3.4 Integration of new metrics

The OpenRefine extension can easily be extended with new quality metrics. To have a metric fully integrated into the extension, a developer would have to do the following steps.

- A new metric class must be created in the com.google.refine.quality.metrics package. It must extend the abstract class AbstractQualityMetric by providing the implementation of its abstract methods. The current implementation of this class provides minimal functionality – enough to use the metric within the extension.

  Optionally, the metric can be enriched with a specific implementation of a related quality problem to end up in the cleaning report generated by the metric implementation. This is achieved by extending the QualityProblem class located in the com.google.refine.quality.problems package. The other improvement to the metric and quality problem pair would be enabling auto-cleaning functionality. It would take place after a simple implementation of the AutoCleanable interface, which has a single abstract method getCleanedStatement with a return value of type Statement (from the Jena RDF API).

- To enable the quality metric in the OpenRefine extension, it must be added to a JSON structure in the ./module/scripts/metrics-dialog.js JavaScript file in the form of a nested JSON data structure, for instance, {
  "LabelsUsingCapitals" : "Label using capitals"
}. The key of this data structure must be the name of the metric class.

The above steps are sufficient to add a metric to the extension. When OpenRefine is running, an instance of the new metric class is created dynamically using using Java’s reflection mechanism. This new metric implementation approach that we are now employing facilitates the declaration of new metrics and improves the maintainability and readability of the code.

5.4 The Cleaning Web Service

D3.2 [6] described the RESTful API of the cleaning web service. The cleaning web service contains the two methods getCleaningSuggestions and clean, to which this deliverable provides an update. The functionality of the methods, as specified in D3.1 [5], remains unchanged.

The cleaning web service is now implemented as an OpenRefine extension to improve the maintainability of the cleaning code. Even to install just the web service, it is thus necessary to install OpenRefine according to the instructions given in section 5.3.2; however, this is straightforward. Internally, OpenRefine v2.6-rc1 Release Candidate 1 uses the Jetty 6.1.22 web server and servlet container6.

The HTTP method for the getCleaningSuggestions method has been changed from POST to GET, as the method retrieves information from the server, whereas POST is for updating existing resources. The semantics of the getCleaningSuggestions method does not foresee any manipulation of resources; in our case it merely returns RDF data. The parameters of the getCleaningSuggestions methods have slightly changed. The response

---

6http://www.eclipse.org/jetty/
The implementation of the cleaning part of the exposed web service led to updates of the POST request and response parameters with regard to D3.1 [5].

The typical usage of the cleaning service follows the workflow outlined for the cleaning application in figure 11. The updated user interface is shown in figure 16.

5.4.1 Exposed Web Service Interfaces

Table 2 provides an overview of the cleaning methods.

Note: HttpServletRequest and HttpServletResponse are interfaces in the javax.servlet.http package.

Method Details
### Table 2: Cleaning Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HttpServletRequest</td>
<td>getCleaningSuggestionsREST(HttpServletRequest request, HttpServletResponse response): GET method returning a set of cleaning suggestions.</td>
</tr>
<tr>
<td>HttpServletRequest</td>
<td>cleanREST(HttpServletRequest request, HttpServletResponse response): POST method removing triples affected by selected quality problems.</td>
</tr>
</tbody>
</table>

**getCleaningSuggestionsREST**: GET method returning a set of cleaning suggestions.

**URL (partial)**: /quality_extension/get_cleaning_suggestions

**Parameters**:

- **HTTP Request message**: A JSON-encoded string having the following structure:
  ```json
  { "download": "Dataset", "metrics": ["metric1", "metric2"] }
  ```

Where

**download**: The URL of a file to be validated.

**metrics**: A list of quality metrics with respect to which cleaning suggestions should be generated.

**Returns**: A Response instance which has a JSON-encoded entity content depending on the input parameter of the method. We distinguish the following cases:

- **HTTP status code 400 Bad Request** and the entity content, if the input parameters or one of them are empty:
  ```json
  { "status": "error", "message": "Request parameters are not complete." }
  ```

- **HTTP status code 200 OK** and the entity content, if the input parameters are correct and the get cleaning suggestion method has not failed:
  ```json
  { "status": "ok", "message": "Quality report in one of the RDF serialisations." }
  ```

- **HTTP status code 400 Bad Request** and the entity content, if the input parameters are not correct:
  ```json
  { "status": "error", "message": "Request parameters cannot be parsed." }
  ```

The quality report is created according to the QR (Quality Report) and QPROB (Quality Problems) ontologies presented in D3.2 [6]. An example of the quality report is shown in Listing 1.
We also extended the quality report by statistics that summarize information about identified quality problems and affected triples. The QR ontology has been extended by the corresponding classes and properties represented in Figure 17.

Figure 17: Quality Statistics in the Quality Report Ontology

Listing 1: An example of a quality report

```rdfs
@prefix qr: <http://purl.org/eis/vocab/qr#> .
@prefix qprob: <http://purl.org/eis/vocab/qprob#> .

<quality> qr:cleaning <reportOnCleanedTriples> ;
  qr:hasProblem
    [ a rdfs:Bag ;
      rdfs:label "Labels Using Capitals Problem" ;
      qprob:cleaningSuggestion
        "Change the capitalization of the literal" ;
      qprob:problemDescription
        "Literal uses a bad style of capitalization" ;
      qr:problematicThing
        [ a rdfs:Statement ;
          rdfs:subject <http://aksw.org/NatanaelArndt> ;
          rdfs:predicate <http://www.w3.org/2004/02/skos/core#hiddenLabel> ;
          rdfs:object "NatanaelArndt"
        ]
    ]
  ] .
```

**clean**: a **POST** method to clean a dataset by applying the rule “delete every triple that is affected by a quality problem w.r.t. at least one of the metrics specified”.

**URL (partial):** /quality_extension/clean

**Parameters:**

- **HTTP Request message**: A JSON-encoded string of the following structure:
  ```json
  {
    "download": "Dataset",
  }
  ```
"metrics": ["metric1", "metric2"],
"delta": "true \| false"
}

Where

**Dataset**: The URI of an RDF file to be validated.

**metrics**: A list of metrics with respect to which cleaning suggestions should be generated.

**Returns**: A Response instance which has a JSON encoded entity content depending on the input parameter of the method. We distinguish the following cases:

- **HTTP status code 400 Bad Request** and the entity content, if the input parameters or one of them are empty:
  
  
  ```json
  {  
    "status": "error",  
    "message": "Request parameters are not complete."
  }
  ```

- **HTTP status code 400 Bad Request** and the entity content, if the input parameters are not correct:
  
  ```json
  {  
    "status": "error",  
    "message": "Request parameters cannot be parsed or an error occurred while applying metrics."
  }
  ```

- **HTTP status code 200 OK** and the entity content, if the input parameters are correct and the getCleaningSuggestion method has not failed:
  
  ```json
  {  
    "status": "ok",  
    "uri": "Cleaned RDF data serialized in Turtle.",  
    "delta": "Removed RDF statements serialized in Turtle."
  }
  ```

A further improvement planned for the near future is not to return serialised RDF data, but to implement a server-side “assets store” within the extension, to which additional REST methods would enable access. Once this is done, it would be sufficient for the `clean` method to merely return the URL of the cleaned dataset, which the client would then download from the assets store.
6 Conclusion

This report contains the final version of the descriptions of the WP3 services delivered in M28 of the DIACHRON project. These services include the change detection service, the change monitoring and propagation service, and the cleaning service. The repairing service has also been delivered, but omitted from this report, as it has no major changes compared to what was reported in Deliverable D3.2 [6].

Apart from the technical documentation about the services, we also provided the final version of the specifications, when relevant, with emphasis on any differences with respect to what was reported in previous deliverables. The services will be formally deployed as part of the integrated DIACHRON environment for Deliverable D6.5 in M30.
References


A Complex Change Creation

In this section we provide a more detailed description of the `defineCCJSON` method which is available under (POST) service `/diachron/complex_change` defined in deliverable D3.2. This service requires a post JSON-encoded message which represents the complex change definition and contains all the required information for its proper definition. Under the new implementation, we made some changes in the format of the JSON message. An indicative JSON message for a complex change definition would be the following:

```
{
    "Complex_Change": "Add Definition",
    "Priority": 1.0,
    "Description": "Definitions which were added on entities."
    "Simple_Changes": [
        {
            "Simple_Change": "ADD_PROPERTY_INSTANCE",
            "Simple_Change_Uri": "1:ADD_PROPERTY_INSTANCE",
            "Is_Optional": false,
            "Selection_Filter": "1:ADD_PROPERTY_INSTANCE:-property = 
<http://www.ebi.ac.uk/efo/definition>",
            "Join_Filter": ""
        }
    ],
    "Complex_Change_Parameters": [
        {
            "subject": "1:ADD_PROPERTY_INSTANCE:-subject"
        },
        {
            "definition": "1:ADD_PROPERTY_INSTANCE:-object"
        }
    ],
    "Version_Filters": [
        {
            "Subject": "1:ADD_PROPERTY_INSTANCE:-subject",
            "Predicate": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            "Object": "http://www.w3.org/1999/02/22-rdf-syntax-ns#Class",
            "Presence": "EXISTS_IN_V2"
        }
    ]
}
```

Next, we analyse the field of the above JSON message and we dictate the values they can take.

**Complex_Change**: The name of the complex change

**Priority**: The priority of the complex change which is a double number. Lower priority values denote more important complex changes thus, these changes are examined first.

**Description**: A human understandable description of the complex change.

**Simple_Changes**: An array of JSON objects which refers on the simple changes which consist said complex change. Each complex change JSON object consists of the following elements:
Simple Change: A unique name of the complex change. We use specific names for the simple changes defined in both the ontological and multidimensional Diachron model. Table 3 shows the change names for the ontological model and Table 4 shows the change names for the multidimensional model along with the parameter names of each simple change which will be also used in the definition.

Simple Change Uri: A unique identifier which is created by appending a serial number (e.g., 1,2,3, ...) with the simple change name in order to separate multiple simple changes with the same name.

Is Optional: A boolean flag which indicates whether the existence of said simple change is optional (true) or mandatory (false) for the detection of the corresponding complex change.

Selection Filter: A filter which assigns selective values upon simple change parameters. Every filter is an equation which has as left part a parameter name of the respective simple change and as a right part a specific URI or literal value. Last, URI values must be enclosed in symbols "<", ">" whereas literal values must be enclosed in equations "", ". Selection filters are not mandatory for the definition of complex changes.

Join Filter: A filter which joins different simple change parameter. In other words, consider a Selection Filter in which the right part of the equation is a parameter of another simple change. In order to separate simple change parameters we append them after each corresponding simple change URI. Join filters are also not mandatory for the definition of complex changes.

Complex Change Parameters: An array of JSON objects in which each objects maps a complex change parameter name (e.g., definition) a simple change parameter name. Again, the simple change parameter name, must append the simple change URI from which it originates. We should mention here that each defined complex change must have at least one complex change parameter. However, it is not necessary that the complex change parameter will be mapped to a simple change parameter. If we map it with an empty string (i.e., "") then the parameter is expected to appear in a version filter. As a result, it will take value from the respective version filter as well.

Version Filters: An array of JSON objects which represent triple patterns which must be satisfied from either the old version or the new version. The presence of the triple pattern is denoted using the field “Presence” and it can take one of the values: EXISTS_IN_V1, EXISTS_IN_V2, NOT_EXISTS_IN_V1, NOT_EXISTS_IN_V2. The triple pattern may consist of a) simple change parameters, b) complex change parameters as mentioned before and c) specific URI values as the ones used in selection filters. Again, we remind that simple change parameters must be accompanied by the corresponding simple change URI.

Table 3: Ontological Simple Changes Codelist

<table>
<thead>
<tr>
<th>Change Name</th>
<th>Parameter Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_COMMENT</td>
<td>subject</td>
</tr>
<tr>
<td></td>
<td>comment</td>
</tr>
<tr>
<td>ADD_DOMAIN</td>
<td>property</td>
</tr>
<tr>
<td></td>
<td>domain</td>
</tr>
<tr>
<td>ADD_LABEL</td>
<td>subject</td>
</tr>
<tr>
<td></td>
<td>label</td>
</tr>
<tr>
<td>ADD_PROPERTY_INSTANCE</td>
<td>subject</td>
</tr>
<tr>
<td>ADD_PROPERTY_INSTANCE</td>
<td>property</td>
</tr>
<tr>
<td></td>
<td>object</td>
</tr>
<tr>
<td>ADD_RANGE</td>
<td>property</td>
</tr>
<tr>
<td></td>
<td>range</td>
</tr>
<tr>
<td>Change Name</td>
<td>Parameter Name</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>ADD_ATTRIBUTE</td>
<td>attribute</td>
</tr>
<tr>
<td>ADD_CODELIST</td>
<td>codelist</td>
</tr>
<tr>
<td>ADD_DATATYPE</td>
<td>subject</td>
</tr>
<tr>
<td>ADD_DIMENSION</td>
<td>dimension</td>
</tr>
<tr>
<td>ADD_DIMENSION_VALUE_TO_OBSERVATION</td>
<td>dimension</td>
</tr>
<tr>
<td>ADD_FACT_TABLE</td>
<td>fact_table</td>
</tr>
<tr>
<td>ADD_GENERIC_ATTRIBUTE</td>
<td>subj</td>
</tr>
<tr>
<td>ADD_GENERIC_ATTRIBUTE</td>
<td>attribute</td>
</tr>
<tr>
<td>ADD_GENERIC_VALUE_TO_OBSERVATION</td>
<td>value</td>
</tr>
<tr>
<td>ADD_HIERARCHY</td>
<td>hierarchy</td>
</tr>
<tr>
<td>ADD_INSCHEME</td>
<td>scheme</td>
</tr>
</tbody>
</table>

Table 4: Multidimensional Simple Changes Codelist
<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_INSTANCE</td>
<td>instance</td>
</tr>
<tr>
<td>ADD_INSTANCE_TO_PARENT</td>
<td>instance</td>
</tr>
<tr>
<td>ADD_LABEL</td>
<td>subj_label</td>
</tr>
<tr>
<td>ADD_MEASURE</td>
<td>measure</td>
</tr>
<tr>
<td>ADD_MEASURE_VALUE_TO_OBSERVATION</td>
<td>measure</td>
</tr>
<tr>
<td>ADD_OBSERVATION</td>
<td>observation</td>
</tr>
<tr>
<td>ADD_RELEVANCY</td>
<td>lst_arg</td>
</tr>
<tr>
<td>ADD_UNKNOWNPROPERTY</td>
<td>subj_un_prop</td>
</tr>
<tr>
<td>ATTACH_ATTR_TO_DIMENSION</td>
<td>dimension</td>
</tr>
<tr>
<td>ATTACH_ATTR_TO_MEASURE</td>
<td>measure</td>
</tr>
<tr>
<td>ATTACH_CODELIST_TO_DIMENSION</td>
<td>dimension</td>
</tr>
<tr>
<td>ATTACH_DATATYPE_TO_DIMENSION</td>
<td>dimension</td>
</tr>
<tr>
<td>ATTACH_DIMENSION_TO_FT</td>
<td>dimension</td>
</tr>
<tr>
<td>ATTACH_HIERARCHY_TO_DIMENSION</td>
<td>dimension</td>
</tr>
<tr>
<td>ATTACH_INSTANCE_TO_CODELIST</td>
<td>codelist</td>
</tr>
<tr>
<td>ATTACH_INSTANCE_TO_HIERARCHY</td>
<td>hierarchy</td>
</tr>
<tr>
<td>ATTACH_INSTANCE_TO_PARENT</td>
<td>instance</td>
</tr>
<tr>
<td>ATTACH_MEASURE_TO_FT</td>
<td>measure</td>
</tr>
<tr>
<td>ATTACH_OBSERVATION_TO_FT</td>
<td>observation</td>
</tr>
<tr>
<td>ATTACH_TYPE_TO_MEASURE</td>
<td>type</td>
</tr>
<tr>
<td>DELETE_ATTRIBUTE</td>
<td>attribute</td>
</tr>
<tr>
<td>DELETE_CODELIST</td>
<td>codelist</td>
</tr>
<tr>
<td>DELETE_DATATYPE</td>
<td>subject</td>
</tr>
<tr>
<td>DELETE_DATATYPE_VALUE_FROM_OBSERVATION</td>
<td>dimension</td>
</tr>
<tr>
<td>DELETE_DIMENSION</td>
<td>dimension</td>
</tr>
<tr>
<td>DELETE_DIMENSION_VALUE_FROM_OBSERVATION</td>
<td>dimension</td>
</tr>
<tr>
<td>DELETE_FACT_TABLE</td>
<td>fact_table</td>
</tr>
<tr>
<td>DELETE_GENERIC_ATTRIBUTE</td>
<td>subj</td>
</tr>
<tr>
<td>Grant Agreement No. 601043</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DELETE_GENERIC_VALUE_FROM_OBSERVATION</th>
<th>attribute</th>
<th>observation</th>
<th>property</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE_HIERARCHY</td>
<td>hierarchy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE_INSHEME</td>
<td>scheme</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE_INSTANCE</td>
<td>instance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE_INSTANCE_FROM_PARENT</td>
<td>instance</td>
<td></td>
<td>parent</td>
<td></td>
</tr>
<tr>
<td>DELETE_LABEL</td>
<td>subj_label</td>
<td>prop_label</td>
<td>obj_label</td>
<td></td>
</tr>
<tr>
<td>DELETE_MEASURE</td>
<td>measure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE_MEASURE_VALUE_FROM_OBSERVATION</td>
<td>measure</td>
<td>measure_val</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE_OBSERVATION</td>
<td>observation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE_RELEVANCY</td>
<td>subject</td>
<td>1st_arg</td>
<td>2nd_arg</td>
<td>datatype</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE_UNKNOWN_PROPERTY</td>
<td>subj_un_prop</td>
<td>prop_un_prop</td>
<td>obj_un_prop</td>
<td></td>
</tr>
<tr>
<td>DETACH_ATTR_FROM_DIMENSION</td>
<td>dimension</td>
<td>attribute</td>
<td>datatype</td>
<td></td>
</tr>
<tr>
<td>DETACH_ATTR_FROM_MEASURE</td>
<td>measure</td>
<td>attribute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETACH_CODELIST_FROM_DIMENSION</td>
<td>dimension</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETACH_DATATYPE_FROM_DIMENSION</td>
<td>dimension</td>
<td>attribute</td>
<td>datatype</td>
<td></td>
</tr>
<tr>
<td>DETACH_DIMENSION_FROM_FT</td>
<td>dimension</td>
<td>fact_table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETACH_HIERARCHY_FROM_DIMENSION</td>
<td>dimension</td>
<td>hierarchy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETACH_INSTANCE_FROM_CODELIST</td>
<td>codelist</td>
<td>hierarchy</td>
<td>instance</td>
<td></td>
</tr>
<tr>
<td>DETACH_INSTANCE_FROM_PARENT</td>
<td>instance</td>
<td></td>
<td>parent</td>
<td></td>
</tr>
<tr>
<td>DETACH_INSTANCE_FROM_HIERARCHY</td>
<td>hierarchy</td>
<td>instance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETACH_MEASURE_FROM_FT</td>
<td>measure</td>
<td>fact_table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETACH_OBSERVATION_FROM_FT</td>
<td>observation</td>
<td>fact_table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DETACH_TYPE_FROM_MEASURE</td>
<td>type</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>