**EU Project No:601043 (Integrated Project (IP))**

**DIACHRON**

<table>
<thead>
<tr>
<th>Dissemination level:</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of Document:</td>
<td>Report</td>
</tr>
<tr>
<td>Contractual date of delivery:</td>
<td>M16</td>
</tr>
<tr>
<td>Actual Date of Delivery:</td>
<td>M16</td>
</tr>
<tr>
<td>Deliverable Number:</td>
<td>D3.2</td>
</tr>
<tr>
<td>Deliverable Name:</td>
<td>Specification and Implementation of Monitoring, Synchronization and Repairing Services</td>
</tr>
<tr>
<td>Deliverable Leader:</td>
<td>FORTH</td>
</tr>
<tr>
<td>Work package(s):</td>
<td>WP3</td>
</tr>
<tr>
<td>Status &amp; version:</td>
<td>Final version 1.0</td>
</tr>
<tr>
<td>Number of pages:</td>
<td>68</td>
</tr>
<tr>
<td>WP contributing to the deliverable:</td>
<td>WP3</td>
</tr>
<tr>
<td>WP / Task responsible:</td>
<td>WP3: T3.1, T3.2, T3.3</td>
</tr>
<tr>
<td>Coordinator (name / contact):</td>
<td>Giorgos Flouris (<a href="mailto:fgeo@ics.forth.gr">fgeo@ics.forth.gr</a>)</td>
</tr>
<tr>
<td>Other Contributors:</td>
<td>Yannis Roussakis (FORTH), Ioannis Chrysakis (FORTH), Michalis Chortis (FORTH), Kostas Stefanidis (FORTH), Christos Pateritas (ATHENA), Natalja Friesen (UBONN), Christoph Lange (UBONN)</td>
</tr>
<tr>
<td>EC Project Officer:</td>
<td>Federico Milani</td>
</tr>
</tbody>
</table>

**Keywords:** Implementation, Specification, Changes, Change Definition, Change Representation, Change Detection, Monitoring, Propagation, Repair, Cleaning

**Abstract:**

This report includes technical documentation regarding the implementation of the WP3 services (detection, repairing, cleaning), as well as the final version of the specifications, with emphasis on any differences with respect to what was reported in Deliverable D3.1.
<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 20th, 2014</td>
<td>Giorgos Flouris</td>
<td>Table of contents, initial assignments</td>
</tr>
<tr>
<td>0.2</td>
<td>July 9th, 2014</td>
<td>Michalis Chortis</td>
<td>Algorithms</td>
</tr>
<tr>
<td>0.3</td>
<td>July 10th, 2014</td>
<td>Giorgos Flouris</td>
<td>Various improvements in all sections</td>
</tr>
<tr>
<td>0.4</td>
<td>July 16th, 2014</td>
<td>Giorgos Flouris, Ioannis Chrysakis, Yannis Roussakis, Michalis Chortis</td>
<td>First draft of Introduction, Detection and Repairing sections</td>
</tr>
<tr>
<td>0.5</td>
<td>July 18th, 2014</td>
<td>Kostas Stefanidis</td>
<td>Proof-read sections 1, 2 and 3</td>
</tr>
<tr>
<td>0.6</td>
<td>July 21st, 2014</td>
<td>All involved</td>
<td>Finalized Section 4, and proof-read all sections</td>
</tr>
<tr>
<td>0.9</td>
<td>July 23rd, 2014</td>
<td>Kostas Stefanidis</td>
<td>Final Proof-read and submission for internal review</td>
</tr>
</tbody>
</table>
# Table of Contents

1 **Introduction** ................................. 5  
1.1 Change Detection Service ......................... 5  
1.2 Change Monitoring and Propagation Service ........ 5  
1.3 Repairing Service ................................... 5  
1.4 Cleaning Service ................................. 6  

2 **The Detection Service** .......................... 7  
2.1 Service Description .................................. 7  
2.1.1 Change Types ..................................... 7  
2.1.2 Assumptions ..................................... 7  
2.1.3 Complex Changes Definition Interface ............. 8  
2.1.4 Ontology of Changes .............................. 9  
2.1.5 Querying Changes .................................. 12  
2.2 Technical Documentation ............................ 13  
2.2.1 Prerequisites .................................... 13  
2.2.2 Deployment Process ............................... 16  
2.2.3 Required Web Service Interfaces ................. 17  
2.2.4 Exposed Web Service Interfaces .................. 18  
2.2.5 Exposed User Interfaces .......................... 22  

3 **The Repairing Service** .......................... 24  
3.1 Service Description .................................. 24  
3.1.1 Diagnosis Component .............................. 24  
3.1.2 Repairing Component ............................. 27  
3.2 Technical Documentation ............................ 28  
3.2.1 Prerequisites .................................... 28  
3.2.2 Deployment Process ............................... 29  
3.2.3 Required Web Service Interfaces ................. 30  
3.2.4 Exposed Web Service Interfaces .................. 30  
3.2.5 Exposed User Interfaces .......................... 32  

4 **The Cleaning Service** ............................ 36  
4.1 Service Description .................................. 36  
4.1.1 Cleaning Workflow ............................... 37  
4.1.2 Quality Problem Identification ................... 38  
4.1.3 Generation of Cleaning Suggestions ............... 40  
4.1.4 Transformation Engine ............................ 42  
4.2 Technical Documentation ............................ 42  
4.2.1 Prerequisites .................................... 42  
4.2.2 Deployment Process ............................... 42  
4.2.3 Required Web Service Interfaces ................. 42  
4.2.4 Exposed Web Service Interfaces .................. 42  
4.2.5 Exposed User Interfaces .......................... 44  
4.2.6 Extension Points ................................. 44  

5 **Conclusion** ....................................... 47  

A **Low-level Changes** .............................. 49  

B **Simple Changes for the Ontological Model** .......... 50
TABLE OF FIGURES

1 Mappings Representation .................................................. 10
2 Example of a Mapping Representation .................................. 10
3 Top-level Concepts .......................................................... 10
4 Ontological Structures for Representing the Definition of Simple Changes ........................................... 11
5 Simple Changes Detection .................................................. 12
6 Ontological Structures for Representing the Definition of Complex Changes ........................................... 13
7 Complex Change Detection .................................................. 14
8 Virtuoso Universal Server Architecture ................................. 15
9 Virtuoso Sesame Provider Architecture ................................. 16
10 Properties File ............................................................... 17
11 User Interface for Complex Change Creation ......................... 22
12 User Interface for Complex Change Deletion ......................... 23
13 The main features of the diagnosis and repair framework ............ 24
14 Example of an interdependency graph ................................... 26
15 Virtuoso Jena Provider Architecture .................................... 29
16 Input UI for the validation method ....................................... 32
17 UI for the results of the validation method execution ................. 33
18 UI for the results of the validation method execution (with returned invalidities) ....................................... 33
19 Input UI for the repair method ............................................. 34
20 UI for the results of the repair method execution ....................... 34
21 UI for the results of the repair method execution (with returned repairing delta) .......................................... 35
22 Cleaning process workflow .................................................. 38
23 The Diachron Quality extension menu to OpenRefine .................. 38
24 Selecting quality metrics for which problems should be identified ...................................................... 39
25 Dataset with the corresponding quality problems and cleaning suggestions ........................................... 39
26 Data filtered by a particular quality problem. Quality problems, sorted by frequency, are displayed in the left panel ................................................................. 39
27 Data transformation view ..................................................... 40
28 General structure of the Quality Report ontology ...................... 40
29 Quality Problem Class of the QR vocabulary ......................... 41
30 Quality problem representation in the QR Vocabulary ................ 41

LIST OF TABLES

1 Complex Change Methods .................................................. 18
2 Change Detection Methods .................................................. 20
3 Transformation of $DL$-Lite$_A$ constraints to FOL queries ............... 26
4 Validation Methods .......................................................... 30
5 Repair Methods .............................................................. 31
6 Quality Problems ............................................................ 45
7 Cleaning Methods ............................................................ 46
1 Introduction

This is a technical deliverable, in which we provide technical documentation regarding the implementation of the first version 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 Deliverable D3.1 [5]. The services 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/detection_repair). Moreover, we have publicly deployed the services under URL http://139.91.183.93:8181 (Apache Tomcat 7.0.54) as a stand-alone module. However, they will also be formally deployed as part of the integrated DIACHRON environment for Deliverable D6.2 in M20. In the following subsections, we describe the services that will be considered in this deliverable (and have been promised in the context of WP3), and a reference to their detailed description in this report. The appendix contains the up-to-date listings of the changes considered in DIACHRON.

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 definitions of the changes listed in the appendices, and its output should store the detected changes in the ontology of changes (cf. [5] and Subsection 2.1).

The change detection service is described in Section 2. 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 Subsection 2.1.

1.2 Change Monitoring and Propagation Service

The change 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 where a dataset is periodically inspected for changes. Change detection is performed by employing the Change Detection module functionality. The user may configure the task by declaring rules on the types of changes and the dataset resources she wishes to monitor. Other users 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 and D6.1 and, as described in these documents, the change monitoring and propagation service is heavily dependent upon the archive and the change detection services. Therefore, the implementation process and especially the testing and the integration phases of the service need to invoke the stable versions of these relying services. The first prototype versions of the archive and change detection services are scheduled to be delivered in M16 of the project. As a result the delivery of the monitoring and propagation service will follow this date and thus be included in the second prototype of the WP3 services scheduled in M28.

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 [5]. This deliverable contains a listing of the related algorithms, as well as some technical documentation on the implementation of the service (see Section 3).
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 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]; this deliverable (in particular, Section 4) provides its technical documentation.
2 The Detection Service

2.1 Service Description

This subsection describes the change detection service. Note that we have made several improvements compared to the specifications described in D3.1 [5], which were dictated by interaction with the partners, efficiency reasons, or technical problems. The detailed description of these improvements are also included here. Familiarity with the contents of D3.1 is assumed.

2.1.1 Change Types

In Deliverable D3.1, we had defined four different levels of changes, namely low-level, basic, simple and complex. Low-level changes represent changes at the triple level and constitute the basis upon which the detection process is based. As explained in D3.1, there are two low-level changes, namely add(t) and delete(t), where t is a triple. The detection of these changes uses the SPARQL queries described in Appendix A (cf. D3.1 [5]).

Basic changes were introduced as a layer to achieve completeness and unambiguity in the change detection process; in a nutshell, completeness guarantees that all changes that happen will be reported, whereas unambiguity guarantees that no change will be reported twice (see D3.1 for more details on these notions). Nevertheless, basic changes were subsequently dropped from our modelling, because the completeness and unambiguity could be easily guaranteed by the definition of simple changes. Therefore there was no longer need for basic changes. In addition, the removal of basic changes from our model made it simpler (three levels as opposed to four) and more efficient.

At the time of writing of Deliverable D3.1, simple changes were still vague; in the present deliverable, we provide the definitive listing of simple changes for the RDF and multidimensional models used by the pilots (see Appendix B and C, respectively). The two listings are different, as different deployments (and data models) will use different simple changes. Both lists have been incorporated in the change detection service implementation. These simple changes were defined in cooperation with the pilots, and are provably complete and unambiguous. Note that for each pilot, the corresponding set of simple changes should be predefined upon the deployment of the Change Detection module and it remains constant during the life-cycle of the module. Subsection 2.2.2 describes, among others, how the simple changes are predefined and “passed” into the Change Detection module.

Complex changes represent custom, user-defined changes, which are defined dynamically (at run-time) by the user. Complex changes are detected as described in D3.1. In this deliverable, we specify the information that the user should provide in order to define a complex change, and illustrate a mockup interface for this information to be submitted (see Subsection 2.2 below). Even though this interface is not strictly under the hood of the change detection service (and WP3), we chose to describe it here for completeness.

2.1.2 Assumptions

The Change Detection module and its corresponding services were implemented under certain assumptions that are met by the DIACHRON model. These assumptions were agreed upon with the pilot partners and the partner ATHENA, who is responsible for the underlying storage model.

Firstly, the change detection module will consider datasets that were provided by the data pilots and have already been ingested and transformed into the corresponding DIACHRON model in a complete and consistent way. For instance, the ontological data which are provided by EMBL were transformed, as dictated in Deliverable D4.1, into the corresponding DIACHRON model via Ontological-to-DIACHRON mappings. As a result, the Change Detection module is agnostic to the original pilot’s data and their internal representation.

One more assumption we considered during the implementation phase is the form that the object URIs should have within DIACHRON platform. More specifically, we assumed that URIs are constructed in such way that when we refer on the “same” objects (i.e., classes, properties) across different versions, their URIs will remain persistent as well. Thus, an entity retains the same URI across different versions, thereby being dereferenceable across all versions. Note that this is required for all pilots. This feature is achieved via the transformation mechanism from the
original pilots’ datasets to DIACHRON data by ignoring the datasets’ version numbers during the transformation process.

2.1.3 Complex Changes Definition Interface

In this subsection, we describe the required input for managing complex changes. We first focus on the most important operation of creating (i.e., defining) a complex change, and then discuss other operations (namely, deleting and editing complex changes). In all cases, we explain what is the required input that the user interface should provide in order to allow the change detection service to manipulate the corresponding change. The mockup for providing this information appears in 2.2.

Creating Complex Changes When creating a new complex change, the following information should be requested via the user interface and provided to the internal function that creates the complex change:

Name: A descriptive name for the complex change (mandatory). Names of changes should be unique, i.e., a new complex change could not have the same name as a previously defined simple or complex change.

Priority: A float number denoting the priority of the change (to help solve ambiguities during the change detection phase – cf. D3.1 [5]); this is also mandatory.

Parameter Names: An ordered set of parameters, defined through their descriptive names (and denoted by $P_1^*, \ldots, P_n^*$). Parameter names should be unique within each change, but it is possible for different (simple or complex) changes to share the same parameter name. It is allowed, though not recommended, for this set to be empty, i.e., for a change to have no parameters.

Mandatory Simple Changes: A set of selected simple change(s) over the total set of simple changes per pilot deployment, that comprise the specified complex change. Note that each simple change has a name ($S_1, \ldots, S_m$), as well as its own parameters ($P_{ij}$, denoting the $j^{th}$ parameter of $S_i$). There must be at least one mandatory simple change.

Optional Simple Changes: A set of selected simple change(s) over the total set of simple changes per pilot deployment that could be optionally “consumed” during the detection of the specified complex change. It is possible that there are no optional simple changes.

Parameter Filters: For each parameter $P_{ij}^*$ of the complex change, we should apply a selection filter (equality operator) over a simple change parameter $P_{ij}$. Thus, a parameter filter is of the form: $P_{ij}^* = P_{ij}$, where $P_{ij}^*, P_{ij}$ are the names for the complex change parameter and simple change parameter, respectively; note that the simple change could be mandatory or optional. This filter associates a complex change parameter with a simple change parameter. Parameter filters are mandatory, i.e., there should be exactly one parameter filter per complex change parameter.

Selection Filters: Different types of selection filters on specified parameters could be optionally added by the user. These should have the form $P_{ij} \ast X$ where $\ast$ can be any SPARQL comparison operator (e.g., $=, >$ etc.) and $X$ can be any URI or literal supported by SPARQL.

Join Filters: Additionally, join filters across different parameters of simple changes can be optionally added. These should be of the form $P_{ij} = P_{i'j'}$, where $P_{ij}, P_{i'j'}$ are different parameter names of the mandatory or optional simple changes comprising the complex one.

Mapping Filters: Some complex changes are based on mappings that express that two different URIs (in different versions) represent the same real-world concept; this is necessary to encode certain types of changes, such as those that represent rename, merge or split operations. To express the fact that such a mapping should exist for a complex change to be detectable, mapping filters are introduced. Such filters may have one of the following forms: $P_{ij} \rightarrow P_{i'j'}$ or $P_{ij} \rightarrow \{P_{i'j'_1}, \ldots, P_{i'j'_n}\}$ or $\{P_{ij_1}, \ldots, P_{ij_m}\} \rightarrow P_{i'j'}$. Note that not all complex changes have mapping filters, i.e., this field is optional.
Version Filters: These filters are used to (optionally) express conditions over triples or URIs that should (or should not) exist in the old and/or the new version. Such filter(s) can be of one of the following forms: “x IN V” or “x NOT IN V” where x can be either a URI or a triple, and V can be either the old or the new version (denoted by $V_{old}, V_{new}$ respectively).

All the above filters are understood conjunctively, but filters combined via disjunction (or a combination of disjunction and conjunction) are also technically possible. However, it is an open issue whether this functionality could be useful for the user, as we believe that this sophisticated version could overwhelm the user with many options. For this reason, disjunctive filters are not supported in this version of the implementation, but their usability will be evaluated in the near future in order to determine whether to include this functionality in the next version of the implementation.

Example 1. The following is an intuitive example which will help to visualize and better understand the process, via clarifying how the above input requirements can be materialized to define a complex change. In particular, we will define the change “Mark as Obsolete”, which is used to encode/capture a standard pattern of changes that the EBI pilot uses when they want to delete a class; in this case, the class is not physically deleted, but marked as “obsolete”, via a label. Formally, the change can be defined as follows:

Name: Mark as Obsolete

Priority: 2

Parameter Names: obs_class, obs_reason

Mandatory Simple Changes: Add_Superclass(subclass, superclass)

Optional Simple Changes: Add_Label(class, label), Add_Property_Instance(subj, prop, obj)

Parameter Filters: obs_class = subclass, obs_reason = obj

Selection Filters: superclass = “ObsoleteClass”, prop = “reason_for_obsolescence”

Join Filters: class = subclass, subj = subclass

Mapping Filters: -

Version Filters: -

Deleting Complex Changes When deleting a complex change, all we need is the name of the deleted complex change. The deletion of a complex change eliminates all references of said change from the ontology of changes, including previous detections.

Editing Complex Changes In this version, editing a complex change is implemented as a deletion followed by a creation of a change. In the second version of the DIACHRON platform (due on M28), we will consider a more efficient implementation.

2.1.4 Ontology of Changes

Ontologies provide the functionality of maintaining (sharing and reuse) linked datasets and allow the flexible tracking of changes with support of query capabilities in a uniform manner. In addition, the ontological representation provides a supervisory look of the detected changes (either simple or complex) and their association with the entities they refer to in the actual datasets. Moreover, this representation provides useful information about the definition of changes.

Based on this motivation, in Deliverable D3.1 [5], we described an ontology of changes for storing the detected changes in a convenient format, which allows the association of information on the changes with the actual data;
this ability is the cornerstone for supporting cross-snapshot queries, for supporting queries that deal with both the data and the changes, and for treating changes as first-class citizens. An additional role for the ontology of changes is to provide a storage mechanism for the definitions (i.e., specifications) of the user-defined complex changes.

Since the delivery of D3.1 in M12 of the project, the ontology of changes underwent certain revisions, dictated by technical problems identified, feedback from pilots and other partners, as well as by the revisions performed in the structure of changes that were described above. Below, we list the revisions made on the ontology of changes as compared to what was reported in D3.1; the new version of the ontology is visualized in Figures 1-7:
1. As basic changes were dropped from our modelling, they were also dropped from the ontology (see Figure 3).

2. For each simple and complex change, as well as for each simple and complex change parameter, we added a property named "name" to hold a user defined descriptive name (see Figures 4-7); this was necessary to allow a more intuitive interaction with the user during the construction of a complex change. For the parameter names in particular, the "name" property has as domain the property that connects the parameter with its value (see Figures 4-7); this was done for efficiency purposes.

3. The "consumes" property was added to denote the fact that certain simple changes are subsumed by complex ones (in technical terms, the detection of a complex change consumes said simple changes – cf. [5] and Figures 6, 7).

4. We simplified the representation of the definition of complex changes, keeping only the information that is absolutely relevant and critical for the system. In particular, the properties "refers_to", "visible", "filter", ...
"comprise_of_mand" and "comprise_of_opt" were dropped, because the information they represent was embedded in the SPARQL query which is used for the detection of said complex change (see Figures 6, 7).

5. A generic "notes/metadata" property was added to changes. This property is not used at this stage, but is reserved for future use in order to store notes related to the change such as an intuitive description, or provenance information.

6. We enhanced the ontology, so as to be able to store mappings between URIs residing in different versions of datasets; these are necessary for the definition of some complex changes that represent rename, merge or split operations (cf. Subsection 2.1.3). Mappings are stored as instances of a new class called "Map", as shown in Figure 1. We use the properties "old_version", "new_version" to denote in which dataset versions a mapping refers to, and the properties "old_value", "new_value" to link the URI(s) in the old and new version. Moreover, we provide the property "notes/metadata" (as with changes) to store any high level information related to the mappings (e.g., notes, provenance etc). An example of storing mappings is illustrated in Figure 2, the left part of which shows a one-to-one mapping (x1→x2) from v1 to v2 (useful for renames), whereas the right part shows a one-to-many mapping (y1→{y2,y3}) from v1 to v2 (useful for splits).

7. We replaced some classes that functioned as ranges for certain properties with more generic ones; in particular, for generic RDF resources, related to data, the classes "rdfs:Class", "rdf:Property" etc were replaced by the more general "rdfs:Resource", and the new class "diachron:Entity" was used as a generic container for DIACHRON entities (see Figures 4–7).

8. The central class Change of the ontology was defined to be a subclass of prov:activity (see the upcoming Deliverable D2.2, due on M16).

2.1.5 Querying Changes

As mentioned above, the ontology of changes was introduced in order to allow the user to perform queries on both the data and the changes, thereby treating changes as first-class citizens, which is one of the main objectives of DIACHRON. To support the user in this task, we created a method allowing query access to the ontology of changes. This GET method is implemented within service /diachron/change_detection and requires the following parameters:
1. the SPARQL query which will be applied upon the ontology of changes
2. the format of the query results i.e., xml, csv, tsv, json

More details on this method can be found in Subsection 2.2.

2.2 Technical Documentation

As already mentioned, 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.

2.2.1 Prerequisites

**Virtuoso RDF Store** The Change Detection Module relies on the existence of a specific RDFStore used to store and manage pilots’ data converted into DIACHRON model. The current version of the module exploits the Virtuoso Universal Server\(^1\), which is an hybrid of a database engine and a middleware that combines the functionality of a

\(^1\)http://virtuoso.openlinksw.com
traditional RDBMS, an ORDBMS, an RDFStore, a virtual database, a web application server and a file server. It is actually a single threaded server process that supports multiple protocols. The following standard Web and Internet protocols have been implemented in Virtuoso: HTTP, HTTPs, WebDav, SOAP, UDDI, SPARQL and SPARUL. In addition, concerning the development of database-based applications and the integration of systems, Virtuoso has implemented a wide variety of industry standard data access APIs, such as ODBC, JDBC, OLE DB and ADO.NET. Virtuoso is made up of various server and client components, which enable the communication of a local or remote Virtuoso server, such as the Conductor, which is Web based Database Administration User Interface, and ISQL and ISQLO utilities. To this end, Virtuoso Universal Server can be exploited to produce a clustered-based system of RDFStores. In addition, there is a specific version of this server, which can be deployed on the Amazon Cloud.

The architecture of the Virtuoso Universal Server is shown in Figure 8. As depicted, Virtuoso exhibits various endpoints (such as a SPARQL endpoint) and has implemented various APIs, which can be exploited for performing a variety of data management operations. The main component in the architecture is the server itself, which comprises two main sub-components: Virtual Database Engine and the Unified Storage Engine. The Virtual Database Engine is responsible for the management of various types of databases, such as SQL and XML databases as well as RDF stores and comprises a free text engine, exploited to perform free text queries on the databases and RDF stores, and particular web services enabling access to these databases and stores. The Unified Storage Engine is responsible for the unified storage of the data in the databases and stores. Another component is the application logic sources which are used for deploying particular application through respective server extensions. In the left part of the architecture, it can be seen which types of databases are currently supported through the interaction with the JDBC and ODBC protocols. Finally, at the bottom part of the architecture, it can be seen that virtuoso actually offers a virtualization of various heterogeneous data sources, such as data spaces and web data sources.

Concerning the specific characteristics of Virtuoso that are of interest to this project, it can be highlighted that Virtuoso not only allows the management of RDF (linked-)data but it enables their querying through the SPARQL language. The same language (actually SPARUL) can be exploited for the updating of the linked-data.
Sesame API  Virtuoso can be used in conjunction with particular RDF Data Management APIs to enable access and management of linked data. Three APIs are currently supported, namely Jena², Sesame³, and Redland⁴. In terms of the implementation of the Change Detection module in this project, the Sesame API was exploited. The architecture of the Virtuoso Sesame Provider, which is a native graph model storage provider for the Sesame framework, is depicted in Figure 9. This provider allows leveraging the Sesame framework in order to enable the linked data management of a Virtuoso RDF store using the Java programming language.

As it can be seen from Figure 9, the Repository API is the main API that can be exploited by an application for importing, exporting, querying, and updating RDF data. This API abstracts all other (low-level) programming details that are handled by other more specific APIs, such as the JDBC Driver Manager and the Virtuoso JDBC driver. In this respect, the developer of the application is equipped with the right abstractions that enable him

²https://jena.apache.org
³http://www.openrdf.org
⁴http://librdf.org
to easily import and export linked data in various forms, as well as to query the RDF store using the SPARQL language and to update the linked data stored using the SPARUL language.

**Implementation Analysis** Based on the above analysis, the Change Detection module was developed by exploiting the Virtuoso Sesame Provider (API). In addition, in order to be exposed as a rest-based web service and have the appropriate rich security mechanisms, the jersey API was exploited.

The jersey API enables not only the deployment of java interfaces as rest-based web services (through the reference implementation of JAX-RS (JSR-311) and the incorporation of the right annotations in the respective Java code), but it also provides a server-side API enabling the retrieval and management of the content of the (http) request, the production of the respective response, the creation of (web application) exceptions, the determination of the life-cycle of the root server classes, and the injection of crucial information, such as security information. However, the injection mechanism does not comply with the standard JSR-330 annotations. Another interesting feature of jersey is that it supports the JSON format.

The code was developed and all the provided services were bundled in a Web Application Archive (WAR) which in turn was deployed on a tomcat web server (Apache Tomcat 7.0.54). In fact, the rest-based service is deployed in tomcat and through the Sesame Repository API communicates with Virtuoso.

### 2.2.2 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 files is usually used as a good-practise way to initialize parameters which remain constant in the whole life-cycle of any application. Next we analyse the parameters which are contained in the properties file and a snapshot of such a file is displayed in Figure 10:
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.

Changes_Ontology: A named graph URI within Virtuoso Endpoint which is used to store the changes definitions (simple and complex) along with the detected changes of each type.

Changes_Ontology_Schema: A named graph URI Virtuoso Endpoint which is used to store only the change definitions.

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 practise, 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.

2.2.3 Required Web Service Interfaces

The current version of the Change Detection module does not use any external web services. We are just using API-calls from the Sesame API, as described above, to access and update Virtuoso. Moreover, we use the JDBC protocol to achieve better performance when we interact directly on Virtuoso Server.
2.2.4 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.

**Complex Changes Management** This service resides in the (partial) URL: /diachron/complex_change and contains all the methods which can be used for the a) definition, b) deletion and c) search of a complex change w.r.t. the ontology of changes which is defined in the configuration file (see Table 1).

<table>
<thead>
<tr>
<th>Methods</th>
<th>Method and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.ws.rs.core.Response</td>
<td>defineCCJSON(java.lang.String inputMessage):</td>
<td>POST method which is responsible for the definition of a complex change and its storage in the ontology of changes.</td>
</tr>
<tr>
<td>javax.ws.rs.core.Response</td>
<td>deleteCCJSON(java.lang.String name):</td>
<td>DELETE method which deletes the complex name with name given as parameter from the ontology of changes.</td>
</tr>
<tr>
<td>javax.ws.rs.core.Response</td>
<td>getCCJSON(java.lang.String name):</td>
<td>GET method which returns high level information about the definition of complex changes.</td>
</tr>
</tbody>
</table>

**Method Details**

**defineCCJSON: POST** method which is responsible for the definition of a complex change and its storage in the ontology of changes.

**URL (partial):** /diachron/complex_change

**Parameters:**
- **inputMessage**: A JSON-encoded string which represents the definition of a complex change and has the following form:

```json
{
    "Complex_Change": "Mark_asObsolete_v2",
    "Complex_Change_Parameters": [
        {
            "obs_class": "sc1:-subclass"
        }
    ],
    "Simple_Changes": [
        {
            "Simple_Change": "ADD_SUPERCLASS",
            "Simple_Change_Uri": "sc1",
            "Is_Optional": false,
            "Selection_Filter": "sc1:-superclass = "",
            "Mapping_Filter": "",
            "Join_Filter": "",
            "Version_Filter": ""
        }
    ]
}
```

where:

18
Complex_Change: the name of the complex change.
Priority: its priority which can be any double number.
Complex_Change_Parameters: the parameter names of the complex change along with the simple change parameter names which are binded with.
Simple_Changes: an array of simple changes which consist the complex change. For each simple change we have the following fields:
Simple_Change: the name of the simple change.
Simple_Change_URI: a URI instance of the simple change.
Is_Optional: a boolean which indicates whether the simple change is optional (true) or mandatory (false).
Selection_Filter: filter which assigns selective values upon simple change parameters.
Mapping_Filter: filter which expresses mappings upon simple change parameters.
Join_Filter: filter which expresses joins across different parameters of simple changes.
Version_Filter: filter which expresses conditions over triples or URIs that should (not) exist in the old and/or the new version.

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

- Error code: 200 and entity content: { "Success": true, "Message": "Complex Change’s definition was inserted in the ontology of changes." } if the complex change is successfully defined and stored.
- Error code: 204 and entity content: { "Success": false, "Message": "There is already defined a Complex Change with the same name." } if there exists a complex change with the same name in the ontology of changes.
- Error code: 400 and entity content: { "Success": false, "Message": "JSON input message could not be parsed." } if the input parameter has not the correct form.
- Error code: 400 and entity content: { "Success": false, "Message": "Error in {Selection|Join|Parameter} Filters." } if one of the issued filters namely, Selection, Join, Parameter has not the correct syntactic form.

getCCJSON: GET method which returns high level information about the definition of a complex change. Such information can be the name of the complex change, the simple changes which compose it, its parameters along with their corresponding filters etc.

URL (partial): /diachron/complex_change/{com_change}

Parameters:
- com_change: Path parameter which refers to the complex change name.

Returns: A Response instance which has a JSON-encoded entity content. We discriminate the following cases:

- Error code: 200 and entity content: { "Result": false, "Message": "Complex change was not found." } if the complex change is not found.
- Error code: 200 and entity content: { "Result": true, "Message": "...." } where an example response message could be:

```json
{
  "Complex_Change": "Mark_as_Obsolete_v2",
  "Complex_Change_Parameters": [
    {
      "obs_class": "sc1:-subclass"
    }
  ]
}```
deleteCCJSON: DELETE method which deletes the complex name with name given as parameter from the ontology of changes. In fact, it deletes all the correlated triples with the corresponding complex change. This means that any detected changes of this complex type will be deleted as well.

URL (partial): /diachron/complex_change/{com_change}

Parameters:
- **com_change**: Path parameter which refers on the complex change name.

Returns: A Response instance which has a JSON-encoded entity content. We discriminate the following cases:

- **Error code: 200** and entity content: { “Result” : true, “Message” : “Complex Change was successfully deleted from the ontology of changes.” } if the complex change is found and successfully deleted.
- **Error code: 204** and entity content: { “Result” : false, “Message” : “Complex Change was not found in the ontology of changes.” } if the complex change is not found.

**Change Detection** This service resides in the (partial) URL: /diachron/change_detection and contains all the methods which can be used for a) the change detection process among any two given dataset versions, and b) the exploitation of any detected changes stored in the ontology of changes (see Table 2).

Table 2: Change Detection Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>javax.ws.rs.core.Response</td>
<td>changeDetectJSON(java.lang.String inputMessage): POST method which is responsible for the change detection process among two dataset versions.</td>
</tr>
<tr>
<td></td>
<td>javax.ws.rs.core.Response</td>
<td>deleteCCJSON(java.lang.String query, java.lang.String format): GET method which applies SPARQL queries on the ontology of changes and returns the results in various formats.</td>
</tr>
</tbody>
</table>

**Method Details**
**changeDetectJSON**: POST method which is responsible for the change detection process among two dataset versions. This method detects any existing simple and complex changes and updates the ontology of changes accordingly.

**URL (partial)**: `/diachron/change_detection`

**Parameters**:
- **inputMessage**: A JSON-encoded string which has the following form:
  ```json
  {
    "Old_Version": "V1",
    "New_Version": "V2",
    "ingest": true,
    "CCs": ["Label_Obsolete", ...]
  }
  ```
  where:
  - **Old_Version**: The old version URI of a DIACHRON entity.
  - **New_Version**: The new version URI of a DIACHRON entity.
  - **ingest**: A flag which denotes whether the service is called due to a new dataset ingestion or not.
  - **CCs**: The set of complex change types which will be considered. If the set is empty, then all the defined complex changes in the ontology of changes will be considered.

**Returns**: A Response instance which has a JSON-encoded entity content depending on the input parameter of the method. We discriminate the following cases:
- **Error code**: 400 and entity content: `{ "Success" : false, "Message" : "JSON input message should have exactly 4 arguments." }` if the input parameter has not four JSON parameters.
- **Error code**: 200 and entity content: `{ "Success" : true, "Message" : "Change detection among versions Old_Version, New_Version was executed." }` if the input parameter has the correct form.
- **Error code**: 400 and entity content: `{ "Success" : false, "Message" : "JSON input message could not be parsed." }` if the input parameter has not the correct form.

**queryChangesOntology**: POST method which applies SPARQL queries on the ontology of changes and returns the results in various formats.

**URL (partial)**: `/diachron/change_detection/query`

**Parameters**:
- **inputMessage**: A JSON-encoded string which has the following form:
  ```json
  {
    "Query": "select ?s ?p . . . ",
    "Format": "json",
  }
  ```
  where:
  - **Query**: A string which represents the SPARQL query which will be applied.
  - **Format**: The format(MIME type) of the query results. The supported formats are: xml, csv, tsv, json.

**Returns**: A Response instance which has a JSON-encoded entity content with the query results in the requested format. We discriminate the following cases:
- **Error code**: 400 if the given SPARQL query contains syntax errors or there was an internal server issue. In any case, the entity content explains the problem’s category.
- **Error code**: 200 and entity content with the query results with the requested format.
• Error code: 406 and entity content “Invalid results format given.” if the requested query results format is not recognized.
• Error code: 400 and entity content: { “Success” : false, “Message” : “JSON input message should have exactly 2 arguments.” } if the input parameter has not two JSON parameters.
• Error code: 400 and entity content: { “Success” : false, “Message” : “JSON input message could not be parsed.” } if the input parameter has not the correct form.

2.2.5 Exposed User Interfaces

Next, we present some mockups which could be implemented and used for the Subsection 2.1.3 and the Complex Changes Management.

**Complex Change Creation** Figure 11 presents a potential mockup which describes a UI for the definition of a Complex Change. This mockup, contains all the functionality described in Subsection 2.1.3 which allows the user to set and instantiate all the Complex Change’s features.

![Figure 11: User Interface for Complex Change Creation](image)

**Complex Change Deletion** Figure 12 also presents a potential mockup for the deletion of a Complex Change. This mockup is much simpler than the one presented in Figure 6b as the only required input from the user is the Complex Change’s name.
Figure 12: User Interface for Complex Change Deletion
3 The Repairing Service

The purpose of the repairing service is to automatically perform diagnosis and repairing of invalidities that appear in a DL-LiteA KB. As explained in D3.1 [5], the service consists of two individual components, namely the diagnosis and the repairing component. The diagnosis component is responsible for detecting inconsistencies between the ABox and the given TBox (along with the integrity constraints it includes), as well as for reporting these invalidities in an intuitive way, in the form of a graph that represents the interdependencies between them. The repairing component takes as input the graph produced by the diagnosis component and is responsible for automatically computing a repairing delta, as well as for applying this repairing delta to render the KB valid. The main features of the diagnosis and repair framework are illustrated in Figure 13. In the figure, we use the notation \( A_i \) to denote a data assertion appearing in the ABox.

In the following, we describe the algorithms related to the diagnosis and repairing components of the repairing service, based on the specifications that appeared in D3.1 [5]. Familiarity with the contents of D3.1 is assumed.

![Figure 13: The main features of the diagnosis and repair framework](image)

3.1 Service Description

3.1.1 Diagnosis Component

The diagnosis algorithm is the core of the diagnosis component and is used to detect all the invalidities in a KB, as well as provide them as output in the form of an interdependency graph. The steps needed to perform diagnosis are illustrated in Algorithm 1.

The diagnosis algorithm starts by computing the closure \( \text{cln}(T) \) of negative inclusions and functionality assertions of the TBox (line 2 of Algorithm 1), in order to get the full set of constraints that need to be checked over the ABox. The process of computing \( \text{cln}(T) \) is defined below:

**Definition 1.** Let \( T \) be a DL-LiteA TBox. We call NI-closure of \( T \), denoted by \( \text{cln}(T) \), the TBox defined inductively as follows:

1. All functionality assertions in \( T \) are also in \( \text{cln}(T) \).
2. All negative inclusion assertions in \( T \) are also in \( \text{cln}(T) \).
3. If \( C_1 \sqsubseteq C_2 \) is in \( T \) and \( C_2 \sqsubseteq \neg C_3 \) or \( C_3 \sqsubseteq \neg C_2 \) is in \( \text{cln}(T) \), then also \( C_1 \sqsubseteq \neg C_3 \) is in \( \text{cln}(T) \).
4. If \( R_1 \sqsubseteq R_2 \) is in \( T \) and \( R_2 \sqsubseteq \neg R_3 \) or \( R_3 \sqsubseteq \neg R_2 \) is in \( \text{cln}(T) \), then also \( R_1 \sqsubseteq \neg R_3 \) is in \( \text{cln}(T) \).
5. If \( R_1 \sqsubseteq R_2 \) is in \( T \) and \( \exists R_2 \sqsubseteq \neg C \) or \( C \sqsubseteq \neg \exists R_2 \) is in \( \text{cln}(T) \), then also \( \exists R_1 \sqsubseteq \neg C \) is in \( \text{cln}(T) \).
Algorithm 1 Diagnosis($K$)

Require: A DL-Lite$_A$ KB $K = \langle T,A \rangle$

Ensure: The interdependency graph of $K$, $IG(K) = (V,E)$

1: $V, E \leftarrow \emptyset$
2: Compute $cln(T)$
3: for all $c \in cln(T)$ do
4: \hspace{1em} $c_r \leftarrow \delta(c)$
5: \hspace{1em} $Ans_{c_r} \leftarrow q_{c_r}$
6: \hspace{1em} for all $\langle i_1,i_2 \rangle \in Ans_{c_r}$ do
7: \hspace{2em} $V \leftarrow V \cup \{i_1,i_2\}$
8: \hspace{2em} $E \leftarrow E \cup \{(i_1,i_2)\}$
9: \hspace{1em} end for
10: end for
11: return $IG(K) = (V,E)$

6. If $R_1 \sqsubseteq R_2$ is in $T$ and $\exists R_2 \sqsubseteq \neg C$ or $C \sqsubseteq \neg \exists R_2^-$ is in $cln(T)$, then also $\exists R_1^- \sqsubseteq \neg C$ is in $cln(T)$.

7. If one of the assertions $\exists R \sqsubseteq \neg \exists Q$, $\exists Q^- \sqsubseteq \neg \exists Q^-$, or $Q \sqsubseteq \neg Q$ is in $cln(T)$, then all three such assertions are in $cln(T)$.

The following example illustrates the computation of the $cln(T)$ follows:

**Example 2.** Consider the following DL-Lite$_A$ KB $K = \langle T,A \rangle$:

$$T = \{(\text{funct } P_1), A_1 \sqsubseteq \neg A_2, A_2 \sqsubseteq \exists P_2\}$$

$$A = \{A_1(x_1), A_2(x_1), P_2(x_1,y_1), P_1(x_3,y_2), P_1(x_3,y_3), P_1(x_3,y_4)\}$$

The closure of negative inclusions and functionality assertions of $T$ ($cln(T)$), is the following:

$$cln(T) = \{(\text{funct } P_1), A_1 \sqsubseteq \neg A_2, \exists P_2 \sqsubseteq \neg A_1\}$$

Each of the constraints in $cln(T)$ is then transformed to a FOL query using predefined patterns, as defined in Table 3 (line 4 of Algorithm 1). The answers to those queries identify the diagnosed invalidities. In Algorithm 1, the execution of each FOL query over the ABox is performed in line 5, where $Ans_{c_r}$ is the answer set of the query, which denotes the set of invalidities that break the constraint $c$. Note that these FOL queries can be easily expressed as SPARQL queries over an ABox stored in a triple store, so that off-the-shelf, optimized tools can be used for query answering.

The last step of the algorithm encodes the invalidities in the form of an interdependency graph (lines 6-9). This graph is produced by iterating over all invalidities (provided by the previous step of the algorithm) and adding the invalid data assertions as vertices (line 7). For each pair of vertices (representing a pair of interdependent invalid data assertions), an edge connecting them is added and labelled with the constraint that this pair breaks (line 8).

Note that the graph does not contain duplicate vertices, meaning that an invalid data assertion will appear at most once in the graph, regardless of how many invalidities it is involved in. Also note that, for each invalidity that said data assertion is involved in, a different edge connecting its vertex with the vertex that represents the other data assertion of the invalidity is added to the graph. As a result, we can easily determine how many invalidities each invalid data assertion is involved in. This information is used by the repairing component for the computation of the repairing delta.

The following example illustrates the diagnosis algorithm in action:

**Example 3.** Consider the KB $K$ and the $cln(T)$ of Example 2. The corresponding FOL queries to check for invalidities, with respect to the constraints in $cln(T)$, are defined as follows:
From the execution of the above three queries over the ABox of Example 2, we get the following sets of invalidities:

\[ q_1(x) \leftarrow P_1(x,y) \land P_1(x,z) \land y \neq z \]
\[ q_2(x) \leftarrow A_1(x) \land A_2(x) \]
\[ q_3(x) \leftarrow P_2(x,y) \land A_1(x) \]

From the execution of the above three queries over the ABox of Example 2, we get the following sets of invalidities:

\[ \text{Ans}_{q_1} = \{ \{ A_1(x_3, y_2), P_1(x_3, y_3) \}, \{ A_1(x_3, y_2), P_1(x_3, y_4) \}, \{ A_1(x_3, y_3), P_1(x_3, y_4) \} \} \]
\[ \text{Ans}_{q_2} = \{ \{ A_1(x_1), A_2(x_1) \} \} \]
\[ \text{Ans}_{q_3} = \{ \{ A_1(x_1), P_2(x_1, y_1) \} \} \]

Figure 14 shows the interdependency graph produced by these sets of invalidities.

Table 3: Transformation of DL-Lite\(_A\) constraints to FOL queries

<table>
<thead>
<tr>
<th>Constraint ((c))</th>
<th>Transformation ((\delta(c)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c = A_1 \subseteq \neg A_2)</td>
<td>(\delta(c) = q(x) \leftarrow A_1(x), \neg A_2(x))</td>
</tr>
<tr>
<td>(c = A_1 \subseteq \neg \exists P_1) (or (c = \exists P_1 \subseteq \neg A_1))</td>
<td>(\delta(c) = q(x) \leftarrow A_1(x), P_1(x,y))</td>
</tr>
<tr>
<td>(c = \exists P_1 \subseteq \neg \exists P_2)</td>
<td>(\delta(c) = q(x) \leftarrow A_1(x), P_1(x,y))</td>
</tr>
<tr>
<td>(c = \exists P_1 \subseteq \neg \exists P_2)</td>
<td>(\delta(c) = q(x) \leftarrow A_1(x), P_1(x,y))</td>
</tr>
<tr>
<td>(c = \exists P_1 \subseteq \neg \exists P_2)</td>
<td>(\delta(c) = q(x) \leftarrow A_1(x), P_1(x,y))</td>
</tr>
<tr>
<td>(c = P_1 \subseteq \neg P_2) (or (c = \exists P_1 \subseteq \neg P_2))</td>
<td>(\delta(c) = q(x,y) \leftarrow P_1(x,y), P_2(x,y))</td>
</tr>
<tr>
<td>(c = P_1 \subseteq \neg P_2)</td>
<td>(\delta(c) = q(x,y) \leftarrow P_1(x,y), P_2(x,y))</td>
</tr>
<tr>
<td>(c = \text{funct } P)</td>
<td>(\delta(c) = q(x) \leftarrow P(x,y), P(x,y))</td>
</tr>
<tr>
<td>(c = \text{funct } P^*)</td>
<td>(\delta(c) = q(x) \leftarrow P(y_1,x), P(y_2,x))</td>
</tr>
</tbody>
</table>
3.1.2 Repairing Component

Given the interdependency graph, the repairing component is responsible for automatically computing and applying a repairing delta, which leads to a consistent KB. Due to the form of DL-Lite_A constraints, repairing is performed by the deletion of either one of the two invalid data assertions that take part in an invalidity. Thus, in terms of the interdependency graph, resolving an invalidity amounts to removing one of the two vertices that are connected by the edge representing this invalidity. A full repair amounts to repeating this process for all edges in the graph.

The problem of identifying the minimum repairing delta is actually the well-known problem of finding the minimum vertex cover [6]. A vertex cover of a graph is a set of vertices, such that each edge of the graph is incident to at least one vertex of this set. By computing a vertex cover of the interdependency graph, the repairing component computes a repairing delta that, when applied to the dataset, leads to a consistent KB. This is due to the fact that the removal of all the vertices (in other words, all the invalid data assertions) in the vertex cover leads to the removal of all the edges (in other words, all the invalidities) from the graph. Note that we are interested in the minimum, with respect to cardinality, vertex cover, in order to guarantee minimum impact of the repairing process on the ABox (i.e., remove the minimum amount of data assertions, cf. [1, 3]).

The repairing algorithm is shown in Algorithm 2; it takes as input the interdependency graph and is responsible for automatically repairing the KB. In its first step (line 2), the algorithm breaks the interdependency graph $IG(K)$ into the set of its connected components (denoted as $CC$ in the algorithm). This way, the computation of the vertex cover of $IG(K)$ can be divided into the separate computation of the vertex covers of the connected components, thus it can be parallelized for better performance.

As a next step, the repairing algorithm computes the vertex cover of each of the connected components (lines 3-5). Recall that the computation of the minimum vertex cover is not efficient, as this is a known NP-COMPLETE problem [9]. However, many approximation algorithms can be used to compute the vertex cover, such as the 2-approximation algorithm, attributed to Gavril and Yannakakis in [9], or the algorithm presented in [7].

We chose to compute the vertex cover in a greedy manner, as presented in [9]. Greedy means that, in each step of the computation, the vertex that is chosen to be included in the cover is the vertex with the highest degree (in other words, the invalid data assertion that is part of the most interdependencies/invalidities). In the case that there exist more than one vertices with the same degree, one of those vertices is arbitrarily chosen. This way, a single vertex cover is returned by the algorithm. This computation is performed in the GreedyVertexCover subroutine, which is presented in Algorithm 3.

The simple approximation algorithm described above guarantees that the best choice is made locally (i.e., in each step of the algorithm) and is very efficient. It can be proven that the above approximation algorithm achieves $O(\log n)$ approximation of the optimal solution, where $n$ is the number of vertices of the graph, with a time complexity of $O(n \log n)$ [9].

The vertices in the computed vertex covers of the connected components represent the invalid data assertions to be removed from the dataset in order to render it valid. It is obvious that, the union of the vertex covers of the connected components of a graph forms a vertex cover of the entire graph. Thus, this union forms the repairing delta (line 4).

The final step of the repairing algorithm is to apply the repairing delta (line 6 of Algorithm 2). This can be

Algorithm 2 Repair($IG(K), A$)

Require: An interdependency graph $IG(K)$ and a DL-Lite_A ABox $A$
Ensure: $K$ in a consistent state
1: $\text{repairing\_delta} \leftarrow \emptyset$
2: $CC \leftarrow \text{ConnectedComponents}(IG(K))$
3: for all $C \in CC$ do
4: \hspace{1em} $\text{repairing\_delta} \leftarrow \text{repairing\_delta} \cup \text{GreedyVertexCover}(C)$
5: end for
6: $A \leftarrow A \setminus \text{repairing\_delta}$
Grant Agreement No. 601043

Algorithm 3 GreedyVertexCover($G$)

Require: A graph $G = (V, E)$
Ensure: The vertices in $V$ that belong to an approximate vertex cover of $G$

1: $cover \leftarrow \emptyset$
2: while $E \neq \emptyset$ do
3: Pick the vertex $v$ with the greatest degree
4: $cover \leftarrow cover \cup v$
5: $V \leftarrow V \setminus v$
6: end while
7: return $cover$

performed in a very efficient manner, by posing a single SPARQL-Update query, containing the deletion of all the invalid data assertions in the repairing delta, over a triple store that stores the ABox.

The following concludes our running example:

Example 4. Consider the interdependency graph of Figure 14. The repairing algorithm will compute the following repairing delta:

$$repairing\_delta = \{A_1(x_1), P_1(x_3, y_2), P_1(x_3, y_3)\}$$

After the application of the repairing delta, the ABox $A$ is in the following state:

$$A = \{A_2(x_1), P_2(x_1, y_1), P_1(x_3, y_4)\}$$

which forms a consistent KB with respect to the $cln(T)$.

3.2 Technical Documentation

3.2.1 Prerequisites

**Virtuoso RDFStore** The Validation module relies on the existence of a specific RDFStore which is exploited in order to store and manage pilots’ data converted into the DIACHRON model. The Change Detection Module exploits the Virtuoso Universal Server for this task. For more details on the Virtuoso Universal Server, refer to Section 2.2.

**Jena API** As noted before, Virtuoso can be used in conjunction with particular RDF Data Management APIs to enable the programmatic access and management of linked data. Three APIs are currently supported, namely Jena, Sesame, and Redland. In terms of the implementation of the Validation module in this project, the Jena API was exploited. The architecture of the Virtuoso Jena Provider, which is a native graph model storage provider for the Jena framework, is depicted in Figure 15.

As depicted in Figure 15, the ARQ API can be used to interact with RDF data, stored in the Virtuoso RDFStore. This API abstracts the low-level functionality, which is handled by other more specific APIs. Therefore, the developer can easily import and export linked data, as well as execute queries over them and update them, using the SPARQL and SPARQL update (SPARUL) languages.

Moreover, the Jena Framework is also equipped with an Ontology API and an Inference API, which can be used together for the interaction with ontologies and for performing reasoning tasks over them.

**JUNG API** JUNG is an open source software library that provides a common and extendible language for the modeling, analysis, and visualization of data that can be represented as a graph or network. It is written in Java, which allows JUNG-based applications to make use of the extensive built-in capabilities of the Java API.

http://jung.sourceforge.net
The JUNG architecture is designed to support a variety of representations of entities and their relations, such as directed and undirected graphs, multi-modal graphs, graphs with parallel edges, and hypergraphs. It provides a mechanism for annotating graphs, entities and relations with metadata. This facilitates the creation of analytic tools for complex data sets that can examine the relations between entities as well as the metadata attached to each entity and relation.

In the context of the Validation module, JUNG is used for the generation and manipulation of the interdependency graph.

**Implementation Analysis**  Based on the above analysis, the Validation module was developed by exploiting the Virtuoso Jena Provider (API) along with the JUNG API. As in Change Detection module, in order to be exposed as a rest-based web service and have the appropriate rich security mechanisms, the jersey API was exploited.

As in the Change Detection module, the provided services were bundled in a Web Application Archive (WAR) which in turn was deployed on a Tomcat web server (Apache Tomcat 7.0.54). In fact, the Restful-based service is deployed in tomcat and through the Jena’s Repository API communicates with Virtuoso (in order to proceed with the proper management of the linked data stored in the Virtuoso’s RDF store).

### 3.2.2 Deployment Process

Similarly to the Change Detection Module, 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. Again, 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 as in the Change Detection module. Next we analyse the parameters which are contained in the properties file:

**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.

3.2.3 Required Web Service Interfaces

The Validation module does not use any external web services. It just uses API-calls from the Jena API to access and update the Virtuoso RDFSStore, as well as from the JUNG API to manipulate the interdependency graph.

3.2.4 Exposed Web Service Interfaces

The Validation module consists of services which can be used for a) the validation of a DIACHRON dataset w.r.t. an ontology which contains a set of integrity constraints and b) the repair of a DIACHRON dataset.

Validation This service resides under the (partial) URL: /diachron/validation and contains all methods that can be used for the diagnosis of the invalidities between an instance of a DIACHRON dataset and the provided integrity constraints (see Table 7).

Table 4: Validation Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>validationJSON</td>
<td>javax.ws.rs.core.Response</td>
<td>validationJSON(java.lang.String inputMessage): POST method which is responsible for the diagnosis of invalidities between an instance of a DIACHRON dataset and the provided integrity constraints.</td>
</tr>
</tbody>
</table>

Method Details

validationJSON: POST method which is responsible for the diagnosis of invalidities between an instance of a DIACHRON dataset and the provided integrity constraints.

URL (partial): /diachron/validation

Parameters:
- inputMessage: A JSON-encoded string which has the following form:
  ```java
  [
  "Dataset" : "Dataset1",
  "Ontology_w_constraints" : "Ontology1",
  "GetInvalidities" : true
  ]
  where:
  **Dataset**: The URI of a DIACHRON entity to be validated.
  **Ontology_w_constraints**: The URI of an ontology describing the dataset, which should also contain the integrity constraints that will be checked over the dataset, in OWL syntax.
  **GetInvalidities**: A flag which denotes if we want the method to return the set of invalidities or just a true/false value if the dataset is valid/invalid. Each invalidity is represented by a pair of triples which cause the invalidity and the constraint which is violated.
Returns: A Response instance which has a JSON-encoded entity content depending on the input parameter of the method. We discriminate the following cases:

- Error code: **400** and entity content: `{ "Success" : false, "Message" : "JSON input message should have exactly 3 arguments." }` if the input parameter has not three JSON parameters.
- Error code: **200** and entity content: `{ "Success" : true, "Result" : true, "Invalidities" : [ ["s11 p11 o11", "s12 p12 o12", "constraint1"], ["s21 p21 o21", "s22 p22 o22", "constraint2"], ... ] }` if the input parameter has the correct form where:
  - **Result**: Flag which denotes whether the dataset is valid (true) or invalid (false).
  - **Invalidities**: Array of triple pairs and each triple pair causes an invalidity in the dataset.
- Error code: **400** and entity content: `{ "Success" : false, "Message" : "JSON input message could not be parsed." }` if the input parameter has not the correct form.

Repair This service resides in the (partial) URL: `/diachron/repair` and contains all the methods which can be used for the repairing of diagnosed invalidities, given an instance of a DIACHRON dataset and the provided integrity constraints (see Table 5).

Table 5: Repair Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>javax.ws.rs.core.Response</code></td>
<td><code>repairJSON(java.lang.String inputMessage)</code>: POST method which is responsible for the repairing of diagnosed invalidities, given an instance of a DIACHRON dataset and the provided integrity constraints.</td>
</tr>
</tbody>
</table>

Method Details

**repairJSON**: POST method which is responsible for the repairing of diagnosed invalidities, given an instance of a DIACHRON dataset and the provided integrity constraints.

**URL (partial)**: `/diachron/repair`

Parameters:

- **inputMessage**: A JSON-encoded string which has the following form:
  ```
  { 
  "Dataset" : "Dataset1", 
  "Ontology_w_constraints" : "Ontology1", 
  "GetDelta" : true 
  }
  ```
  where:
  - **Dataset**: The URI of a DIACHRON entity to be validated.
  - **Ontology_w_constraints**: The URI of an ontology describing the dataset, which should also contain integrity constraints that will be checked over the dataset, in OWL syntax.
  - **GetDelta**: A flag which denotes if we want the method to return the delta applied to the dataset. The delta consists of the triples which have to be deleted in order the dataset to become valid.

**Returns**: A Response instance which has a JSON-encoded entity content depending on the input parameter of the method. We discriminate the following cases:
• Error code: 400 and entity content: { “Success” : false, “Message” : “JSON input message should have exactly 3 arguments.” } if the input parameter has not three JSON parameters.

• Error code: 200 and entity content: { “Success” : true, “RepairApplied” : true, “Delta” : [“s1 p1 o1”, “s2 p2 o2”, …] } if the input parameter has the correct form where:
  
  **RepairApplied**: Flag which denotes whether the dataset was invalid, thus a repair was found and applied (true), false otherwise.
  
  **Delta**: Array of triples which had to be removed in order the dataset to become valid. However, whether these triples will be returned or not depends on the GetDelta flag which is given in the inputMessage parameter.

• Error code: 400 and entity content: { “Success” : false, “Message” : “JSON input message could not be parsed.” } if the input parameter has not the correct form.

### 3.2.5 Exposed User Interfaces

The following figures are mockups of a possible UI that uses the validation and the repair methods.

**Validation input**  Using this UI (Figure 16), the user will be able to define the input parameters for the validation method. More specifically, the user can define the named graph that stores the ontology with the constraints that will be diagnosed over the dataset, as well as the named graph that hold the dataset to be validated. Finally, the user can choose if he wants to get the detected invalidities as a result of the execution of the validation method.

![Figure 16: Input UI for the validation method](image)

**Validation results**  With this UI (Figure 17), the user is informed if the provided dataset is valid or not, with respect to the given ontology with constraints.

**Validation results (with returned invalidities)**  With this UI (Figure 18) the user is informed if the provided dataset is valid or not, with respect to the given ontology with constraints. Moreover, the user is able to download the detected invalidities.
Repair input Using this UI (Figure 19), the user will be able to define the input parameters for the repair method. More specifically, the user can define the named graph that stores the ontology with the constraints that will be diagnosed over the dataset, as well as the named graph that hold the dataset to be repaired. Finally, the user can choose to get the repairing delta as a result of the execution of the repair method.

Repair results With this UI (Figure 20), the user is informed if the provided dataset was successfully repaired.

Repair results (with returned repairing delta) Finally, with this UI (Figure 21) the user is informed if the provided dataset was successfully repaired. Moreover, the user is able to download the triples that were removed from the dataset, during the repairing process (i.e., the repairing delta).
Figure 19: Input UI for the repair method

Figure 20: UI for the results of the repair method execution
Figure 21: UI for the results of the repair method execution (with returned repairing delta)
4 The Cleaning Service

This section presents an automated cleaning service, which focuses on identifying problems that can be cleaned (based on the report of the quality assessment service described in Deliverable 5.2 [4]) and automatically cleaning an RDF dataset by removing all problematic triples, as well as an interactive cleaning application, which supports a human user in resolving more complex cleaning problems.

The general functionality as well as architecture of the cleaning module, which have been introduced in Deliverable 3.1 [5], remains unchanged. The cleaning process is composed of the three following steps:

1. Identifying quality problems in a dataset (carried out by the implementation described in Deliverable 5.2 [4])
2. Suggesting transformations to fix these problems
3. Applying these transformations to the dataset

The cleaning service takes care of steps (1) and (2), where the “transformation rule” is simplified to “delete all triples affected by quality problems”. In contrast to repairing, which applies unambiguously specified rules to data that violate logical constraints, data cleaning cannot be performed fully automatically, because the correction of errors and inconsistencies requires user involvement as well as detailed knowledge of the data. Therefore, to provide full support for cleaning, we have additionally developed an interactive (review and approval of cleaning suggestions by the user) and iterative (sequential processing of the identified problems) cleaning application.

Designing and implementing the cleaning process raises three main difficulties. First, since human resources are very expensive it is necessary to automate as much work as possible and give the user maximal support so that a well-founded decision can be taken. Second, because of the iterative nature of the cleaning process it is essential to be able to set up the cleaning process according to the user’s needs. Third, allowing the user to work interactively – in the cleaning application – requires the availability of a user friendly interface.

The Diachron cleaning service and application address these requirements in the following ways:

1. The cleaning service is designed in a way to provide as much automated operation as possible. For that the identification of quality problems and generation of cleaning suggestions are done automatically in the background. We maintain concrete user advices and data transformation rules in an extensible ontology.
2. The interface both of the service and of the application enables the user (human or machine) to specify quality metrics he is interested in (see more details in section 4.1).
3. We implemented the cleaning application as an extension to the well-known cleaning tool OpenRefine\(^6\). This enables to use a wide range of functionalities provided in OpenRefine, such as defining and applying transformation on the data in a spreadsheet like interface (see Figure 25). Transformation effects are immediately displayed on the screen and each single transformation can be undone.

4.1 Service Description

The main components of the cleaning service and the cleaning application were defined according to the three steps of the cleaning process, presented above, namely: i) a component for quality problems identification, ii) a component for generating cleaning suggestions, and iii) a component for applying cleaning rules.

The general description of the cleaning service and application, as well as architecture design, have been introduced in Deliverable 3.1 [5]. Hereafter we present a detailed description of the particular service components as well as design decisions and implementation details that have been made. Since the quality assessment service already exists in Diachron (see Deliverable 5.2 [12] for more details), the cleaning service and application were designed to operate on its output. In order to provide the user with an interactive application for data cleaning that enables to define and apply cleaning operations in a spreadsheet-like interface we decided to implement the cleaning application as an extension of the well-known data cleaning tool OpenRefine.

\(^6\)http://openrefine.org/
**OpenRefine** (formerly Google Refine) is a powerful tool for working with data containing inconsistencies and invalidities. It serves a wide range of functionalities for: detecting and fixing inconsistencies; transforming data from one structure or format to another; looking up further information from web services; and linking it to databases like Freebase. OpenRefine is a stand-alone desktop application. It supports “expressions” to transform existing data or to create new data based on existing data. OpenRefine supports several languages for writing expressions. It has its own native language called OpenRefine Expression Language (GREL), but other languages such as Jython\(^7\) are also supported. Detailed information of how to work with OpenRefine is presented in [10].

Combining the Diachron quality assessment framework and the OpenRefine cleaning framework has two advantages: the former can automatically identify inconsistencies and anomalies in data, while the latter serves a wide range of functionalities designed for interactive data cleaning.

Both the data cleaning service and the data cleaning application accept as input a possibly erroneous and inconsistent data set and output a cleaned data set. The complete transformation from the input data into the output view is decomposed into three steps, each of which corresponds to the specific service component, described below in more detail. First, we present the complete cleaning workflow using the Diachron quality extension for OpenRefine.

### 4.1.1 Cleaning Workflow

Figure 22 shows the whole workflow of the cleaning process. Hereafter we explain it step by step, illustrated by screenshots of the cleaning application. Typical usage of the cleaning service, whose API is presented separately in section 4.2.4, follows a similar but simplified workflow. For now, we take the perspective of the user controlling the whole workflow from inside OpenRefine: the user loads a dataset, has its quality assessed, and then cleans. For subsequent stages of code integration, we are aiming at enabling external applications to embed our cleaning application into more complex workflows. See, e.g., Deliverable 5.2 [4, section 4.3] for a full quality assessment workflow, one of whose steps is cleaning.

The user starts cleaning by uploading his data and creating a new project. OpenRefine supports several data formats, including RDF and Turtle. The uploaded data set will be displayed to the user; now the cleaning process can begin.

Clicking on the ‘Diachron Quality’ extension button and then selecting ‘Identify Quality Problems’ tab (Figure 23), the user launches the interface where he can specify the quality metrics of his interest (Figure 24). If he only wants to work on specific metric dimensions or even single metrics, he can click the corresponding checkboxes.

Once the selection of quality metrics has been confirmed by clicking the ‘OK’ button, the quality assessment service will be invoked to perform quality assessment and return a quality report. The quality report is an input for the ‘cleaning suggestion generation’ component. Cleaning suggestions are generated based on the quality report ontology, presented in more detail in section 4.1.3. Each quality problem in the ontology is assigned with an appropriate cleaning suggestion and, if available, a cleaning rule (see section 4.1.3 for more details). In preparation for the next cleaning step, the original data set is displayed in a spreadsheet view with the following columns: Subject, Predicate, Object, Quality Problem, Problem Description, Cleaning suggestion and Cleaning Rule (Figure 25).

This view enables the user to work separately on each single triple component, e.g. to apply a transformation only to the predicate column or to perform changes in the object column. OpenRefine supports faceted browsing as a mechanism for filtering down to just a subset of rows to be transformed. Faceting enables to filter all the triples affected by a particular quality problem, and to apply a proposed data transformation rule to relevant triples only. After defining a text facet over the possible values of the Quality Problem column, the left panel will display all identified quality problems with the number of their occurrences (Figure 26). Sorting them the user can sequentially clean the dataset w.r.t. these problems, starting with the most frequent one.

Unfortunately, it is not always possible to define an appropriate data transformation as GREL rule. For some problems, such as ‘EmptyAnnotationValue’, the user has to fill in the value manually, or he should remove the corresponding triple. The proposed rule can be applied in the corresponding view (see Figure 27). After putting

\(^7\)https://github.com/OpenRefine/OpenRefine/wiki/Jython
the proposed expression into the expression field, the user can directly see the effect of the targeted transformation. After cleaning, the user can download the cleaned data set in the original format.

4.1.2 Quality Problem Identification
As mentioned above, the core of the Quality Problem identification component is the Diachron quality assessment service, which was introduced in Deliverable 5.1 [12] and is described in detail in Deliverable 5.2 [4]. The quality assessment framework enables to reflect different aspects of data quality, with regard to a wide range of quality metrics. In parallel to computing any quality metric, it collects those triples that violate the quality criteria defined by the particular metric. Two different outputs are provided: the quality metadata, i.e. an RDF representation of metric values, and the quality report, which is composed of identified quality problems. The quality report (QR) is represented in terms of the QR ontology, which we developed for this purpose, and which is described in
An ontology represents the concepts and their relations that are relevant for a given domain [11]. It consists of a representational vocabulary with precise definitions of the meanings of the terms in the vocabulary plus a set of axioms. The QR ontology was designed to specify the concepts related to quality problems and possible solution for them.

Figure 28 shows the structure of the quality report vocabulary. The Quality report is computed on a Resource and contains a set of Quality problems.

With respect to Deliverable 3.1 [5] the QR ontology was extended to support cleaning issues as described in
Figure 27: Data transformation view.

Figure 28: General structure of the Quality Report ontology

the next section.

4.1.3 Generation of Cleaning Suggestions

Figure 29 presents the extended structure of the QualityProblem class. A QualityProblem is described by the corresponding Metric and affects an RDF triple, which we reify for the purpose of referring to it. A problem's problemDescription property aims to provide the user with a brief and clear description of the problem. The cleaningSuggestion property contains a general natural language recommendation how to solve the identified problem, while the optional cleaningRule property contains a cleaning rule. The latter property has language-specific subproperties, such as grelCleaningRule for cleaning rules expressed in GREL.
The quality report contains a set of quality problems that are associated with the affected triples. As soon as we are able to precisely refer to a quality problem by URI, it is straightforward to provide the user with an appropriate cleaning suggestion and with additional information (such as an exact description description of the problem) to help the user understand. The property `describedBy` is not bijective, as one quality metric may correspond to one or more quality problems. Consider, for example, the `MisusedOwlDatatypeOrObjectProperty` metric. From a high-level quality assessment perspective it is not important to separately count datatype properties and object properties that were misused; any triple with any such misuse just counts as one “bad” triple. However, it does play an important role from the cleaning perspective, as the cleaning service and application aim at giving the user precise feedback. We therefore relate to the two quality problems `MisusedDatatypeProperty` and `MisusedObjectProperty`, each of which has a different cleaning suggestion (see Table 6). Some quality problems require additional properties in order to precisely specify the problem. Figure 30 shows an example for such kind of quality problems. In order to provide the user with an appropriate cleaning rule for `IncompatibleDatatypeRangeProblem` we need to memorize which datatype is required. This is realized using the `qr:expectedDatatype` property.

The descriptions of concrete quality problems are available as a linked dataset at http://purl.org/eis/vocab/qprob.

---

**Figure 29:** Quality Problem Class of the QR vocabulary.

**Figure 30:** Quality problem representation in the QR Vocabulary.

**Listing 1:** An example of a quality problem

```reasoning
qr:WhitespaceInAnnotationProblem a rdfs:Class;
  rdfs:subClassOf qr:QualityProblem;
  qr:describedBy dqm:WhitespaceInAnnotationProblem;
  qr:problemDescription "Literal value contains white spaces";
  qr:cleaningSuggestion "Trim leading and trailing whitespace";
```
4.1.4 Transformation Engine

The user performs data transformation by direct manipulation on the OpenRefine spreadsheet interface. He first chooses an appropriate column and selects rows affected by a particular quality problem using facets. Working step by step on the quality problems identified, the user can undo particular transformations if they do not have the expected effect. This positively affects the accuracy of cleaning results. Having performed all necessary transformations on the data, the user can export the cleaned data to their original RDF format.

4.2 Technical Documentation

4.2.1 Prerequisites

Apache Jena

Apache Jena is an open source Semantic Web framework for Java. It provides a programmatic environment for RDF, RDFs and OWL, SPARQL and includes a rule-based inference engine. It provides an API to extract data from and write to RDF graphs. The graphs are represented as an abstract “model”. A model can be populated with data from files, databases, URLs or a combination of these. A model can be queried using SPARQL.

Ontologies

The daQ, DQM, and QR ontologies developed in the context of the Diachron project⁸ are required to understand the quality report created by the quality assessment component and then to produce cleaning suggestion for the corresponding problems.

OpenRefine

OpenRefine⁹ is a desktop application for data cleaning, which provides the foundation for the cleaning application. It is intended to be installed and run on the user’s local machine, but has a web-based user interface. Unlike most other desktop applications, it runs as a small web server on the local machine. The user has to point the web browser at that web server in order to use OpenRefine.

4.2.2 Deployment Process

Similarly to deploying the repairing service, as explained in section 3.2.2, deploying the cleaning service involves the deployment of a WAR file, and defining the following parameters in a properties file:

Quality_Assessment_Service: The URL of the quality assessment web service required (see section 4.2.3).

The cleaning application is implemented as an extension to OpenRefine and provided as a ZIP file. Installing it requires extracting that ZIP file to the extensions folder of the OpenRefine workspace and restarting OpenRefine.

4.2.3 Required Web Service Interfaces

The cleaning service requires the quality assessment service explained in Deliverable 5.2 [4, section 2.4].

4.2.4 Exposed Web Service Interfaces

Deliverable 6.1 was missing a description of the RESTful API of the cleaning service, but explained the general design of these APIs for the Diachron services [8]. The cleaning service API is similar to the API of the quality assessment service described in Deliverable 5.2 [4].

The cleaning service provides the two methods /diachron/get_cleaning_suggestions and /diachron/clean, of which we give a precise description below. The typical usage of the cleaning service follows the workflow outlined for the cleaning application in figure 22: In the input to the get_cleaning_suggestions method, the client specifies the dataset and the metrics w.r.t. which the dataset should be cleaned; the method returns a set of

⁸http://github.com/diachron/quality/
⁹http://openrefine.org/
cleaning suggestions. In the input to the clean method, the client one more specifies a set of metrics. The dataset is then cleaned by applying the rule “delete every triple that is affected by a quality problem w.r.t. to at least one of the metrics specified”. This results in a cleaned dataset. Optionally, the clean method returns – similarly to the repair method introduced in section – a list of all triples that were removed by cleaning.

Method Details

getCleaningSuggestionsJSON: POST method which returns a set of cleaning suggestions.

URL (partial): /diachron/get_cleaning_suggestions

Parameters:

- **inputMessage**: A JSON-encoded string which has the following form:
  
  ```json
  {
  "Dataset" : "Dataset1",
  "MetricsConfiguration" : "MetricsConfiguration",
  }
  
  where:
  
  Dataset: The URI of a DIACHRON entity to be validated.
  MetricsConfiguration: a list of quality metrics w.r.t. which cleaning suggestions should be generated.

  The format of this list is defined by the quality assessment service and explained in [4, section 2.4].

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

- Error code: **400** and entity content: { “Success”: false, “Message”: “JSON input message should have exactly 2 arguments.” } if the input parameter has not two JSON parameters.
- Error code: **200** and entity content: { “Success”: true, “Suggestions”: [ subject: s1, predicate: p1, object: o1, problem: QP1, additionalProperty1: additionalValue1, ..., subject: s2, predicate: p2, object: o2, problem: QP2, ..... ] } if the input parameter has the correct form where:
  
  **Suggestions**: Array of “problem/suggestion” records, each of which contains a reified triple from the dataset, the URI of the QualityProblem this triple is affected from (where multiple quality problems may correspond to one quality metric), plus additional problem-specific properties such as expectedDataType (cf. section 4.1.3) in key/value form. A triple that is affected by multiple quality problems may occur more than once. The client can look up the actual cleaning suggestions from the quality report ontology, which is published as a linked dataset at http://purl.org/eis/vocab/qprob.
- Error code: **400** and entity content: { “Success”: false, “Message”: “JSON input message could not be parsed.” } if the input parameter has not the correct form.

cleanJSON: POST method which which cleans a dataset (similar to the repair method described in section 3.2.4). The dataset is cleaned by applying the rule “delete every triple that is affected by a quality problem w.r.t. to at least one of the metrics specified”.

URL (partial): /diachron/clean

Parameters:

- **inputMessage**: A JSON-encoded string which has the following form:
  
  ```json
  {
  "Dataset" : "Dataset1",
  "MetricsConfiguration" : "MetricsConfiguration",
  "GetDelta": true
  }
  
  where:
  
  Dataset: The URI of a DIACHRON entity to be validated.
  MetricsConfiguration: a list of quality metrics w.r.t. which cleaning suggestions should be generated.

  The format of this list is defined by the quality assessment service and explained in [4, section 2.4].

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

- Error code: **400** and entity content: { “Success”: false, “Message”: “JSON input message should have exactly 2 arguments.” } if the input parameter has not two JSON parameters.
- Error code: **200** and entity content: { “Success”: true, “Suggestions”: [ subject: s1, predicate: p1, object: o1, problem: QP1, additionalProperty1: additionalValue1, ..., subject: s2, predicate: p2, object: o2, problem: QP2, ..... ] } if the input parameter has the correct form where:
  
  **Suggestions**: Array of “problem/suggestion” records, each of which contains a reified triple from the dataset, the URI of the QualityProblem this triple is affected from (where multiple quality problems may correspond to one quality metric), plus additional problem-specific properties such as expectedDataType (cf. section 4.1.3) in key/value form. A triple that is affected by multiple quality problems may occur more than once. The client can look up the actual cleaning suggestions from the quality report ontology, which is published as a linked dataset at http://purl.org/eis/vocab/qprob.
- Error code: **400** and entity content: { “Success”: false, “Message”: “JSON input message could not be parsed.” } if the input parameter has not the correct form.
**Dataset:** The URI of a DIACHRON entity to be validated.

**MetricsConfiguration:** a list of quality metrics w.r.t. which cleaning suggestions should be generated.

The format of this list is defined by the quality assessment service and explained in [4, section 2.4].

**GetDelta:** a flag which denotes, in analogy to the repair service, if we want the method to return the delta applied to the dataset, i.e. the list of triples that have been deleted during cleaning.

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

- Error code: **400** and entity content: `{ “Success” : false, “Message” : “JSON input message should have exactly 3 arguments.” }` if the input parameter has not three JSON parameters.

- Error code: **200** and entity content: `{ "CleaningApplied": true, “Delta” : [ "s1 p1 o1", "s2 p2 o2", ... ] }` if the input parameter has the correct form where:

  **CleaningApplied:** flag which denotes whether the dataset was cleaned, i.e. triples were removed (true), false otherwise

  **Delta:** if GetDelta was true, this will be an array of those triples that were removed during cleaning.

- Error code: **400** and entity content: `{ “Success” : false, “Message” : “JSON input message could not be parsed.” }` if the input parameter has not the correct form.

### 4.2.5 Exposed User Interfaces

The screenshots of the user interface of the cleaning application are presented in section 4.1.1. From a technical point of view, the two parts of the interface can be summarized as follows:

**Quality assessment interface** enables the user to define a set of metrics w.r.t. which the quality of the data will be assessed. More specifically, the user can select a particular quality dimension; then all quality problems w.r.t. metrics in this dimension will be identified.

In the **ExportRDF interface** the user has to specify the path the cleaned dataset will be exported to.

Since OpenRefine operates on its own data model, which is different from the Diachron data model, the data transformation unit of the cleaning application implements the following two methods:

- **retrieverDFData** loads an RDF dataset into the OpenRefine data model for cleaning.

- **exportAsRDFData** exports a cleaned data set into its original RDF format.

### 4.2.6 Extension Points

Any third party can extend the cleaning service by extending the quality assessment framework by additional metrics (e.g. further ones from Deliverable 5.1 [12] or 5.2 [4]) as well as extending the quality report ontology by the corresponding cleaning suggestions. In addition, the cleaning application can be extended by adding cleaning rules in further expression languages, such as Jython\(^\text{10}\).

---

\(^{10}\)https://github.com/OpenRefine/OpenRefine/wiki/Jython
Table 6: Quality Problems

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Quality Problem</th>
<th>Problem Description</th>
<th>Cleaning suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>SPARQL Accessibility</td>
<td>SPARQL Endpoint can not be accessed</td>
<td>Check the SPARQL Endpoint URI or remove the triple</td>
</tr>
<tr>
<td></td>
<td>RDF Dump Accessibility</td>
<td>RDF dump is not available</td>
<td>Check the path to RDF dump or remove the triple</td>
</tr>
<tr>
<td></td>
<td>NotValidURI</td>
<td>URI is not valid or link is broken</td>
<td>Check the way the vocabulary is published or use a different one</td>
</tr>
<tr>
<td></td>
<td>UnstructuredData</td>
<td>Dead link</td>
<td></td>
</tr>
<tr>
<td>Accuracy</td>
<td>MalformedDatatypeLiteral</td>
<td>Literal value is not consistent with its data type</td>
<td>Convert literal to expected datatype</td>
</tr>
<tr>
<td></td>
<td>expectedDatatype rdfs:DatatypeRange</td>
<td>Literal value is not consistent with property range</td>
<td>Convert literal to expected datatype</td>
</tr>
<tr>
<td></td>
<td>HomogeneousDatatypes</td>
<td>Property has literals of different datatypes as its objects</td>
<td>Convert literals to unique datatype</td>
</tr>
<tr>
<td></td>
<td>MisplacedClass</td>
<td>Instance of rdfs:Class used in property position</td>
<td>Use instance of rdfs:Property</td>
</tr>
<tr>
<td></td>
<td>MisplacedProperty</td>
<td>Instance of rdfs:Property used in object position</td>
<td>Use instance of rdfs:Class</td>
</tr>
<tr>
<td></td>
<td>MisusedOwlDatatypeProperty</td>
<td>owl:DatatypeProperty is used with rdfs:Resource</td>
<td>Use class of owl:ObjectProperty or change object to rdfs:Literal</td>
</tr>
<tr>
<td></td>
<td>MisusedOwlObjectProperty</td>
<td>owl:ObjectProperty is used with rdfs:Literal</td>
<td>Use class of owl:DatatypeProperty or change object to rdfs:Resource</td>
</tr>
<tr>
<td>Consistency</td>
<td>WhitespaceInAnnotation</td>
<td>Literal value contains leading or trailing whitespaces</td>
<td>Trim leading and trailing whitespace</td>
</tr>
<tr>
<td></td>
<td>EmptyAnnotationValue</td>
<td>Literal value is empty</td>
<td>Remove the triple or add literal value</td>
</tr>
<tr>
<td></td>
<td>LabelsUsingCapitals</td>
<td>Literal uses a bad style of capitalization</td>
<td>Change the capitalization of the literal</td>
</tr>
<tr>
<td></td>
<td>BlankNodeObject</td>
<td>Object is a blank node (i.e. neither a literal nor a resource having a URI)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BlankNodeSubject</td>
<td>Subject is a blank node</td>
<td>Remove the triple or change it</td>
</tr>
<tr>
<td></td>
<td>OntologyHijacking</td>
<td>The triple redefines an element that already exists in an external vocabulary</td>
<td></td>
</tr>
</tbody>
</table>
Table 7: Cleaning Methods

<table>
<thead>
<tr>
<th>Methods</th>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>javax.ws.rs.core.Response</td>
<td>getCleaningSuggestionsJSON(java.lang.String inputMessage): POST method which returns a set of cleaning suggestions.</td>
</tr>
<tr>
<td></td>
<td>javax.ws.rs.core.Response</td>
<td>cleanJSON(java.lang.String inputMessage): POST method which cleans a dataset (similar to the repair method described in section 3.2.4).</td>
</tr>
</tbody>
</table>
5 Conclusion

This report contains the technical documentation regarding the first version of the WP3 services delivered in M16 of the DIACHRON project; these are the detection service, the repairing service and the cleaning service. Apart from the technical documentation about the services, we also provided the final version of the specifications, with emphasis on any differences with respect to what was reported in Deliverable D3.1 [5]. These services have already been implemented and are publicly deployed as a stand-alone module at http://139.91.183.93:8181. They will be also formally deployed as part of the integrated DIACHRON environment for Deliverable D6.2 in M20.
References


A Low-level Changes

As reported in Deliverable D3.1 [5], we have two low-level changes, one reporting triples that were added and one reporting triples that were deleted. These low-level changes refer to the representation of the dataset in the internal DIACHRON representation (under the DIACHRON model).

The low-level changes are stored in two specific named graphs within Virtuoso Server namely, http://added and http://deleted. Consider now the case in which we desire to detect the low-level changes among two dataset versions e.g., V1, V2. The corresponding SPARQL queries which will update the above named graphs will be:

**Added low-level changes:**

```sparql
INSERT INTO <http://added> {?s ?p ?o} WHERE {
    GRAPH <V2> {?s ?p ?o }
    FILTER NOT EXISTS { GRAPH <V1> {?s ?p ?o} }
}
```

**Deleted low-level changes:**

```sparql
INSERT INTO <http://deleted> {?s ?p ?o} WHERE {
    GRAPH <V1> {?s ?p ?o }
    FILTER NOT EXISTS { GRAPH <V2> {?s ?p ?o} }
}
```
**B Simple Changes for the Ontological Model**

In this appendix, we list the simple changes referring to the Ontological DIACHRON model. We should note that these changes are almost identical to the Basic Changes which described in Deliverable D3.1.

Following the same approach as in Deliverable D3.1 in the tables below we list, for each change:

- Its name.
- The intuition it captures, described in terms of the RDF model.
- Its parameters, and the intuition behind each parameter.
- The SPARQL query which will be used for its detection.
- The low-level changes ($\delta^+$, $\delta^-$) which are required to be present in the low-level delta ($\Delta^+$, $\Delta^-$ respectively) in order for said change to be detected.
- A logical condition ($\phi$) which must be satisfied in order for the change to be detected. For simplicity, the conditions in the following tables use clauses like “$a$ is a class in $V_1$”, rather than the more verbose (and formal) statement: $(a, \text{rdf: type}, \text{rdfs: class}) \in V_1 \land (a, \text{rdfs: subClassOf}, \text{rdfs: resource}) \in V_1$.

<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Type_Class(a)</th>
<th>Delete_Type_Class(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Add object $a$ of type rdfs: class</td>
<td>Delete object $a$ of type rdfs: class</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>$a =$ The added object</td>
<td>$a =$ The deleted object</td>
</tr>
<tr>
<td><strong>$\delta^+$</strong></td>
<td>$(r, \text{diachron: subject}, a)$, $(r, \text{diachron: hasAttribute}, \text{ratt})$, $(\text{ratt}, \text{diachron: Property}, \text{rdf: type})$, $(\text{ratt}, \text{diachron: object}, \text{rdfs: class})$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td><strong>$\delta^-$</strong></td>
<td>$\emptyset$</td>
<td>$(r, \text{diachron: subject}, a)$, $(r, \text{diachron: hasAttribute}, \text{ratt})$, $(\text{ratt}, \text{diachron: Property}, \text{rdf: type})$, $(\text{ratt}, \text{diachron: object}, \text{rdfs: class})$</td>
</tr>
<tr>
<td><strong>$\phi$</strong></td>
<td>$a$ does not appear in $V_1$</td>
<td>$a$ does not appear in $V_2$</td>
</tr>
</tbody>
</table>
The changes Add_Type_Property and Delete_Type_Property are defined analogously with the exception that \((ratt, \text{diachron:object, rdfs:resource})\) should be in \(\delta^+\) (\(\delta^-\)) instead of \((ratt, \text{diachron:object, rdfs:resource})\).
<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Superproperty(a,b)</th>
<th>Delete_Superproperty(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Parent b of property a is added</td>
<td>Parent b of property a is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The property (b = ) The new parent</td>
<td>a = The property (b = ) The old parent</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>((r, \text{diachron:hasAttribute, } \text{ratt})), ((\text{ratt, diachron:Property, rdfs:subPropertyOf}), (\text{ratt, diachron:object, } b))</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>(\emptyset)</td>
<td>((r, \text{diachron:hasAttribute, } \text{ratt})), ((\text{ratt, diachron:Property, rdfs:subPropertyOf}), (\text{ratt, diachron:object, } b))</td>
</tr>
<tr>
<td>(\phi)</td>
<td>Triple ((r, \text{diachron:subject, } a) \in V^2)</td>
<td>Triple ((r, \text{diachron:subject, } a) \in V^2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Type_To_Individual(a,b)</th>
<th>Delete_Type_From_Individual(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Type b of individual a is added</td>
<td>Type b of individual a is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The individual (b = ) The new type (class)</td>
<td>a = The individual (b = ) The old type (class)</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>((r, \text{diachron:hasAttribute, } \text{ratt})), ((\text{ratt, diachron:Property, rdf:type}), (\text{ratt, diachron:object, } b))</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>(\emptyset)</td>
<td>((r, \text{diachron:hasAttribute, } \text{ratt})), ((\text{ratt, diachron:Property, rdf:type}), (\text{ratt, diachron:object, } b))</td>
</tr>
<tr>
<td>(\phi)</td>
<td>Triple ((r, \text{diachron:subject, } a) \in V^2) &amp; (b \notin {\text{rdfs:resource, rdfs:resource, rdfs:resource, rdfs:resource}})</td>
<td>Triple ((r, \text{diachron:subject, } a) \in V^2) &amp; (b \notin {\text{rdfs:resource, rdfs:resource, rdfs:resource, rdfs:resource}})</td>
</tr>
<tr>
<td>Change</td>
<td>Add_Property_Instance($a_1, a_2, b)</td>
<td>Delete_Property_Instance($a_1, a_2, b)</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Intuition</td>
<td>Add property instance of property $b$</td>
<td>Delete property instance of property $b$</td>
</tr>
<tr>
<td>Parameters</td>
<td>$a_1 = \text{The subject}$</td>
<td>$a_1 = \text{The subject}$</td>
</tr>
<tr>
<td></td>
<td>$a_2 = \text{The object}$</td>
<td>$a_2 = \text{The object}$</td>
</tr>
<tr>
<td></td>
<td>$b = \text{The property}$</td>
<td>$b = \text{The property}$</td>
</tr>
</tbody>
</table>
| SPARQL used for detection | `SELECT ?a1 ?a2 ?b WHERE {` | `SELECT ?a1 ?a2 ?b WHERE {`
| | `{GRAPH (added) {?r diachron:subject ?a1.}} UNION {` | `{GRAPH (deleted) {?r diachron:subject ?a1.}} UNION {`
| | `GRAPH <V1> {?r diachron:subject ?a1.}} ``` | `GRAPH <V1> {?r diachron:subject ?a1.}} ```
| | `GRAPH (added) {` | `GRAPH (deleted) {
| | `?r diachron:hasAttribute ?ratt.` | `?r diachron:hasAttribute ?ratt.`
| | `FILTER(?(b != rdfs:domain) &&` | `FILTER(?(b != rdfs:domain) &&`
| | `b != rdfs:range)`} UNION `{GRAPH <deleted> {?r diachron:subject ?a1.}} ``` | `b != rdfs:range)`} UNION `{GRAPH <deleted> {?r diachron:subject ?a1.}} ```
| $\delta^+$ | `(r, diachron : hasAttribute, ratt). (ratt, diachron : Property, b), (ratt, diachron : object, a2)` | `$\emptyset$` |
| $\delta^-$ | `$\emptyset$` | `(r, diachron : hasAttribute, ratt), (r, diachron : Property, b), (r, diachron : object, a2)` |
| $\phi$ | `Triple (r, diachron : subject, a) \in V_2 ∧ b \notin \{rdfs : subClassOf, rdfs : subPropertyOf, rdf : type, rdfs : comment, rdfs : label, rdfs : domain, rdfs : range\}` | `Triple (r, diachron : subject, a) \in V_2 ∧ b \notin \{rdfs : subClassOf, rdfs : subPropertyOf, rdf : type, rdfs : comment, rdfs : label, rdfs : domain, rdfs : range\}` |

<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Domain($a$, b)</th>
<th>Delete_Domain($a$, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Domain $b$ of property $a$ is added</td>
<td>Domain $b$ of property $a$ is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>$a = \text{The property}$</td>
<td>$a = \text{The property}$</td>
</tr>
<tr>
<td></td>
<td>$b = \text{The domain}$</td>
<td>$b = \text{The domain}$</td>
</tr>
</tbody>
</table>
| SPARQL used for detection | `SELECT ?a ?b WHERE {` | `SELECT ?a ?b WHERE {`
| | `{GRAPH (added) {?r diachron:subject ?a.}} UNION {` | `{GRAPH (deleted) {?r diachron:subject ?a.}} UNION {`
| | `GRAPH <V1> {?r diachron:subject ?a.}} ``` | `GRAPH <V1> {?r diachron:subject ?a.}} ```
| | `GRAPH (added) {` | `GRAPH (deleted) {
| | `?r diachron:hasAttribute ?ratt.` | `?r diachron:hasAttribute ?ratt.`
| | `FILTER(?(b != rdfs:domain) && b != rdfs:range)`} UNION `{GRAPH <deleted> {` | `FILTER(?(b != rdfs:domain) && b != rdfs:range)`} UNION `{GRAPH <deleted> {`}
| $\delta^+$ | `(r, diachron : hasAttribute, ratt), (r, diachron : Property, rdf : domain), (r, diachron : object, b)` | `$\emptyset$` |
| $\delta^-$ | `$\emptyset$` | `(r, diachron : hasAttribute, ratt), (r, diachron : Property, rdf : domain), (r, diachron : object, b)` |
| $\phi$ | `Triple (r, diachron : subject, a) \in V_2` | `Triple (r, diachron : subject, a) \in V_2` |

The changes $\text{Add\ Range}$ and $\text{Delete\ Range}$ are defined analogously with the exception that $(r, \text{diachron : Property}, \text{rdfs : range})$ should be in $\delta^+$ ($\delta^-$) instead of $(a, \text{rdfs : domain}, b)$. 

---

Grant Agreement No. 601043
<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Comment(a,b)</th>
<th>Delete_Comment(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Comment b of object a is added</td>
<td>Comment b of object a is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The object</td>
<td>a = The object</td>
</tr>
<tr>
<td>b = The new comment</td>
<td>b = The old comment</td>
<td></td>
</tr>
</tbody>
</table>

δ⁺ \((r, \text{diachron}: \text{hasAttribute}, \text{ratt}), (\text{ratt}, \text{diachron}: \text{Property}, \text{rdfs}: \text{comment}), (\text{ratt}, \text{diachron}: \text{object}, b)\)

δ⁻ \(\emptyset\)

φ Triple \((r, \text{diachron}: \text{subject}, a) \in V_2\)
C Simple Changes for the Multidimensional Model

In this appendix, we list the simple changes referring to the multidimensional DIACHRON model. Note that some of them have already been presented in Deliverable 3.1 [5], but in this appendix we present the final list of simple changes to use in the multidimensional model.

The list is presented according to the corresponding building entity of the model. Thus, we present changes for the following entities: dimension, observation, codelist/hierarchy (CH) measure, fact table and attribute.

Additionally, at the end we list a category of changes that are more generic, and are necessary to capture cases of changes useful for pilots (e.g., Add_Label(S, P, O)) or unusual cases (e.g., Add_Unknown_Property(S, P, O)). These changes ensure completeness at the detection phase. Also the conditions appeared in each change ensure unambiguity.

Following the same approach as in Deliverable D3.1 in the tables below we list, for each change:

- Its name.
- The intuition it captures, described in terms of the multidimensional model.
- Its parameters, and the intuition behind each parameter.
- The SPARQL query which will be used for its detection.
- The low-level changes ($\delta^+$, $\delta^-$) which are required to be present in the low-level delta ($\Delta^+$, $\Delta^-$ respectively) in order for said change to be detected.
- A logical condition ($\phi$) which must be satisfied in order for the change to be detected.
**Entity: Dimension**

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Dimension(D)</th>
<th>Delete Dimension(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Add a new dimension (not assigned to a fact table)</td>
<td>Reverse of Add Dimension(D)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>D = The added dimension</td>
<td>D = The deleted dimension</td>
</tr>
<tr>
<td><strong>SPARQL used for detection</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>δ⁺</td>
<td>( (D, \text{rdf:type, diachron:DimensionProperty}) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>δ⁻</td>
<td>( \emptyset )</td>
<td>( (D, \text{rdf:type, diachron:DimensionProperty}) )</td>
</tr>
<tr>
<td>θ</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach Datatype to Dimension(D, T)</th>
<th>Detach Datatype from Dimension(D, T)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Associates a datatype with an existing dimension</td>
<td>Reverse of Attach Datatype to Dimension(D, T)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>D = The dimension in which the datatype is attached</td>
<td>D = The dimension in which the datatype is detached</td>
</tr>
<tr>
<td></td>
<td>T = The datatype which is attached to dimension</td>
<td>T = The datatype which is detached from dimension</td>
</tr>
<tr>
<td><strong>SPARQL used for detection</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>δ⁺</td>
<td>( (D, \text{rdfs:range, T}) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>δ⁻</td>
<td>( \emptyset )</td>
<td>( (D, \text{rdfs:range, T}) )</td>
</tr>
<tr>
<td>θ</td>
<td>( (D, \text{rdf:type, diachron:DimensionProperty}) ) in ( V_2 )</td>
<td>( (D, \text{rdf:type, diachron:DimensionProperty}) ) in ( V_1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach Attr to Dimension(D, attr)</th>
<th>Detach Attr from Dimension(D, attr)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Add an attribute property to dimension</td>
<td>Reverse of Attach Attr to Dimension(D, attr)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>D = The dimension in which the attribute is attached</td>
<td>D = The dimension in which the attribute is detached</td>
</tr>
<tr>
<td></td>
<td>attr = The attribute which is attached to dimension</td>
<td>attr = The attribute which is detached from dimension</td>
</tr>
<tr>
<td><strong>SPARQL used for detection</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>δ⁺</td>
<td>( (D, \text{diachron:hasAttribute, attr}) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>δ⁻</td>
<td>( \emptyset )</td>
<td>( (D, \text{diachron:hasAttribute, attr}) )</td>
</tr>
<tr>
<td>θ</td>
<td>( (D, \text{rdf:type, diachron:DimensionProperty}) ) in ( V_2 )</td>
<td>( (D, \text{rdf:type, diachron:DimensionProperty}) ) in ( V_1 )</td>
</tr>
</tbody>
</table>
### Entity: Observation

<table>
<thead>
<tr>
<th>Change</th>
<th>$Add_{Observation}(O)$</th>
<th>$Delete_{Observation}(O)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Add an observation entity</td>
<td>Reverse of $Add_{Observation}(O)$</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>$O =$ The observation which is added</td>
<td>$O =$ The observation which is deleted</td>
</tr>
<tr>
<td><strong>SPARQL used for detection</strong></td>
<td>SELECT ?o WHERE { GRAPH &lt;added&gt; { ?o rdf:type diachron:Observation. } }</td>
<td>SELECT ?o WHERE { GRAPH &lt;deleted&gt; { ?o rdf:type diachron:Observation. } }</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(O, \text{rdf:type, diachron:Observation})$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>$(O, \text{rdf:type, diachron:Observation})$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>$Attach_{Observation_to_FT}(O, FT)$</th>
<th>$Detach_{Observation_from_FT}(O, FT)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Attach an observation to a fact table</td>
<td>Reverse of $Attach_{Observation_to_FT}(O, FT)$</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>$O =$ The observation which attached to fact table</td>
<td>$O =$ The observation which is detached from fact table</td>
</tr>
<tr>
<td>$FT =$ The fact table</td>
<td>$FT =$ The fact table</td>
<td></td>
</tr>
<tr>
<td><strong>SPARQL used for detection</strong></td>
<td>SELECT ?o ?ft WHERE { GRAPH &lt;added&gt; { ?o diachron:subject ?ft. } }</td>
<td>SELECT ?o ?ft WHERE { GRAPH &lt;deleted&gt; { ?o diachron:subject ?ft. } }</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(O, \text{diachron:subject, FT})$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>$(O, \text{diachron:subject, FT})$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
### Change 1: Add Measure Value to Observation (O, M, V)

**Intuition**: Add value to a measure in a specific observation

**Parameters**
- \(O\) = The observation
- \(M\) = The measure on which the observation refers
- \(V\) = The value of the measure

**SPARQL used for detection**

```sparql
SELECT ?o ?m ?v WHERE {
  GRAPH <V2> {
    ?ratt rdf:type diachron:RecordAttribute.
    ?o diachron:hasRecordAttribute ?ratt.
    ?ratt diachron:predicate ?m.
    GRAPH <added> {
      ?m rdf:type diachron:MeasureProperty.
    }
  }
}
```

**\(\delta^+\)**
- \((O, \text{diachron} : \text{hasRecordAttribute}, \text{ratt1})\)
- \((\text{ratt1}, \text{rdf} : \text{type}, \text{diachron} : \text{RecordAttribute})\)
- \((\text{ratt1}, \text{diachron} : \text{predicate}, M)\)
- \((\text{ratt1}, \text{diachron} : \text{object}, V)\)

**\(\delta^-\)**
- \(\emptyset\)

**\(\phi\)**
- \((M, \text{rdf} : \text{type}, \text{diachron} : \text{MeasureProperty})\) in \(V_2\)
- \((M, \text{rdf} : \text{type}, \text{diachron} : \text{MeasureProperty})\) in \(V_1\)

### Assumption:
We assume that the transformation of the original data to the DIACHRON model ensures that for each record attribute \(\text{ratt1}\) all the following triples should appear:

- \((O, \text{diachron} : \text{hasRecordAttribute}, \text{ratt1})\)
- \((\text{ratt1}, \text{rdf} : \text{type}, \text{diachron} : \text{RecordAttribute})\)
- \((\text{ratt1}, \text{diachron} : \text{predicate}, M)\)
- \((\text{ratt1}, \text{diachron} : \text{object}, V)\)

### Change 2: Delete Dimension Value from Observation (O, D, V)

**Intuition**: Delete value from a dimension in a specific observation

**Parameters**
- \(O\) = The observation
- \(D\) = The dimension on which the observation refers
- \(V\) = The value of the dimension

**SPARQL used for detection**

```sparql
SELECT ?o ?d ?v WHERE {
  GRAPH <V2> {
    ?o diachron:hasRecordAttribute ?ratt.
    ?ratt rdf:type diachron:RecordAttribute.
    ?ratt diachron:predicate ?d.
    GRAPH <deleted> {
      ?ratt diachron:hasRecordAttribute ?ratt.
      ?ratt rdf:type diachron:RecordAttribute.
      ?ratt diachron:predicate ?d.
    }
  }
}
```

**\(\delta^+\)**
- \((O, \text{diachron} : \text{hasRecordAttribute}, \text{ratt1})\)
- \((\text{ratt1}, \text{rdf} : \text{type}, \text{diachron} : \text{RecordAttribute})\)
- \((\text{ratt1}, \text{diachron} : \text{predicate}, M)\)
- \((\text{ratt1}, \text{diachron} : \text{object}, V)\)

**\(\delta^-\)**
- \(\emptyset\)

**\(\phi\)**
- \((D, \text{rdf} : \text{type}, \text{diachron} : \text{DimensionProperty})\) in \(V_2\)
- \((D, \text{rdf} : \text{type}, \text{diachron} : \text{DimensionProperty})\) in \(V_1\)
### Entity: Codelist/Hierarchy

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Codelist(C)</th>
<th>Delete Codelist(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new codelist</td>
<td>Reverse of Add Codelist(C)</td>
</tr>
<tr>
<td>Parameters</td>
<td>C = The added codelist</td>
<td>C = The deleted codelist</td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?c WHERE { GRAPH &lt;added&gt; { ?c rdf:type diachron:CodeList. }}</td>
<td>SELECT ?c WHERE { GRAPH &lt;deleted&gt; { ?c rdf:type diachron:CodeList. }}</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>(C, rdf : type, diachron : Codelist)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>(C, rdf : type, diachron : Codelist)</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Hierarchy(H)</th>
<th>Delete Hierarchy(H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new hierarchy</td>
<td>Reverse of Add Hierarchy(H)</td>
</tr>
<tr>
<td>Parameters</td>
<td>H = The added hierarchy</td>
<td>H = The deleted hierarchy</td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?h WHERE { GRAPH &lt;added&gt; { ?h rdf:type diachron:Hierarchy. }}</td>
<td>SELECT ?h WHERE { GRAPH &lt;deleted&gt; { ?h rdf:type diachron:Hierarchy. }}</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>(H, rdf : type, diachron : Hierarchy)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>(H, rdf : type, diachron : Hierarchy)</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach Codelist to Dimension(D, C)</th>
<th>Detach Codelist from Dimension(D, C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Assign a codelist to a dimension</td>
<td>Reverse of Attach Codelist to Dimension(D, C)</td>
</tr>
<tr>
<td>Parameters</td>
<td>D = The dimension in which the codelist is attached</td>
<td>D = The dimension in which the codelist is detached</td>
</tr>
<tr>
<td>C = The codelist which is attached to dimension</td>
<td>C = The codelist which is detached from dimension</td>
<td></td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?d ?c WHERE { GRAPH &lt;added&gt; { ?d diachron:hasCodelist ?c. }}</td>
<td>SELECT ?d ?c WHERE { GRAPH &lt;deleted&gt; { ?d diachron:hasCodelist ?c. }}</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>(D, diachron : hasCodelist, C)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>(D, diachron : hasCodelist, C)</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>Change</td>
<td>Attach_Hierarchy_to_Dimension(D, H)</td>
<td>Detach_Hierarchy_from_Dimension(D, H)</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>Intuition</td>
<td>Assign an hierarchy to a dimension</td>
<td>Reverse of Attach_Hierarchy_to_Dimension(D, H)</td>
</tr>
<tr>
<td>Parameters</td>
<td>( D = ) The dimension in which the hierarchy is attached ( H = ) The hierarchy which is attached to dimension</td>
<td>( D = ) The dimension in which the hierarchy is detached ( H = ) The hierarchy which is detached from dimension</td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?d ?h WHERE { GRAPH &lt;added&gt; { ?d diachron:hasHierarchy ?h. } }</td>
<td>SELECT ?d ?h WHERE { GRAPH &lt;deleted&gt; { ?d diachron:hasHierarchy ?h. } }</td>
</tr>
</tbody>
</table>

\( \delta^+ \) \((D, \text{diachron:hasHierarchy}, H)\) 0

\( \delta^- \) 0 \((D, \text{diachron:hasHierarchy}, H)\)

\( \phi \) –

<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Instance(I)</th>
<th>Delete_Instance(I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new instance</td>
<td>Reverse of Add_Instance(I)</td>
</tr>
<tr>
<td>Parameters</td>
<td>( I = ) The added instance</td>
<td>( I = ) The deleted instance</td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?i WHERE { GRAPH &lt;added&gt; { ?i rdf:type diachron:CodeListTerm. } }</td>
<td>SELECT ?i WHERE { GRAPH &lt;deleted&gt; { ?i rdf:type diachron:CodeListTerm. } }</td>
</tr>
</tbody>
</table>

\( \delta^+ \) \((I, \text{rdf:type}, \text{diachron:CodeListTerm})\) 0

\( \delta^- \) 0 \((I, \text{rdf:type}, \text{diachron:CodeListTerm})\)

\( \phi \) –

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach_Instance_to_Codelist(C, I)</th>
<th>Detach_Instance_from_Codelist(C, I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new instance into a codelist</td>
<td>Reverse of Attach_Instance_to_Codelist(C, I)</td>
</tr>
<tr>
<td>Parameters</td>
<td>( C = ) The codelist in which the instance is attached ( I = ) The instance which is attached to codelist</td>
<td>( C = ) The codelist in which the instance is attached ( I = ) The instance which is attached to codelist</td>
</tr>
</tbody>
</table>

\( \delta^+ \) \((I, \text{diachron:inScheme}, C)\) 0

\( \delta^- \) 0 \((I, \text{diachron:inScheme}, C)\)

\( \phi \) \((C, \text{rdf:type}, \text{diachron:CodeList})\) in \( V_2 \)

\((C, \text{rdf:type}, \text{diachron:CodeList})\) in \( V_1 \)

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach_Instance_to_Hierarchy(H, I)</th>
<th>Detach_Instance_from_Hierarchy(H, I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new instance into a hierarchy</td>
<td>Reverse of Attach_Instance_to_Hierarchy(H, I)</td>
</tr>
<tr>
<td>Parameters</td>
<td>( H = ) The hierarchy in which the instance is attached ( I = ) The instance which is attached to hierarchy</td>
<td>( H = ) The hierarchy in which the instance is attached ( I = ) The instance which is attached to hierarchy</td>
</tr>
</tbody>
</table>

\( \delta^+ \) \((I, \text{diachron:inScheme}, H)\) 0

\( \delta^- \) 0 \((I, \text{diachron:inScheme}, H)\)

\( \phi \) \((H, \text{rdf:type}, \text{diachron:Hierarchy})\) in \( V_2 \)

\((H, \text{rdf:type}, \text{diachron:Hierarchy})\) in \( V_1 \)
<table>
<thead>
<tr>
<th>Change</th>
<th>Attach_Instance_to_Parent(I, P)</th>
<th>Detach_Instance_to_Parent(I, P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Associate an instance with its parent</td>
<td>Reverse of Attach_Instance_to_Parent(I, P)</td>
</tr>
</tbody>
</table>
| Parameters | I = The instance  
P = The parent which is added to instance | I = The instance  
P = The parent which is deleted from instance |
| SPARQL used for detection | SELECT ?i ?p WHERE {  
GRAPH <added> {  
}  
} | SELECT ?i ?p WHERE {  
GRAPH <deleted> {  
}  
} |
| δ⁺ | \((I, \text{diachron:hasParent}, P)\) | \(\emptyset\) |
| δ⁻ | \(\emptyset\) | \((I, \text{diachron:hasParent}, P)\) |
| θ | – | – |
### Entity: Measure

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Measure(M)</th>
<th>Delete Measure(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Add a new measure</td>
<td>Reverse of Add Measure(M)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>M = The measure which is added</td>
<td>M = The measure which is deleted</td>
</tr>
<tr>
<td><strong>SPARQL used for detection</strong></td>
<td><code>SELECT ?m WHERE { GRAPH &lt;added&gt; { ?m rdf:type diachron:MeasureProperty. }}</code></td>
<td><code>SELECT ?m WHERE { GRAPH &lt;deleted&gt; { ?m rdf:type diachron:MeasureProperty. }}</code></td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(M, \text{rdf:type, diachron:MeasureProperty})$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>$(M, \text{rdf:type, diachron:MeasureProperty})$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach Type to Measure(T, M)</th>
<th>Detach Type from Measure(T, M)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Add a new datatype to measure</td>
<td>Reverse of Attach Type to Measure(T, M)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>T = The added type to measure</td>
<td>T = The deleted type from measure</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>M = The measure in which the type is added</td>
<td>M = The measure in which the type is deleted</td>
</tr>
<tr>
<td><strong>SPARQL used for detection</strong></td>
<td><code>SELECT ?m ?t WHERE { GRAPH &lt;V2&gt; { ?m rdf:type diachron:MeasureProperty. GRAPH &lt;added&gt; { ?m rdfs:range ?t. }} }</code></td>
<td><code>SELECT ?m ?t WHERE { GRAPH &lt;V1&gt; { ?m rdf:type diachron:MeasureProperty. GRAPH &lt;deleted&gt; { ?m rdfs:range ?t. }} }</code></td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(M, \text{rdfs:range, T})$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>$(M, \text{rdfs:range, T})$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$(M, \text{rdf:type, diachron:MeasureProperty})$ in $V_2$</td>
<td>$(M, \text{rdf:type, diachron:MeasureProperty})$ in $V_1$</td>
</tr>
</tbody>
</table>
### Entity: Fact Table

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Fact Table(FT)</th>
<th>Delete Fact Table(FT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new fact table</td>
<td>Reverse of Add Fact Table(FT)</td>
</tr>
<tr>
<td>Parameters</td>
<td>FT = The fact table is added</td>
<td>FT = The fact table which is deleted</td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?f WHERE { GRAPH &lt;added&gt; { ?f rdf:type diachron:FactTable. } }</td>
<td>SELECT ?f WHERE { GRAPH &lt;deleted&gt; { ?f rdf:type diachron:FactTable. } }</td>
</tr>
<tr>
<td>δ⁺</td>
<td>(FT, rdf:type, diachron:FactTable)</td>
<td>0</td>
</tr>
<tr>
<td>δ⁻</td>
<td>0</td>
<td>(FT, rdf:type, diachron:FactTable)</td>
</tr>
<tr>
<td>φ</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach Measure to Fact Table(M,FT)</th>
<th>Detach Measure from Fact Table(M,FT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a measure property to fact table</td>
<td>Reverse of Attach Measure to Fact Table(M,FT)</td>
</tr>
<tr>
<td>Parameters</td>
<td>M = The added measure to fact table</td>
<td>M = The deleted measure to fact table</td>
</tr>
<tr>
<td>FT = The fact table in which the measure is added</td>
<td>FT = The fact table in which the measure is deleted</td>
<td></td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?m ?ft WHERE { GRAPH &lt;added&gt; { ?ft diachron:hasMeasure ?m. } }</td>
<td>SELECT ?m ?ft WHERE { GRAPH &lt;deleted&gt; { ?ft diachron:hasMeasure ?m. } }</td>
</tr>
<tr>
<td>δ⁺</td>
<td>(FT,diachron:hasMeasure,M)</td>
<td>0</td>
</tr>
<tr>
<td>δ⁻</td>
<td>0</td>
<td>(FT,diachron:hasMeasure,M)</td>
</tr>
<tr>
<td>φ</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Attach Dimension to Fact Table(D,FT)</th>
<th>Detach Dimension from Fact Table(D,FT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a dimension property to fact table</td>
<td>Reverse of Attach Dimension to Fact Table(D,FT)</td>
</tr>
<tr>
<td>Parameters</td>
<td>D = The added dimension to fact table</td>
<td>D = The deleted dimension to fact table</td>
</tr>
<tr>
<td>FT = The fact table in which the dimension is added</td>
<td>FT = The fact table in which the dimension is deleted</td>
<td></td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?d ?ft WHERE { GRAPH &lt;added&gt; { ?ft diachron:hasDimension ?d. } }</td>
<td>SELECT ?d ?ft WHERE { GRAPH &lt;deleted&gt; { ?ft diachron:hasDimension ?d. } }</td>
</tr>
<tr>
<td>δ⁺</td>
<td>(FT,diachron:hasDimension,D)</td>
<td>0</td>
</tr>
<tr>
<td>δ⁻</td>
<td>0</td>
<td>(FT,diachron:hasDimension,D)</td>
</tr>
<tr>
<td>φ</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
### Entity: Attribute

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Attribute(attr)</th>
<th>Delete Attribute(attr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new attribute</td>
<td>Reverse of Add Attribute(attr)</td>
</tr>
<tr>
<td>Parameters</td>
<td>attr = The attribute which is added</td>
<td>attr = The attribute which is deleted</td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?a WHERE { GRAPH &lt;added&gt; { ?a rdf:type diachron:AttributeProperty. }}</td>
<td>SELECT ?a WHERE { GRAPH &lt;deleted&gt; { ?a rdf:type diachron:AttributeProperty. }}</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>(attr rdf : type, diachron : AttributeProperty)</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$\emptyset$</td>
<td>(attr rdf : type, diachron : AttributeProperty)</td>
</tr>
<tr>
<td>$\phi$</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>
## Generic Changes

<table>
<thead>
<tr>
<th>Change</th>
<th>( Add_Relevancy(X,Y) )</th>
<th>( Delete_Relevancy(X,Y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a property to denote relevancy between two arguments</td>
<td>Reverse of ( Add_Relevancy(X,Y) )</td>
</tr>
<tr>
<td>Parameters</td>
<td>( X = ) The first argument&lt;br&gt;( Y = ) The second argument</td>
<td>( X = ) The first argument&lt;br&gt;( Y = ) The second argument</td>
</tr>
</tbody>
</table>
| SPARQL used for detection | SELECT ?x ?y WHERE {
GRAPH \(<added>\) {
\(?x \text{ diachron:isRelevantTo } ?y\)
}
} | SELECT ?x ?y WHERE {
GRAPH \(<deleted>\) {
\(?x \text{ diachron:isRelevantTo } ?y\)
}
} |
| \( \delta^+ \) | \( (X, \text{diachron:isRelevantTo}, Y) \) | \( \phi \) |
| \( \delta^- \) | \( \emptyset \) | \( (X, \text{diachron:isRelevantTo}, Y) \) |
| \( \phi \) | \( \) | \( \) |

---

<table>
<thead>
<tr>
<th>Change</th>
<th>( Add_Label(S,P,O) )</th>
<th>( Delete_Label(S,P,O) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new label with the specified name (different than ( rdfs:label ))</td>
<td>Reverse of ( Add_Label(S,P,O) )</td>
</tr>
<tr>
<td>Parameters</td>
<td>( S = ) The subject in which the label is added&lt;br&gt;( P = ) The property (name of label)&lt;br&gt;( O = ) The object which is related with the subject via the property</td>
<td>( S = ) The subject in which the label is deleted&lt;br&gt;( P = ) The property (name of label)&lt;br&gt;( O = ) The object which is related with the subject via the property</td>
</tr>
</tbody>
</table>
| SPARQL used for detection | SELECT ?s ?p ?o WHERE {
GRAPH \(<V2>\) {
\{ ?p rdfs:subClassOf rdfs:label \}
}
UNION
GRAPH \(<added>\) {
\(?s \text{ ?p } ?o.\)
}
} | SELECT ?s ?p ?o WHERE {
GRAPH \(<V1>\) {
\{ ?p rdfs:subClassOf rdfs:label \}
}
UNION
GRAPH \(<deleted>\) {
\(?s \text{ ?p } ?o.\)
}
} |
<p>| ( \delta^+ ) | ( (S, P, O) ) | ( \emptyset ) |
| ( \delta^- ) | ( \emptyset ) | ( (S, P, O) ) |
| ( \phi ) | ( ) | ( ) |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>Add Unknown Property (S, P, O)</th>
<th>Delete Unknown Property (S, P, O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a new property with specified subject and object related</td>
<td>Reverse of Add Unknown Property (S, P, O)</td>
</tr>
<tr>
<td>Parameters</td>
<td>S = The subject in which the property is added</td>
<td>S = The subject in which the property is deleted</td>
</tr>
<tr>
<td></td>
<td>P = The property</td>
<td>P = The property</td>
</tr>
<tr>
<td></td>
<td>O = The object which is related with the subject via the property</td>
<td>O = The object which is related with the subject via the property</td>
</tr>
</tbody>
</table>

**SPARQL used for detection**

```sql
SELECT ?s ?p ?o WHERE {
  FILTER NOT EXISTS { GRAPH <V2> { ?p rdfs:subClassOf rdfs:label} } 
  FILTER (?p != rdf:type || 
    diachron:Observation).
  FILTER (?p != diachron:hasMeasure).
  FILTER (?p != diachron:hasDimension).
  FILTER (?p != diachron:hasParent).
  FILTER (?p != diachron:hasHierarchy).
  FILTER (?p != diachron:inScheme).
  FILTER (?p != diachron:hasCodelist).
  FILTER (?p != diachron:hasMeasure).
  FILTER (?p != diachron:hasRelevant).
  FILTER (?p != rdf:type). 
  diachron:DimensionProperty).
  FILTER (?p != rdf:type).
  diachron:RecordAttribute).
  FILTER (?p != rdf:type).
  diachron:CodeList).
  FILTER (?p != rdf:type).
  diachron:CodeListTerm).
  FILTER (?p != rdf:type).
  diachron:MeasureProperty).
  FILTER (?p != rdf:type).
  diachron:FactTable).
  FILTER (?p != rdf:type).
  diachron:AttributeProperty).
}
```

```sql
SELECT ?s ?p ?o WHERE {
  FILTER NOT EXISTS { GRAPH <V1> { ?p rdfs:subClassOf rdfs:label} } 
  UNION 
  { GRAPH <V1> { FILTER (?p != rdfs:label). } }
  GRAPH <added> { 
    GRAPH <V2> { FILTER (?p != rdfs:label). }
  }
}
```

---

Δ  (S, P, O)  ∅

Δ  0

(p # rdfs:range ∧ 
  p # diachron:hasAttribute ∧
  p # diachron:hasRecordAttribute ∧
  p # diachron:subject ∧ 
  p # diachron:object ∧
  p # diachron:hasHierarchy ∧
  p # diachron:inScheme ∧ 
  p # diachron:hasParent ∧
  p # diachron:hasDimension ∧
  p # diachron:hasMeasure ∧
  p # diachron:isRelevant ∧
  p # rdfs:label ∧
  (p, rdfs:subClassOf, rdfs:label) not in V₂) ∧

(p # rdfs:range ∧
  p # diachron:hasAttribute ∧
  p # diachron:hasRecordAttribute ∧
  p # diachron:subject ∧
  p # diachron:object ∧
  p # diachron:hasHierarchy ∧
  p # diachron:inScheme ∧
  p # diachron:hasParent ∧
  p # diachron:hasDimension ∧
  p # diachron:hasMeasure ∧
  p # diachron:isRelevant ∧
  p # rdfs:label ∧
  (p, rdfs:subClassOf, rdfs:label) not in V₂) ∧

(p # rdfs:range ∧
  p # diachron:hasAttribute ∧
  p # diachron:hasRecordAttribute ∧
  p # diachron:subject ∧
  p # diachron:object ∧
  p # diachron:hasHierarchy ∧
  p # diachron:inScheme ∧
  p # diachron:hasParent ∧
  p # diachron:hasDimension ∧
  p # diachron:hasMeasure ∧
  p # diachron:isRelevant ∧
  p # rdfs:label ∧
  (p, rdfs:subClassOf, rdfs:label) not in V₂) ∧
### Intuition

Add a data type to a given subject

Reverse of Add_Datatype($X$, $T$)

### Parameters

$X$ = The subject in which the datatype is added

$T$ = The added datatype

$X$ = The subject in which the datatype is deleted

$T$ = The deleted datatype

### SPARQL used for detection

#### Add_Datatype($X$, $T$)

```sparql
SELECT ?x ?t WHERE {
    GRAPH <V2> {
        FILTER NOT EXISTS {
            {?x rdf:type diachron:DimensionProperty.}
            UNION
            {?x rdf:type diachron:MeasureProperty.}
        }
    }
    GRAPH <added> {
        ?x rdfs:range ?t.
    }
}
```

#### Delete_Datatype($X$, $T$)

```sparql
SELECT ?x ?t WHERE {
    GRAPH <V1> {
        FILTER NOT EXISTS {
            {?x rdf:type diachron:DimensionProperty.}
            UNION
            {?x rdf:type diachron:MeasureProperty.}
        }
    }
    GRAPH <deleted> {
        ?x rdfs:range ?t.
    }
}
```

### Change

Add a Generic Attribute to a given subject

Reverse of Add_Generic_Attribute($X$, $attr$)

### Parameters

$X$ = The subject in which the attribute is added

$attr$ = The added attribute

$X$ = The subject in which the attribute is deleted

$attr$ = The deleted attribute

### SPARQL used for detection

#### Add_Generic_Attribute($X$, $attr$)

```sparql
SELECT ?x ?att WHERE {
    GRAPH <V2> {
        FILTER NOT EXISTS {
            {?x rdf:type diachron:DimensionProperty.}
            UNION
            {?x rdf:type diachron:MeasureProperty.}
        }
    }
    GRAPH <added> {
        ?x diachron:hasAttribute ?att.
    }
}
```

#### Delete_Generic_Attribute($X$, $attr$)

```sparql
SELECT ?x ?t WHERE {
    GRAPH <V1> {
        FILTER NOT EXISTS {
            {?x rdf:type diachron:DimensionProperty.}
            UNION
            {?x rdf:type diachron:MeasureProperty.}
        }
    }
    GRAPH <deleted> {
        ?x diachron:hasAttribute ?att.
    }
}
```
<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Generic_Value_to_Observation(O,P,V)</th>
<th>Delete_Generic_Value_from_Observation(O,P,V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a generic value to observation</td>
<td>Reverse of Add_Generic_Value_to_Observation(O,P,V)</td>
</tr>
<tr>
<td>Parameters</td>
<td>$O = $The observation $P = $The property related as predicate of the observation $V = $The added value</td>
<td>$O = $The observation $P = $The property related as predicate of the observation $V = $The deleted value</td>
</tr>
</tbody>
</table>
| SPARQL used for detection | SELECT ?x ?p ?v WHERE {
|                          | GRAPH <V2> {
|                          | {p rdf:type diachron:DimensionProperty.} UNISON
|                          | {p rdf:type diachron:MeasureProperty.} }
|                          | GRAPH <added> {
|                          | ?o diachron:hasRecordAttribute ?ratt.
|                          | ?ratt rdf:type diachron:RecordAttribute.
|                          | }
|                          | }                                                                                                          | SELECT ?x ?p ?v WHERE {
|                          | GRAPH <V1> {
|                          | {p rdf:type diachron:DimensionProperty.} UNISON
|                          | {p rdf:type diachron:MeasureProperty.} }
|                          | GRAPH <deleted> {
|                          | ?o diachron:hasRecordAttribute ?ratt.
|                          | ?ratt rdf:type diachron:RecordAttribute.
|                          | }                                                                                                          |
| $\delta^+$                | $(O, diachron:hasRecordAttribute, ratt1) (ratt1, rdf:type, diachron:RecordAttribute) (ratt1, diachron:object, V) | $\emptyset$                                                                                               |
| $\delta^-$                | $\emptyset$                                                                                              | $(O, diachron:hasRecordAttribute, ratt1) (ratt1, rdf:type, diachron:RecordAttribute) (ratt1, diachron:object, V) |
| $\emptyset$               | $(P, rdf:type, diachron:DimensionProperty) not in V$_2$ $\land$ $(P, rdf:type, diachron:MeasureProperty) not in V$_2$ | $(P, rdf:type, diachron:DimensionProperty) not in V$_1$ $\land$ $(P, rdf:type, diachron:MeasureProperty) not in V$_1$      |

**Assumption:** We assume that the transformation of the original data to the DIACHRON model ensures that for each record attribute `ratt1` all the following triples should appear:

- $(O, diachron:hasRecordAttribute, ratt1) $
- $(ratt1, rdf:type, diachron:RecordAttribute)$ $
- $(ratt1, diachron:attribute, P)$ $
- $(ratt1, diachron:object, V)$

<table>
<thead>
<tr>
<th>Change</th>
<th>Add_inScheme(X,S)</th>
<th>Delete_inScheme(X,S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add a scheme to a given subject</td>
<td>Reverse of Add_inScheme(X,S)</td>
</tr>
<tr>
<td>Parameters</td>
<td>$X = $The subject in which the scheme is added $S = $The added scheme</td>
<td>$X = $The subject in which the scheme is deleted $S = $The deleted scheme</td>
</tr>
</tbody>
</table>
| SPARQL used for detection | SELECT ?x ?s WHERE {
|                          | GRAPH <V2> {
|                          | {?s rdf:type diachron:Codelist.} UNISON
|                          | {?s rdf:type diachron:Hierarchy.} }
|                          | GRAPH <added> {
|                          | ?x diachron:inScheme ?s.}
|                          | }                                                                                                          | SELECT ?x ?s WHERE {
|                          | GRAPH <V1> {
|                          | {?s rdf:type diachron:Codelist.} UNISON
|                          | {?s rdf:type diachron:Hierarchy.} }
|                          | GRAPH <deleted> {
|                          | ?x diachron:inScheme ?s.}
|                          | }                                                                                                          |
| $\delta^+$                | $(X, diachron:inScheme, S)$                                                                                 | $\emptyset$                                                                                               |
| $\delta^-$                | $\emptyset$                                                                                              | $(S, rdf:type, diachron:Codelist) not in V$_2$ $\land$ $(S, rdf:type, diachron:Hierarchy) not in V$_2$       |
| $\emptyset$               | $(S, rdf:type, diachron:Codelist) not in V$_1$ $\land$ $(S, rdf:type, diachron:Hierarchy) not in V$_1$       | $(S, rdf:type, diachron:Codelist) not in V$_1$ $\land$ $(S, rdf:type, diachron:Hierarchy) not in V$_1$       |

Grant Agreement No. 601043