CLEENEX: Iterative Data Cleaning with User Intervention

Cátia Borges Ormonde
catia.ormonde@tecnico.ulisboa.pt

Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal

October 2017

Abstract

Data cleaning is the process of correcting data quality problems, in datasets, to foster their fitness for use. Most data cleaning solutions’ tendency is to automate their tasks. However, data cleaning processes may require human knowledge to achieve optimal results. Hence, to provide results of excellence, data cleaning processes should be iterative and support user involvement. CLEENEX is a data cleaning framework that addresses the need for cleaning data iteratively. It enables the specification of data cleaning processes as graphs of data transformations and supports user intervention, recurring to Data Cleaning Graphs (DCGs) equipped with Quality Constraints (QCs) and Manual Data Repairs (MDRs). This thesis’ objective is to allow the iterative execution of DCGs, improving its user intervention, allowing the support for MDRs and overall reducing the user effort required. The exhaustive experiments conducted proved to validate the implementations done over this framework, reducing the user effort successfully, and evidencing its effectiveness.

Keywords: Data Cleaning; Data Quality; User Intervention; Quality Constraint; Manual Data Repair; Iterative Execution

1. Introduction

Data is all around us, and with the emergence of innovative techniques and solutions to extract value from it, it is crucial that datasets are fit for use; particularly, they should conform to domain-specific quality standards [11]. In order to resolve the data quality problems found in databases, data cleaning processes are conducted.

There is already a good arsenal of approaches for data cleaning at our disposal [1]. Data cleaning solutions that completely automate the data cleaning tasks may not always be adequate - namely for large quantities of data - because it is difficult to create a data cleaning process, at a first try, that addresses all the data quality problems that exist in a dataset. Data cleaning processes usually require refinement to produce data with the idealized quality [12]. Hence, data cleaning should be seen as an iterative task, that can be gradually refined by a user with solid domain knowledge.

CLEENEX is a data cleaning framework, whose fundamentals were introduced in [7]. It is based on the Ajax [5][6] framework, and its goal is to solve data quality problems. CLEENEX allows the specification of data cleaning processes as DCGs, which are graphs of data transformations.

In CLEENEX, data cleaning processes are meant to be refined and guided by users. To complement DCGs, and provide support for user intervention, QCs and MDRs can be used.

Upon the start of this thesis, we identified some limitations of the CLEENEX prototype, in what concerns the execution of the DCGs and the incorporation of user interaction. For addressing such limitations, we proposed the contributions: (i) Adding support for the MDRs, in order for CLEENEX to allow the creation and application of all the supported MDR actions (insert, update and delete) chosen by the user; (ii) Implementation of the iterative execution of data cleaning processes, according to the defined operational semantics; and inclusion of the notions of deterministic and non deterministic attributes, in the data cleaning programs’ definition; (iii) Incorporation of a feature to allow the persistent storage of MDRs and a complementary option to allow the user to pause and reload the data cleaning process; (iv) Inclusion of the notion of MDR instance conflicts, as well as a functionality to allow the detection and resolution of the MDR instance conflicts; (v) Inclusion of new Graphical User Interface (GUI) components, and update existing ones; to conclude, we performed
(vi) an exhaustive experimental validation, in order to prove that, with the functionalities implemented, the user effort required is reduced, and the data cleaning process’ effectiveness is as expected.

This document is organized in six Chapters, as follows. Chapter 2 presents the CLEENEX framework. Chapter 3 summarizes relevant related work. Chapter 4 explains the contributions done, in the context of the thesis. Chapter 5 describes the experimental validations conducted. Chapter 6 presents our conclusions, as well as relevant guidance for future work.

2. CLEENEX

The CLEENEX framework allows the specification of data cleaning processes through its specification language, based on Structured Query Language (SQL).

This framework’s goal is to allow the user to intervene in a data cleaning process, in order to guarantee the output of data with the best quality possible.

2.1. Logical and Physical Levels

Typically, the development of a data cleaning process encloses two phases: (i) designing the data transformations’ graph; and (ii) designing and setting the adequate heuristics. Considering this, the CLEENEX framework is divided in two corresponding levels: (i) logical level: where the graph is described, using a declarative language, based on SQL; and (ii) physical level: where specific algorithms can be selected, to perform the logical operations.

2.2. Logical Operators

The five logical operators supported by CLEENEX, for the data transformations, are:

1. **Mapping**: takes a single relation as input and outputs one or more relations. It is able to standardize data formats or create records with a more suitable format;

2. **Matching**: considering two input relations, it locates the pairs of records that seem to refer the same object, through the usage of matching criteria, for their comparison;

3. **Clustering**: groups highly-matching record pairs together, according to a similarity value, given by a grouping criteria (e.g., by transitive closure);

4. **Merging**: returns a unique record, for each set of records included in a cluster (i.e., outputs a representative for each cluster);

5. **View**: corresponds to an arbitrary SQL query, augmented with some integrity checking over its results.

2.3. Quality Constraints and Manual Data Repairs

CLEENEX allows the intermediate data (along the graph) to be inspected and manually repaired. For such purpose, and with the intention of guiding the user, a DCG can be complemented by the constructs: QCs, that consist of quality integrity constraints that data should satisfy, and MDRs, which correspond to actions (insert, update and delete) that can be manually applied, over specifically defined updatable views, by the user.

By complementing a DCG with QCs and MDRs and using the MDRs as templates for user actions over the blamed records, we are constraining the choices a user has (i.e., when inspecting the uncleaned data), therefore making it easier to provide feedback. An example of a DCG’s excerpt is presented in Figure 1. This DCG’s objective is to clean a fictional dataset (i.e., the relation SurveyRes, presented in Figure 2). SurveyRes includes basic information about people who answered a car survey (i.e., their name, birth year, country, and email address), and their response to the survey (i.e., their preferred car brand, pBrand). The DCG uses two additional tables, Brand and Manufacturer - that refer to cars’ brands and manufacturers, respectively.

![Figure 1: SurveyRes’ DCG.](image)

![Figure 2: Subset of the SurveyRes relation, that contains a car survey results.](image)
which is used to reinforce that data records’ name, bYear and email attributes should be unique; and MDR1, which states that a possible manual repair to perform over violations to QC1 is to delete them. This DCG’s goal is to: (i) find if there are any duplicate records regarding the unique set of attributes name, bYear, email (QC1) and, if so, enable the user to eliminate them (creating an instance of MDR1); (ii) get each person’s preferred car brand manufacturer ID (transformation T2 and relation R2); (iii) make sure that each survey response (i.e., per email address) is associated to only one manufacturer name (T3 is used to get the manufacturer’s name); etc., and then (iv) save the cleaned data, with an updated schema, to its output (CleanSurveyRes table).

2.4. Architecture

Figure 3 depicts the CLEENEX framework’s main components. The components are described as:

- **GUI**: responsible for displaying information to the user (e.g., by graphically representing the data cleaning process);
- **Parser**: responsible for parsing the data cleaning processes specified by the programmer (e.g., performing the syntactical validation and interpretation);
- **Optimizer**: responsible for selecting the optimal execution plans, for processes’ execution;
- **Scheduler**: responsible for scheduling the execution of the tasks that constitute the execution plan (chosen by the Optimizer);
- **Catalog Manager**: responsible for generating the internal representation of the data cleaning processes;
- **Debugger**: responsible for triggering an inspection trail mechanism;
- **QC Manager**: responsible for parsing the QCs defined by the user, and managing their usage (e.g., computing the respective blamed tuples);
- **MDR Manager**: responsible for parsing the MDRs defined by the user, and managing their usage (e.g., enabling the MDRs’ usage, for a certain relation, when the respective conditions apply);
- **Database Manager**: responsible for communicating with the underlying Relational Database Management System (RDBMS). It conducts, for example, the creation of relations, and execution of SQL statements (as requested by the other components).

The CLEENEX framework’s execution phases are explained as follows:

1. **Library Specification**: allows the action of adding new, externally-defined algorithms and/or functions to the CLEENEX’ functions’ library, which exists for extensibility purposes;

2. **Program Specification**: the act of a programmer specifying a data cleaning process in CLEENEX (i.e., defining a DCG and respective QCs and MDRs);

3. **Optimization**: the act of CLEENEX to compile and optimize the specified data cleaning process;

4. **Execution**: after the compilation, CLEENEX proceeds to the execution of the previously defined data cleaning process. In this phase, during the execution of some operations, it can invoke externally-defined algorithms/functions, that may exist in the functions’ library. During the execution, the user can check the execution status (i.e., by watching the various elements of the graph’s workflow);

5. **Testing & Debugging**: after execution, the user can use the data derivation mechanism to navigate in the graph, in order to inspect the origins of data and the transformations applied to them;

6. **Refinement**: as the user tests and debugs the data along the DCG, some opportunities for data cleaning criteria refinement might be found. If such opportunities are identified, then the cleaning process can be better tailored, and re-executed.

3. Related Work

Data cleaning tasks can be automated or human guided. The latter implies considering user intervention to repair the data. It is also possible that data may have quality problems because the
data cleaning rules defined are outdated; therefore some approaches also focus on cleaning the corresponding integrity rules.

In the context of user involvement, we analyzed state of the art research and commercial solutions. To classify them, we used a taxonomy (Figure 4), which considers three main questions involved in data cleaning [4] (What to repair?, How to repair?, When to repair?).

Figure 4: Taxonomy for classifying data cleaning techniques, inspired by [3][8].

In respect of research solutions, we considered the works: Guided Data Repairs, Holistic Data Cleaning, NADEEF, LLUNATIC, Potter’s Wheel, TAILOR, Unified Repair, and Continuous Data Cleaning.

In what regards the commercial solutions, we considered those by the leading developers in the field (according to the Gartner Magic Quadrant for data quality); namely: Informatica, IBM, SAP, SAS, Talend and Oracle.

The majority of the research solutions analyzed are focused on integrity rules; while others consider both (e.g., LLUNATIC and CLEENEX), since the criteria is not mutually exclusive. On the other hand, in what regards the commercial tools, all of them use data transformations to conduct the data cleaning processes.

There has been significant progress in the design and implementation of data cleaning tools. This has led to an increase in the importance of providing means for: debugging and data validation; the interleaving of analysis and cleaning, during data cleaning processes; and the possibility of reducing the user effort required. The tendency is to enable the data cleaning systems to facilitate and automate rapid human-in-the-loop interactivity [9][10][2]. We noted that most tools integrate user feedback, provide a rich GUI, and some of them even have mechanisms for learning the user’s behavior (i.e., to automatize the decision making).

After the research conducted, we acknowledged that CLEENEX, apart from not using Machine Learning techniques, neither cleaning the data quality rules, nor following a holistic approach, provides support for the user to execute a data cleaning process, and intends for his/her effort to be reduced. CLEENEX’s goal is to enable an iterative execution of data cleaning processes; while integrating user actions, and re-applying them accordingly. In terms of commercial tools (which all use data transformations), we noted they were all automatic, focused on data transformations, and provided a rich GUI. Neither of them considered the repair of data quality rules. Additionally, not all the commercial tools provided debugging facilities - something that is available in CLEENEX.

4. Iterative Execution of a Data Cleaning Program

The CLEENEX framework prototype had various limitations, which were restraining the data cleaning processes from being executed properly. Such limitations were targeted by this thesis’ contributions, particularly: the prototype did not yet enable users to iteratively clean data effectively; neither to provide feedback (i.e., creating MDR instances), nor to have their feedback integrated into the data cleaning process. Another problem identified was that the mutability of data, upon the re-execution of a DCG, was being ignored. Additionally, there were some faults in terms of the GUI, specifically the lack of means to keep the user informed, and to support the creation and application of the MDR instances.

To enable the iterative execution of the DCGs, while incorporating user intervention, we addressed the existing limitations, with various contributions, categorized into three topics: (i) the support for MDRs, (ii) the iterative execution of a DCG, and (iii) GUI.

An illustration of CLEENEX’ main components is shown in Figure 5. The various implementations performed were done both in the server and the client sides of the CLEENEX framework.

Figure 5: Overview diagram of CLEENEX’ main components.

4.1. Support for MDRs

In the context of the support for MDRs, we started by enabling the creation and application of MDR instances. In order to support the three MDR actions (i.e., insert, delete and update), it was necessary to perform the “bridging” between the CLEENEX’ Core, and the GUI. Furthermore, it was necessary to refine and add components to the graphical interface, in order to allow the user to manually clean the data records intuitively, and with minor effort.

A visual representation of the workflow of information in CLEENEX, when a user creates an MDR instance, is depicted in Figure 6. The communication between the GUI and CLEENEX’ Core is done through various web services, which have differ-
ent purposes. For example, in this figure we are representing two of them: one to receive MDR instances’ creation requests, and another to notify the GUI of the MDR instances’ creation status (i.e., failure or success).

![MDR instance creation workflow](image)

Currently, whenever a user tries to create a new MDR instance, through CLEENEX’ GUI, a creation request is sent to CLEENEX’ Core (i.e., to the respective web service component). The information sent to the web service (“MDRs Controller”) is carried through a Data Transfer Object (DTO), which specifically contains information about the MDR to instantiate, the action to perform, the target tuple’s original values (if applicable), and the attributes that the user wants to execute the action on (e.g., the attributes to update). When the request reaches CLEENEX’ Core’s “MDRs Controller”, it is passed on to the Java class that is responsible for managing the MDRs (i.e., the MDR manager). Such class then creates an object to represent the MDR instance requested. While doing this, a SQL statement (depending on the MDR action chosen) is constructed, in order to be passed on to the RDBMS, for further execution.

After creating the MDR instance Java object, and sending the SQL statement to the RDBMS: upon the SQL statement’s execution, the MDR manager analyzes the SQL statement’s execution result, and notifies the “MDRs Instances Creation Reporter” that the request has been completed. The “MDRs Instances Creation Reporter” then sends a message to the GUI, to inform the user about the status of the MDR instance creation request.

In the CLEENEX’ Core, we distinguish the various types of MDR instances that can be instantiated by the MDR Manager. This is advantageous because provides some extensibility, for example, if we wish to add support for additional actions, in the future. The composition of the SQL statements, for each MDR instance created, is done according to the action chosen. For example: (i) an update MDR instance will have a SQL statement which corresponds to an update of the target tuple(s) on the RDBMS; (ii) an insert MDR instance will have a SQL statement that corresponds to a tuple insertion on the RDBMS; and (iii) a delete MDR instance will have a SQL statement that corresponds to a tuple removal.

The construction of the SQL statements was not being done correctly. To resolve it, we added a component, MdrQueryUtils, that contains utility functions related to the construction of SQL statements. For example, the composition of the where and set SQL clauses is now being done in that class. We opted for creating a single Java class in order to reuse code that was similar, for the MDR actions that use such SQL clauses.

4.2. Iterative Execution of a DCG

To implement the iterative execution of a DCG we had to ensure the operational semantics determined in [7] was being followed. The operational semantics is as described in Definition 1.

**Definition 1.** Let $\mathcal{G} = \langle G, \{Q, M\} \rangle$ be a data cleaning graph for a set $R_1, ..., R_n$ of input relations. Let $r_1, ..., r_n$ be instances of these relations and $M$ be a manual data repair state for $\mathcal{G}$, i.e., a function that assigns to every relation $R \in rels(G)$, a list of instances of manual data repairs over $R$. The result of executing $\mathcal{G}$ over $r_1, ..., r_n$ and $M$ is $\{\{\text{tuples}(R), \text{tuple}^{bl}(R)\} : R \in rels(G)\}$ calculated as follows:

1. for $i = 1$ to $n$
2. for each $r_i \in M(R_i)$ do
3. \( \text{vr} \leftarrow \text{compute_view(view}(i), \text{tuples}(R_i)) \)
4. \( \text{apply_mdr}(i, \text{vr}) \)
5. \( \text{tuples}(R_i) \leftarrow \text{propagate}(	ext{vr}) \)
6. end for
7. end for
8. for $i = 1$ to $n$
9. \( \text{tuples}^{bl}(R_i) \leftarrow \text{blamed}(	ext{tuples}(r_i)) \)
10. end for
11. for each $T \in \text{trans}(G)$ do
12. let \( \{R_i, R_n\} = ^{*}\{R\} \)
13. \( \text{tuples}(T^*) \leftarrow \text{T}(\text{tuples}(R_1), ..., \text{tuples}(R_n)) \)
14. for each $r^{*} \in M(T^*)$ do
15. \( \text{vr} \leftarrow \text{compute_view(view}(i), \text{tuples}(T^*)) \)
16. \( \text{apply_mdr}(i, \text{vr}) \)
17. \( \text{tuples}(T^*) \leftarrow \text{propagate}(	ext{vr}) \)
18. end for
19. \( \text{tuples}^{bl}(T^*) \leftarrow \text{blamed}(	ext{tuples}(T^*)) \)
20. end for
According to the operational semantics presented, the execution of a DCG should be as follows: considering each of the DCG's input relations, we traverse through the list of MDR instances created for it (if there are any). For each MDR instance that is assigned to that relation node, we compute the relation node's view, and apply the MDR instance. The application of the MDR instance results in an updated view (i.e., considering the action performed). If there are various MDR instances associated to the relation node, their application is done in the original order (i.e., order of creation). After applying the MDR instances, the blamed tuples are computed, for each node.

To proceed: for each transformation node that exists in the DCG, and considering its input relations, we compute (i.e., transform, according to the operation chosen) the resulting output relation node. Then, for each MDR instance associated to the outputted relation, we apply it over the computed relation view; and then propagate the resulting view, to the transformation node's output (i.e., to the upcoming nodes in the graph). After applying all the node's MDR instances, and having an updated view (i.e., the node's output), the blamed tuples are re-computed.

In CLEENEX, a new data cleaning process begins without any MDR instances. On its first execution, the system starts by computing the list of blamed tuples of a table (considering the QCs that are associated to that relation). If blamed tuples are found, the system then proceeds to compute the resulting view. After such view is computed, the user is able to consult the data records marked as blamed, as well as that view's results. When analyzing the data (namely the blamed tuples), if the user decides to perform MDRs, he/she creates the desired MDR instances. After creating the MDR instances successfully, the views are updated. When the user requests a re-execution of the DCG, the MDR instances created previously are re-applied to the respective views, in order of creation.

### 4.2.1 The DCG re-execution problem

An important problem related to the re-execution of MDR instances was found: essentially, not all the DCGs' MDR instances were being correctly re-executed because, eventually, the original values of the targeted tuples would change (i.e., the input dataset had been updated). This way, the existing MDR instances would become deprecated, and not be executed successfully, because the targeted data was not found (i.e., it had changed upon the re-execution).

The DCG re-execution problem was happening when the tuples' values were generated by certain externally defined functions (chosen by the user), that were, for example, returning variable/randomly generated values, even if the input was the same (i.e., they were returning non deterministic values). Due to such tuples' non deterministic values, when the DCG was re-executed, the corresponding MDR instances' actions would not be executed, because the initially targeted tuples had changed (i.e., upon re-execution). This situation - of an MDR instance's target tuple becoming deprecated/outdated - was named as an MDR instance conflict.

The CLEENEX framework allows its users to plug in any user-defined function. Therefore, it is crucial to keep in mind that the situation described above (i.e., the "MDR instance conflicts") can happen. We consider that it is the responsibility of the users who define the DCGs to identify if the external functions used are deterministic or non deterministic.

To solve this issue, we came up with a solution that deals with this kind of situations, in order to prevent the loss of MDR instances, and guarantee the correct execution of the MDRs' actions: (i) identifying the attributes that are non deterministic, in the data cleaning process' specification, and (ii) upon the re-execution of the DCG, detecting MDR instances that target records with non deterministic values; to verify if those MDR instances need to be updated or not (i.e., if an MDR instance conflict appeared). This verification, to detect the MDR instance conflicts, is done by a new component, the MDR conflict manager. It not only detects the conflicts, but also resolves them automatically, when possible; or provides means for resolving them, by requesting the user to pick the best option for the resolution (i.e., through a new GUI component).

When an MDR instance conflict is found, the user is notified through the GUI, as the respective node (to which the MDR instance is associated) has its color changed to orange, and a warning symbol appears (as exemplified in Figure 7).

When the “Resolve” button is clicked, the user is shown another pop-up window (Figure 8), that...
allows him/her to select the new (updated) source tuple (i.e., according to the new input data - which resulted from the DCG’s re-execution). After selecting the correct data record to which the MDR instance should be applied to, the user must select the “Apply” button, to conclude the updating of the whole information regarding that MDR instance, and apply it with success.

Another functionality implemented was the persistent storage of the MDRs and respective MDR instances. By performing this way, we are able to keep track of the data records. Additionally, we added a complementary feature to recover from a paused data cleaning process - by reloading all the MDR instances previously created.

4.3. Graphical User Interface
In terms of the GUI we did not only add support for the new features, but we also improved and corrected faults that existed in the prototype.

We changed the GUI to include new components for: the resolution of MDR instance conflicts - as shown in Figure 8; for the creation of MDR instances (insert and delete actions); and for displaying notifications to the user, in order for him/her to be more informed during the application of MDR instances.

Additionally, we corrected the filtering and sorting functions, for the tabular data’s columns (i.e., in the data browser component). We also enabled the automatic resizing of some GUI components (e.g., the buttons); and updated others. For example, we updated the JavaScript code related to the update MDR instances.

4.4. Discussion
The various implementations performed over the CLEENEX framework, in the context of this thesis, had the objective of enabling the iterative execution of the DCGs, while allowing the user to intervene in the data cleaning process, through MDRs.

To fully enable the two-way guidance that CLEENEX can provide, we started by performing the necessary implementations, to ensure the correct incorporation of the feedback given by the user. That is, we added support for the MDR instances’ creation and application.

In order to achieve the thesis’ goals, we came across various impediments. The main limitations found were related to the concept of MDRs. Particularly, the fact that an MDR did not consider the mutability of the data targeted by the respective MDR instances. To resolve this issue, we started by providing means for identifying the data that was prone to change (i.e., which we called the non deterministic attributes). We included support for the specification of non deterministic attributes, in the data cleaning programs, in order to allow CLEENEX to be aware of the possibility of having errors in the re-application of the MDR instances. Then, we added the notion of MDR instance conflicts, to flag eventual errors caused because of those non deterministic attributes (i.e., the errors that happen when an MDR instance is not applied because the target tuple(s) have changed). When MDR instance conflicts are found, they are associated to the respective MDRs. By having this notion, and performing the detection and resolution of the MDR instance conflicts that appear (i.e., during the execution of a DCG), we managed to create a functionality that allows the user to recover from errors (i.e., errors in the application of the MDR instances), without losing the work previously performed.

In terms of the re-execution of the MDR instances, during the iterative execution of a DCG, we found it important to store persistently all the MDRs and respective instances. We implemented the persistent storage and retrieval of the MDR-related data, and added an additional feature, to enable the recovery of the previously created MDR instances; which can be quite helpful in case the user wishes to pause the data cleaning process and recovery its status later.

In what regards the GUI, we improved some of the existing components (e.g., correcting some faults), and added others, to support the functionalities that were updated and/or implemented.

Overall, we consider the functionalities imple-
mented to be potential value adders, because they are enabling what was addressed by this thesis, and may have a positive impact in what regards the user experience (e.g., by providing better guidance to the user and reducing the effort required).

5. Experimental Validation
Although supplying feedback (i.e., through MDRs) to the data cleaning process requires effort, the actions performed by the user may be crucial to obtain data with higher quality, at the end of a data cleaning process. The goal is to minimize the user effort required, such that the trade-off between the data quality obtained and the effort required is worthwhile. Regarding this, we measure both the user effort required and the data cleaning processes’ effectiveness (i.e., considering the data quality produced).

In order to validate the implemented functionalities, we performed three distinct experiments. Each experiment was done for two datasets, from different business domains, and with distinct data quality problems.

5.1. Experimental Scenarios
Each experiment consists in the execution of a set of data cleaning processes. Table 1 contains information about each experiment and the respective main functionalities that they target (i.e., as shown in the first column, Targeted Validation), as well as the metrics used for such purpose (i.e., as mentioned in the second column, Metrics of Interest). A description of the validation metrics is presented in Section 5.3.

5.2. Data Cleaning Processes
The various experiments’ data cleaning processes’ conditions are explained accordingly:
- Experiment (A) From Manual to DCG:
  1. Manual: the user manually cleans the dataset. We consider that its output is completely accurate.
  2. DCG, with Further Cleaning: the data cleaning process is modeled and executed through a DCG, without neither QCs nor MDRs. The user manually cleans the data (i.e., to fix the remaining data quality problems), after the DCG’s execution.
  3. DCG, with QCs and MDRs: a data cleaning process is modeled and executed through a DCG with MDRs and/or QCs. During the execution of the process, the user interacts with the DCG, providing feedback.
- Experiment (B) Conflict Detection and Resolution: A data cleaning process is modeled and executed through a DCG with MDRs and/or QCs. During the execution of the process, the user interacts with the DCG, providing feedback.
  3. DCG, with QCs and MDRs: the MDR instance conflict detection and resolution functionality is not used.
  4. DCG, with QCs, MDRs and MDR instance Conflicts: the MDR instance conflict detection and resolution functionality is used.
- Experiment (C) Work Loss and Recovery: For this scenario we consider that the user’s machine undergoes an outage, when half of the required MDR instances had already been created.
  4.(a) With Work Loss: the user does not use the MDRs recovery functionality. The MDR instances created are irrecoverably lost.
  4.(b) With Work Loss and Recovery: the user takes advantage of the MDRs recovery functionality.

5.3. Validation Measures
The metrics considered are as follows:
- Effectiveness: we consider Precision, Recall and F1 Score, as follows.
  \[
  \text{Precision} = \frac{\#TP}{\#TP + \#FP} \quad (1)
  \]
  \[
  \text{Recall} = \frac{\#TP}{\#TP + \#FN} \quad (2)
  \]
  \[
  F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (3)
  \]
- User Effort:
  1. Manual Data Inspection:
     \- Inspection Effort: average number of characters inspected.
  2. Manual Data Update:
     \- Updating Effort: (i) number of characters modified; (ii) number of characters added; (iii) number of characters deleted.
  3. Repeated Manual Data Update: when the user has to repeat the work done (e.g., by re-creating the MDR instances lost) he/she is repeating actions. This results in an extra user effort, measured in terms of:
     \- Extra Updating Effort: (i) number of MDR instances created; (ii) number of MDR instances re-applied (with success).
Table 1: Experiments’ targeted functionalities and metrics of interest.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Targeted Validation</th>
<th>Metrics of Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) From Manual to DCG</td>
<td>✓</td>
<td>- Inspection Effort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Updating Effort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- F1 Score</td>
</tr>
<tr>
<td>(B) Conflict Detection and Resolution</td>
<td>✓</td>
<td>- Updating Effort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Extra Updating Effort</td>
</tr>
<tr>
<td>(C) Work Loss and Recovery</td>
<td>✓</td>
<td>- Updating Effort</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Extra Updating Effort</td>
</tr>
</tbody>
</table>

5.4. Results

Although the experiments were performed for two distinct datasets, we will only be presenting the results for one of them: the Publications dataset; because there was consistency in what regards the results (i.e., we noted similar improvements).

- Experiment (A): the results obtained, in terms of user effort, are shown in Table 2. The results regarding the effectiveness of the data cleaning processes executed are shown in Table 3. The results obtained for Precision, Recall, and F1 Score are “1”, which is the ideal value.

- Experiment (B): the results obtained, in terms of user effort, are shown in Table 4. We conclude that by using the MDR instance conflicts’ detection and resolution functionality, the User Effort was greatly reduced, by at least 38%.

- Experiment (C): the results obtained, in terms of user effort, are shown in Table 5. By analyzing its content we conclude that the usage of the MDR instances’ recovery is useful, as it guarantees less User Effort (i.e., both for Updating Effort and Repetition Effort), in case the user needs to “reload” the MDR instances.

Considering the various results obtained, for the experiments performed, we concluded that, with the new functionalities, the required user effort was proven to reduce. Furthermore, the data quality obtained always corresponded to what was idealized. Therefore, we consider that the work done apropos of this thesis is validated.

6. Conclusions

Considering the initial limitations of the CLEENEX prototype, we presented our contributions, which were made to improve the CLEENEX framework at several levels. The implementations done over the CLEENEX framework addressed the following: (i) support for MDRs, (ii) iterative execution of a DCG, and (iii) inclusion and/or refinement of GUI components. In terms of the (i) support for MDRs, we enabled the creation and execution of MDR instances, for the three supported actions (insert, update and delete). In what regards the (ii) iterative execution of a DCG, we implemented the necessary changes to ensure its conformity to the CLEENEX’ operational semantics; we implemented the persistent storage of MDRs; we added the notion of non deterministic attributes - in order for the framework to be aware of MDR instance conflicts; we added support for an important non functional requirement (which was previously being ignored) of enabling the recovery of MDRs’ application errors (i.e., by detecting and resolving those errors automatically). We also added a complementary feature, to allow the user to pause a data cleaning process. Regarding the (iii) GUI, we improved some limitations and/or faults it had, and added graphical components, to support the comprehension and usage of the newly added functionalities.

Overall, the functionalities implemented over the CLEENEX framework are value adders, because they do not only allow a correct execution of the DCGs, but also enable the incorporation of user feedback; which can be essential for the data cleaning processes to be effective, producing data with the idealized quality.

For future work, we believe the following topics would be advantageous: (i) Data Transformations’ Incremental Execution: to reduce the computational effort and the execution time, the application of data transformations could be done solely on the new data (i.e., the data that is new to the current iteration of the DCG); (ii) MDR Instances’ GUI Management: it could be advantageous to implement an additional GUI component, to allow the user to manage the MDR instances created. For example, to allow the user to discard (i.e., rollback) certain MDR instances; and (iii) Additional Evaluations: it would be interesting to conduct additional experiments, in order to calculate the computational overhead required for the execution of some of the new functionalities (e.g., the recovery of MDR instance conflicts).
Table 2: Experiment (A) results, regarding User Effort, for the Publications dataset.

<table>
<thead>
<tr>
<th>Data Cleaning Process</th>
<th>Updating Effort</th>
<th>Repetition Effort</th>
<th>Visualization Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Characters Modified</td>
<td># Characters Added</td>
<td># Characters Deleted</td>
</tr>
<tr>
<td>1. Manual</td>
<td>217107</td>
<td>111076</td>
<td>106029</td>
</tr>
<tr>
<td>2. DCG, with Further Manual Cleaning</td>
<td>193698</td>
<td>127143</td>
<td>66555</td>
</tr>
<tr>
<td>3. DCG, with QCs and MDRs</td>
<td>132910</td>
<td>118322</td>
<td>14588</td>
</tr>
</tbody>
</table>

Table 3: Experiment (A) results, regarding Effectiveness, for the Publications dataset. NA = Not Applicable.

<table>
<thead>
<tr>
<th>Data Cleaning Process</th>
<th>True Positive (TP)</th>
<th>True Negative (TN)</th>
<th>False Positive (FP)</th>
<th>False Negative (FN)</th>
<th>Precision</th>
<th>Recall</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Manual</td>
<td>152</td>
<td>216</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2. DCG, with Further Cleaning</td>
<td>152</td>
<td>216</td>
<td>NA</td>
<td>NA</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3. DCG, with QCs and MDRs</td>
<td>152</td>
<td>216</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Experiment (B) results, regarding User Effort, for the Publications dataset.

<table>
<thead>
<tr>
<th>Data Cleaning Process</th>
<th># MDR Instances Created</th>
<th># MDR Instances Re-applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. DCG, with QCs and MDRs</td>
<td>608</td>
<td>608</td>
</tr>
<tr>
<td>4. DCG, with QCs, MDRs and MDR Conflicts</td>
<td>608</td>
<td>608</td>
</tr>
</tbody>
</table>

Table 5: Experiment (C) results, regarding User Effort, for the Publications dataset.

<table>
<thead>
<tr>
<th>Data Cleaning Process</th>
<th># MDR Instances Created</th>
<th># MDR Instances Re-applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. (a) With Work Loss</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4. (b) With Work Loss and Restoration</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

References


