Towards Event Log Querying for Data Quality
Let's Start with Detecting Log Imperfections

Robert Andrews(✉), Suriadi Suriadi, Chun Ouyang, and Erik Poppe

Queensland University of Technology, Brisbane, Australia
{r.andrews,s.suriadi,c.ouyang,e.poppe}@qut.edu.au

Abstract. Process mining is, by now, a well-established discipline focussing on process-oriented data analysis. As with other forms of data analysis, the quality and reliability of insights derived through analysis is directly related to the quality of the input (*garbage in - garbage out*). In the case of process mining, the input is an event log comprised of event data captured (in information systems) during the execution of the process. It is crucial then that the event log be treated as a first-class citizen. While data quality is an easily understood concept little effort has been directed towards systematically detecting data quality issues in event logs. Analysts still spend a large proportion of any project in 'data cleaning', often involving manual and *ad hoc* tasks, and requiring more than one tool. While there are existing tools and languages that query event logs, the problem of different approaches for different log imperfections remains. In this paper we take the first steps to developing QUELI (Querying Event Log for Imperfections) a log query language that provides direct support for detecting log imperfections. We develop an approach that identifies capabilities required of QUELI and illustrate the approach by applying it to 5 of the 11 event log imperfection patterns described in [29]. We view this as a first step towards operationalising systematic, automated support for log cleaning.

Keywords: Process mining · Event log query language
Data quality · Event log imperfection patterns

1 Introduction

Process mining is, by now, a well-established discipline focussing on process-oriented data analysis. As with other forms of data analysis, the quality and reliability of insights derived through analysis is directly related to the quality of the input (*garbage in - garbage out*). In the case of process mining, the input is an event log comprised of event data captured (in information systems) during the execution of the process. It is crucial then that the event log be treated as a first-class citizen.

H. Panetto et al. (Eds.): OTM 2018 Conferences, LNCS 11229, pp. 116–134, 2018.
https://doi.org/10.1007/978-3-030-02610-3_7

From it's inception to the present day, process mining has focused on three key areas; *discovery*, (taking an event log and generating a process model), *conformance* (comparing an existing process model with an event log drawn from the same process), and *enhancement* (extending an existing process model using information recorded in an event log) [3]. Process mining then is a model-based discipline with the event log being seen as the enabler for model development and subsequent process analysis. The Process Mining Manifesto [2] provides a star-rating for event logs in terms of their readiness for use in a process mining analysis. Existing data quality frameworks for event logs that have been proposed [9,21] are useful in classifying/labelling identified quality issues, but are not useful in detecting specific examples of quality issues.

Historically, analysts have devoted much time and effort to cleaning data (i.e. ensuring the data is 'fit for purpose') prior to analysis. A recent survey revealed data scientists spend more than half their time collecting, labeling, cleaning and organising data [11] and, while some cleaning tasks can be automated, "far too much handcrafted work...is still required"[18] with (compatibility) issues arising through the use of multiple tools [30] in collecting and preparing data.

Quality issues commonly found in event logs have been described [9,29] and, although solutions in the form of filtering, detection and repair algorithms [13, 15,19,31] have been proposed, the problem (different approaches to deal with different log imperfections), persists. A common feature of all these approaches however is an underlying capability to, in some way, query the log for the presence (or absence) of certain characteristics. Hence, our long-term goal is to develop an event log query language that can directly detect log imperfection issues. We call this Querying Event Log for Imperfections (QUELI). We take into consideration the view expressed by Behashti et al. [6] that "querying methods need to enable users to express their data analysis and querying needs using process-aware abstractions rather than other lower level abstractions". This simply means that more than being **somehow possible**, it should be **actually convenient** to encode event log and process constructs in a query language.

The key questions we consider are (i) what are the capabilities required to achieve QUELI?, and (ii) how do we identify these capabilities? In Sect. 3 we address these questions as a three stage approach involving (i) considering *how* log imperfections manifest in a log (i.e. what to look for in detecting log imperfections), (ii) stepwise refinement (narrative to rigorous algorithmic) of detection strategies, and (iii) consolidation of the algorithms to abstract 'building blocks'.

The major contributions of this paper are:

- definition of algorithms to detect 5 out of 11 log imperfection patterns described in [29];
- a preliminary (and by no means complete) consolidation of building blocks needed by QUELI; and
- a systematic approach to identifying remaining/further building blocks needed by QUELI.

The remainder of this paper is structured as follows: Sect. 2 describes the background to the project with a focus on (event log) data quality and discusses

related work in the areas of detecting log quality issues and event log querying. In Sects. 3 and 4 we introduce our approach to developing detection strategies and algorithms for the selected log imperfection patterns and in Sect. 5 we consolidate the pattern-at-a-time requirements into a set of QUELI constructs and briefly assess a range of existing log query languages against their ability to support the QUELI constructs. Section 6 concludes the paper with a brief discussion and some thoughts on future work.

2 Background and Related Work

Data quality is an easily understood concept, at least to the extent that "high" quality data is generally regarded as being desirable and "poor" quality data undesirable as input for any analysis. However, the numerous published attempts to objectively describe the characteristics of high/poor quality data (see [4,16,34] for various surveys of data quality research), testify to the fact that the 'devil is in the detail' as far as a universal understanding of data quality goes. High quality data has been defined as "data that is fit for use by data consumers" [28] with data quality considered as "the degree to which the characteristics of data satisfy stated and implied needs when used under specified conditions" [1]. Data quality is frequently described as being a multi-dimensional concept [33] with dimensions (i.e. *measurable* data quality properties) such as accuracy/correctness, completeness, unambiguity/understandability and timeliness/currentness [1,5,33] being frequently mentioned.

Table 1. Manifestation of quality issues in event log entities [9]

	Event log entities			
Quality issues	Missing data	Incorrect data	Imprecise data	Irrelevant data
Case	I1	I10		I26
Event	I2	I11		I27
Relationship	I3	I12	I19	
Case attrs.	I4	I13	I20	
Position	I5	I14	I21	
Activity name	I6	I15	I22	
Timestamp	I7	I16	I23	
Resource	I8	I17	I24	
Event attrs.	I9	I18	I25	

The issue of data quality for event logs was first considered in [2] with event log quality frameworks being proposed in [9,21,32]. Mans et al. [21] describes event log quality as a two-dimensional spectrum with the first dimension concerned about the *level of abstraction* of the events and the second one concerned with the *accuracy* of the timestamp (in terms of its (i) *granularity*, (ii) *directness of registration* (i.e. the currency of the timestamp recording) and (iii) *correctness*. Bose et al. [9] identify four broad categories of issues affecting process mining event log quality: Missing, Incorrect, Imprecise and Irrelevant data. The authors then show where each of these issues may manifest themselves in the various entities of an event log resulting in 27 separate data quality issues (see Table 1). Such frameworks are useful in **classifying** quality issues, however, they do not provide guidance as to how to **discover** quality issues in a log, nor do they provide a mechanism to quantify the extent to which a log is affected by any identified quality issue. For instance, how many cases are affected by missing events?

In Suriadi et al. [29] the authors describe a set of 11 *log imperfection patterns*[1] that capture some data quality issues commonly found in event logs (see Table 2 for relationship between patterns and quality issues in Table 1). In this paper we use 5 of the 11 log imperfection patterns described in [29] as the basis for understanding the requirements of an event log query language.

2.1 Related Work

Table 2. Relationship between individual patterns and quality framework.

Pattern	Quality issue/s
Form-based event capture	I16, I27
Inadvertent time travel	I16
Unanchored event	I23, I16
Scattered event	I2
Elusive case	I3
Scattered case	I12
Collateral events	I27
Polluted label	I15, I17
Distorted label	I15
Synonymous labels	I15
Homonymous label	I22

Suriadi et al. [29] adopt a patterns-based approach to describing event log quality issues and provide *indicative rules* for detecting the presence of each described log imperfection pattern. As the name suggests, an indicative rule describes conditions that make it likely that the related imperfection pattern is present in the log. The indicative rules are not however, at a low enough level to be immediately operationalised to provide direct support for pattern detection. Lu et al. [19] apply so-called *behaviour patterns* ("*partial orders* of activities where direct and indirect succession of activities are specified") to detect and visualise areas of complexity in an event log. While not specifically designed to detect log quality issues, the approach can be used to visualise event concurrency and hence provide an indication of the possibility of the existence of Form-based Event Capture and Collateral Events patterns [29]. Unsupervised event log pattern detection approaches such as [8,17] use statistical methods to detect frequently occurring behaviours in logs. These approaches suffer from "pattern explosion" (return many patterns) and require the user to sift through returned patterns to decide which are interesting. As these approaches are targeted at frequently occurring behaviours, infrequent behaviours (which may represent data quality issues) are harder to detect. Mannhardt et al. [20] by contrast, describe an approach involving manual specification of behavioural *activity patterns* which encode assumptions about how high-level activities manifest in lower-level events in a log. This approach is neither quality focused nor domain agnostic (requires domain knowledge), and relies on the expertise of the user in specifying patterns with the risk that the user may miss important patterns. Research in the area of record-linkage, data matching and ontology matching exists [10,27], but generally deals with less complex issues, e.g. only matching labels based on similarity measures.

Log repair, by definition, involves rectifying some identified quality issue. Log repair approaches necessarily require a filtering or querying step to identify log elements that are the subject of repair actions. As an example, in [13] the authors discuss three indicators of event order-related quality issues in event logs (mixed

[1] http://www.workflowpatterns.com/patterns/logimperfection/.

granularity timestamps, unusual or low-frequency directly followed relations or statistical anomalies in timestamp values) and describe techniques by which the log may be queried to detect each anomaly.

Unlike approaches that implement only one (or a limited range of) pattern types, event log query languages can potentially be used to encode multiple types of patterns (depending on the language constructs and the formulation of the event log). For instance, where the event log is represented as a table, SQL may be applied to log querying. In [12] the authors point out that formulating conceptually simple, but nevertheless fundamental, process related questions such as 'retrieve all directly follows relations between events' is inefficient and difficult to phrase in standard SQL (requiring joining the events table to itself and a NOT EXISTS nested sub-query) and has performance implications. Dijkman et al. [12] then propose (but do not implement) a relational algebaraic operator to extract the 'directly follows' relation from a log. In [25] the authors exploit *RelationalXES* [26] a relational database architecture for storing event log data and show how conventional SQL can be used to encode declarative constraints to extract process knowledge from event logs stored in relational tables. The approach, implemented as SQLMiner, has at least the following limitations: (i) data-perspective constraints have not yet been implemented, (ii) intermediate results are not available for follow-on queries, and (iii) native SQL is not process-aware, therefore the encoded declarative constraints are complex (and possibly beyond the ability of all but expert level SQL users).

FQSPARQL [7] is a process event query language and graph-based querying process engine derived from SPARQL [24]. The FPSPARQL approach models an event log as a graph of typed nodes and edges. In [22] de Murillas, Reijers and van der Aalst describe DAPOQ, the Data-Aware Process Oriented Query Language for querying event data. DAPOQ was purpose built for process querying and so has the advantages of improved query development time and readability of queries. The language supports the traditional process view (events, instances and processes), with the data perspective (data models, objects and object versions). Process Instance Query Language (PIQL) [23] describes a query language specifically designed to report on various Process Performance Indicators (PPIs). PPI queries are formulated in PIQL to return the number of process instances or tasks that are (i) (not) finalized, (ii) (not) cancelled, (iii) executed by {name}, (iv) start before {date}, etc. The fact that the language is specifically designed to report on a PPIs limits its generality. XSLT (Extensible Stylesheet Language Transformations) is a declarative data transformation language developed by the W3C and used for transforming XML documents. If the event log is encoded as a tree structure (a log contains several cases and each case contains one or more events) in XES/XML, XSLT can be used to filter, transform and query the event log. Durand et al. [14] leverage this prevalence of XML-based standards for encoding event logs to build an XML vocabulary and execution language on top of XSLT, that can be used to query and analyse event sequences.

3 Approach

Our approach for identifying log query constructs, which are key building blocks for a query language that is suitable for detecting imperfections in event logs, is illustrated in Fig. 1. By 'constructs', we mean a collection of functionality that is required by a language (e.g. a log query language) to serve its purpose (e.g. to detect log imperfections).

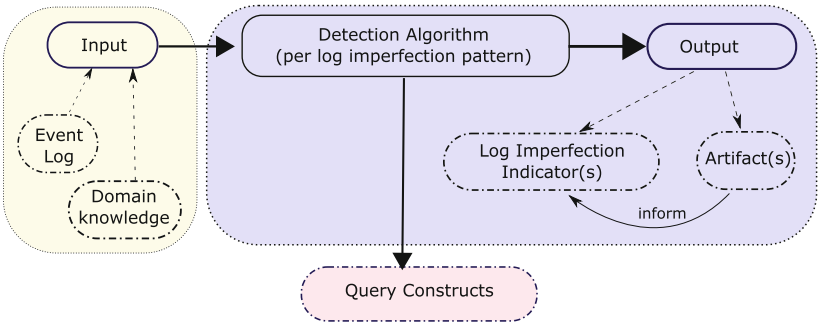

Fig. 1. Approach for identifying query constructs for log imperfection detection

Inputs. The inputs to our approach are an event log and, if available, domain knowledge (e.g. the valid ranges of values for the timestamps of certain activities). An event log is a collection of events. An event can be defined as a tuple consisting of various attributes. An attribute may be mandatory or optional. For example, there are three mandatory event attributes, known as *case identifier*, *activity label*, and *timestamp*, required for process mining.

Definition 1 (Event). Let $Case$ be the set of case identifiers, Act the set of activity labels, and $Time$ the set of timestamps. $\mathscr{E} \subseteq Case \times Act \times Time$ is the set of events. For any event $e \in \mathscr{E}$, $case(e) \in Case$, $act(e) \in Act$, and $time(e) \in Time$, represent the case identifier, activity label, and timestamp of event e. □

Definition 2 (Event Log). An event log $\mathcal{L} \subseteq \mathscr{E}$ is a set of events. □

Detection Algorithms. In order to identify the constructs needed for querying an event log for the existence of various imperfections, we start by developing detection strategies to address certain log imperfections. These detection strategies should be *generic*, i.e. independent of any specific language and system/tool, and *rigorously* defined in the form of an algorithm (for each strategy per log imperfection pattern) so that they can be implemented in a precise and unambiguous way. In the next section of this paper, we elaborate on the detection strategies for five event log imperfection patterns among those defined in [29].

Outputs. Each algorithm will produce some artifacts (such as statistical summary of certain characteristics in an event log, or the derivation of sub-logs that meet certain criteria) that will then be used to reason about the existence, or lack thereof, of certain log imperfections.

Query Constructs. Having defined the algorithms for detecting various log imperfections, we consolidate them to identify which part(s) of the algorithms can potentially be used as *query constructs*. These constructs provide direct support to querying capabilities and thus they are key building blocks for the intended query language. As an indication of the generality and reusability of such a construct, it should be applicable to several log imperfection detection algorithms. In Sect. 5, we explain how we consolidate the algorithms to identify potential query constructs. It is worth noting that these constructs are still at their early stage, and further analysis of other log imperfections is needed in order for one to be reasonably assured that a comprehensive collection of constructs has been identified.

4 Detection Strategies for Log Imperfection Patterns

In this section we propose the design of strategies for detecting the presence of log imperfection patterns. For each pattern, we start with describing how the pattern manifests in an event log, what could be the main reason that has led to the existence of the pattern in the log, and how the pattern exemplifies a data quality issue that is defined at a more abstract level as discussed in Sect. 2. From there, we continue to present our detection strategy for the pattern which consists of an outline of the underlying detection mechanism in natural language and then a conceptual design of the strategy specified in the form of an algorithm.

4.1 Form-Based Event Capture

The pattern manifests in an event log as multiple occurrences of groups of events characterised by a common set of activity labels with each event in the group having the same or very nearly the same timestamp (allowing for physical logging by the system) in the same case. An example of the pattern is shown in Fig. 2. A main reason that may cause this pattern to arise is that when process-related data is entered into fields on a computerised form, updates to form field values are logged as *separate* events (e.g. one event per field) in response to a user action (e.g. clicking 'Save' on the form). In this case, the activity labels are usually informed directly by the corresponding form field names. It can be observed, taking the example in Fig. 2, that the presence of

Episode ID	Activity	Timestamp	Description	...
ID1	Primary Survey	2012-11-23 15:42:38	.....	....
ID1	Airway Clear	2012-11-23 15:42:38	.....	....
ID1	...	2012-11-23 15:42:38	.....	....
	Primary Survey	2012-11-24 09:58:33	.....	....
	Pupils Responsive	2012-11-24 09:58:33	.....	....
	...	2012-11-24 09:58:33	.....	....
	Procedure 1	2012-11-24 09:58:33	Completed on 2012-11-24 06:58:34	....

These events are recorded on a form ...

... and all have the same timestamp.

Fig. 2. Example of *form-based event capture*

Algorithm 1. DetectForm-basedEventCapture

```
input : event log ℒ, timestamp difference Δt
output: ℱ_𝒢
begin
    𝒢 ← findSimultaneousEvents(ℒ, Δt) /* Step-1: 𝒢 is a set of event groups */
    𝒜 ← ∅ /* Step-2.a: To compute a set of groups of activity labels 𝒜 */
    foreach G_i ∈ 𝒢 do
        𝒜 ← 𝒜 ∪ {getActLabels(G_i)}
    𝒟 ← 𝒜 /* Step-2.b: To compute a set of distinct groups of activity labels 𝒟 */
    do
        𝒟_t ← 𝒟
        D_0 ← getOneSetElement(𝒟)
        𝒟 ← 𝒟 \ {D_0}
        foreach D_i ∈ 𝒟 do
            if D_0 ∩ D_i ≠ ∅ then
                D_0 ← D_i ∪ D_0
                𝒟 ← 𝒟 \ {D_i}
        𝒟 ← 𝒟 ∪ {D_0}
    while 𝒟 ≠ 𝒟_t;
    if |𝒟| = 1 then
        ℱ_𝒢 ← {𝒟, |ℒ|} /* Detection ends if only one distinct group of activity labels */
    else
        ℱ_𝒢 ← ∅ /* Step-3: To compute the likelihood of form existence ℱ_𝒢 */
        foreach D ∈ 𝒟 do
            count_D ← 0
            foreach G ∈ 𝒢 do
                if getActLabels(G) ⊆ D then
                    count_D++
            ℱ_𝒢 ← ℱ_𝒢 ∪ (D, count_D)
```

this pattern in an event log leads to the recording of incorrect timestamps of affected events representing the time the form was saved, not the time each of affected events actually occurred. As such, this pattern presents a more concrete example of data quality issue *I16 - Incorrect data: timestamp*.

Detection Strategy: The main idea is to discover the existence of a form by identifying the group of activity labels that are likely informed by the corresponding fields on that form. An outline of our detection strategy follows.

Step-1. Due to the fact that these activities are logged as events that have the same or very nearly the same timestamps in the same case, the first step of detection is to find groups of such *simultaneous events*.
Step-2. It is important to realise that updates may be applied to different fields on a form in different cases. For example, as shown in Fig. 2, in one case (Episode ID1) 'Primary Survey' and 'Airway Clear' are among those updated, and in another case (Episode ID2) update occurred to 'Primary Survey' and 'Pupils

Responsive' on the form. Hence, it is necessary to find the group of all possible fields on each form (given the scope of the data available in a log).

This can be achieved by (a) extracting the groups of activity labels from the groups of simultaneous events and (b) traversing the activity labels group by group and merging the groups that have at least one overlapping label between them. As such, the second step of detection will yield *distinct* groups of activity labels meaning that each activity label belongs to only one of the groups.

However, it is possible that only *one* distinct group of activity labels is identified and the detection process will then end with an output of mainly this group of activity labels. In this case, a conclusion likely to be drawn by end users is that either all the events in the log are recorded from a single form or no form exists.

Step-3. This is to help understand how likely each group of activity labels may inform the existence of a form. Certain quantitative measures can be computed to provide reasonable indication for the likelihood of form existence. An example of such a measure is to count how often each group of activity labels, including its sub-groups, have appeared in the event log. This can help end users to make their decision given necessary domain knowledge. E.g., if the above count of a specific group of activity labels is larger than a certain threshold value, a conclusion can be drawn that there exists a form that contains the corresponding field names.

4.2 Collateral Events

This pattern manifests as an event log containing groups of activities with timestamps that are very close to each other (e.g. within seconds). The problem may be introduced into the log through incorrect or too fine grained logging of event data. As illustrated by the example in Fig. 3, the snippet of an event log contains a list of micro-steps of a process activity, whereas it is the activity, but not its micro-steps, that is of interest to process analysis. As such, this pattern presents a more concrete example of data quality issue *I27 - Irrelevant data: event.*

caseID	Activity	Timestamp
1234567	Adjust recovery cost	19/06/2014 12:15:18
1234567	Adjust recovery cost	19/06/2014 12:16:53
1234567	Email	19/06/2014 12:19:25
....	....	.....
1234567	Pay assessor fee	19/06/2
1234567	Adjust admin cost	19/06/2014 12:22:48

All events refer to single process step 'Pay Insurance Claim Assessor'.

Fig. 3. Example of *collateral events*

Detection Strategy: The key objective is to discover a high-level (parent) activity by identifying the list of micro-steps (instead of the activity) that are possibly recorded in the event log. Hence, the detection strategy is similar to that of form-based event capture. The differences are: (1) the central objects for detection of collateral events are parent process activities and their micro-steps (instead of forms and their fields in the case of form-based event capture); and (2) the input time difference Δt for detection of collateral events has a greater value than that of form-based event capture. Algorithm 1 can be re-used for detection of collateral events, because it is generic (e.g. being independent of specific objects to identify, such as a form and its fields vs. a process activity and its micro-steps) and configurable (e.g. Δt being a user input variable).

4.3 Inadvertent Time Travel

This pattern manifests as a number of cases in the log where the temporal ordering of events deviates significantly from the majority of cases in the log or from a mandatory temporal ordering property. For example, Fig. 4 shows a snippet of a log with this imperfection pattern: the activity Arrival first hospital (henceforth referred to as activity A) was recorded to take place on 2011-09-08 00:30:00; however, the time of the Injury activity (henceforth referred to as activity B), which triggered the patient being sent to the hospital, was recorded to take place more than 23 h *later* (at 2011-09-08 23:47:01). The cause of this problem is the 'midnight problem' whereby a hospital staff recorded the correct 'time' of patient arrival but failed to change the 'date' portion of the timestamp (it should have been 2011-09-09). The occurrence of this pattern is often associated with manual recording of timestamp data and results from the 'proximity' between correct value and the recorded, incorrect value. Proximity errors occur through a user pressing an incorrect key on a keyboard, or as in our example above, a user failing to recognise the recently-crossed date/time boundary (such as the mentioned midnight problem, or a new year). This pattern negatively impacts the attribute accuracy of the log in that the temporal ordering of the events no longer reflects the actual ordering of events. Therefore, this pattern is a manifestation of the data quality issue *I16 - Incorrect data: timestamp*.

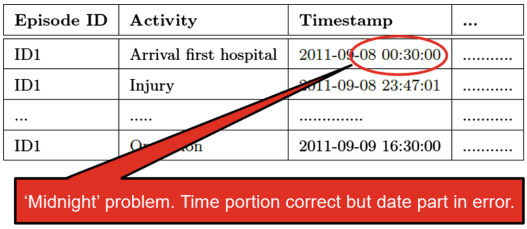

Fig. 4. Example of *inadvertent time travel*

Detection Strategy: The main idea is to discover the existence of pairs of activities, within the same case, with 'unusual' temporal ordering, i.e., it either deviates from the majority of the cases or violates some mandatory ordering. Once such pairs of activities are discovered, we then extract statistical summary information (such as the proportion of cases with the deviant temporal ordering) to be presented to users to determine if the unusual temporal ordering is indeed a data quality issue. An outline of our detection strategy is as follows.

Step-1. Using the example from Fig. 4, an unusual temporal ordering of two events is seen when in one or more cases, the activity B *succeeds* A, while in the majority of cases B (the injury event) *precedes* A. In other words, we say A and B occurred in any order (in parallel). The first detection step is thus to identify *all pairs of activity names* that can occur in any order.

Step-2. Next, for each pair of activity names that can occur in any order, we extract the corresponding *pairs of events*. Using our example above, the idea here is to obtain two sets of pairs of events: the first set consists of all pairs of events where A was followed by B, and the second group consists of all event pairs where B was followed by A.

Step-3. Finally, we obtain some statistical summary of those two groups. The intended statistical summary includes information such as the proportion of cases of usual vs. unusual temporal ordering and the frequency of each pair of events.

This statistical summary information is then presented to the user to determine if it is an acceptable deviance or if it is an event log quality issue.

Algorithm 2. DetectInadvertentTimeTravel

input : event log $\mathcal{L}$
output: $\mathbb{S}_{(\mathsf{A}||\mathsf{B})}$
begin
  /* *Step-1*: $\mathcal{A}_{||}$ is a set of activity names that can occur in any order */
  $\mathcal{A}_{||} \leftarrow$ `findParallelEventPairs`$(\mathcal{L})$.
  $\mathbb{S}_{(\mathsf{A}||\mathsf{B})} \leftarrow \emptyset$ /* Initialise the return value */
  foreach $(a, b) \in \mathcal{A}_{||}$ **do**
    /* *Step-2*: $\mathcal{L}_{(\mathsf{a}||\mathsf{b})}$ and $\mathcal{L}_{(\mathsf{b}||\mathsf{a})}$ are the corresponding sets of pairs of events with activities that can occur in any order */
    Let $\mathcal{L}_{(\mathsf{a}||\mathsf{b})} = \{(e, e') \in \mathcal{L} \times \mathcal{L} \mid \exists_{(\mathsf{a},\mathsf{b})\in\mathcal{A}_{||}} : act(e) = a \wedge act(e') = b\}$
    Let $\mathcal{L}_{(\mathsf{b}||\mathsf{a})} = \{(e, e') \in \mathcal{L} \times \mathcal{L} \mid \exists_{(\mathsf{a},\mathsf{b})\in\mathcal{A}_{||}} : act(e) = b \wedge act(e') = a\}$
    /* *Step-3*: Calculate the statistical summary */
    $\mathtt{StatSumm}_{(\mathsf{a}||\mathsf{b})} \leftarrow$ `getStatSummary`$(\mathcal{L}_{(\mathsf{a}||\mathsf{b})})$
    $\mathtt{StatSumm}_{(\mathsf{b}||\mathsf{a})} \leftarrow$ `getStatSummary`$(\mathcal{L}_{(\mathsf{b}||\mathsf{a})})$
    $\mathbb{S}_{(\mathsf{A}||\mathsf{B})} \leftarrow \mathbb{S}_{(\mathsf{A}||\mathsf{B})} \cup \{\mathtt{StatSumm}_{(\mathsf{a}||\mathsf{b})}\} \cup \{\mathtt{StatSumm}_{(\mathsf{b}||\mathsf{a})}\}$

4.4 Synonymous Labels

This pattern manifests as the existence of multiple values of a particular attribute that seem to share a similar meaning but are nevertheless, distinct. For example, Fig. 5 shows a snippet of a log with this imperfection pattern: the activities Medical Assign and DrSeen refer to the activity of consulting a medical doctor. However, the labels (or names) given to the activity are different.

This log imperfection pattern may arise when an event log is constructed from multiple source logs, each of which represents the same process, but uses a different label to represent essentially the same process step. The existence of multiple names for the same attribute creates ambiguity in an event log. As such, this imperfection pattern is a manifestation of the *I22 - Imprecise data: event attributes* quality issue.

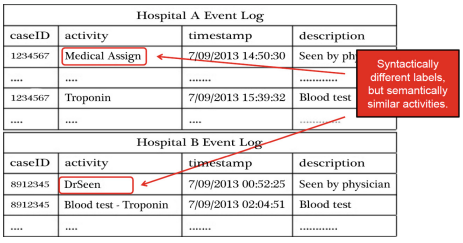

Fig. 5. Example of *synonymous labels*

Detection Strategy: The main idea is to discover the existence of pairs of activities that never occur together within the same case. Using the example above, the underlying assumption is that some cases were recorded in one particular system using the activity name Medical Assign while other cases were recorded in another system using the activity name DrSeen. Then, we examine

Algorithm 3. DetectSynonymousLabels

Input : event log $\mathcal{L}$
Output: $\{\mathcal{A}_{\texttt{synonymous}}\}$
begin
  /* *Step-1*: $\mathcal{A}_{\#}$ is a set of activity names that are in conflict */
  $\mathcal{A}_{\#} \leftarrow$ `findConflictPairs`($\mathcal{L}$) is a set consisting of pairs of events with conflict relation.
  $\mathcal{A}_{\texttt{synonymous}} \leftarrow$ `NULL` /* Initialise the return value */
  foreach $(a, b) \in \mathcal{A}_{\#}$ **do**
    /* *Step-2*: Obtain the corresponding events for each pair of activity names that are in conflict */
    Let $\mathcal{L}_{\#\texttt{a}} = \{e \in \mathcal{L} \mid \exists_{(\texttt{a},\texttt{b}) \in \mathcal{A}_{\#}} : act(e) = a\ \}$
    Let $\mathcal{L}_{\#\texttt{b}} = \{e \in \mathcal{L} \mid \exists_{(\texttt{a},\texttt{b}) \in \mathcal{A}_{\#}} : act(e) = b\ \}$
    /* *Step-3*: Obtain the context variable and check for similarity of the context variables */
    $\mathcal{C}_{\texttt{context(a)}} =$ `getContextVariables`($\mathcal{L}_{\#\texttt{a}}, \mathcal{L}$)
    $\mathcal{C}_{\texttt{context(b)}} =$ `getContextVariables`($\mathcal{L}_{\#\texttt{b}}, \mathcal{L}$)
    if $\mathcal{C}_{context(a)} \approx \mathcal{C}_{context(b)}$ **then**
      $\mathcal{A}_{\texttt{synonymous}} \leftarrow \mathcal{A}_{\texttt{synonymous}} \cup (\texttt{a}, \texttt{b})$

the contextual variables surrounding this pair of activity names. Contextual variables include the surrounding activity names preceding, succeeding, or running in parallel with the activity name being examined. If the contextual variables are similar, then this pair of activity names may be candidate for synonymous label. An outline of our detection strategy is as follows.

Step-1. The first step in our detection strategy is to identify those activity labels that never occur together within the same case, across all cases seen in the event log. When two activity labels never occur together within the same case, we call these activity labels to be *in conflict*. By examining the whole event log, we will get a list of pairs of activity names that are in conflict.

Step-2. Next, for each pair of activity names that are in conflict, we extract the corresponding contextual variables as explained above. To do so, we need to extract two groups of events: each group consists of all events whose activity names is the same as one of those activity names in conflict. Using our example above, the first group of events will be those events whose activity names are equal to Medical Assign, while the other group consists of all events with activity name DrSeen.

Step-3. For each group of events extracted in the previous step, we obtain the contextual variables and then compare them to see if they are similar. Using the example above, if the contextual variables between the activity names DrSeen and Medical Assign are similar enough, we store this pair of activity names into a list to be presented to users for determination.

4.5 Homonymous Labels

This pattern manifests as the existence of an activity name being repeated multiple times within a case (i.e. the same activity name applied to each occurrence of the activity), but the interpretation of the activity, from a process perspective, differs across the various occurrences. For example, in Fig. 6, the activity name Triage Assessment occurred multiple times within the same case. The second and third occurrences happened roughly 7 days after the first. From a process perspective, the first occurrence referred to an actual triage activity of a patient, while the second and third occurrences referred to a doctor reviewing the triaging decision made earlier (it is impossible to be triaged again after a patient has been discharged from the hospital). This log imperfection pattern may arise when the original logging or the subsequent event log extraction does not record the context information necessary to distinguish between the different occurrences of the activity. For instance, in our example, the first triage activity activity should have been further qualified by using the name Triage - Initial, while the second and third should be further qualified by using the name Triage - Review. The occurrence of this log imperfection pattern makes certain activity names too coarse to reflect the different connotations associated with the recorded events. As such, this is a manifestation of the *I12 - Imprecise data: activity name* data quality issue.

caseID	activity	timestamp	Description
1234567	Triage Assessment	06/09/2013 12:33:17	.....
1234567	Progress Note	06/09/2013 13:10:23	.....
1234567	Discharged	06/09/2013 13:15:00	.....
1234567	Triage Assessment	13/09/2013 07:24:36	.....
1234567	Triage Assessment	13/09/2013 07:28:51	.....

Fig. 6. Example of *homonymous labels*

Detection Strategy: A homonymous label pattern manifests itself as the occurrence of an activity name at rather 'odd' places within a case, e.g. the occurrence of a triage activity after the patient was discharged (in the example above). In other words, similar to the *Inadvertent Time Travel* pattern, the first detection step is thus to discover the existence of pairs of activities, within the same case, with 'unusual' temporal ordering. Next, from those pairs of activities with unusual temporal ordering, we identify those activities that are repeated at least twice within the same case (a homonymous label pattern will only be seen if the activity is repeated). Finally, for each activity that is repeated, we obtain the contextual variable to identify if we see one or more distinct contextual variables. We then present the contextual variables along with the activity names to users to determine if there is a homonymous label pattern in the log. An outline of our detection strategy is as follows.

Step-1. As explained above, the first step in our detection strategy is to identify those pairs of activity names with unusual temporal ordering. As explained in the detection of *Inadvertent Time Travel* pattern, we can identify such pairs of activity names by extracting those pairs of activities that happened in any order.

Step-2. The logic behind this step is similar to *Step-2* of the *Inadvertent Time Travel* pattern: for each pair of activity names that can occur in any order, we extract the corresponding *pairs of events*.

Step-3. Next, for each pair of activities that can occur in any order, we extract those duplicated activity names within a case. This step is needed to narrow down the list of potential homonymous activity labels for further examination. As explained above, homonymous label pattern exists for an activity name that occurs at least twice within a case.
Step-4. Finally, for the set of duplicated activity names extracted from *Step-3*, we obtain the contextual variables for each occurrence of this activity name. We then return the set of contextual variables for each duplicated activity name from *Step-3* to users to determine the existence of the homonymous label pattern.

5 Towards Required Capabilities for QUELI

By presenting detection strategies for a number of event log imperfection patterns we have shown that interacting with event log data in a way that enables the detection of data quality issues is often non-trivial. Rather than expecting users to manually apply these strategies to their data sets it would be useful to provide them with "building blocks" that they can use to apply these strategies

Algorithm 4. DetectHomonymousLabels

Input : event log $\mathcal{L}$
Output: $\mathbb{N}_{\texttt{context}}$
begin
  /* *Step-1*: $\mathcal{A}_{||}$ is a set of activity names that can occur in any order */
  $\mathcal{A}_{||} \leftarrow \texttt{findParallelEventPairs}(\mathcal{L})$.
  /* *Step-2*: $\mathcal{L}_{(\texttt{a}||\texttt{b})}$ and $\mathcal{L}_{(\texttt{b}||\texttt{a})}$ are the corresponding sets of pairs of events for all activities that can occur in any order */
  $\mathcal{L}_{(\texttt{a}||\texttt{b})} \leftarrow \emptyset$
  $\mathcal{L}_{(\texttt{b}||\texttt{a})} \leftarrow \emptyset$
  foreach $(a, b) \in \mathcal{A}_{||}$ **do**
    Let $L_{(\texttt{a}||\texttt{b})} = \{(e, e') \in \mathcal{L} \times \mathcal{L} \mid \exists_{(\texttt{a},\texttt{b})\in\mathcal{A}_{||}}\colon act(e) = a \wedge act(e') = b\}$
    Let $L_{(\texttt{b}||\texttt{a})} = \{(e, e') \in \mathcal{L} \times \mathcal{L} \mid \exists_{(\texttt{a},\texttt{b})\in\mathcal{A}_{||}}\colon act(e) = b \wedge act(e') = a\}$
    $\mathcal{L}_{(\texttt{a}||\texttt{b})} \leftarrow \mathcal{L}_{(\texttt{a}||\texttt{b})} \cup L_{(\texttt{a}||\texttt{b})}$
    $\mathcal{L}_{(\texttt{a}||\texttt{b})} \leftarrow \mathcal{L}_{(\texttt{a}||\texttt{b})} \cup L_{(\texttt{b}||\texttt{a})}$
  /* *Step-3*: Identify duplicated activity names in a case. The set $\mathcal{A}_{\texttt{candidates}}$ consists of all activity names that can be duplicated within a case. */
  $\mathcal{A}_{\texttt{candidates}} = \texttt{getDuplicateNames}(\mathcal{L}_{(\texttt{a}||\texttt{b})}, \mathcal{L}_{(\texttt{b}||\texttt{a})})$
  /* Initialise the return value */
  $\mathbb{N}_{(\texttt{context})} \leftarrow \emptyset$
  foreach $a \in \mathcal{A}_{candidates}$ **do**
    /* *Step-4*: Extract the contextual variables. */
    $\mathcal{L}_{\texttt{candidates}} = \{\ e \in \mathcal{L} \mid e.act = a\ \}$
    $\mathcal{C}_{context(a)} \leftarrow \texttt{getContextVariables}(\mathcal{L}_{\texttt{candidates}}, \mathcal{L})$
    $\mathbb{N}_{\texttt{context}} \leftarrow \mathbb{N}_{\texttt{context}} \cup (a, \mathcal{N}_{context(a)})$

through querying. Our detection strategies already show reoccurring operations and data structures used in the detection. In Table 3 we aggregate the primitives used in the presented detection strategies, generalise them, and show which primitives are relevant to detecting more than one pattern in order to identify some potential "building blocks".

Table 3. Aggregated primitives to detect log imperfection patterns [29]

Primitive: `findSimultaneousEvents`($\mathcal{L}$, Δt)
Relates to: Form-based Event Capture, Collateral Events
Primitive: `getOneSetElement`($\mathcal{D}$)
Relates to: Form-based Event Capture, Collateral Events
Primitive: `findRelationshipPairs`($\mathcal{L}$,[$\|\|, \#, <, >$])
Relates to: Inadvertent Time Travel, Synonymous Label, Homonymous Label
Generalisation: We combine `findParallelEventPairs`($\mathcal{L}$) and `findConflictPairs`($\mathcal{L}$) and anticipate the need to extract direct-follow and direct-precede relations
Primitive: `getActLabels`($\mathcal{L}$,$A \subseteq \mathcal{A}$, $\delta(\mathcal{L})$)
Relates to: Form-based Event Capture, Collateral Events
Primitive: `getStatSummary`($\mathcal{L}$, $\mathcal{L}$)
Relates to: Inadvertent Time Travel, Homonymous Label
Primitive: `getContextVariables`($a \in \mathcal{A}$,$\mathcal{L}$)
Relates to: Synonomous Label, Homonymous Label
Primitive: `getDuplicateNames`($\mathcal{L}$, $\mathcal{L}$)
Relates to: Homonymous Label

On a higher level, we can summarise that, to apply the presented detection strategies, a mixture of high-level language features are required. These are (i) support for selection/projection of data, (ii) support for aggregation of results, (iii) support for set-operations, (iv) support for loops, and (v) support for event-relations (e.g. parallel, conflict) (see Table 4). In the following we show that the required high-level language features are not supported by any single query language. As at least one of the referenced languages is Turing complete and can therefore theoretically perform any operation that can be defined as an algorithm, we more specifically check for direct support. We define *direct support* for a primitive as the availability of parameterised function calls, so that the query can be performed without writing procedural code. An example of such a parameterised operator for filtering events in a log is the WHERE clause in an SQL statement.

The getActLabel() primitive returns activity labels associated with events. Hence, this primitive requires selection (to identify events) and projection (to return the activity label attribute values) functionality. All query languages support this functionality. All our proposed detection algorithms make use of set

operations. As a set-based language, SQL provides support for set operations. PIQL supports querying the numbers of process or task instances and is therefore not able to provide the actual sets of event (tasks) as required for our primitives. The getStatSummary() primitive returns aggregates of low-level data. SQL provides support for aggregation through the GROUP BY clause and built-in functions such as MIN(), MAX(), AVG(). XSLT is Turing complete and should therefore in theory be able to provide support for all our primitives, however, aggregations are, in fact, not well supported and are practically infeasible. All our detection algorithms involve some form of repetition, generally count-controlled, with two algorithms (Form-based Event Capture and Collateral Events) requiring condition-controlled iteration. Only XSLT provides direct support for both forms of repetition, DAPOQ supports only count-controlled repetition, while the other languages do not support either form of repetition. The findRelationshipPairs() primitive requires identifying relationships between pairs of events. While this is possible in a language such as SQL, the query is complex and, using only standard SQL features, requires manual specification of pairs of events and relationship type. Hence, it is not reasonable to conclude that SQL provides direct support for this primitive. Only FPSPARQL, through its notion of paths, provides direct support for this primitive.

Table 4. High-level features required for application of detection strategies

Features	SQL	FPSPARQL	DAPOQ	PIQL	XSLT
Support for selection/projection of data *Relates to*: getActLabels()	Y	Y	Y	Y	Y
Support for set-operations *Relates to*: All strategies	Y	Y	Y	N	N
Support for aggregation of results *Relates to*: getStatSummary()	Y	Y	N	Y	N
Support for repetition *Relates to*: Form-based Event Capture, Collateral Events	N	N	N	N	Y
Support for event-relations *Relates to*: findRelationshipPairs()	N	Y	N	N	N

6 Conclusion

This work was motivated by our experiences in data preparation for multiple process mining case studies. For each study, the objective was to construct event logs with the highest possible data quality. Often, the starting point was source log(s) (i) drawn from non-process aware information systems, (ii) all of which exhibited a mixture of the issues described in [9,21,29] and (iii) required the

use of multiple off-the-shelf tools and, sometimes, custom-developed software to identify and rectify quality issues (with the associated save, open, save-as different format, import operations to move from one environment to the next). In this paper we have outlined an approach that identified a small set of function primitives for detecting a range of data quality issues commonly found in event logs, *viz.* log imperfection patterns. We note that, unsurprisingly, none of the tools or query languages we considered provide **direct** support for **all** of the functional requirements we derived. The detection strategies and algorithms provided in this paper meet our aim of providing guidance to process analysts in detecting the log imperfection patterns and form the basis of future work in implementing the primitives in QUELI.

Acknowledgement. The contributions to this paper of Robert Andrews and Chun Ouyang were supported through ARC Discovery Grant DP150103356.

References

1. ISO/IEC 25010:2011: Systems and software engineering - Systems and software product Quality Requirements and Evaluation (SQuaRE) - System and software quality models (2011)
2. van der Aalst, W., et al.: Process mining manifesto. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM 2011. LNBIP, vol. 99, pp. 169–194. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28108-2_19
3. van der Aalst, W.: Process Mining: Discovery Conformance and Enhancement of Business Processes. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19345-3
4. Batini, C., Palmonari, M., Viscusi, G.: Opening the closed world: a survey of information quality research in the wild. In: Floridi, L., Illari, P. (eds.) The Philosophy of Information Quality. SL, vol. 358, pp. 43–73. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07121-3_4
5. Batini, C., Scannapieco, M.: Data Quality: Concepts, Methodologies and Techniques. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-33173-5
6. Beheshti, S.-M.-R., Benatallah, B., Motahari-Nezhad, H.R.: Scalable graph-based OLAP analytics over process execution data. Distrib. Parallel Datab. **34**(3), 379–423 (2016)
7. Beheshti, S.-M.-R., Benatallah, B., Motahari-Nezhad, H.R., Sakr, S.: A query language for analyzing business processes execution. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 281–297. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23059-2_22
8. Jagadeesh Chandra Bose, R.P., van der Aalst, W.M.P.: Abstractions in process mining: a taxonomy of patterns. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 159–175. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03848-8_12
9. Jagadeesh Chandra Bose, R.P., Mans, R.S., van der Aalst, W.M.: Wanna improve process mining results? CIDM **2013**, 127–134 (2013)
10. Christen, P.: Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31164-2

11. CrowdFlower: 2017 Data Scientist Report (2017). https://visit.crowdflower.com. Accessed 25 July 2018
12. Dijkman, R., Gao, J., Grefen, P., ter Hofstede, A.: Relational algebra for in-database process mining. arXiv preprint arXiv:1706.08259 (2017)
13. Dixit, P.M., et al.: Detection and interactive repair of event ordering imperfection in process logs. In: Krogstie, J., Reijers, H.A. (eds.) CAiSE 2018. LNCS, vol. 10816, pp. 274–290. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91563-0_17
14. Durand, J., Cho, H., Moberg, D., Woo, J.: XTemp: event-driven testing and monitoring of business processes. In: Proceedings of Balisage, The Markup Conference 2011, vol. 7. Balisage Series on Markup Technologies (2011)
15. Günther, C.W., Rozinat, A.: Disco: discover your processes. BPM (Demos) **940**, 40–44 (2012)
16. Laranjeiro, N., Soydemir, S.N., Bernardino, J.: A survey on data quality: classifying poor data. In: PRDC 2015, pp. 179–188. IEEE (2015)
17. Leemans, M., van der Aalst, W.M.P.: Discovery of frequent episodes in event logs. In: Ceravolo, P., Russo, B., Accorsi, R. (eds.) SIMPDA 2014. LNBIP, vol. 237, pp. 1–31. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27243-6_1
18. Lohr, S.: For big-data scientists, 'janitor work' is key hurdle to insights. New York Times, 17 August 2014
19. Lu, X., et al.: Semi-supervised log pattern detection and exploration using event concurrence and contextual information. In: Panetto, H., et al. (eds.) OTM On the Move to Meaningful Internet Systems, pp. 154–174. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69462-7_11
20. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P., Toussaint, P.J.: From low-level events to activities - a pattern-based approach. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNCS, vol. 9850, pp. 125–141. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_8
21. Mans, R.S., van der Aalst, W.M., Vanwersch, R., Moleman, A.: Process Support and Knowledge Representation in Health Care. LNCS, vol. 7738, pp. 140–153. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36438-9
22. González López de Murillas, E., Reijers, H.A., van der Aalst, W.M.P.: Everything you always wanted to know about your process, but did not know how to ask. In: Dumas, M., Fantinato, M. (eds.) BPM 2016. LNBIP, vol. 281, pp. 296–309. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58457-7_22
23. Perez-Alvarez, J.M., Gomez-Lopez, M.T., Parody, L., Gasca, R.M.: Process instance query language to include process performance indicators in DMN. In: EDOCW 2016, pp. 1–8. IEEE (2016)
24. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C recommendation, January 2008 (2008)
25. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and customisable declarative process mining with SQL. In: Nurcan, S., Soffer, P., Bajec, M., Eder, J. (eds.) CAiSE 2016. LNCS, vol. 9694, pp. 290–305. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39696-5_18
26. Shabani, S., et al.: Relational XES: data management for process mining. In: CAiSE 2015. CEUR-WS. org (2015)
27. Shvaiko, P., Euzenat, J.: Ontology matching: state of the art and future challenges. IEEE Trans. Knowl. Data Eng. **25**(1), 158–176 (2013)
28. Strong, D.M., Lee, Y.W., Wang, R.Y.: Data quality in context. Commun. ACM **40**(5), 103–110 (1997)

29. Suriadi, S., Andrews, R., ter Hofstede, A., Wynn, M.: Event log imperfection patterns for process mining: towards a systematic approach to cleaning event logs. Inf. Syst. **64**, 132–150 (2017)
30. Suriadi, S., Wynn, M.T., Ouyang, C., ter Hofstede, A.H.M., van Dijk, N.J.: Understanding process behaviours in a large insurance company in australia: a case study. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 449–464. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38709-8_29
31. Vázquez-Barreiros, B., Mucientes, M., Lama, M.: Mining duplicate tasks from discovered processes. In: ATAED@ Petri Nets/ACSD, pp. 78–82 (2015)
32. Verhulst, R.: Evaluating quality of event data within event logs: an extensible framework. Ph.D. thesis, Technische Universiteit Eindhoven (2016)
33. Wand, Y., Wang, R.Y.: Anchoring data quality dimensions in ontological foundations. Commun. ACM **39**(11), 86–95 (1996)
34. Wang, R.Y., Storey, V., Firth, C.: A framework for analysis of data quality research. IEEE Trans. Knowl. Data Eng. **7**(4), 623–640 (1995)