Process Conformance Checking by Relaxing Data Dependencies

Montserrat Estañol[1,2], Mirjana Mazuran[3] Xavier Oriol[1], Letizia Tanca[3], and Ernest Teniente[1] *

[1] Universitat Politècnica de Catalunya, Barcelona, Spain
{estanyol|xoriol|teniente}@essi.upc.edu
[2] SIRIS Lab, Research Division of SIRIS Academic, Spain
[3] Politecnico di Milano, Milan, Italy
{mirjana.mazuran|letizia.tanca}@polimi.it

Abstract. Given the events modeled by a business process, it may happen in the presence of alternative execution paths that the data required by a certain event determines somehow what event is executed next. Then, the process can be modeled by using an approximate functional dependency between the data required by both events. We apply this approach in the context of conformance checking: given a business process model with a functional dependency (FD) that no longer corresponds to the observed reality, we propose corrections to the FD to make it exact or at least to improve its confidence and produce a more accurate model.

1 Introduction

Conformance checking aims at determining whether the actual execution of a business process, as recorded within an event log, conforms to its model. This approach helps bridge the gap between process specifications and their implementation, by detecting deviations between models and reality [1]. Several approaches take data into consideration when looking for these misalignments [2,6,8,9,11]. They assume that the data is provided through global attributes, and do not deal with structured data, which instead is usually considered in business process specification. Moreover, these approaches are able to highlight the deviations between the real executions and the process model, but do not propose alternatives to modify the model so that it becomes correct again.

In contrast, our work addresses conformance checking by considering an artifact-centric perspective of process definition where the data is represented by a UML class diagram and the processes are described in BPMN. So, we use standard and common formalisms for conceptual representation of data and processes, and are aligned with recent proposals aimed at specifying business process conceptually to facilitate understanding of these models by the domain experts [4, 5]. We assume in this paper that the activities in the BPMN diagram and in the event log are the same (no new activity is taking place in the

* This work is supported by project TIN2014-52938-C2-2-R, project 2014 SGR 1534, H2020 IT2Rail grant. 636078 and Italian project SHELL CTN01 00128 111357.

actual execution of the business process) and we concentrate on analyzing the correctness of the conditions specfied in the condition points of a BPMN.

With our approach, not only can we identify current deviations between the model and the actual execution of the process, but we can also propose modifications to the models to correct this misalignment. This is achieved by analyzing, for each event in the log, the functional dependency holding between its contextual data and the next branch after it.

The following example illustrates our contribution. Assume that a Taxi Service has developed an application to coordinate its taxi drivers. When they are in service, they may receive offers for new rides. Then, depending on the distance and the price of the offer, the drivers decide to accept/reject it. The BPMN model in Figure 1 shows this behavior.

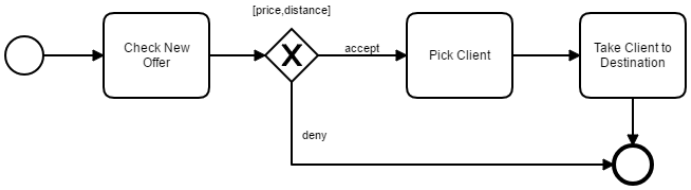
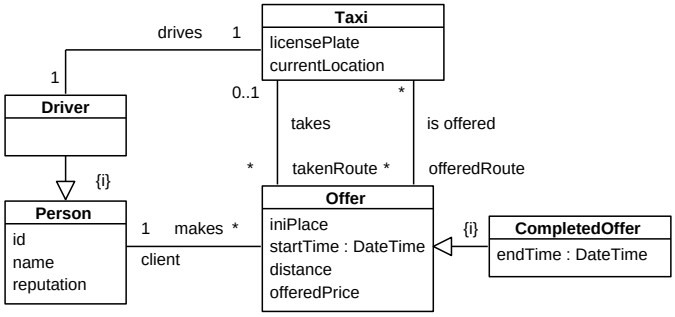


Fig. 1. BPMN model to handle taxi requests

In order to execute artifact-centric approaches, activities need to handle data. In particular, the data required by previous activities is shown in the UML diagram in Figure 1. Note that it contains information regarding the taxi and its driver, and also regarding the offers and the clients that make them. Additionally, we include an OCL operation that defines how the activity *check new offer* shown in the BPMN updates the data represented in the class diagram [4].




```
Operation: checkNewOffer(Offer o, Taxi t)
post: t.offeredRoute->includes(o)
```

Fig. 2. UML diagram describing data/operations required to handle a request

When the application is running, it may come up with the log in Table 1, which shows the attributes related to the *new offer* event (as given by the class *Offer* in Figure 1), and the name of the branch taken in the BPMN diagram (i.e. the decision made by the Taxi driver).

Table 1. Price, distance, and acceptance relation

timestamp	*price*	*distance*	*client reputation*	*driver reputation*	*Next event*
21-03-17 12:01:01	78,13	4,8	4	2	Pick client
21-03-17 14:45:12	35,12	34	2	3	Refuse client
21-03-17 14:56:27	34,14	35	5	4	Pick client
21-03-17 11:51:21	37,89	32	1	3	Refuse client

From the log we observe that some offers with distances and prices similar to those that are accepted are instead rejected, thus actually taxi drivers DO NOT necessarily decide to accept or reject an offer just based on the distance and price: clearly, this implies that the previous BPMN model is incorrect. At a closer observation of the data in the log, we eventually note that the decision depends also on the client's reputation. While this seems rather obvious with few data, in practice, with huge amounts of data, a manual analysis is unfeasible, and error prone because of the difficulty of the analysis itself.

In this paper we propose an approach that uses the techniques for evolving approximate functional dependencies proposed in [10] to compute *to what extent* the event log conforms to the business process model specified, and to identify the attributes (possibly missing from the BPMN diagram) that determine the next branch taken in a decision point of a process execution.

2 Preliminaries

Conformance Checking It receives as input a business process model and the *footprints* of the processes executed in the information system in terms of an event log. An execution of a process (also known as *process instance* or *case*) is represented as a sequence of activities called *trace*. We assume, without loss of generality, that each trace can be identified with a (surrogate) *id*, and each activity within the trace with a *timestamp* and *activity name*. An event log is a set of traces, representing the behavior observed in the information system during the execution of the process.

Definition 1 (Trace, Event Log). *Let $A \subseteq \mathcal{U}_A$ be a set of events A over a universe of events $\mathcal{U}_A$. A trace $\sigma \in A^*$ is a sequence of events. An event log $L \in \mathcal{P}(A^*)$ is a set of traces, where $\mathcal{P}(A^*)$ represents the power set of A^*.*

Functional Dependencies Let r be an instance of relation $R(A_1, A_2, ..., A_n)$, $|r|$ denotes the number of tuples in r, $t[A_i]$ the value of the attribute A_i in the tuple t and $\pi_X(r)$ is the projection of r on the attributes of X. A functional dependency F over R has the form $F : X \to Y$ where X and Y are two subsets of the attributes of R. Given an instance r of R, r satisfies an FD F defined on R if, for every pair of tuples t_1, t_2 in r, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. An instance r is inconsistent with respect to F if it does not satisfy it. Each FD can be characterized by its *confidence* $c_{F,r}$ and its *goodness* $g_{F,r}$ [10]:

Definition 2. *Let r be an instance of a relation R, X and Y two subsets of the attributes of R and $F : X \to Y$ a functional dependency over R. Then:*

$$c_{F,r} = |\pi_X(r)|/|\pi_{XY}(r)| \text{ and } g_{F,r} = |\pi_X(r)| - |\pi_Y(r)|$$

Depending on the confidence value we have the following definition:

Definition 3. *Given a relation R, an instance r of R, a functional dependency F over R and its confidence $c_{F,r}$, we say that F is an* exact functional dependency *iff $c_{F,r} = 1$, otherwise it is an* approximate functional dependency*.*

3 Our Approach

We start from the signature of the events of the BPMN to be analyzed and from the database storing the data of the process. Then, we automatically build the *enriched log table*, which is a new database table that essentially contains the contextual data of the initial event applied, together with the subsequent events. At runtime, the enriched log is filled with the real data of the processes. Then, to have the data ready for analysis, we have to discretize the continuous values of the *enriched log* since we are not interested in how a particular continuous value determines the next event, but on how a *kind* of values does. The resulting *discrete enriched log* is the basis for our conformance checking analysis.

3.1 Enriching the Log

The enriched log table should contain all necessary attributes to store the *contextual data* of each event execution, plus an additional attribute to store the name of the successive event. The contextual data consist of the values given in the event's input, together with all those values from the database that can be obtained through joins following the *foreign keys*.

We obtain these contextual attributes by looking at the UML types of the event's arguments. Then, we pick the attributes in the relational tables implementing these UML types and chase the *foreign keys* of these tables recursively to obtain more tables (and attributes). Note that, since this process is essentially a chase of *foreign keys*, we need to pay special attention to *foreign key* cycles. Indeed, chasing *foreign key* cycles never stops, thus, we should eventually break these cycles at some predefined point. Intuitively, this is done by deciding how many nested levels of joins we want to perform.

This process can be performed at compile time since the contextual attributes are fully determined by the user's given operation signature, the database schema, and the mapping between the UML schema and the underlying relational schema. At runtime we store each execution of the transition under study in a single row (tuple) of the enriched log. That is, whenever we detect an execution of the targeted transition, we query the database using the given event argument values to obtain the values for all its attributes. In the same row, we also store the next event applied by the user.

3.2 Discretization

The existence of a functional dependency (FD) depends, of course, on the data, and indeed, two different discretizations of the data may affect the fact that the dependency holds or not in the obtained table. Consider the FD $F : Distance \rightarrow Outcome$, meaning that a taxi driver decides to accept or reject a request only based on the distance. Suppose that the attribute *Distance* has been discretized to assume the values $\{short, long\}$ and that the data satisfy the FD. If we change the discretization of the *Distance* to the domain $\{short, medium, long\}$, then it is likely that the data will not satisfy the FD any longer, because now some tuples having *medium* distance will belong to the *accepted* category, while others with different, but still medium, distance, will be among the *rejected* ones.

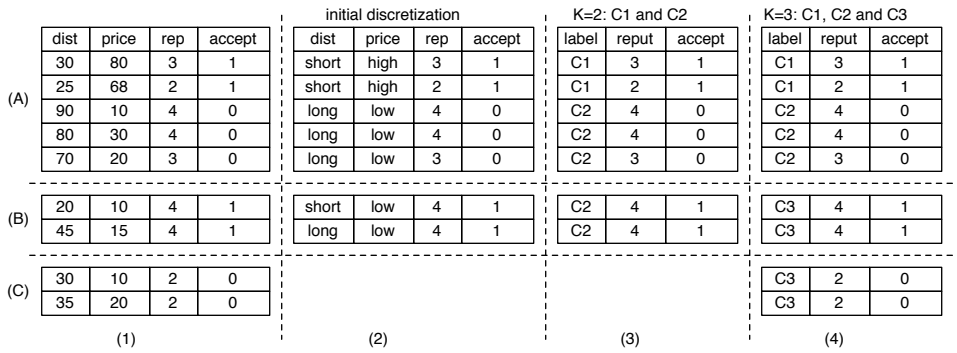
These observations also hold for FDs that involve more than one attribute. Consider $F : Distance, Price \rightarrow Outcome$, that is, a taxi driver decides to accept or reject a request based on the distance and price of the offer. It is also possible that we find an "appropriate" discretization of the two attributes *Distance* and *Price* such that the tuples satisfy the FD. Note that most discretization techniques are univariate, that is, they consider only one feature at a time and can be used to discretize *Distance* and *Price* independently of one another. However, the two attributes are involved in the FD together, thus a multivariate discretization, performed on the two attributes together, would be more appropriate.

One way to perform multivariate discretization is by means of clustering techniques: data can be clustered according to the values of the attributes *Distance* and *Price*, thus, each cluster contains data with a certain combination of ranges of the domains of the two attributes. Then each cluster can be given a label that is the symbolic representation of such combination of domain ranges. This label can be interpreted as *a single categorical attribute*, that takes the place of the two attributes *Distance* and *Price*. Therefore, a FD is influenced by the discretization process, but can also be used, conversely, to guide the discretization task. In fact, given a FD, we can try and find a discretization of the antecedent such that the FD would hold on the current data: for the FD to hold on the data, the antecedent must have at least the same amount of distinct values as the consequent (ideally each of them should be related to exactly one of the distinct values of the consequent) or more (more than one distinct value of the antecedent is related to the same value of the consequent). If this is true, then the confidence of the FD will be 1 (thus an exact FD) otherwise it will be less than 1 (thus an approximate FD). The greater the amount of distinct values of the antecedent and of the consequent, the worse the goodness of the FD.

Let r be an instance of a relation R, $F : X \rightarrow Y$ an FD defined on it and $K = |\pi_Y(r)|$ the number of distinct values of the consequent of the FD. The process of discretization, under the guidance of F, consists in finding K homogeneous clusters of the tuples based on their values of X. If such clustering exists it represents a multivariate discretization of X such that F holds on the discretized data.

3.3 Applying Functional Dependency Evolution

Given a FD that is no longer valid, there are two ways to capture the modeled reality again: (i) try to change the way the attributes in the FD antecedent are discretized, so that the dependency holds again, (ii) keep the same discretization and "strengthen" the antecedent of the FD by adding an attribute to it. Consider the example of Section 3.2 and suppose both price and distance vary in the range [1,98] (both 1 and 98 are included) and that, initially, the taxi service company has a conceptual discretization of these ranges such that, for the *distance*, [1,32] means *short* and [33,98] means *long*, while for the *offeredPrice* [1,65] means *low* and [66,98] means *high*. Moreover, $F : distance, offeredPrice \rightarrow accepted$ is defined on the data. Suppose the initial data, without discretization are



dist	price	rep	accept
30	80	3	1
25	68	2	1
90	10	4	0
80	30	4	0
70	20	3	0

dist	price	rep	accept
short	high	3	1
short	high	2	1
long	low	4	0
long	low	4	0
long	low	3	0

label	reput	accept
C1	3	1
C1	2	1
C2	4	0
C2	4	0
C2	3	0

label	reput	accept
C1	3	1
C1	2	1
C2	4	0
C2	4	0
C2	3	0

dist	price	rep	accept
20	10	4	1
45	15	4	1

dist	price	rep	accept
short	low	4	1
long	low	4	1

label	reput	accept
C2	4	1
C2	4	1

label	reput	accept
C3	4	1
C3	4	1

dist	price	rep	accept
30	10	2	0
35	20	2	0

label	reput	accept
C3	2	0
C3	2	0

Fig. 3. Taxi service running example

the tuples in Figure 3(1)(A). After discretization we obtain the data in (2)(A) satisfying F with c_F=1 and g_F=0. By running the K-means algorithm (with K=2) we find two homogeneous clusters: C_1 with centroid *distance*=29 and $offeredPrice$=79.333 where all tuples have *accepted*=1 and C_2 with centroid *distance*=75 and $offeredPrice$=25 where all tuples have *accepted*=0. Thus, $C1$ represents (accepted) offers with short distance and high price while $C2$ represents (rejected) offers with long distance and low price. We can also discretize the data assigning them the label of the cluster to which they belong (as shown in Figure 3(3)(A)).

Now, suppose that the two tuples in Figure 3(1)(B) arrive. The first tuple still satisfies the FD but the second one does not. Therefore, the FD is not satisfied any longer. By running the K-means algorithm again on the data we find C_1 (centroid *distance*=29, $offeredPrice$=79.333), where all tuples have *accepted*=1; and C_2 (centroid *distance*=75, $offeredPrice$=25), where 60% of tuples have *accepted*=0 and the rest *accepted*=1, i.e., a non-homogeneous cluster. Thus, C_1 still represents accepted offers with short distance and high price while C_2 mixes tuples having different ranges of distance and price and different outcomes.

Now, we can choose different strategies: (i) accept approximate FDs, and decide that c_F=2/3 is still a good confidence for the FD: nothing should be

changed in this case; (ii) try to find a different discretization of the data such that the FD would hold on them: the data discretization should be changed; (iii) choose to evolve the FD by looking for attributes that can be added to its antecedent in order to repair it: the FD should be changed.

Suppose that we decide that the new confidence of the FD is too low and try to re-cluster the data with K=3. Then, we find 3 homogeneous clusters whose centroids are: C_1: *distance*=27.5 and $offeredPrice$=74 with *accepted*=1; C_2: *distance*=80 and $offeredPrice$=20 with *accepted*=0; C_3: *distance*=32.5 and $offeredPrice$=12.5 with *accepted*=1. In fact, if we associate each tuple with the label of the cluster it belongs to (see Figure 3(4)(A-B)), instead of *distance* and $offeredPrice$, we can see that the functional dependency F holds on these data with c_F=1 and g_F=1. As data continue to arrive the confidence of the FD might continue to decrease: we could try to change the discretization again but there might be a point where this would not be possible anymore. For example, suppose the two new tuples shown in Figure 3(1)(C) are added to the data: a homogeneous clustering is found only at K=9, when each cluster contains only one tuple! Thus, no discretization can bring the data to satisfy the FD. At this point we can choose to change the FD, looking for a minimal set of attributes that can be added to its antecedent in order to restore a high confidence value. To this end we apply the technique for evolving FD in [10] and add one attribute of the relation at a time to the antecedent of F, evaluate the confidence of the newly built FD and decide what is the confidence value we consider acceptable. Consider the tuples in Figure 3(4): since with the discretization we have performed the data no longer satisfies F, we try adding *reputation* to the antecedent of F. The obtained FD: $label, reputation \rightarrow accepted$ has confidence 1 and goodness 4, hence the reputation allows us to discriminate in particular inside the C_3 cluster. By remembering its centroid (*distance*=32.5 and $offeredPrice$=12.5) we can say, intuitively, that it contains medium-distance offers with low price. Thus, by repairing the FD, we are saying that these offers are accepted if the client has high reputation and are rejected otherwise.

4 Related Work

Several proposals perform conformance checking over process models and logs considering the data. Most of them use procedural models [6, 9, 11] and are usually based on Petri nets with data, although they are scalable to other representations such as BPMN or EPC. More specifically, [11] deals with event logs with deviating behavior and more complex control-flow constructs. [7] follows a different approach by splitting the log when doing the conformance checking, to achieve better efficiency. All of [6,9] incorporate resources into the conformance checking, but [6, 9] also consider time constraints in the process. On the other hand, [2, 8] perform conformance checking and process discovery on declarative process models which consider data. They are based on the Declare language, which has formal semantics and a graphical representation and it allows defining many different types of dependencies between tasks. All these approaches do

not deal with structured data, but rather they consider data as a set of global attributes. Moreover, none of them deals with variable discretization. In terms of the data itself, note that the approaches performing conformance checking highlight the deviations between the real executions, as recorded in the log, and the process model, but do not propose alternative conditions like we do. [8, 11] do discover data conditions from the log, but both are focused on discovery, not on conformance. In addition, [8] uses a declarative specification instead of a procedural one, like we do. Our method for analyzing, for each event in the log, the (approximate) functional dependency holding between its contextual data and the next branch is based on [10], which employs the two measures *confidence* and *goodness* to evaluate the degree of approximation of a FD. This is rather efficient because it only requires to count tuples. However, we plan to study whether other measures of approximation [3] produce suggestions of model modifications that are "closer" to the intention of the designer.

5 Conclusions

Our approach for conformance checking of artifact-centric BPMN models determines whether the actual execution of a business process conforms to the model and, by analyzing the data dependencies, proposes changes to the model accordingly.

References

1. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes (2011)
2. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. 65, 194–211 (2016)
3. Caruccio, L., Deufemia, V., Polese, G.: Relaxed functional dependencies - A survey of approaches. IEEE Trans. Knowl. Data Eng. 28(1), 147–165 (2016)
4. De Giacomo, G., Oriol, X., Estañol, M., Teniente, E.: Linking data and BPMN processes to achieve executable model. In: CAiSE (2017), accepted for publication
5. Estañol, M., Sancho, M., Teniente, E.: Verification and validation of UML artifact-centric business process models. In: CAiSE. pp. 434–449 (2015)
6. de Leoni, M., van der Aalst, W.M.P.: Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In: 11th BPM. pp. 113–129 (2013)
7. de Leoni, M., Munoz-Gama, J., Carmona, J., van der Aalst, W.M.P.: Decomposing alignment-based conformance checking of data-aware process models. In: On the Move to Meaningful Internet Systems: OTM. pp. 3–20 (2014)
8. Maggi, F.M., Dumas, M., García-Bañuelos, L., Montali, M.: Discovering data-aware declarative process models from event logs. In: 11th BPM. pp. 81–96 (2013)
9. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Balanced multi-perspective checking of process conformance. Computing 98(4), 407–437 (2016)
10. Mazuran, M., Quintarelli, E., Tanca, L., Ugolini, S.: Semi-automatic support for evolving functional dependencies. In: EDBT. pp. 293–304 (2016)
11. Rozinat, A., van der Aalst, W.M.P.: Decision mining in prom. In: 4th International Conference on Business Process Management. pp. 420–425 (2006)