# Declarative Process Mining for DCR Graphs*

Søren Debois
IT University of Copenhagen
Copenhagen, Denmark
debois@itu.dk

Thomas T. Hildebrandt
IT University of Copenhagen
Copenhagen, Denmark
hilde@itu.dk

Paw Høvsgaard Laursen
IT University of Copenhagen
Copenhagen, Denmark
pawh@itu.dk

Kenneth Ry Ulrik
IT University of Copenhagen
Copenhagen, Denmark
kulr@itu.dk

## ABSTRACT

We investigate process mining for the declarative Dynamic Condition Response (DCR) graphs process modelling language. We contribute (a) a process mining algorithm for DCR graphs, (b) a proposal for a set of metrics quantifying output model quality, and (c) a preliminary example-based comparison with the Declare Maps Miner. The algorithm takes a contradiction-based approach, that is, we initially assume that all possible constraints hold, subsequently removing constraints as they are observed to be violated by traces in the input log.

## CCS Concepts

**•Information systems → Data mining; •Theory of computation → Logic; •Computing methodologies → Knowledge representation and reasoning;**

## Keywords

Declarative process mining; DCR graphs

## 1. INTRODUCTION

Business process management (BPM) technologies [32] support the management and digitalisation of workflows and business processes by employing explicit process models, following a cycle of process (re)design, validation, execution and monitoring.

*Process mining* algorithms [31] have been proposed for the identification of process models from process logs, supporting both process design and compliance monitoring.

Most industrial BPM tools and process miners describe processes as imperative flow diagrams such as BPMN. However, flow diagrams tend to get either too rigid or too complex, in particular for knowledge work processes having a high degree of variation [27]. Moreover, flow diagrams only describe *how* to perform a process, leaving a gap to the legal regulations and guidelines, that are often more *declarative* in nature, describing *why* the process must be performed in certain ways, not how exactly it must be performed. For instance, a clinical guideline may state, that a patient must consent to a blood transfusion [13]. It does not state exactly when such consent should be obtained, only "prior to the transfusion".

For this reason, it is recommended to use flow diagrams only for routine processes, or for describing common standard practices and allow deviations [27]. It has been advocated that declarative notations should be used as output of process mining (e.g. [17]) and for run-time process support (e.g. [24, 23, 28]). For the former, one hopes to extract from a process log the rules obeyed in practice (the "why") as opposed to a flow-diagram describing the usual executions (the "how"). For the latter, one hopes to guide knowledge workers to activities in conformance with rules and regulations.

Implementation techniques for most declarative models such as Declare [26] and DecSerFlow [30], rely on translating the declarative constraints to an imperative model (e.g., an automaton [20]) to enable execution. Such translation usually entail a state-space explosion, and run-time adaptation of constraints becomes more difficult, because the automaton must be recomputed when constraints change.

A notable exception is the Dynamic Condition Response (DCR) graphs process language [11, 29]. DCR graphs can be executed without intermediate transformation to an imperative model creating the entire transition graph, and more directly support run-time adaptive case management [23, 5]. DCR graphs are supported by industrial design and case management tools (see e.g. dcrgraphs.net and [5]).

In the present paper, we present the first process mining algorithm for DCR graphs.

## 2. DCR GRAPHS

In this Section, we briefly recall DCR graphs. For a formal introduction and applications, refer to [11, 22, 29, 3, 5, 6].

Dynamic Condition Response graphs is a declarative modelling notation describing at the same time a process and its run-time state. The core notation comprises activities, activity states, and four relations between activities. An activity state comprises three booleans, indicating respectively whether the activity has been executed, is included, and

*Authors listed alphabetically. This work supported in part by Velux Foundation grant #33295 and Exformatics A/S.



*SAC 2017 April 03-07, 2017, Marrakech, Morocco*



ACM ISBN 978-1-4503-4486-9/17/04.

DOI: http://dx.doi.org/10.1145/3019612.3019622

is pending. Intuitively, activities that are not *included* are treated as temporarily absent from the workflow; activities that are pending must eventually be executed or excluded before the workflow may complete.

Relations between activities govern whether an activity can currently be executed and how executing one activity modifies the state of another. A *condition* $A \bullet\!\leftarrow B$ means that the activity $A$ cannot execute unless $B$ was previously executed, i.e., the the executed-state of $B$ is true. Executing an activity clears its pending-state and sets its execution-state. The *response* $A \bullet\!\!\rightarrow B$ means that whenever $A$ executes, the pending-state of $B$ is set. An *inclusion* $A \rightarrow\!+ B$ means that whenever $A$ executes, the inclusion-state of $B$ is set, and conversely, an *exclusion* $A \rightarrow\!\% B$ means that whenever $A$ is executed, the inclusion-state of $B$ is cleared.

Note that excluding an activity voids it as both a condition and a response: If $A \bullet\!\leftarrow B$ and $B$ is not executed but also not included, $A$ is free to execute. Conversely, an activity which is pending but also not included does not prevent the workflow from being completed.

While the condition and response relations has the same meaning as the corresponding relations in DECLARE [25] or DecSerFlow [30], the inclusion and exclusion relations provide the ability to dynamically include and remove conditions and response obligations. They have no direct counterpart in other declarative notations.

# 3. MODEL METRICS

In this Section, we present quality measures quantifying the appropriateness of a DCR graph $G$ for a given log $l$. We take as starting point the already established metrics of *fitness*, *precision*, *generality*, and *simplicity* introduced in [1] in the context of (internally binary) process trees.

## 3.1 Fitness

*Replay fitness* is defined in [1] as the normalised ratio of how an alignment between the input process tree and the event log differs over the maximum possible alignment for the model given an arbitrary event log. A variant of this approach was successfully applied to declarative models in [2].

However, within Adaptive Case Management, the core application area of DCR graphs [3, 5, 6], we use declarative models specifically to encompass *all* admissible behaviours. In this context, we take the view that the appropriate notion of "replay fitness" is simply the ability of the model to replay the traces of the input log *exactly*. As such, we define fitness to be simply the ratio of input traces in the log $l$ replayable by the DCR model $G$:

$$\mathsf{fitness}(G,l) = \frac{\#\mathsf{ReplayableTraces}(G,l)}{\#\mathsf{Traces}(l)}$$

## 3.2 Precision

*Precision* is defined in [1] essentially as a tally of the amount of behavioural options unused by the log. This idea is straightforward to apply to DCR graphs: replay the log and record, for each reached state in the graph, the activities that are executable in that state as well as how many of these executable activities were actually executed at some point.

We transfer this idea directly to DCR graphs, measuring for each visited state the number of enabled activities actually executed in that state:

$$\mathsf{precision}(G,l) = \frac{\sum_{s\in\mathsf{VisitedStates}(l)} \#\mathsf{ActivitiesExecuted}(G,s,l)}{\sum_{s\in\mathsf{VisitedStates}(l)} \#\mathsf{ExecutableActivities}(G,s,l)}$$

As a technical note, "#ActivitiesExecuted" is only counted up the first time an activity is seen executed in a certain state. If it is observed to be executed from the same state multiple times, we only count the one execution.

However, we question the usefulness of this measure in the context of Adaptive Case Management. One advantage of declarative models in this context is that they afford flexibility for case workers to handle infrequent outlier cases. By definition, these happen only seldom; we cannot expect all such cases to be represented in the input log. Encompassing them, then, entails supporting a very large amount of *potential* such outlier cases. So it would be the expectation and not the exception that a log uses only a tiny fragment of the options available in the model.

This thinking was confirmed in [6], where a commercial system based on DCR graphs supported at least five orders of magnitude more states than observed in actual logs.

## 3.3 Simplicity

Simplicity for process trees is defined in [1] (roughly) as the ratio of the size of the internal binary process tree to the amount of activities in the input log. This notion of simplicity was partly motivated by previous findings that size is the main driver of errors in process models [21].

However, these findings have to the best our knowledge *not* been replicated in the context of *declarative* process models [9, 10, 33], where key impediments to understandability appear to be the number of constraints as opposed to the number of activities. Moreover, measuring the number of activities in DCR graphs is not a proxy for semantic complexity the way measuring duplicate activity representation is in a process tree is—large graphs are not necessarily complex.

Accordingly, we measure the simplicity of a DCR graph by (1) the number of pairs of related activities (Relation Pairs: RP); (2) the total amount of relations. Note that (2) is greater than (1) when some activities are related by more than one relation. Under this measure, a simplest possible graph is any graph with no relations.

$$\mathsf{simplicity}(\mathsf{G}) = \frac{(1 - \frac{\#\mathsf{Relations}}{\#\mathsf{PossibleRelations}})}{2} + \frac{(1 - \frac{\#\mathsf{RPs}}{\#\mathsf{PossibleRPs}})}{2}$$

Note that because the number of activities in a declarative model is not necessarily correlated with its complexity, in contrast to [1], we can define simplicity *without* reference to the events in the particular log $l$.

We have ignored in this measure (and in this paper) complexity of DCR graphs stemming from nesting [12]. While nesting generally enhance perceived understandability (see, e.g., [34, 35]), it may also implicitly introduce more relations. We leave open the question of exactly what a good measure of simplicity in the presence of nesting might be.

## 3.4 Generality

The notion of *generality* is defined in [1] for process trees as the frequency with which each node of the process tree

must be visited in order to produce the given log. Infrequently visited nodes of the process tree *decreases* generality.

This notion is specific to the notion of process trees and, to a lesser extent, imperative models. DCR graphs have no notion resembling the "inner nodes" of a process tree that can be considered "visited" during executions.

Moreover, generalisation is intended to assess "*the extent to which the resulting model will be able to reproduce future behaviour*" [1, p. 2]. This is an extremely important quality for both declarative models in general and ACM models in particular. However, we contend that it cannot reasonably be measured without appeal to domain-knowledge: We cannot from the logs alone determine which are useful generalisations (e.g., swapping the order of obtaining authorisation signatures in a loan application) and which are not (e.g., swapping the order of granting the loan and obtaining authorisation).

Altogether, we leave the definition of a notion of generalisation for DCR graphs as future work.

# 4. DCR MINING

In this Section we present a mining algorithm for DCR graphs: Given a log $l$, produce a DCR graph $G$. We take a "contradiction-based" approach to mining for constraint-based modelling languages: Begin with the set of activities and all possible constraints, and remove a constraint whenever the input log has a trace violating it. This approach has proven successful for DECLARE [8, 16, 2, 18], although requiring non-trivial enhancements to curb combinatorial explosion because of the large number of possible DECLARE constraints; to avoid contradictory models [7]; and to avoid unhelpful vacuously satisfied constraints [19]. DCR graphs have only 4 relations; checking those for each pair of activities across all input traces *is* a viable option.

Because include relations by definition trump exclude relations, we do not take as starting point a graph with every possible constraint. Rather, in the interest of beginning with the most restrictive possible graph, we retain exclusions and omit inclusions. Altogether, our initial, restricted over-approximation will have conditions, exclusions, and responses between any pair of events.

In DCR, we have to account not only for constraints, but also initial state. Following the principle behind contradiction-based mining, we opt for the most restrictive possible starting state: each activity is initially not executed, not included, and pending.

## 4.1 Algorithm

The core mining algorithm is given in Algorithm 1. We comment on specifics below. In the algorithm, for a given trace $t$, we write $t_0, t_1, \ldots$ for the sequence of activities in $t$.

**Include- and exclude-relations.** When we observe an activity at the start of a trace, we set the initial included-state of that activity to true. When an activity is observed after the start of a trace, we replace the exclude-relation from the preceding event with an include, again to allow the two activities to be executed in succession.

**Response relations.** At the completion of a trace, we check that for each activity execution in that trace whether all the activities that had response relations installed have been executed later in the trace. If not, we remove the offending response relations. Moreover, we clear the initial pending-state for all activities not seen in that trace.

The latter of these rules is an over-approximation; pending activities may be discharged *either* by execution *or* by being excluded. We make the present choice partly to make an initially-pending state signal that the activity *has* to be executed in any and all traces, not just excluded, partly to facilitate dynamic mining, see Section 4.2.

**Condition relations.** When we observe an activity execution in a trace, we remove conditions from non-executed activities in the trace in question.

**Algorithm 1** Core DCR mining algorithm

```
1: function MINE(log)
2:     G := activities(log)
3:     for all x where x activity of G do
4:         set x excluded, pending, not executed in G
5:     end for
6:     for all (x, y) where x, y activities of G do
7:         G := G ∪ {x •← y, x •→ y, x →% y}
8:     end for
9:     for all t ∈ traces(log) do
10:        set t_0 included in G
11:        remove all conditions to t_0 from G
12:    end for
13:    for all t ∈ traces(log) do
14:        p := t_0
15:        for i from 1 to |t| − 1 do
16:            remove p →% t_i from G
17:            add p →+ t_i to G
18:            for all x where t_i •← x ∈ G do
19:                if x ∉ {t_j | j < i} then
20:                    remove t_i •← x from G
21:                end if
22:            end for
23:            for all x where t_i •→ x do
24:                if x ∉ {t_j | j > i} then
25:                    remove t_i •→ x from G
26:                end if
27:            end for
28:            p := t_i
29:        end for
30:        for all a ∉ t do
31:            set a not pending in G
32:        end for
33:    end for
34:    return G
35: end function
```

## 4.2 Correctness

Removal of a DCR constraint in general *does not* preserve admissibility of workflows. Here are two counterexamples:

1. The graph $A \rightarrow\!+\ B$ where $B$ is initially excluded admits the traces $A?+A(A \mid B)^+$. Removing the inclusion relation reduces the set of admitted traces to $A^*$.

2. The graph $B \bullet\!\leftarrow\ A \mid C \rightarrow\!\%\ A$ admits the trace $CB$; removing $C \rightarrow\!\%\ A$ makes that trace inadmissible.

This non-monotonicity is a central difference between DCR and DECLARE; it was studied in [4]. It follows that in a naive DCR-miner, whenever we remove a constraint, we

must re-check all previously processed traces to ensure that they are still admissible. Such a naive approach would lead to practically unacceptable running-times.

Our mining algorithm rests on the observation that these two examples exemplify the *only* two ways removing a constraint from a DCR graph may reduce its set of accepted traces; Algorithm 1 does not remove such constraints when it is dangerous to do so.

We use this insight to prove Algorithm 1 correct. Write $G \models t$ if a DCR graph $G$ accepts a trace $t$; write $\mathcal{L}(G)$ for the set $\{t \mid G \models t\}$.

PROPOSITION 4.1. *Let $G$ be a label-deterministic DCR graph, and let $G'$ be the DCR graph obtained by removing a single constraint $\gamma$ from $G$. Suppose $t$ is a trace s.t. $G \models t$. Then $G' \not\models t$ implies either*

1. *$\gamma = A \rightarrow+ B$ for some $A, B$, or*
2. *$\gamma = A \rightarrow\% B$ and $C \bullet\!\leftarrow B$ for some $A, B, C$, and there exists $i$ s.t. $t_i = C$ but for no $j < i$ do we have $t_j = B$.*

PROOF. Suppose $G \not\models l$. We proceed by cases on $\gamma$. If $\gamma$ is a condition or a response, clearly $\mathcal{L}(G) \subseteq \mathcal{L}(G')$; contradiction. If $\gamma$ is an inclusion we are done. So suppose finally $\gamma = A \rightarrow\% B$. If for no $C$ we have $C \bullet\!\leftarrow B$ it follows easily that $\mathcal{L}(G) \subseteq \mathcal{L}(G')$, so we must have $C \bullet\!\leftarrow B$ for some $C$. Suppose for a contradiction that for all such $C$, we have for all $t_i$ either $t_i \neq C$ or $t_i = C$ and for some $j < i$ we have $t_i = B$. In either case, it is straightforward to prove by induction on $t$ that $G' \models l$; contradiction. □

LEMMA 4.2. *Let $G$ be the value of $G$ at line 16 and $G'$ the value of $G$ at line 28 in Algorithm 1 within the same iteration of the loop. Then $\forall t \in \log.\ G \models t \implies G' \models t$.*

PROOF. For the *removed* relations, by Proposition 4.1 it is sufficient to verify that we remove no constraint satisfying Items 1 and 2 of that theorem. By inspection, Algorithm 1 does not remove inclusions, and so cannot violate Item 1. By inspection, we see that when the algorithm removes an exclusion (line 16) it also removes conditions that would violate Item 2 (lines 18-22).

For the *added* inclusion at line 17, it is sufficient to note that adding inclusion may only lead to inadmissible traces if it includes a left-hand side of a condition; however, by line 18–22 only conditions that were executed are retained. □

THEOREM 4.3 (CORRECTNESS). *Let $G$ be the output of Algorithm 1 on a log $l$. Then forall $t \in l$ we have $G \models t$.*

PROOF. Using Lemma 4.2, it is straightforward to verify by induction on each $t \in l$ that $t_i$ was enabled after $t_{i-1}$ in $G$ at line 28, and that $G$ is accepting for $t_{|t|-1}$ at line 32. □

## 4.3 Weighing of constraints

Algorithm 1 does not take into account noise in the log, since we remove every violated constraint. Moreover, in some applications, we may desire not a completely fitting model, but rather one that characterises the "common execution": We may want to trade off fitness for precision.

Following common approaches to process mining, we only remove a relation when our *confidence* in removing that constraint is above a certain *threshold*. Each constraint is therefore assigned two values: an invocation counter and a violation counter. The invocation counter tallies the number of traces in which the constraint was invoked, e.g., the number of traces where the source activity of an exclude-relation was executed. The violation counter simply tallies the number of traces in which the constraint was violated.

Exact criteria for invocations and violations are given in Table 1. The ratio of violations to invocations define our confidence in the removal of a constraint. A threshold below 0% will remove all constraints, resulting in a flower model. A threshold of exactly 0% retains only constraints satisfied in every trace. A threshold of 100% will remove no constraints; the output model will allow no runs.

Experimentally, the desired trade-off between precision and fitness occurs in the 0-15 % range. A threshold larger than 20 % would result in a large amount of the log being unsupported by the resulting graph.

## 4.4 Post-processing

To improve simplicity of the core algorithm's output, we remove *redundant constraints*, i.e., relations that never have an effect on what the output DCR-graph allows. Redundant relations are closely related to vacuous constraints in Declare mining [8, 16, 2, 18, 19], but turn out to be much easier to detect in DCR graphs.

This implementation does ad-hoc removal of redundant relations by replaying logs against the output of the core mining algorithm, removing those inclusion, exclusion and response relations that never modify the state of their target activities; as well as removing those conditions that never inhibit execution of their target activities.

To further improve Simplicity, one might consider introducing nested graphs [29, 12] when they reduce relations.

# 5. EXPERIMENTAL RESULTS

An implementation of Algorithm 1 with rudimentary redundancy removal is available at [14]. For an experimental comparison with the Declare Maps Miner, consider the log in Table 2. For the sake of clarity, the log consists of only ten traces and is based on a relatively simple regular expression. For a larger log, see the on-line results at [15].

The test log represents a basic process flow; parallels may be drawn to a real-world process where A is registration for an exam, B, C and D are answers to a multiple-choice question, and the student either passes (E) or fails (F). Failed students may retry if they wish, but if they pass, they can no longer re-take the exam.

Given the sample log, our algorithm, with a constraint-violation threshold of 15 %, returns the DCR-graph depicted in Figure 1. Because the log contains only a single occurrence of A followed by D, the exclusion constraint between them remains intact: the one in ten traces do not yield a sufficient statistical percentage of violation ($10 < 15$). Thus, as no other activity includes D, it is removed entirely from the result-graph as a result of redundancy removal.

The removal of activity D means that the trace A–D–F is no longer allowed, leaving the Fitness measure down at 90 %. This is, however, an acceptable trade-off for an increase of precision from 72.73 % if the threshold were set below 10 % to the final 78.57 %, as the two measures are now closer to each other. The main cause for this effect on the precision measure is the observed state-space that the execution of D involved. This, along with the fact that paths involving executions of B and C are quite well-traversed, results in a slightly higher, final precision measure.

Table 1: Threshold-dependant constraint removal

| Constraint | Invocation | Violation | Result |
|---|---|---|---|
| Excluded-state | Each trace | A is first in a Trace | A is Included |
| Exclude-relation | A is executed in a trace | B executed immediately after A | A → B exclude is changed to include |
| Condition-relation | B is executed in a trace | A is not executed before B | A → B condition is removed |
| Response-relation | A is executed in a trace | B is not executed after A | A → B response is removed |
| Pending-state | Each trace | A is not executed | A is not Pending |

Table 2: Example log. Follows the regular expression `(A(B+|C|D)F)*(A(B+|C|D)E)?`

| | | | |
|---|---|---|---|
| 1 | ABE | 6 | ACF |
| 2 | ACFABBF | 7 | ABFACFACE |
| 3 | ACE | 8 | ABBBF |
| 4 | ADF | 9 | ABBE |
| 5 | ABFABE | 10 | ACFACE |

## 5.1 Result comparison: Declare Maps Miner

For comparison, we show the the result graph of running the Declare Maps Miner [18] on the same log (Table 2) in Figure 2. The result is computed using a Declare Maps Miner support of 85 %, i.e., any constraint must be supported by at least 85 % of traces. This corresponds to the constraint-violation threshold of 15 % used by our algorithm above, as the contradiction-based method uses the threshold to tell when to remove a constraint, while Declare uses support for when to include a constraint.

- In the Declare model a trace must begin with A, followed by either B or C and then possibly ending in E, after which it is not permitted to go back to A.
- If C is chosen after A, it is also possible to continue to F, instead of E, and then possible to return to A and start over.
- If B is chosen instead of C, it is then not possible to choose F, despite four instances of this succession in the log.
- The exclusive choice constraint between A and D, combined with A being the initial activity, means that it is not possible to ever execute D. This is similar to the DCR miner never including D.
- Additionally, the Declare model does not have a terminal state. If E is executed, A and F cannot subsequently occur, but the same does not seem to apply for B and C. Thus, these three can be executed arbitrarily after E, even though all traces in the log end in E.

This last point marks the primary difference between the two resulting models. Overall, the results seem to suggest that our miner is slightly better in terms of closely reflecting the underlying process of the test log (its regular expression).

We emphasise that these results are only for this single, simple example, and may not necessarily generalise.

# 6. CONCLUSIONS

We have presented the first process mining algorithm for DCR graphs and a set of metrics quantifying output model quality. The algorithm has been implemented and a preliminary example-based comparison with the Declare Maps Miner has been carried out. We plan as future work to extend the evaluation and use of the algorithm to real-time distributed process mining.

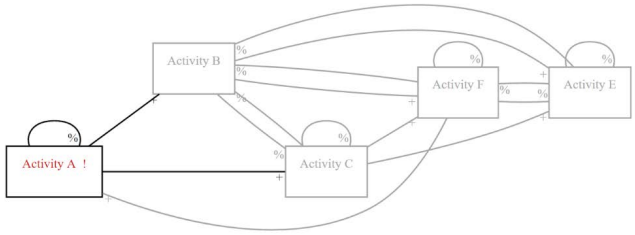


Figure 1: DCR model. Obtained by running Algorithm 1 (extended with weighted constraints, threshold 15 %) on the log of Table 2 and removing redundancy. Fitness: 90.00 %, Precision: 78.58 %, Simplicity: 40.39 %

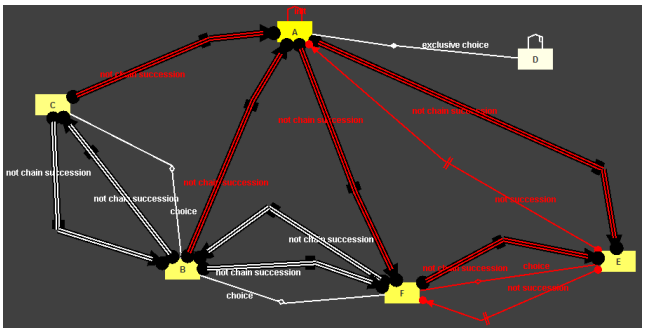


Figure 2: Declare model. Obtained by running the Declare Maps Miner on the log of Table 2.

# 7. REFERENCES


[1] J. Buijs, B. van Dongen, and W. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *OTM '12*, volume 7565 of *LNCS*, pages 305–322. Springer, 2012.

[2] M. de Leoni, F. M. Maggi, and W. M. P. van der Aalst. An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. *Information Systems*, 47:258–277, Jan. 2015.

[3] S. Debois, T. T. Hildebrandt, M. Marquard, and T. Slaats. Hybrid process technologies in the financial sector. In *BPM '15*, pages 107–119, 2015.

[4] S. Debois, T. T. Hildebrandt, and T. Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In *FM '15*, pages 143–160, 2015.

[5] S. Debois, T. T. Hildebrandt, T. Slaats, and M. Marquard. A case for declarative process modelling: Agile development of a grant application

system. In *EDOC Workshops '14*, pages 126–133. IEEE Computer Society, 2014.

[6] S. Debois and T. Slaats. The analysis of a real life declarative process. In *Symposium Series on Computational Intelligence*, pages 1374 – 1382, Cape Town, Dec 2015. IEEE.

[7] C. Di Ciccio, F. M. Maggi, M. Montali, and J. Mendling. Ensuring model consistency in declarative process discovery. In *BPM '15*, pages 144–159, 2015.

[8] C. Di Ciccio and M. Mecella. Mining constraints for artful processes. In W. Abramowicz, D. Kriksciuniene, and V. Sakalauskas, editors, *BIS*, volume 117 of *Lecture Notes in Business Information Processing*. Springer, 2012.

[9] D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative versus imperative process modeling languages: The issue of understandability. In *Enterprise, Business-Process and Information Systems Modeling*, volume 29 of *Lecture Notes in Business Information Processing*, pages 353–366. Springer, 2009.

[10] C. Haisjackl, I. Barba, S. Zugal, P. Soffer, I. Hadar, M. Reichert, J. Pinggera, and B. Weber. Understanding declare models: strategies, pitfalls, empirical results. *Software & Systems Modeling*, pages 1–28, 2014.

[11] T. Hildebrandt and R. R. Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *Post-proceedings of PLACES 2010*, 2010.

[12] T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Nested dynamic condition response graphs. In *FSEN '11*, April 2011.

[13] Consent for blood transfusion. *Joint United Kingdom (UK) Blood Transfusion and Tissue Transplantation Services Professional Advisory Committee (JPAC)*, http://www.transfusionguidelines.org/transfusion-practice/consent-for-blood-transfusion-1, accessed Sept. 28th, 2016.

[14] P. H. Laursen and K. R. Ulrik. DCR miner. https://github.com/Kirluu/UlrikHovsgaardAlgorithm.

[15] P. H. Laursen and K. R. Ulrik. Hospital log study. https://github.com/Kirluu/UlrikHovsgaardAlgorithm/tree/master/Hospital%20log%20result.

[16] F. Maggi, R. Bose, and W. van der Aalst. Efficient discovery of understandable declarative models from event Logs. In *CAiSE*, pages 270–285, 2012.

[17] F. Maggi, A. Mooij, and W. van der Aalst. User-guided discovery of declarative process models. In *CIDM*, pages 192–199, 2011.

[18] F. M. Maggi. Declarative process mining with the Declare component of ProM. In M. Fauvet and B. F. van Dongen, editors, *BPM DEMO '13*, volume 1021 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[19] F. M. Maggi, M. Montali, C. Di Ciccio, and J. Mendling. Semantical vacuity detection in declarative process mining. In M. L. Rosa, P. Loos, and O. Pastor, editors, *BPM '16*, volume 9850 of *LNCS*, pages 158–175. Springer, 2016.

[20] F. Maria Maggi, M. Montali, M. Westergaard, and W. M. P. van der Aalst. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In *BPM '11*, volume 6896 of *LNCS*, pages 32–147, 2011.

[21] J. Mendling, H. M. W. Verbeek, B. F. van Dongen, W. M. P. van der Aalst, and G. Neumann. Detection and prediction of errors in EPCs of the SAP reference model. *Data & Knowledge Engineering*, 64(1):312–329, Jan. 2008.

[22] R. R. Mukkamala. *A Formal Model For Declarative Workflows - Dynamic Condition Response Graphs*. PhD thesis, IT University of Copenhagen, March 2012.

[23] R. R. Mukkamala, T. Hildebrandt, and T. Slaats. Towards trustworthy adaptive case management with dynamic condition response graphs. In *EDOC '13*, 2013.

[24] M. Pesic, H. Schonenberg, and W. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *EDOC '07*, pages 287–. IEEE, 2007.

[25] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. Van Der Aalst. Constraint-based workflow models: change made easy. In *On the Move 2007*, OTM'07, pages 77–94, Berlin, 2007. Springer.

[26] M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *Proc. of the 2006 international conference on Business Process Management Workshops*, BPM'06, pages 169–180. Springer, 2006.

[27] M. Reichert and B. Weber. *Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies*. Springer, 2012.

[28] I. Rychkova. Towards automated support for case management processes with declarative configurable specifications. In *Business Process Management Workshops*, volume 132 of *Lecture Notes in Business Information Processing*, pages 65–76. Springer, 2013.

[29] T. Slaats. *Flexible Process Notations for Cross-organizational Case Management Systems*. PhD thesis, IT University of Copenhagen, January 2015.

[30] W. M. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *WS-FM '06*, volume 4184 of *LNCS*, pages 1–23. Springer, 2006.

[31] W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[32] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[33] S. Zugal, J. Pinggera, and B. Weber. Assessing Process Models with Cognitive Psychology. In *EMISA*, pages 177–182, 2011.

[34] S. Zugal, J. Pinggera, B. Weber, J. Mendling, and H. A. Reijers. Assessing the Impact of Hierarchy on Model - A Cognitive Perspective. In *EESSMod*, 2011.

[35] S. Zugal, P. Soffer, C. Haisjackl, J. Pinggera, M. Reichert, and B. Weber. Investigating expressiveness and understandability of hierarchy in declarative business process models. *Software & Systems Modeling*, June 2014.