Discovering Data-Aware Declarative Process Models from Event Logs

Fabrizio Maria Maggi[1], Marlon Dumas[1], Luciano García-Bañuelos[1], and Marco Montali[2]

[1] University of Tartu, Estonia
{f.m.maggi,marlon.dumas,luciano.garcia}@ut.ee
[2] KRDB Research Centre, Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

Abstract. A wealth of techniques are available to automatically discover business process models from event logs. However, the bulk of these techniques yield procedural process models that may be useful for detailed analysis, but do not necessarily provide a comprehensible picture of the process. Additionally, barring few exceptions, these techniques do not take into account data attributes associated to events in the log, which can otherwise provide valuable insights into the rules that govern the process. This paper contributes to filling these gaps by proposing a technique to automatically discover declarative process models that incorporate both control-flow dependencies and data conditions. The discovered models are conjunctions of first-order temporal logic expressions with an associated graphical representation (Declare notation). Importantly, the proposed technique discovers underspecified models capturing recurrent rules relating pairs of activities, as opposed to full specifications of process behavior – thus providing a summarized view of key rules governing the process. The proposed technique is validated on a real-life log of a cancer treatment process.

Keywords: Automated Process Discovery, Predicate Mining, Linear Temporal Logic, Declare.

1 Introduction

Business processes in modern organizations are generally supported and controlled by information systems. These systems usually record relevant events, such as messages and transactions, in the form of event logs. Process mining aims at exploiting these event logs in order to model and analyze the underlying processes. One of the most developed family of process mining techniques is automated process discovery. Automated process discovery aims at constructing a process model from an event log consisting of traces, such that each trace corresponds to one execution of the process. Each event in a trace consists as a minimum of an event class (i.e., the task to which the event corresponds) and generally a timestamp. In some cases, other information may be available such as the originator of the event (i.e., the performer of the task) as well as data produced by the event in the form of attribute-value pairs.

F. Daniel, J. Wang, and B. Weber (Eds.): BPM 2013, LNCS 8094, pp. 81–96, 2013.

The Process Mining Manifesto [9] argues that one of the open challenges in process mining is *to find a suitable representational bias (language) to visualize the resulting models*. The suitability of a language largely depends on the level of standardization and the environment of the process. Standardized processes in stable environments (e.g., a process for handling insurance claims) are characterized by low complexity of collaboration, coordination and decision making. In addition, they are highly predictable, meaning that it is feasible to determine the path that the process will follow. On the other hand, processes in dynamic environments are more complex and less predictable. They comprise a very large number of possible paths as process participants have considerable freedom in determining the next steps in the process (e.g., a doctor in a healthcare process).

As discussed in [24,20,23], procedural languages, such as BPMN, EPCs and Petri nets, are suitable for describing standardized processes in stable environments. Due to their predictability and low complexity, these processes can be described under a "closed world" assumption, meaning that it is feasible to explicitly represent all the allowed behavior of the process. In contrast, the use of procedural languages for describing processes in dynamic environments leads to complex and incomprehensible models. In this context, declarative process modeling languages are more appropriate [23]. Unlike their procedural counterparts, declarative models describe a process under an "open world" assumption, such that everything is allowed unless it is explicitly forbidden. Accordingly, a declarative model focuses on capturing commitments and prohibitions that describe what must or must not occur in a given state of the process.

Previous work on automated discovery of declarative process models [16,14] has focused on mining control-flow dependencies, such as "the execution of a task entails that another task must eventually be executed". This prior work, as well as the bulk of process discovery techniques for procedural languages, ignores data attributes attached to events, besides the event class. Hence, the resulting models lack insights into the role of data in the execution of the process.

The importance of data in business processes, particularly dynamic ones, is paramount as it is often data that drives the decisions that participants make. In dynamic processes, the fact that a task A is executed often tells us little about what must or must not happen later. It is only when considering the data produced by task A and other data associated to the process that we can state that something must or must not happen later. This holds in particular for healthcare processes, which according to Rebuge et al. [21] involve numerous variables that determine how a specific patient should be treated (e.g., age, gender, type of disease).

This paper addresses the above gap by presenting a technique to discover data-aware declarative process models, represented using an extension of the *Declare* notation [17]. Declare is a declarative language that combines a formal semantics grounded in Linear Temporal Logic (LTL) on finite traces,[1] with a graphical representation. In essence, a Declare model is a collection of LTL rules, each capturing a control-flow dependency between two activities. Declare itself

[1] For compactness, we will use the LTL acronym to denote LTL on finite traces.

is not designed to capture data aspects of a process. Accordingly, for the sake of discovering data-aware models, we extend Declare with the ability to define data conditions (predicates). The extended (data-aware) Declare notation is defined in terms of LTL-FO (First-Order LTL) rules, each one capturing an association between a task, a condition and another task. An example of such rule is that if a task is executed and a certain data condition holds after this execution, some other task must eventually be performed.

The proposed approach relies on the notion of *constraint activation* [3]. For example, for the constraint "every request is eventually acknowledged" each request is an activation. This activation becomes a fulfillment or a violation depending on whether the request is followed by an acknowledgement or not. In our approach, we first generate a set of candidate constraints considering the constraints that are most frequently activated. Then, we apply an algorithm to replay the log and classify activations (with their data snapshots) into fulfillments and violations. Given the resulting classification problem, we use invariant discovery techniques to identify the data conditions that should hold for a constraint activation to be fulfilled.

The paper is structured as follows. Section 2 introduces the basic Declare notation as well as the techniques used to discover data conditions. Next, Section 3 introduces the proposed data-aware extension of Declare and the technique for automated discovery of data-aware Declare models. In Section 4, we validate our approach in a real-life scenario. Finally, Section 5 discusses related work and Section 6 concludes and spells out directions for future work.

2 Background

In this section, we introduce some background material needed to present our proposed approach. In Section 2.1, we give an overview of the Declare language and introduce the notion of activation, fulfillment and violation for a Declare constraint. We describe the data condition discovery technique we use in our discovery algorithm in Section 2.2.

2.1 Declare: Some Basic Notions

Declare is a declarative process modeling language first introduced by Pesic and van der Aalst in [18]. A Declare model is a set of constraints that must hold in conjunction during the process execution. Declare constraints are equipped with a graphical notation and an LTL semantics. Examples of Declare constraints are $\mathit{response}(A, B)$ (formally: $\Box(A \rightarrow \Diamond B)$), $\mathit{responded\ existence}(A, B)$ (formally: $\Diamond A \rightarrow \Diamond B$) and $\mathit{precedence}(A, B)$ (formally: $(\neg B \sqcup A) \lor \Box(\neg B)$). We refer the reader to [19] for a complete overview of the language.

Constraint $\mathit{response}(A, B)$ indicates that if A occurs, B must eventually follow. Therefore, this constraint is satisfied for traces such as $t_1 = \langle A, A, B, C \rangle$, $t_2 = \langle B, B, C, D \rangle$ and $t_3 = \langle A, B, C, B \rangle$, but not for $t_4 = \langle A, B, A, C \rangle$ because, in this case, the second A is not followed by a B.

Note that, in t_2, $response(A, B)$ is satisfied in a trivial way because A never occurs. In this case, we say that the constraint is *vacuously satisfied* [11]. In [3], the authors introduce the notion of *behavioral vacuity detection* according to which a constraint is non-vacuously satisfied in a trace when it is activated in that trace. An *constraint activation* in a trace is an event whose occurrence imposes, because of that constraint, some obligations on other events in the same trace. For example, A is an activation for $response(A, B)$ because the execution of A forces B to be executed eventually.

A constraint activation can be classified as a *fulfillment* or a *violation*. When a trace is perfectly compliant with respect to a constraint, every constraint activation in the trace leads to a fulfillment. Consider, again, constraint $response(A, B)$. In trace t_1, the constraint is activated and fulfilled twice, whereas, in trace t_3, the same constraint is activated and fulfilled only once. On the other hand, when a trace is not compliant with respect to a constraint, a constraint activation in the trace can lead to a fulfillment but also to a violation (and at least one activation leads to a violation). In trace t_4, for example, $response(A, B)$ is activated twice, but the first activation leads to a fulfillment (eventually B occurs) and the second activation leads to a violation (the target event class B does not occur eventually).

In [3], the authors define two metrics to measure the conformance of an event log with respect to a constraint in terms of violations and fulfillments, called *violation ratio* and *fulfillment ratio* of the constraint in the log. These metrics are valued 0 if the log contains no activations of the considered constraint. Otherwise, they are evaluated as the percentage of violations and fulfillments of the constraint over the total number of activations.

2.2 Discovery of Data Conditions

Given a set of Declare constraints extracted from an event log, a key step of the proposed technique is to generate a set of data-aware constraints, meaning constraints that incorporate conditions based on data attributes found in the logs. This problem can be mapped to a classification problem as follows. Given a Declare constraint and a set of traces, we can determine by "replaying" the log, the points in each trace of the log where the constraint is fulfilled or violated. In other words, we can construct a set of *trace snapshots* where the constraint is fulfilled and another set where the constraint is violated, where a snapshot is an assignment of values to each attribute appearing in the log (possibly including "null" values). Given these two sets, classification techniques, such as decision tree learning, can be used to discover a condition on the data attributes that discriminates between fulfillments and violations. The discovered condition is then used to enrich the initial (control-flow) Declare constraint.

A similar principle is used in ProM's Decision Miner [22] for the purpose of discovering conditions that can be associated to branches of a decision point of a business process model. ProM's Decision Miner applies decision tree learning to discover conditions consisting of atoms of the form 'variable op constant', where 'op' is a relational operator (e.g., $=$, $<$, or $>$). Given the capabilities of standard decision tree learning techniques, this approach does not allow us to discover expressions of the form ‘variable op variable’ or conditions involving linear combinations of variables. This limitation is lifted in our previous work [5], where we combine standard decision tree learning with a technique for the discovery of (likely) invariants from execution logs, i.e., Daikon [7]. Daikon allows us to discover invariants that hold true at a given point in a program, where a program point may be a method call, a field access or some other construction of the target programming language. The execution logs that serve as input to Daikon are commonly generated by instrumented code that monitors the program’s points of interest, but they can also come from other sources. Given such execution logs, Daikon discovers invariants consisting of linear expressions with up to three variables as well as expressions involving arrays.

The technique described in [5] uses Daikon as an oracle to discover conditions that, given a decision point (e.g., XOR-split), discriminates between the cases where one branch of the decision point is taken and those where the other branch is taken. In a nutshell, this technique works as follows: given a set of traces S, a process model M discovered from S and a task T in this process model, Daikon is used to discover invariants that hold true before each execution of task T. Given a decision point between a branch starting with task $T1$ and a branch starting with task $T2$, the invariants discovered for branch $T1$ and those discovered for branch $T2$ are combined in order to discover a conjunctive expression that discriminates between $T1$ and $T2$. In order to discover disjunctive expressions, decision tree learning is employed to first partition the observation instances where $T1$ (or $T2$) are executed into disjoint subsets. One conjunctive expression is then discovered for each subset.

In this paper, this technique is employed to discover conditions that discriminate between violations and fulfillments of a constraint as detailed in Section 3.2.

3 Discovering Data-Aware Declare Models

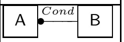
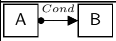
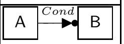
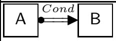
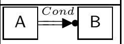
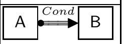
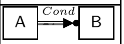
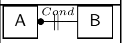
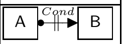
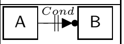
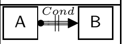
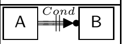
In this section, we first define a semantics to enrich Declare constraints with data conditions based on First-Order Linear Temporal Logic (LTL-FO). Then, we present an algorithm for discovering Declare models with data.

3.1 LTL-FO Semantics for Declare

We now define a semantics to extend the standard Declare constraints with data conditions. To do this, we use First-Order Linear Temporal Logic (LTL-FO), which is the first-order extension of propositional LTL. While many reasoning tasks are clearly undecidable for LTL-FO, this logic is appropriate to unambiguously describe the semantics of the data-aware Declare constraints we can generate by using our algorithm.

The defined semantics (shown in Table 1) is quite straightforward. In particular, the original LTL semantics of a Declare constraint is extended by requiring an additional condition on data, $Cond$, to hold when the constraint is activated.

Table 1. LTL-FO semantics and graphical representation for some Declare constraints extended with data conditions

constraint	description	formalization	notation
responded existence(A,B,Cond)	if A occurs and Cond holds, B must occur before or after A	$\Diamond(A \wedge Cond) \rightarrow \Diamond B$	
response(A,B,Cond)	if A occurs and Cond holds, B must occur afterwards	$\Box((A \wedge Cond) \rightarrow \Diamond B)$	
precedence(A,B,Cond)	if B occurs and Cond holds, A must have occurred before	$(\neg(B \wedge Cond) \sqcup A) \vee \Box(\neg(B \wedge Cond))$	
alternate response(A,B,Cond)	if A occurs and Cond holds, B must occur afterwards, without further As in between	$\Box((A \wedge Cond) \rightarrow \bigcirc(\neg A \sqcup B))$	
alternate precedence(A,B,Cond)	if B occurs and Cond holds, A must have occurred before, without other Bs in between	$((\neg(B \wedge Cond) \sqcup A) \vee \Box(\neg(B \wedge Cond)))$ $\wedge \Box((B \wedge Cond) \rightarrow \bigcirc(\neg B \sqcup A))$	
chain response(A,B,Cond)	if A occurs and Cond holds, B must occur next	$\Box((A \wedge Cond) \rightarrow \bigcirc B)$	
chain precedence(A,B,Cond)	if B occurs and Cond holds, A must have occurred immediately before	$\Box(\bigcirc(B \wedge Cond) \rightarrow A)$	
not resp. existence(A,B,Cond)	if A occurs and Cond holds, B can never occur	$\Diamond(A \wedge Cond) \rightarrow \neg\Diamond B$	
not response(A,B,Cond)	if A occurs and Cond holds, B cannot occur afterwards	$\Box((A \wedge Cond) \rightarrow \neg\Diamond B)$	
not precedence(A,B,Cond)	if B occurs and Cond holds, A cannot have occurred before	$\Box(A \rightarrow \neg\Diamond(B \wedge Cond))$	
not chain response(A,B,Cond)	if A occurs and Cond holds, B cannot be executed next	$\Box((A \wedge Cond) \rightarrow \neg \bigcirc B)$	
not chain precedence(A,B,Cond)	if B occurs and Cond holds, A cannot have occurred immediately before	$\Box(\bigcirc(B \wedge Cond) \rightarrow \neg A)$	

Cond is a closed first-order formula with the following structure: $\exists x_1, \ldots, x_n.\ curState(x_1, \ldots, x_n) \wedge \Phi(x_1, \ldots, x_n)$, where $curState/n$ is a relation storing the n data available in the system (considering both case attributes and event attributes in the log) and Φ/n is a first-order formula constraining such data by means of conjunctions, disjunctions and relational operators.

For example, $response(A, B, Cond)$ specifies that whenever A occurs and condition $Cond$ holds true, then a corresponding occurrence of B is expected to eventually happen. Constraint $precedence(A, B, Cond)$ indicates that whenever B occurs and $Cond$ holds, then an occurrence of A must have been executed beforehand. The semantics for negative relations is also very intuitive. For example, *not responded existence*$(A, B, Cond)$ indicates that if an instance of A occurs and $Cond$ holds, then no occurrence of B can happen before or after A. Note that some Declare constraints derive from the conjunction of other constraints. For example, the *succession* constraint is the conjunction of *response* and *precedence*. In this case, we have a condition on the attribute values of A and a condition on the attribute values of B. These two conditions can be, in principle, different.

Based on this semantics, the notion of constraint activation changes. Activations of data-aware Declare constraints are all those constraint activations (according to the standard definition) for which $Cond$ is true. For example, $response(A, B, Cond)$ is activated when A occurs and, also, $Cond$ is valid. On the other hand, $precedence(A, B, Cond)$ is activated when B occurs and $Cond$ is valid. The definitions of fulfillments and violations are also adapted accordingly.

3.2 Discovery Algorithm

In a nutshell, our approach aims at *discovering data-aware Declare constraints with fulfillment ratio close to 1 from an event log.* We thus start from event logs where the process execution traces and their events are equipped with data, modeled as attribute-value pairs.

More specifically, the algorithm takes as input an event log, which is a set of execution traces. Each execution trace represents, as usual, the sequence of events characterizing a specific instantiation of the process. Our focus is on *case data*, i.e., we consider data to be attached to the case and their values to be manipulated by the corresponding events. For this reason, a case can be associated to a set of key-value pairs defining the initial values for some of the data. These can be extracted by applying the $\mathsf{caseAtts}/1$ function to a trace. The other data mentioned in the events of the log are implicitly considered to have an initial null value.

Events are meant to manipulate such case data. Specifically, each event ev is associated to: *(i)* a *class* that represents the task to which the event refers to and that can be extracted with $\mathsf{evClass}(\mathsf{ev})$; *(ii)* a *timestamp*; *(iii)* a set of attribute-value pairs that denotes the impact of the event in terms of case data and that can be extracted with $\mathsf{evAtts}(\mathsf{ev})$. We follow the classical *commonsense law of inertia*: given a data attribute a, its value remains constant until it is explicitly overridden by an event that provides a new value for a.

The discovery of data-aware Declare constraints is based on a supervised learning approach. Before discussing the details of the algorithm, we introduce a short example that summarizes its key aspects. The algorithm requires the user to choose the constraint types she is interested in. In the following, we assume that *response* is selected. Consider an event log constituted by the following execution traces (we use triples to represent the events):

$$\begin{array}{l}
\{(A, 1, \{x = 1, y = 1\}),\ (B, 5, \{x = 2, y = 2\}),\ (C, 8, \{x = 3, y = 3\})\} \\
\{(A, 1, \{x = 1, y = 2\}),\ (B, 3, \{x = 1, y = 2\})\} \\
\{(A, 1, \{x = 2, y = 1\}),\ (C, 7, \{x = 2, y = 4\})\}
\end{array}$$

The event log contains three event classes: A, B and C. Therefore, in principle, all possible pairs of event classes could be involved in response constraints: response from A to B, from A to C, from B to A, from B to C, from C to A and from C to B. Among all these possibilities, only those that are "relevant" are considered to be candidate constraints. Relevance is measured in terms of number of activations, which, in the case of response, correspond to the execution of the source activity.

For example, response constraints with source A are activated once in each trace present in the log above, whereas response constraints with source B or

C are activated in only two traces out from three. Assuming to filter away those constraints with number of activations < 3, only response constraints with source A are kept. For each of those, the activations are classified as fulfillments or violations, depending on whether there is an event that refers to the target activity and occurs after it.

In the case of $response(A, B)$, the activations in the first two traces are marked as fulfilled, whereas the one for the third trace is not (in fact, no B is present in the third trace). This means that this constraint is not fully supported by the log. The classification of activations into fulfillments and violations is used as input of the approach discussed in Section 2.2. With this approach, we try to improve the support of a constraint by discovering finer-grained data conditions, used to restrict the context of application for the constraint. For example, we could learn that $response(A, B)$ is fully supported by the log whenever at the time A is executed, the value for attribute x is 1.

The full algorithm is shown in Fig. 1. It takes as input an event log, a set of constraint types userTypes previously selected by the user, a threshold minRatio representing the minimum expected fulfillment ratio for a constraint to be discovered and a threshold minActivations representing the minimum number of activations for a constraint to be considered as a candidate.

All the information needed for the discovery is collected by traversing the log twice. In the first iteration, the event classes and the (event and case) attributes with their types are collected (lines 2-9). To start the second iteration, we invoke function generateConstraints to generate the set of possible candidate constraints given the required minimum level of activation support, minActivations (line 10). This function produces all possible constraints of the form $Constr(A, B)$, where $Constr$ is one of the constraint types in userTypes and A and B are event classes in eClasses (the one corresponding to the constraint activation with at least minActivations occurrences).

In the second iteration (lines 14-28), we process each event in the log with a twofold purpose: constructing a snapshot that tracks the values of data obtained after the event execution and classifying constraint activations into fulfillments and violations. These two sources of information are used to select the final constraints and decorate them with data-aware conditions. In particular, when we start replaying a trace trace, we create a set state_0 of pairs (attribute,value), where each event/case attribute is firstly initialized to null (line 15) and each case attribute present in trace is then associated to the corresponding value (line 16). Given a trace/event x and an attribute a, we use function value(x,a) to extract the corresponding value. When an event occur at position p, the value of each event attribute is replaced by the new value attached to the event just occurred (through the update of curState, line 20), so as to reconstruct the effect of the event in terms of data values update. In this way, we associate each event occurring in trace at position p to snapshot[p], calculated by updating the previous state with the contribution of that event (line 21).

In parallel with the construction of snapshots, constraint activations are classified into fulfillments and violations. For every trace, each candidate constraint

```
 Algorithm Discovery
Input: log, an event log
       userTypes, a set of Declare constraint types
       minRatio, the minimum expected fulfillment ratio for a constraint to be discovered
       minActivations, the minimum number of activations for a constraint to be considered as a candidate
1: eClasses = $\emptyset$; cAtts = $\emptyset$; model = $\emptyset$; prunedModel = $\emptyset$;
2: for each trace in log do
3:     cAtts = cAtts $\cup$ caseAtts(trace);
4:     for each trace in log do
5:         for each ev in trace do
6:             cAtts = cAtts $\cup$ evAtts(ev); eClasses = eClasses $\cup$ evClass(ev);
7:         end
8:     end
9: end
10: constraints = generateConstraints(userTypes,eClasses,minActivations);
11: for each c in constraints do
12:     fulfSnapshots(c) = $\emptyset$; violSnapshots(c) = $\emptyset$;
13: end
14: for each trace in log do
15:     $\text{state}_0 = \{(a, null) \mid a \in \text{cAtts}\}$;
16:     for each a in caseAtts(trace) do $\text{state}_0 = (\text{state}_0 \setminus \{(a, null)\}) \cup \{(a, \text{value}(trace, a))\}$;
17:     curState = $\text{state}_0$;
18:     snapshot = new Array(length(trace));
19:     for (p=0; p < length(trace); p++) do
20:         for each a in evAtts(trace[p]) do curState = $(\text{curState} \setminus \{(a, \_)\}) \cup \{(a, \text{value}(trace[p], a))\}$;
21:         snapshot[p] = curState;
22:         for each c in constraints do classifyActivations(candidate, id(trace), p);
23:     end
24:     for each c in constraints do
25:         for each fp in getFulfPositions(c) do fulfSnapshots(c) = fulfSnapshots(c) $\cup$ snapshot[fp];
26:         for each vp in getViolPositions(c) do violSnapshots(c) = violSnapshots(c) $\cup$ snapshot[vp];
27:     end
28: end
29: for each c in constraints do
30:     if ($\min\{|\text{fulfSnapshots}(c)|, |\text{violSnapshots}(c)|\} \geq 10 \times |\text{cAtts}|$)
31:       dataCondition= callDaikon(fulfSnapshots(c),violSnapshots(c));
32:       model= model $\cup$ (c, dataCondition);
33:     end
34: end
35: for each c in model do complianceCount(c) = 0;
36: for each trace in log do
37:     for each c in model do
38:         if (checkCompliance(trace, c)) complianceCount(c)++;
39:     end
40: end
41: for each c in model do
42:     if ($\frac{\text{complianceCount}(c)}{|\text{log}|} \geq$ minRatio) prunedModel = prunedModel $\cup$ c; 42
43: end
44: return prunedModel;
```

Fig. 1. Discovery algorithm for data-aware Declare

is associated to a set of activations. Internally, every activation is a quadruple (candidate, id(trace), p, curState) indicating that in position p of the trace identified by id(trace), an event occurs activating constraint candidate and that snapshot(id(trace), p) = curState in the same position. These quadruples are classified into fulfillments and violations by leveraging on function classifyActivations (line 22). This function depends on the constraint type.

In particular, there is a difference when we are processing an event for a constraint looking at the past (e.g., *precedence*) and for constraints looking at the future (e.g., *response*). For constraints looking at the past, we store each scanned event as possible target in a sorted list. The same event will be an activation for some candidate constraints. In particular, it will be a fulfillment if the list of the events already occurred contains a possible target and a violation if the list does not contain such an event. For constraints looking at the future, we process an event by considering it as a "pending" activation waiting for a possible target to be classified as a fulfillment. The same event can be, on the other hand, a target for a pending activation. All the activations that are still pending when the trace has been completely replayed are classified as violations (indeed, no further events can occur to fulfill them). Note that undirected constraints (e.g., *responded existence*) use an hybrid approach. Furthermore, for each negative constraint the same algorithm used for the corresponding positive constraint is adopted, by substituting fulfillments with violations and vice-versa.

As an example, consider constraint (response, A, B). Activation ((response,A,B), 123, 4, curState) is added to the list of pending activations whenever in trace 123 at position 4, activity A is executed. This activation is pending, since it expects a consequent execution of B. If B occurs in 123 at a later position, say, 12, then the activation at position 4 is classified as a fulfillment. On the other hand, if we evaluate constraint (not response, A, B) on the same trace, ((not response,A,B), 123, 4, curState) would be classified as a violation (indeed, *not response* would forbid the presence of B after A).

When the processing of a trace is completed, the aforementioned functions have calculated, for each constraint c, the set of positions at which an activation for c was classified as a fulfillment or as a violation. These two sets can then be retrieved by respectively calling function getFulfPositions(c) and getViolPositions(c). Starting from these positions, we can in turn obtain the corresponding snapshots, globally accumulating them into two sets fulfSnapshots(c) and violSnapshots(c) (lines 24-27).

With the information collected in fulfSnapshots(c) and violSnapshots(c), we proceed with the discovery of data-aware conditions using the approach discussed in Section 2.2 (lines 29-34). It is well known that the quality of decision trees is sensible to the amount of the observations for each class being considered and so is the method used for discovering data conditions. To filter cases with not enough observations, we use a common heuristic as described in [10]. According to this heuristic, the number of samples for classifier learning should be at least 10 times the number of features. Hence, we filter out candidate constraints that have a number of fulfillments and a number of violations (i.e., number of positive and negative samples) lower than 10 times the number of attributes in the log.

Finally, the resulting data-aware Declare model can be further pruned by means of threshold minRatio, i.e., the minimum expected fulfillment ratio for a discovered data-aware constraint. Function checkCompliance/2 is called to check whether the aforementioned ratio is above minRatio or not. If so, the constraint is maintained in the final model and discarded otherwise (lines 41-43).

Table 2. Discovered response constraints

A	B	data condition
Milk acid dehydrogenase LDH kinetic	squamous cell carcinoma using eia	(((Diagnosis code == “M13”) || (Diagnosis code == “822”)) || (Diagnosis code == “M12”))
First outpatient consultation	teletherapy - megavolt photons bestrali	(((org:group == “Radiotherapy”) || (Treatment code == “113”)) || ((Diagnosis == “Gynaecologische tumoren”) || (Diagnosis == “Maligne neoplasma cervix uteri”) || (Diagnosis == “maligniteit cervix”)))
bilirubin-total	squamous cell carcinoma using eia	((Diagnosis code == “M13”) || (Diagnosis code == “822”))
gammaglutamyl-transpeptidase	squamous cell carcinoma using eia	(((Diagnosis code == “M13”) || (Diagnosis code == “822”)) || (Diagnosis code == “M12”))
unconjugated bilirubin	squamous cell carcinoma using eia	(((Diagnosis code == “M13”) || (Diagnosis code == “822”))|| (Diagnosis code == “M12”))
outpatient follow-up consultation	differential count automatically	(Specialism code == “13”)
CEA - tumor marker using meia	squamous cell carcinoma using eia	(((((Diagnosis == “Maligne neoplasma cervix uteri”) || (Diagnosis == “maligniteit cervix”)) || (Diagnosis code == “M13”)) || ((Diagnosis == “Plaveiselcelca. vagina st II”) || (Diagnosis == “maligniteit vagina”))) || ((Diagnosis == “Plav.celcarc. vulva: st II”) || (Diagnosis == “maligne melanoom van de vulva”)))

Table 3. No. of activations, fulfillments and fulfillment ratio (response)

A	B	# activ no data	# activ data	# fulf. no data	# fulf. data	fulf. ratio no data	fulf. ratio data
Milk acid dehydrogenase LDH kinetic	squamous cell carcinoma using eia	1282	**474**	420	**315**	0.32	**0.66**
First outpatient consultation	teletherapy - megavolt photons bestrali	1200	**646**	530	**452**	0.44	**0.69**
bilirubin-total	squamous cell carcinoma using eia	1253	**499**	419	**321**	0.33	**0.64**
gammaglutamyl-transpeptidase	squamous cell carcinoma using eia	1442	**595**	479	**372**	0.33	**0.62**
unconjugated bilirubin	squamous cell carcinoma using eia	967	**406**	361	**284**	0.37	**0.69**
outpatient follow-up consultation	differential count automatically	6860	**2575**	2096	**1345**	0.30	**0.52**
CEA - tumor marker using meia	squamous cell carcinoma using eia	465	**132**	145	**103**	0.31	**0.78**

Table 4. Discovered not response constraints

A	B	data condition
rhesus factor d - Centrifuge method - email	ABO blood group antigens other than rhesu	(Age >= 46)
rhesus factor d - Centrifuge method - email	cde phenotyping	(Age >= 46)
Milk acid dehydrogenase LDH kinetic	teletherapy - megavolt photons bestrali	((((((Diagnosis code == “M16”) || (Diagnosis code == “821”)) || ((Diagnosis == “Maligne neoplasma adnexa uteri”) || (Diagnosis == “Maligne neoplasma vulva”) || (Diagnosis == “maligniteit vulva”))) || (Diagnosis code == “823”)) || ((Diagnosis == “Plaveiselcelca. vagina st II”) || (Diagnosis == “maligniteit vagina”))) || (Diagnosis code == “M11”))
bilirubin - total	teletherapy - megavolt photons bestrali	((((((Diagnosis code == “M16”) || (Diagnosis code == “821”)) || ((Diagnosis == “Maligne neoplasma adnexa uteri”) || (Diagnosis == “Maligne neoplasma vulva”) || (Diagnosis == “maligniteit vulva”))) || (Diagnosis code == “823”)) || (Diagnosis code == “M11”)) || (Diagnosis == “maligniteit myometrium”))
unconjugated bilirubin	teletherapy - megavolt photons bestrali	((((((Diagnosis code == “M16”) || (Diagnosis code == “821”)) || (Diagnosis code == “M11”)) || ((Diagnosis code == “M13”) && (Diagnosis == “maligniteit cervix”))) || (Diagnosis code == “839”)) || (Treatment code == “503”))
alkaline phosphatase-kinetic-	teletherapy - megavolt photons bestrali	(((((Diagnosis code == “M16”) || (Diagnosis code == “821”)) || ((Diagnosis == “Maligne neoplasma adnexa uteri”) || (Diagnosis == “Maligne neoplasma vulva”) || (Diagnosis == “maligniteit vulva”))) || (Diagnosis code == “823”)) || (Diagnosis code == “M11”))
ABO blood group and rhesus factor	ABO blood group antigens other than rhesu	(Age >= 46)
ABO blood group and rhesus factor	cde phenotyping	(Age >= 46)

Table 5. No. of activations, fulfillments and fulfillment ratio (not response)

A	B	# activ no data	# activ data	# fulf. no data	# fulf. data	fulf. ratio no data	fulf. ratio data
rhesus factor d - Centrifuge method - email	ABO blood group antigens other than rhesu	1558	**1071**	1271	**1041**	0.81	**0.97**
rhesus factor d - Centrifuge method - email	cde phenotyping	1558	**1071**	1273	**1043**	0.81	**0.97**
Milk acid dehydrogenase LDH kinetic	teletherapy - megavolt photons bestrali	1191	**541**	908	**528**	0.76	**0.97**
bilirubin - total	teletherapy - megavolt photons bestrali	1166	**518**	880	**504**	0.75	**0.97**
unconjugated bilirubin	teletherapy - megavolt photons bestrali	909	**457**	676	**441**	0.74	**0.96**
alkaline phosphatase-kinetic-	teletherapy - megavolt photons bestrali	1326	**557**	1001	**544**	0.75	**0.97**
ABO blood group and rhesus factor	ABO blood group antigens other than rhesu	1558	**1071**	1271	**1041**	0.81	**0.97**
ABO blood group and rhesus factor	cde phenotyping	1558	**1071**	1273	**1043**	0.81	**0.97**

4 Validation

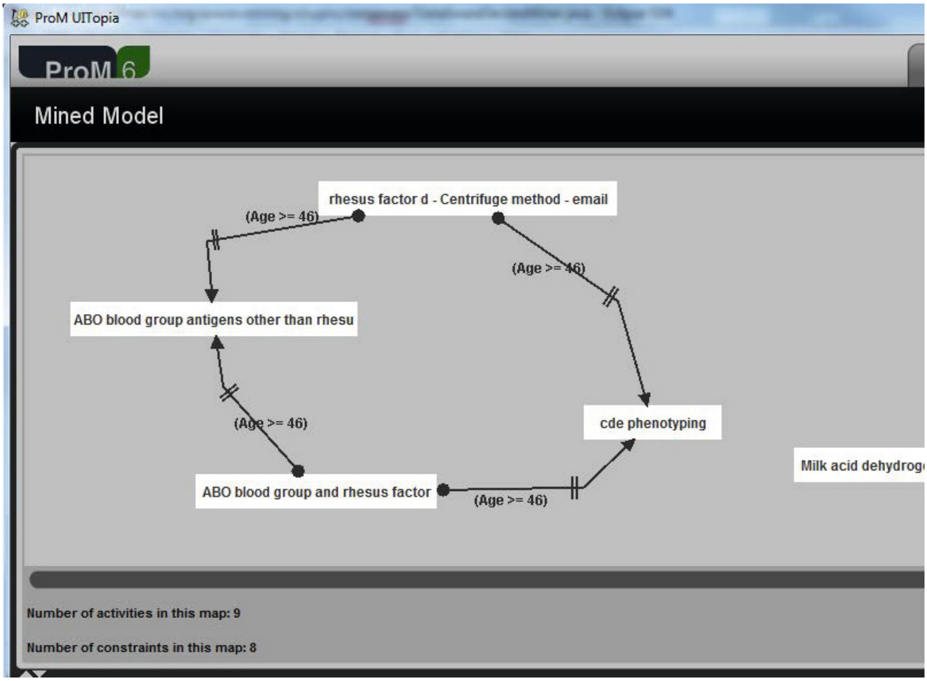
We implemented the approach as a plug-in of the process mining tool ProM.[2] As a proof of concept, we validated the approach with the event log used in the BPI challenge 2011 [1] that records the treatment of patients diagnosed with cancer from a large Dutch hospital. The event log contains 1143 cases and 150,291 events distributed across 623 event classes. Moreover, the event log contains a total of 13 domain specific attributes, e.g., Age, Diagnosis Code, Treatment code, in addition to the standard XES attributes, i.e., concept:name, lifecycle:transition, time:timestamp and org:group. In our experiments, we take into consideration only the domain specific attributes.

In a first experiment,[3] we discovered data-aware *response* constraints from the event log, with a fulfillment ratio of at least 0.5. Since the log contains 13 data attributes, the candidate constraints must have at least 130 fulfillments and 130 violations (i.e., 10 times the number of attributes, as explained in Section 3.2). The execution time for this experiment was 9.6 minutes for the first traversal of the log (gathering of data snapshots, fulfillments and violations for each candidate constraint) and 15.3 minutes for the discovery of data-aware conditions. The constraints discovered are summarized in Table 2.

In Table 3, we compare the number of activations and fulfillments for the discovered constraints, first without considering the data conditions and then considering the data conditions (in bold). As expected, both the number of activations and the number of fulfillments decrease when the data conditions are considered. However, the decrease in the number of fulfillments is less pronounced than the decrease in the number of activations. If we interpret the fulfillment ratio as a measure of goodness of a constraint, we obtain better results when considering the data conditions (see the last two columns of Table 3).

[2] `www.processmining.org`

[3] The experiments were performed on a standard, 2.6 GHz dual-core processor laptop.


ProM UITopia
ProM 6
Mined Model
rhesus factor d - Centrifuge method - email
(Age >= 46)
(Age >= 46)
ABO blood group antigens other than rhesu
(Age >= 46)
cde phenotyping
Milk acid dehydrog
ABO blood group and rhesus factor
(Age >= 46)

Number of activities in this map: 9

Number of constraints in this map: 8

Fig. 2. Some of the discovered not response constraints in ProM

In a second experiment, we considered the discovery of *not response* constraints. It is worth noting that negative constraints are interesting because they specify forbidden scenarios that usually result in extremely complex representations when using procedural modeling languages. For this experiment, we decided to discover data-aware *not response* constraints with a fulfillment ratio of at least 0.95. The execution time for this experiment was 13.3 minutes for the first traversal of the log (to collect data snapshots and fulfillments and violations for each candidate constraint) and 14.1 minutes for the discovery of data conditions. The *not response* constraints discovered are summarized in Table 4. Interestingly, in this experiment we discovered more complex data conditions. For instance, the *not response* constraint between *unconjugated bilirubin* and *teletherapy - megavolt photons bestrali* has a data condition associated with a combination of conjunctions and disjunctions.

In Table 5, we compare the number of activations and the number of fulfillments for the constraints discovered in the second experiment. Similarly to the results obtained in the first experiment, we can clearly observe a lift in the fulfillment ratio when the data conditions are considered. In Fig. 2, we present a screenshot of ProM with the data-aware Declare model discovered in the second experiment. For example, the *not response* constraint between *rhesus factor d - Centrifuge method - email* and *ABO blood group antigens other than rhesu* indicates that, if the age of the patient is greater than or equal to 46, when *rhesus factor d - Centrifuge method - email* occurs, then *ABO blood group antigens other than rhesu* can no longer occur.

5 Related Work

Several algorithms have been proposed to discover declarative process models. Some of these algorithms [12,8,4] assume that every trace in the input log is labeled as a "positive" or a "negative" case, where a negative case is one that should not occur. The problem of mining a declarative model is mapped to one of discriminating between positive and negative cases. The assumption of a pre-existing labeling of positive and negative cases enables the separation of constraint fulfillments and violations. However, this assumption often does not hold as negative cases are generally not explicitly present in a real-life event log. In [16,14], LTL model checking techniques are used to classify negative and positive cases (i.e., constraint violations and fulfillments), thus avoiding the need for a preprocessing step to explicitly label the traces. The approach presented in this paper extends the one in [16,14] by using data attributes in order to enrich candidate control-flow constraints with data conditions. We have shown in the case study that this enrichment leads to constraints with higher fulfillment ratio.

The work reported in [6] provides an alternative approach to declarative process mining that does not assume explicit labeling of positive and negative cases. In this approach, each Declare constraint is mapped to a regular expression. The regular expressions are used to generate a set of matrices of fulfillments and these matrices are used to generate a Declare model. It would be worth investigating the combination of this approach with our data enrichment algorithm. To this end, the approach in [6] would first have to be extended to reconstruct the constraint activations and the corresponding fulfillments and violations.

Automated discovery of behavioral models enhanced with data conditions has been addressed recently in [13,22,5]. In [13], a technique is presented to mine finite state machines extended with data. This work builds on top of a well-known technique to mine finite state machines that incrementally merges states based on automata equivalence notions (e.g., trace equivalence). However, this approach is not suitable for discovering business process models, as automata do not capture concurrency and concurrency is common in business processes. ProM's decision miner [22] embodies a technique to discover data-aware procedural process models, based on existing techniques for discovering "control-flow" process models (e.g., Petri nets) and decision trees. [5] extends ProM's decision miner in order to discover more general conditions as discussed in Section 2.2.

6 Conclusion and Future Work

This paper has presented a technique to automatically discover data-aware declarative models consisting of LTL-FO rules from event logs. A validation on real-life logs from a cancer treatment process demonstrates that the technique can discover more precise rules (higher fulfillment ratio) compared to a technique for discovering declarative models without data conditions.

As future work, we will carry out a more extensive experimentation with new datasets. Furthermore, some optimizations of the presented technique are warranted. For example, it may be possible to prune the discovered models through transitive reduction. In [15], the authors use an algorithm for transitive reduction of cyclic graphs to prune a Declare model discovered from a log. This approach, however, can be used when the model only includes Declare constraints without data conditions. For data-aware Declare models different reduction algorithms should be used. For example, approaches for transitive reduction of weighted graphs like the one presented in [2] could be adopted.

Another avenue for future work is to optimize the performance of the proposed technique, for example by reducing the number of invocations made to Daikon. This could be achieved by caching some of the invariants discovered by Daikon for a given constraint and reusing them for other constraints. Such optimization should be based however on a case-by-case analysis of which invariants can be reused for a given constraint type.

Finally, we plan to extend the technique so that it can discover a larger set of LTL-FO rule templates such as the existence templates and the non-binary relation templates in Declare as well as templates beyond the standard set included in Declare.

Acknowledgment. This research is supported by the EU's FP7 Programme (ACSI Project).

References

1. 3TU Data Center. BPI Challenge, Event Log (2011), doi: 10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54
2. Bonaki, D., Odenbrett, M.R., Wijs, A., Ligtenberg, W.P.A., Hilbers, P.A.J.: Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors. BMC Bioinformatics 13, 281 (2012)
3. Burattin, A., Maggi, F.M., van der Aalst, W.M.P., Sperduti, A.: Techniques for a Posteriori Analysis of Declarative Processes. In: EDOC, pp. 41–50 (2012)
4. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 278–295. Springer, Heidelberg (2009)
5. de Leoni, M., Dumas, M., García-Bañuelos, L.: Discovering Branching Conditions from Business Process Execution Logs. In: Cortellessa, V., Varró, D. (eds.) FASE 2013 (ETAPS 2013). LNCS, vol. 7793, pp. 114–129. Springer, Heidelberg (2013)
6. Di Ciccio, C., Mecella, M.: Mining constraints for artful processes. In: Abramowicz, W., Kriksciuniene, D., Sakalauskas, V. (eds.) BIS 2012. LNBIP, vol. 117, pp. 11–23. Springer, Heidelberg (2012)
7. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Software Eng. 27(2), 99–123 (2001)
8. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust process discovery with artificial negative events. JMLR 10, 1305–1340 (2009)

9. IEEE Task Force on Process Mining. Process Mining Manifesto. In: Algebraic Semantics. LNBIP, vol. 99, pp. 169–194. Springer (2011)
10. Jain, A.K., Duin, R.P.W., Mao, J.: Statistical pattern recognition: A review. IEEE Trans. on Pattern Analysis and Machine Intelligence 22(1), 4–37 (2000)
11. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. Int. Journal on Software Tools for Technology Transfer, 224–233 (2003)
12. Lamma, E., Mello, P., Riguzzi, F., Storari, S.: Applying Inductive Logic Programming to Process Mining. In: Blockeel, H., Ramon, J., Shavlik, J., Tadepalli, P. (eds.) ILP 2007. LNCS (LNAI), vol. 4894, pp. 132–146. Springer, Heidelberg (2008)
13. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proc. of ICSE, pp. 501–510. IEEE (2008)
14. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: Efficient discovery of understandable declarative models from event logs. In: Ralyté, J., Franch, X., Brinkkemper, S., Wrycza, S. (eds.) CAiSE 2012. LNCS, vol. 7328, pp. 270–285. Springer, Heidelberg (2012)
15. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: A knowledge-based integrated approach for discovering and repairing declare maps. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 433–448. Springer, Heidelberg (2013)
16. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: Proc. of CIDM, pp. 192–199. IEEE (2011)
17. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. In: Proc. of EDOC, pp. 287–300. IEEE (2007)
18. Pesic, M., van der Aalst, W.M.P.: A Declarative Approach for Flexible Business Processes Management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
19. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Controls to Users. PhD thesis, Beta Research School for Operations Management and Logistics, Eindhoven (2008)
20. Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., Reijers, H.A.: Imperative versus declarative process modeling languages: An empirical investigation. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM Workshops 2011, Part I. LNBIP, vol. 99, pp. 383–394. Springer, Heidelberg (2012)
21. Rebuge, A., Ferreira, D.R.: Business process analysis in healthcare environments: A methodology based on process mining. Inf. Syst. 37(2), 99–116 (2012)
22. Rozinat, A., van der Aalst, W.M.P.: Decision mining in ProM. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 420–425. Springer, Heidelberg (2006)
23. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science - R&D, 99–113 (2009)
24. Zugal, S., Pinggera, J., Weber, B.: The impact of testcases on the maintainability of declarative process models. In: BMMDS/EMMSAD, pp. 163–177 (2011)