A Meta-model for Operational Support

Joyce Nakatumba, Michael Westergaard[⋆], and Wil M.P. van der Aalst

Eindhoven University of Technology, The Netherlands
{jnakatum,m.westergaard,w.m.p.v.d.aalst}@tue.nl

Abstract. Process mining techniques can be used to extract information from event logs helping organizations to gain insights into the operation of their business processes. It is now possible to provide such analysis information for ongoing executions in an on-line setting, yielding predictions about the current execution or recommendations of actions to take, commonly referred to as *operational support*. This paper defines a *meta-model for operational support* making it possible for users or tools to receive operational support without detailed knowledge of the algorithm providing answers. The meta-model defines four types of queries: A *simple* query asks diagnostic information about the current execution and a *compare* query compares the current execution with other similar executions. A *predict* query considers the future of previous executions similar to the current one to make a prediction about its future. A *recommend* query recommends the best next action to take based on predictions. We formally define our meta-model and take care to provide implementation suggestions for each defined concept. We present our implementation of the meta-model in the workflow system Declare and the process mining framework ProM.

Keywords: Process mining, operational support meta-model, prediction, recommendation, ProM, Declare.

1 Introduction

More and more business processes are being supported by *Process-Aware Information Systems* (PAISs) [6], i.e., software systems that manage and execute these processes. PAISs record information about the different processes they support in *event logs*. An event log is a set of *trace*s and each trace consists of *events* from start to end of the trace execution. Event logs provide an excellent source of information useful for *process mining* [1]. Process mining is a technique used to extract non-trivial and useful information from event logs. Hence using process mining necessary insights can be gained to manage, control and improve business processes.

Many existing process mining techniques work in an off-line setting, where the past executions of traces are analyzed without considering the current running

[⋆] This research is supported by the Technology Foundation STW, applied science division of NWO and the technology program of the Dutch Ministry of Economic Affairs.

traces in a workflow system, i.e, only traces that have completed are considered. It is also possible to use process mining in an on-line setting where support is provided for traces that are still running in a workflow system. We refer to this as *operational support* [2]. During operational support, a client sends a *partial execution trace* along with a *query* to the process mining framework. A query is simply a question to which a *response* is received.

Work in operational support usually focuses on either predicting the future of a current execution or on providing recommendations. Such predictions or recommendations are typically very specialized and a user has to interact with different systems in order to make use of different algorithms. In [16] we presented a protocol and architecture making it possible to access different algorithms using a common protocol and in this paper we propose a meta-model making it possible to issue queries without any knowledge of the algorithm providing answers. To do that, we introduce four main queries that can be handled under operational support. A *simple* query checks the current performance of the current partial execution trace, for example, the total execution time. A *compare* query compares the performance of the current partial trace to other similar traces. For example, is the execution time of the current trace to this point higher or lower than the average. A *predict* query considers the future of traces similar to the current and uses that to provide predictions about the current trace. For example, what is the expected total execution time for this trace. Finally, a *recommend* query gives the best possible next action to be done based on the current partial trace. For example, what is the best action to execute in order to complete the execution as fast as possible.

The infrastructure for operational support shown in Fig. 1 is implemented in ProM[1]. A *Client* communicates with a *Workflow System*, and an operational support service (*OS Service*; OSS in the following). The *Client* sends one of the four queries to OSS, which forwards it to a number of operational support providers (*OS providers*; providers in the following), which may implement different algorithms. Responses are sent back to the OSS and forwarded to the *Client*.

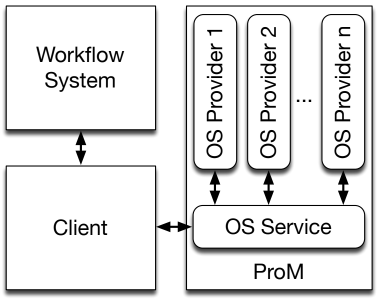


Fig. 1: Architecture of the operational support in ProM.

The main contribution of this paper is that we provide a generic way to describe operational support in terms for four kinds of queries. This approach uses the information in a partial execution trace and a model of the running business process to provide support to the user. We define our operational support meta-model formally, but also take care to provide concrete implementation suggestions in each case. We base our approach entirely on open standards, including XML, XES, and XQuery. The ideas in this paper have been implemented

[1] ProM is an extensible process mining framework . See `www.promtools.org` for more information.

in Declare[2] and ProM. We provide examples showing how the four queries are implemented and handled in Declare and ProM.

The remainder of the paper is organized as follows. First, we introduce preliminaries in Sect. 3. Sect. 2 discusses related work. In Sect. 4, we describe and formally define the meta-model for operational support. Sect. 5 discusses the implementation in the ProM framework and the Declare workflow system. Finally, Sect. 6 concludes the paper, and provides directions for future work.

2 Related Work

Work related to operational support focuses on either providing predictions or recommendations about the future based on a current execution. One example is the prediction engine of Staffware [13] which uses simulation to complete audit trails with expected information about future steps. This approach is however unreliable, since it is based on one run through the system using a copy of the actual engine and it does not provide a means of learning to make better predictions over time. A more refined approach focusing on transient behavior is presented in [11]. It supports operational decision making using process mining techniques and simulation in the context of YAWL. In [2] a concrete approach to operational support using process mining is given. Here, process mining is used in an active way to check the performance of cases that have not completed, predict the future from the current execution, and provide recommendations about the next steps to take in order to achieve a certain goal. This is supported by learning a transition system annotated with time information.

In the context of the world wide web, there are a number of approaches to run time support. Examples include the monitoring based on business rules [8], event calculus [9] etc. Further on, there are various recommender systems that support users in decision making [10]. These systems generate recommendations based on the user's preferences and are becoming an essential part of e-commerce and information seeking activities. In [12] a recommendation engine implemented in ProM is presented. It learns historic information from event logs for guiding a user about the next work item to select. Similarly, the authors in [7] extend the recommendation strategies introduced in [12] with additional ones and also study the effect of log quality on recommendation quality. This work is also related to the case based reasoning approach presented in [15] where a prototype CBRFlow is presented. This prototype is able to adapt a process model to changing situations at run-time and provide the workflow system with learning capabilities. Recommendations can also be based on a Product Data Model as discussed in [14] but these are specifically for product based workflows. The authors in [3] propose a recommendation system based on a constraint-based approach extended to consider not only the control-flow, but also the resource perspective in order to optimize performance goals of business processes. In [5], a self-adjusting approach for building context-sensitive recommendations on the

[2] Declare is a workflow system based on declarative workflow modeling and LTL. See more at `declare.sf.net`.

most suitable next steps based on user behavior analysis, crowd processes, and its application to process detection, is proposed.

3 Preliminaries

In order to understand the operation of the operational support service, we first introduce a sample process we use in the remainder of this paper. The model in made in Declare, but our approach is independent of the modeling language used. To describe our operational support implementation, we introduce a concrete representation of event logs and queries. For logs, we use the *eXtensible Event Stream* (XES) format, an XML-based standard for event logs[3]. As XES is based on XML, we find it natural to use the XML query language XQuery [4] to represent queries.

Running Example. As a running example, we use a simple process that models the life of a student. The underling idea is that a young person should get an education. Furthermore, a person is only young once and can only get one MSc degree. Finally, in order to become a true Master of Business Process Modeling, one needs an MSc degree in BIS. This process model is shown in Fig. 2 and is expressed in the *Declare* language. It consists of a number of tasks and constraints. Tasks are shown as rectangles and represent actions performed by a user, e.g., being *Young*, and constraints are shown as arcs between tasks or as annotations of tasks. We will not go into details of the Declare language, but only mention that the *0. . . 1* annotation of *Young* models the requirement that a person is only young once, the hyper-arc from *Young* to the three degrees models the constraint that if a person is young, they should get an education (at least one of the three). The arc between the two *MSc* degrees indicates that if one is present in an execution, the other cannot be, and, finally, the arc from *MSc,BIS* to *Master of BPM* models that only after completing a *MSc,BIS* can you become a true *Master of BPM.*

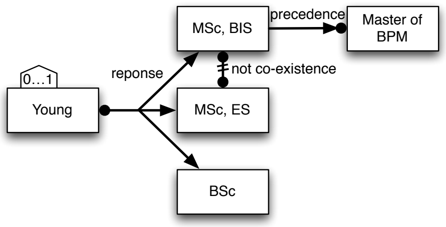


Fig. 2: Simple study process model.

The XES Log Format. Listing 1 shows an example of a partial trace in the XES format. The trace obtained by executing the model shown in Fig. 2. XES is an XML-based standard for event logs and has a reference implementation named OpenXES. The basic hierarchy of XES consists of one *log* object at the top that contains all the event information related to a specific process (e.g., the study process). The log contains a number of *trace* objects, each trace describing

[3] Its full description and a reference implementation is available at `www.xes-standard.org`.

the execution of one specific instance (case or execution) of the logged process (e.g., a specific student belonging to the study program). A trace contains a number of *event* objects and each event corresponds to an activity in the business process (e.g., *Young*).

The trace and event objects have attributes describing them. Attributes are described in *extensions* of the XES standard. In Listing 1, one trace object is annotated by *concept:name* defined by the *Concept* extension. The value of this attribute for the trace is *Case1*. The attribute *concept:name* can be used to name both traces and events. The name of the first event in Listing 1 is *Young*. The next attribute is defined by the *Lifecycle* extension which specifies the type of the event as a lifecycle transition. An event may refer to the start, completion, cancellation, etc. of an activity. Here, the transition recorded is *complete.* Other attributes are defined in various extensions as well. In Listing 1, we also see the *org:resource*, describing which user (or other resource) executed the event, the *time:timestamp* describing when the event was executed, and the non-standard attribute *Sex*, describing the sex of the resource.

XQuery. XQuery is query language for XML data [4] similar to SQL. XQuery uses *FLWOR* expressions that to select and perform computations on nodes of the XML tree. FLWOR expressions are constructed from five clauses after which it was named. The *For* clause iterates through a sequence of nodes and the *Let* clause can make computations for each element iterated over. The *Where* clause filters the values, retaining only those satisfying desired conditions, the *Order by* clause sorts the values iterated over, and the *Return* clause builds the result of the entire expression for each value of the sequence. XQuery has a number of built-in functions used for string values, numeric values, date-time comparison, node, sequence and boolean manipulation, and can also support user defined functions.

An example of an XQuery FLWOR expression is shown in Listing 4 (a full description of this query is given in Sect. 5). This query returns the service times of the executed events from a partial trace like the one shown in Listing 1.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="1.0" xmlns="http://www.xes-standard.org">
  <trace>
  <string key="concept:name" value="Case1"/>
    <event>
      <string key="concept:name" value="Young"/>
      <string key="lifecycle:transition" value="complete"/>
      <string key="org:resource" value="user1"/>
      <date key="time:timestamp" value="2004-10-04T08:05:00.000+02:00"/>
      <string key="Sex" value="Male"/>
    </event>
     ...
  </trace>
   ...
</log>
```

Listing 1: Part of a partial trace expressed in XES format.

4 A Meta-Model for Operational Support

In the operational support setting assumed in this paper, a client sends a partial trace and a query to the operational support service. The OSS handles four main kinds of queries: *simple*, *compare*, *predict* and *recommend* as shown in Fig. 3. A simple query (a) only looks at the current trace, shown by the wide arrow. A compare query (b) also looks at similar prefixes at the same position of the execution, as shown by the narrower arrows of the same length as the current trace. A predict query (c) not only looks at prefixes of similar traces, but the entire traces, as indicated by the longer lines with no arrowhead (to indicate they are not partial prefixes of traces, but finished executions). Finally, a recommend query (d) expands the current trace with all possible events (A and B), and makes predictions for all such expansions, shown by the lines expanding each of the two possible extensions, and recommends the expansion yielding the better prediction. In this section, we formally define these four queries and their responses. We do this by first giving textual examples in the context of our running example from Fig. 2, then introducing the formal definition, and finally exemplifying the formal definition. In the next section, we turn to discussing implementation details, including providing examples in our implementation of all concepts and examples given in this section.

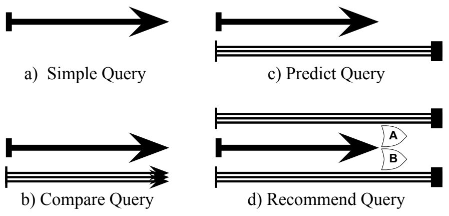


Fig. 3: Operational Support actions.

First we need to define a trace and a model. One can think of a trace as a trace in XES format. Formally, a trace is a string of executed events:

Definition 1 (Trace). *For an alphabet Σ, a* ***trace*** *τ over Σ is a finite string over Σ, i.e., $\tau \in \Sigma^*$.*

We use the notation $\tau = e_0 e_1 \ldots e_{n-1}$, $|\tau| = n$, $\tau(i) = e_i$ (for $0 \leq i < |\tau|$), and $\tau = \tau_1\tau_2$ if $\tau(i) = \tau_1(i)$ (for $0 \leq i < |\tau_1|$), $\tau(i + |\tau_1|) = \tau_2(i)$ (for $0 \leq i < |\tau_2|$), $|\tau| = |\tau_1| + |\tau_2|$. The empty trace is ε (i.e., $|\varepsilon| = 0$).

To make our meta-model independent of any concrete modeling language, we represent a model as a (possibly infinite) set of possible traces:

Definition 2 (Model). *For an alphabet Σ, a* ***model*** *over Σ is a set $M \subseteq \Sigma^*$.*

4.1 Simple Query

A *simple* query allows arbitrary computations on the current execution trace. This, in particular, does not require an underlying model. Examples of simple queries for the example in Fig. 2 are:

i. the degrees a student has completed so far,
ii. the degree that a student spent the longest time on,

iii. how long it has taken for the student to complete a *MSc, BIS*,
iv. the average execution time for all the degrees so far, and
v. the total time since the start of the current execution.

Each of these queries performs a simple computation on a given trace. A simple query Q is a function evaluating a trace to an element of a given set. For example, to compute the total execution time, we would write a function returning the difference between the timestamp of the last and first events. The result of this function would be real numbers (the number of seconds).

Definition 3 (Simple Query). *Given an alphabet, Σ, a* ***simple query*** *over Σ is a pair $Q = (q, \alpha)$, where α is a* ***result set*** *and $q : \Sigma^* \to \alpha$ is a function mapping traces into the result set. For a trace $\tau \in \Sigma^*$, the result of a simple query $Q = (q, \alpha)$ is the value $Q(\tau) := q(\tau) \in \alpha$.*

Example 1 (Time Since Start of Execution). To compute the total time spent (ignoring possible idle time) on a trace, we define the function:

$$totalTime(\tau) = \begin{cases} 0 & \text{if } |\tau| < 2, \text{ and} \\ timestamp(\tau(|\tau| - 1)) - timestamp(\tau(0)) & \text{otherwise,} \end{cases}$$

assuming the *timestamp* function extracts the value of the *time:timestamp* attribute. We use the function in a query $Q_{totalTime} = (totalTime, \mathbb{R})$.

4.2 Compare Query

A simple query only deals with a single (partial) trace. Often we wish to make comparisons with traces with a similar prefix as a given partial trace. One such query is to compare the performance of the current execution to other similar executions. A *compare* query allows such comparisons.

Compare queries need a partial trace and the model as input. In a compare query, we only consider traces from the model that have a similar past as the current partial trace. The filter criteria can be based on a resource, task, data, or other attributes from the partial trace. Examples of use cases that can be handled by a compare query are:
i. in how many similar traces has a *Master of BPM* been already executed,
ii. how long (on average) other similar traces spent to finish a *BSc*, and
iii. how long time has similar executions on average spent to get to where the current execution is.

To capture what we mean by similar traces, we introduce a *projection mapping*, which is a mapping abstracting away information from the original trace. Projection mappings essentially remove uninteresting attributes and events, for example:
i. *name:* consider only event names and ignore all other event attributes,
ii. *name and resource:* consider event and resource names, and
iii. *name of complete events:* consider the event name of events whose transition type is *complete*, and remove all events which are not complete events and all other attributes.

To support online computation, we require that applying a projection mapping more than once does not remove further information and that it is compositional, i.e., that the projection does not depend on the entire trace, but only on the events that it is applied to. Formally:

Definition 4 (Projection Mapping). *A mapping $p : \Sigma^* \to \Sigma^*$ is a* ***projection mapping*** *if it is a homomorphism wrt. trace composition, i.e., $p(\tau_1\tau_2) = p(\tau_1)p(\tau_2)$ for all $\tau_1, \tau_2 \in \Sigma^*$.*

Due to the compositional nature of projection mappings, it suffices to specify how to map traces of length 1 to traces of length 0 or 1. Thus, we can specify a projection mapping as:

Example 2 (Name of Complete Events). To project onto the names of complete events only, we define:

$$p_{nameComplete}(e) = \begin{cases} \varepsilon & \text{if } transition(e) \neq complete, \text{ and} \\ name(e) & \text{otherwise.} \end{cases}$$

Given a projection mapping and a partial trace, we define a notion of "all similar traces" by introducing all prefixes matching the trace under the projection mapping:

Definition 5 (Similar Prefixes). *Given a trace τ, a projection mapping p, and a model $M \subseteq \Sigma^*$ the similar prefixes of τ in M is the set $prefixes_M(p, \tau) := \{\tau' \mid \tau'\tau'' \in M \ , \ p(\tau) = p(\tau')\}$.*

As $prefixes_M(p, \tau)$ may be infinite for some partial trace τ of a model M, we cannot necessarily provide a closed expression for a compare query. Instead, we give an estimate in the form of a probability function. Using the notion of similar prefixes, we are now ready to formally define a compare query:

Definition 6 (Compare Query). *Given an alphabet, Σ, a* ***compare query*** *is a triple $C = (M, Q, p)$, where M is a model, $Q = (q, \alpha)$ is a simple query, and p is a projection mapping over Σ. For a (partial) trace $\tau \in \Sigma^*$, the result of a compare query (the* ***comparison****), $C(\tau)$, is a probability function $C(\tau) : \alpha \to [0, 1]$ such that for $a \in \alpha$, $C(\tau)(a)$ is the probability that $Q(\tau') = a$ for a random similar trace prefix $\tau' \in prefixes_M(p, \tau)$.*

We put no requirement on the probability measure $C(\tau)$, other than requiring that it is indeed a probability measure. In particular, we do not require that it assumes that all traces in M are equally probable.

Example 3 (Average Time to Reach Current Position). To compute the average time spent for other traces with the same sequence of complete event names to get where we are, we would use the compare query $C_{averageTime} = (M, Q_{totalTime}, p_{nameComplete})$ using the simple query and projection mapping defined in Examples 1 and 2. We get the average by computing the expectation for the returned probability function.

4.3 Predict Query

A *predict query* is similar to a compare query, except instead of just considering a prefix of similar traces of a model, it considers all possible futures or completions of such traces. Examples of predict queries include:

i. total execution time after becoming a *Master of BPM*,
ii. probability of becoming a *Master of BPM*, and
iii. expected total execution time of the current trace.

For predict queries, we still use projection mappings to define similar traces, but we no longer only consider similar prefixes, but the completion of such similar executions. To do this, we define the possible completions of a partial trace in a model:

Definition 7 (Completion). *Given a trace τ, a projection mapping p, and a model $M \subseteq \Sigma^*$ the* ***completion*** *of τ in M is the set $completion_M(p, \tau) := \{\tau'\tau'' \in M \mid p(\tau) = p(\tau')\} = \{\tau'\tau'' \in M \mid \tau' \in prefixes_M(p, \tau)\}$.*

Definition 8 (Predict Query). *Given an alphabet, Σ, a* ***predict query*** *is a triple $P = (M, Q, p)$, where M is a model, $Q = (q, \alpha)$ is a simple query, and p is a projection mapping over Σ. For a (partial) trace $\tau \in \Sigma^*$, the result of a predict query (the* ***prediction****), $P(\tau)$, is a probability function $P(\tau) : \alpha \to [0, 1]$ such that for $a \in \alpha$, $P(\tau)(a)$ is the probability that $Q(\tau') = a$ for a random similar trace $\tau' \in completion_M(p, \tau)$.*

Example 4 (Expected Total Execution Time). To compute the expected total execution time for traces with the same prefix of complete event names, we use the predict query $P_{totalTime} = (M, Q_{totalTime}, p_{nameComplete})$. We note that the query is the same as the compare query in Example 3; only the query type has changed.

4.4 Recommend Query

Where a predict query can predict the future, a *recommend query* tries to recommend the best next action to take to achieve a desired goal. The result of a recommend query is based on a given goal that has to be achieved. For the running example shown in Fig. 2, examples of such goals include:

i. to highest chance of becoming a *Master of BPM*,
ii. to finish a *MSc, BIS* in the cheapest possible way, or
iii. finish execution as fast as possible.

The basic idea is to extend a given trace τ with all possible next events, predict the goal in each trace, and recommend actions resulting in the best prediction. In order for this to make sense, we need to define which prediction is best. We could do this by having a user impose a total order of probability functions, but we instead assign a value to any prediction and use this to order the next events. We do this because we expect most of the predicted values to be either numerical values or non-numerical values from a small finite set, and assigning a value is easier for a user to understand than defining a total order. Examples of evaluations would be:

i. highest median (value would be the median),
ii. lowest average (value would be the expectation times minus one), or
iii. highest 95% percentile (value would be 95% percentile).

Formally, an evaluation just assigns a real value to a probability function:

Definition 9 (Evaluation). *Given a set of values, α, an* ***evaluation*** *over α is a function $E : [0,1]^\alpha \to \mathbb{R}$ assigning to a probability measure a real number, the* ***value*** *of the probability measure.*

Example 5 (Lowest Average). An evaluation optimizing for the lowest average of a numerical prediction would be $E_{lowestAverage}(\mathcal{P}) = -E(\mathcal{P})$.

Definition 10 (Recommend Query). *Given an alphabet, Σ, a* ***recommend query*** *is a pair $R = (P, E)$, where $P = (M, Q, p)$ is a predict query over Σ with $Q = (q, \alpha)$ and E is an evaluation over α. We let $next(\tau) := \{e \in \Sigma \mid \tau e \tau' \in M\} \cup \{\varepsilon \mid \tau \in M\}$ denote all possible continuations, and define $predictions_M(\tau) = \{P(\tau\tau') \mid \tau' \in next(\tau)\}$ as all predictions of possible continuations. We let $m_\tau = \max\{E(\mathcal{P}) \mid \mathcal{P} \in predictions_M(\tau)\}$ be the maximal evaluation of a prediction. The result of the recommend query (the* ***recommendation****) is $R(\tau) := \{\tau' \in next_M(\tau) \mid E(P(\tau\tau')) = m_\tau\}$. If the trace contains no valid continuation, $\emptyset$ is returned.*

We note that the complexity of the definition stems from the fact that we need to include the empty trace in the set of possible continuations if terminating the trace is a valid choice, and that we may have multiple continuations yielding the highest evaluation. We note that while we here compute a maximum over a possibly infinite set, it is finite in practise as we often can compute the set of enabled actions using a model as a small finite set, and in our implementation we require clients to provide candidates among which the continuations should be found.

Example 6 (Fastest Execution). To optimize for the fastest execution we could use $R_{fastest} = (P_{totalTime}, E_{lowestAverage})$.

5 Implementation

In the previous section, we defined and exemplified a meta-model for operational support. Here, we provide concrete technology suggestions for implementing the various parts, including how to represent the data required by the meta-model, how to implement a provider, and we show an example client application, namely Declare.

5.1 Data Representation

Looking at the meta-model in discussed in Sect. 4, we see the need to be able to represent: *queries* and the *result set* of queries (for simple queries), *models* and *projection mappings* (for compare and predict queries), and *evaluations* (for recommend queries). Furthermore, we need to be able to specify (partial) traces (as the query parameters), values of result sets and probability functions (for the results of queries).

For the representation of traces we use the XES format. If a model comprises a finite set of traces, it can also be represented using XES. Alternatively, it can be implemented using, e.g., a finite automaton or a Petri net. We leave the exact representation up to the individual providers, but we logically view any process model as a (possibly infinite) XES log.

For queries, result sets, and values of result sets, we could define our own language, but this is tedious and error-prone, os we rely on pre-existing standards. As our traces and models are represented using XES and hence XML (at least conceptually), we have chosen to use XQuery. By using XQuery, we directly inherit descriptions of the query, the result set, and the value of the result set. XQuery also provides representations of the projection mappings and evaluations as they can be viewed as a particularly simple kind of queries.

The final consideration is how to represent probability functions, which is a bit more difficult as the result sets often are infinite. Instead of representing probability functions explicitly, we distinguish between two kinds of observations: continuous values and observations from a finite set. For finite sets, we represent the probability of each element of the set, and for continuous values, we represent common statistical notions, such as the mean, the median, the 25% and 75% quartiles, as well as various confidence intervals. We allow individual providers to supply more information (such as which statistical distribution is assumed and its parameters), but require at least the basic information, which can be represented finitely.

5.2 Provider Implementation

In Fig. 1, we saw the infrastructure of the operational support framework implemented in ProM. The exact protocol used for communication is explained in [16]; here we only explain parts relevant for understanding the implementation and considerations for the meta-model. The *Provider* interface shown in Listing 2 reflects the meta-model defined earlier based on the four main queries handled by operational support service. We have methods for session management, *accept*, *destroy*, and *updateTrace*, and for actual queries. The model is provided once (in *accept*), and the current trace is built incrementally in the *Session* using *updateTrace*. All queries take a log containing all continuations suggested by the client and a query (representing a simple query, as the return set is implicitly defined by the query in XQuery). In addition to projection mappings and evaluations as prescribed by the meta-model, we also include a boolean value indicating whether the trace is complete or not, which is important for some queries.

5.3 Example Client

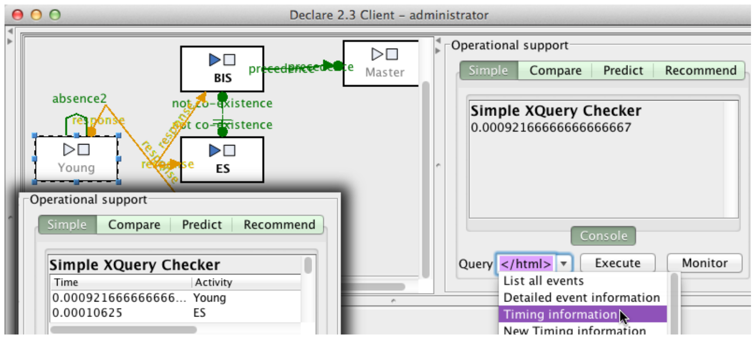
We have implemented an operational support client in Declare as shown in Fig. 4. The Declare client executes tasks from a given process model and can at any

```
public interface Provider extends Serializable {
   boolean accept(Session s, List<String> modelLanguages,
       List<String> queryLanguages, Object model);
   void destroy(Session s);

   <R, L> R simple(Session s, XLog availableItems, L query, boolean done);
   <R, L, P> Prediction<R> compare(Session s, XLog availableItems,
       L query, P projection, boolean done);
   <R, L, P> Prediction<R> predict(Session s, XLog availableItems,
       L query, P projection, boolean done);
   <R, L, P, E> Recommendation<R> recommend(Session s, XLog availableItems,
       L query, P projection, E evaluation, boolean done);

   void updateTrace(Session session, XTrace trace);
}
```

Listing 2: Provider interface.



Simple XQuery Checker
0.00092166666666666667

Simple XQuery Checker

Time	Activity
0.000921666666666...	Young
0.00010625	ES

Fig. 4: Declare client showing results from the query from Listing 3 and Listing 4 (inset).

point during the execution send queries to the OSS in ProM. The parameters correspond closely to what providers expect from Listing 2. If we consider the study process from Fig. 2, we may be interested in seeing how long we have taken since the beginning of the execution. We can write a simple XQuery expression doing this, shown in Listing 3. The implementation corresponds to the one of Example 1. We extract the timestamp of the first and last events of the trace and return the difference. The query is simple we do not need to use the FLOWR syntax as we just compute a single value. We use a simplified syntax for logs, where we have direct access to attributes of events instead of having to go through the *string* elements as shown in Listing 5. The current trace is referred to using the variable *$trace*. For presentation, we divide by one day (1440 minutes) to obtain a result expressed in days. We see the result of executing this query after performing task *Young* in Fig. 4. We also see that Declare provides a set of standard queries.

A more interesting query is shown in Listing 4. Here, we compute the execution time of all executed tasks and need more of the FLOWR syntax for XQuery. In the query we iterate two variables (*$e* and *$f*) over all events in the trace (l. 1). We ensure that of the two events, one is a start event and the other is a complete event (l. 2) and that they belong to the same action instance and have

```
(xs:dateTime($trace/event[last()]/time:timestamp) -
 xs:dateTime($trace/event[position() = 1]/time:timestamp))
                                          div xs:dayTimeDuration('PT1440M')
```

Listing 3: Simple XQuery extracting the duration of a trace in days.

the same name (l. 3). For each such pair, we construct an item which contains a node for the time (computed similarly to Listing 3 except we use the events *\$e* and *\$f*), and a node with the name of the events. Declare uses heuristics for displaying tabular data, and it automatically displays such data as seen in the inset at the bottom left of Fig. 4. Note that for the inset we have additionally executed the *ES* task.

Advanced Queries. In the previous example, we discussed a simple query where computations are done on a single partial trace. In that case, we referred to the current trace as *\$trace*. To perform anything but simple queries, we also need to consider the model. We conceptually consider the model as a (possibly infinite) set of traces, and use queries using the exact same syntax to simplify reusing them. We provide examples of predict and recommend queries in Fig. 5. Both queries require a projection mapping, which is just an XQuery function mapping each event to a domain of choice, considering events equivalent if their activity names are identical. In the example we consider events equivalent if they share name and ignore all events that are not complete events, using the mapping in Listing 6. The prediction provides an estimate and a confidence interval, and the recommendation contains a recommended action (we are optimizing for lowest average) and predictions for all possible next actions.

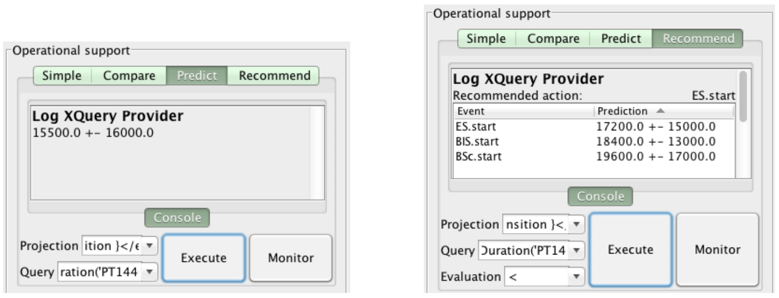
The simple provider presented here uses a historical log as model, and simply computes values for all matching traces of the log, assuming a standard distribution of the results for computation of confidence intervals. As we see, the confidence intervals are very large compared to the predicted values, indicating that the predicted values are not very trustworthy. This can be improved by instead discovering or enhancing a model from the log (this can, e.g., be a Petri net or a transition system), but this may lead to models which comprise infinitely many possible executions.

```
1  for $e in $trace/event, $f in $trace/event
2  where $e/lifecycle:transition='start' and $f/lifecycle:transition='complete'
3  and $e/concept:instance=$f/concept:instance and $e/concept:name=$f/concept:name
4  return <item>
5          <time>{ (xs:dateTime($f/time:timestamp)-xs:dateTime($e/time:timestamp))
6                                   div xs:dayTimeDuration('PT1440M')
7          }</time>
8          <activity>{ $e/concept:name }</activity>
9         </item>
```

Listing 4: Simple XQuery extracting the execution time of tasks.

```
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="1.0" xmlns="..." xmlns:concept="..." ...>
  <trace concept:name = "Case1" >
    <event
      concept:name = "Young"
      lifecycle:transition = "complete"
      org:resource = "user1"
      time:timestamp = "2004-10-04T08:05:00.000+02:00"
      Sex = "Male" />
    ...
  </trace>
  ...
</log>
```

Listing 5: Simplified representation of the partial trace shown in Listing 1.



```
Log XQuery Provider
15500.0 +- 16000.0

Projection  ition }</e
Query       ration('PT144
```

```
Log XQuery Provider
Recommended action:              ES.start
Event          Prediction
ES.start       17200.0 +- 15000.0
BIS.start      18400.0 +- 13000.0
BSc.start      19600.0 +- 17000.0

Projection  nsition }<
Query       Duration('PT14
Evaluation  <
```

Fig. 5: Declare client showing predict (left) and recommend (right) results from the query shown in Listing 3.

We leave the exact interpretation of how to compute the probability distribution up to the provider; if the model is finite as here, they can provide an exact response and otherwise providers may use statical analysis of the queries and derive a closed expression for the result over the infinite model, they may assume that the result of the query is absolutely convergent as we execute loops and iterate until a desired accuracy is achieved, or they may use sampling based on some heuristics.

6 Conclusion and Future Work

In this paper, we have presented a meta-model for operational support. The meta-model defines four kinds of queries of increasing complexity and power, including simple queries providing statistics about the current execution, compare queries comparing the current partial execution to similar partial executions,

```
if ($event/lifecycle:transition = 'complete')
then $event/concept:name
else ()
```

Listing 6: Simple projection mapping.

predict queries yielding predictions about the outcome of the current execution based on the outcome of completed executions with similar prefix, and recommend queries providing recommendations of what to do to achieve a goal. We have provided implementations suggestions based on open standards for all concepts used in the definitions, and presented a concrete implementation is the workflow system Declare and the process mining framework ProM. While we have used Declare as example modeling formalism, our approach is independent of a concrete modeling language as we conceptually view models as sets of possible execution traces.

Future work includes testing and comparing algorithms for operational support. This is now possible due to a unified interface. Also, here we have only considered a single-client/single-case scenario where we have a single client working on a single case. It would be interesting to consider situations where two or more clients work on a single case, a single client works on multiple cases, or multiple clients working on multiple cases, both from a meta-model perspective and from an implementation and evaluation perspective.

References

1. W.M.P. van der Aalst. *Process Mining-Discovery, Conformance and Enhancement of Business Processes.* Springer, 2011.
2. W.M.P. van der Aalst, M. Pesic, and M.S. Song. Beyond Process Mining: From the Past to Present and Future. In *Proc. of CAiSE*, volume 6051 of *LNCS*, pages 38–52. Springer, 2010.
3. I. Barba, B. Weber, and C. Del Valle. Supporting the Optimized Execution of Business Processes through Recommendations. In *Proc. of BPI Workshop*, 2011.
4. M. Brundage. *XQuery: the XML query language.* Addison-Wesley, 2004.
5. C. Dorn, T. Burkhart, D. Werth, and S. Dustdar. Self-adjusting Recommendations for People-Driven Ad-Hoc Processes. In *Proc. of BPM*, volume 6336 of *LNCS*, pages 327–342. Springer, 2010.
6. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology.* Wiley, 2005.
7. C. Haisjackl and B. Weber. User Assistance during Process Execution - An Experimental Evaluation of Recommendation Strategies. In *Proc. of BPI Workshop*, volume 66 of *LNBIP*, pages 135–145. Springer, 2011.
8. A. Lazovik, M. Aiello, and M. Papazoglou. Associating Assertions with Business Processes and Monitoring their Execution. In *ICSOC*, pages 94–104. ACM Press, 2004.
9. K. Mahbub and G. Spanoudakis. A Framework for Requirents Monitoring of Service Based Systems. In *ICSOC*, pages 84–93. ACM Press, 2004.
10. P. Resnick and H.R. Varian. Recommender Systems. *Comm. of the ACM*, 40(3):56–58, 1997.
11. A. Rozinat, M.T. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C. Fidge. Workflow Simulation for Operational Decision Support. *Data and Knowledge Engineering*, 68(9):834–850, 2009.
12. M.H. Schonenberg, B. Weber, B.F. van Dongen, and W.M.P. van der Aalst. Supporting Flexible Processes Through Recommendations Based on History. In *Proc. of BPM*, volume 5240 of *LNCS*, pages 51–66. Springer, 2008.

13. Staffware. *Staffware Process Suite Version 2 – White Paper*. Staffware PLC, 2003.
14. I.T.P. Vanderfeesten, H.A. Reijers, and W.M.P. van der Aalst. Product Based Workflow Support: Dynamic Workflow Execution. In *Proc. of CAiSE*, volume 5074 of *LNCS*, pages 571–574. Springer, 2008.
15. B. Weber, W. Wild, and R. Breu. CBRFlow: Enabling Adaptive Workflow Management Through Conversational Case-Based Reasoning. In *Advances in CBR*, volume 3155 of *LNCS*, pages 89–101. Springer, 2004.
16. M. Westergaard and F.M. Maggi. Modelling and Verification of a Protocol for Operational Support using Coloured Petri Nets. In *Proc. of ATPN*, volume 6709 of *LNCS*. Springer, 2011.