An Infrastructure for Cost-Effective Testing of Operational Support Algorithms Based on Colored Petri Nets

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. *Operational support* is a specific type of process mining that assists users while process instances are being executed. Examples are *predicting* the remaining processing time of a running insurance claim and *recommending* the action that minimizes the treatment costs of a particular patient. Whereas it is easy to evaluate prediction techniques using cross validation, the evaluation of recommendation techniques is challenging as the recommender influences the execution of the process. It is therefore impossible to simply use historic event data. Therefore, we present an approach where we use a *colored Petri net* model of user behavior to drive a real workflow system and real implementations of operational support, thereby providing a way of evaluating algorithms for operational support before implementation and a costly test using real users. In this paper, we evaluate algorithms for operational support using different user models. We have implemented our approach using Access/CPN 2.0.

1 Introduction

Some business processes are unstructured or only very loosely structured. This is the case where a process has never been formalized or where the process requires a lot of freedom. Examples of such processes are processes in small very agile companies, or processes in disaster handling or healthcare, where flexibility and experience plays a more prominent role than a strictly structured process. While such freedom may be good for an experienced user, it may not provide enough support for less experienced users. Using process mining, it is possible to provide *operational support* [3] for running processes of this kind. Under operational support, users are provided with on-line information about the running process, and can even be given recommendations about the next actions to be taken in order to arrive at a goal [15].

In [11], we defined a meta-model for operational support. Here, a client sends a *partial execution trace* along with a *query* to an operational support service.

* 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.

S. Haddad and L. Pomello (Eds.): PETRI NETS 2012, LNCS 7347, pp. 308–327, 2012.

A query is simply a question to which a *response* is received. Operational support allows four types of queries. A *simple* query checks the performance of the current partial execution trace, for example, what is the total time since the start of current execution? 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 looks into 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?

It is easy to evaluate algorithms for the first two types of queries (simple and compare). These are basically lookup functions combined with standard operators computing average, variance, etc. As shown in [1, 4] it is also possible to evaluate predict queries using cross-validation. For example, when using k-fold cross-validation, the set of process instances is partitioned in k parts. $k-1$ parts are used to learn a predictive model. The instances in the remaining part are used to evaluate the quality of the predictive model. Because historic data is used, it is possible to compare the predicted value with the real value. This experiment can be repeated k times thus providing insight into the quality of the predictive algorithm.

Algorithms providing recommendations are much more difficult to evaluate. Since recommendations influence the execution, it is impossible to directly use historical data. Users will change their behavior based on these recommendations, so it is impossible to simply use the observed behavior where users got no recommendations.

Here, we introduce a general setting for testing recommendations as shown in Fig. 1. At the bottom right we have a User. The user is executing a process. The process may be implemented using a Workflow System and as shown here or it may be ad-hoc. The user consults Operational Support to get advice about which step to execute. The idea we present here is to model the user using a colored Petri net (CPN) [8] model and have that model interact directly with a real workflow system, i.e., Declare [2, 7] and real implementations of operational support in the ProM framework [13, 19]. That way we do not need real users, making the approach much more affordable, yet still interact with real systems, so we get realistic results. Using real systems instead of modeled counter-parts also makes it much easier to do the modeling, as we only have to focus on the user behavior, and not on replicating already existing systems and algorithms. Our approach also allows rapid prototyping of algorithms by implementing them using a CPN model and directly integrating them in a real tool for operational support. We can then use the algorithms directly in tools acting as clients for operational support, including workflow systems and our testing platform.

The contribution of this paper is two-fold: First and most importantly, we present the test suite modeled in colored Petri nets that provides a means to

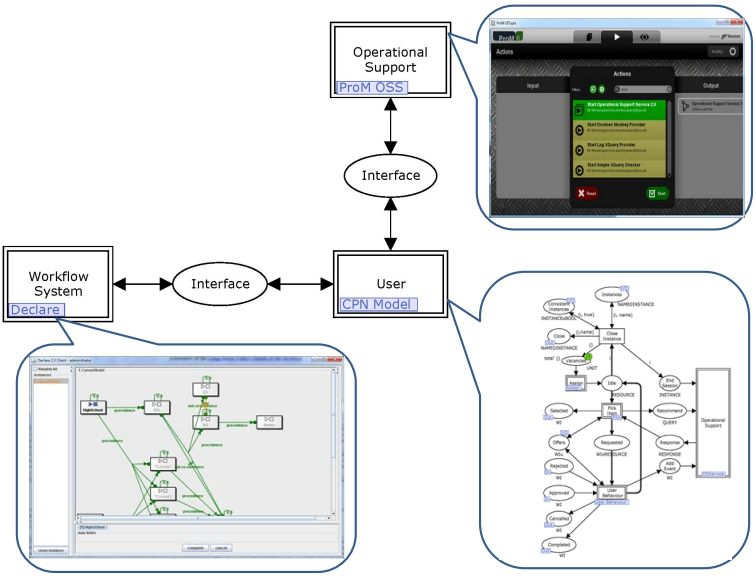

Fig. 1. Abstract Testing Platform

test simple algorithms for operational support in a cost-effective way. The algorithms we present here are not intended as real examples of algorithms for providing operational support, but only to illustrate the test-suite. Second, we provide a framework based on colored Petri nets making it very easy to prototype algorithms for operational support. Through log generation of the CPN model, we are able to also provide an evaluation of simple recommendation algorithms for operational support in a simple but non-trivial setting.

The term *operational support* refers to a collection of process mining techniques [1] executed while people are still working on cases. Several papers describe techniques to predict the remaining total execution time of running cases. See [3, 4] for pointers to such techniques. Another type of operational support, more relevant for this paper, is providing *recommendations*. In [16] simple ad-hoc models are created to support recommendation. In [17], case-based reasoning is applied to find similar cases. See [14] for a more general overview of recommenders. To the best of the authors' knowledge, there is no preexisting work on unified testing recommenders except for ad-hoc testing of individual recommenders compared to no support.

In [11] a generic framework for operational support based on queries is proposed. This is used in this paper. In [19] the operational support service of ProM is modeled and analyzed using CPN Tools [6]. The integration of CPN Tools with other components was described in [18]. In [9] it was shown how CPN components and workflow components can be exchanged for testing and simulation. This approach will also be followed in this paper.

We use Declare [2,7] as an example of a workflow system. It is selected because it allows more flexibility than procedural approaches to process modeling and therefore benefits from recommendation techniques.

The remainder of the paper is organized as follows: In Sect. 2, we provide the background needed to understand the remainder of the paper including a running example. In Sect. 3, we discuss the user model, focusing on the modeling of the time needed to execute tasks. In Sect. 4, we discuss the recommendation algorithms tested in this paper including presenting a generic CPN model that can be used as a starting point for rapid prototyping of operational support algorithms. Section 5 provides a description of the experiments carried out to evaluate the algorithms described using the various user models. Finally, Sect. 6 summarizes our conclusions and provides directions for future work.

2 Background

In this section we provide the background needed to understand the rest of this paper. We briefly introduce the Declare language which is an example of a workflow language that we use for modeling our running example. We also introduce an architecture for operational support. We summarize the Access/CPN 2.0 [18] library for running a CPN model together with software components.

Running Example and Declare. Consider the example in Fig. 2, modeling a study process. This example is created in Declare [7], which is a workflow system based on a declarative language. Compared to conventional procedural workflow systems, Declare allows for much more flexibility [2]. In Declare, *tasks* are shown as rectangles and can initially be executed in any order. Tasks are constrained by *constraints*, shown as arcs. We shall not go into details about the constraints of Declare but refer the interested reader to [2,7]. Here, we just supply an abstract overview of the behavior of the model. Basically, a student can choose either an academic or a practical path to a degree. A student can initially either choose to go to HighSchool or to get a job (Work). Going to high school allows students to be admitted for a BSc (the academic path). Alternatively, a student may decide to get a job. Having had a job allows the student to enter two practical supplementary courses (PCourse1 and PCourse2). In order to be admitted to the four theoretical courses (TCourse1–TCourse4), a student must both have had a job and also been to high school. Having completed all six supplemental courses is a prerequisite to Qualify for starting a master's study. A student is also allowed to start a master's study if he has completed a BSc. Out of the two master's degrees offered, only one can be completed (for financial reasons). Only after completing a master's degree in business information systems (MSc, BIS) can a student become a true Master of BPM.

In our example, we can optimize towards at least two goals: getting a master's degree as fast as possible or becoming a master of BPM. The difficulty for a student is that he has at any point a lot of freedom. For example, he can at any point in time decide to get a job which may open new possibilities. During

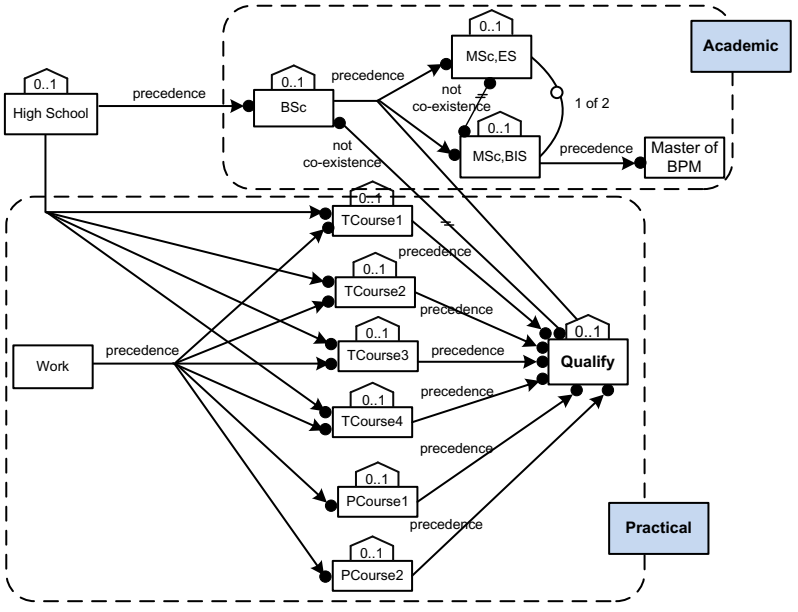

Fig. 2. A study process model in Declare

the execution many activities may be allowed (e.g., after going to high school and having a job, BSc, the 6 courses and Work are allowed actions), making it difficult to select the best action to take. Operational support aims to assist users in making such decisions. For example, based on historic information we can recommend particular actions in a given context.

Operational Support Architecture. Operational support can be implemented in many ways. For example, one way is just to suggest executing a random task and another way is to look at what students did before. In order to support any present and future algorithms in a coherent way, we use the architecture for operational support shown in Fig. 3. Here, a Client communicates with a Workflow System and with the operational support service (OS Service; OSS in the following). The OSS forwards requests to a number of operational support providers (OS providers; providers in the following), which may implement different algorithms. The OSS receives responses from the providers which it sends back to the Client. In [19] we

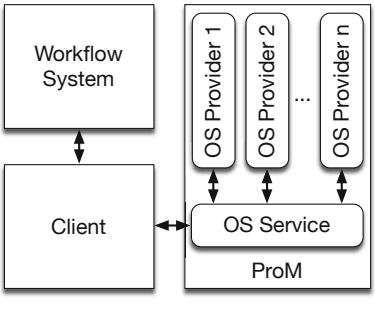

Fig. 3. Architecture of the operational support in ProM

```
1  public interface Provider extends Serializable {
2      boolean accept(Session s);
3      void destroy(Session s);
4
5      <R, L> Recommendation<R> recommend(Session s,
6                                         XLog availableItems, L query);
7
8      void updateTrace(Session session, XTrace trace);
9  }
```

Listing 1. Provider interface

presented a protocol and architecture making it possible to access different algorithms using a common protocol and this architecture. In [11] we defined a common meta-model for operational support, allowing a common interface to all algorithms.

In order to implement an algorithm for the operational support service, we need to implement the interface in Listing 1. The interface also has methods for the other kinds of queries, but we have hidden them as we are not interested in them here. accept and destroy control the life-cycle of the provider, and recommend handles actual queries. Queries get a set of all availableItems to pick among and a query to optimize towards. The result is a Recommendation, which basically is an event recommended to execute. updateTrace is called whenever the client has executed more events. All methods have an additional session parameter which can be used to store case-local data.

Cosimulation Using Access/CPN 2.0. Our goal is to take a model similar to the one in Fig. 1 and refine the User using a sub-module described as a colored Petri net. This is already supported by CPN Tools [6], a tool for editing and analysis of colored Petri nets. Furthermore, we want to use the actual Declare workflow system as a replacement for the substitution transition Workflow System and the actual implementation of operational support in ProM as a replacement for the Operational Support substitution transitions. Access/CPN 2.0 [18] is a library for interaction between CPN models and Java programs, and supports exactly this kind of interaction.

Using Access/CPN 2.0, it is possible to implement a simple interface and have the library run a *cosimulation*, where the model is executed and synchronized with the Java code.

3 User Behavior Modeling

In this section, we show how we model user behavior. Our model is a concrete implementation of the abstract testing platform in Fig. 1. The model is parameterized and allows different user behaviors. While we allow configurability of probabilities of completing or cancelling a current task, more interesting

configuration options include which timing model to use and whether a user uses operational support. Our entire model comprises 14 pages, 127 places, and 41 transitions.

The top level of our model can be seen in Fig. 4. It consists of the Workflow System (left) and the User (right). The workflow system has exposes a number of Instances of the process to be executed (cf. instance 1 for the Study process in Fig. 4). For each instance, it also indicated whether the instance is Consistent, i.e., if it can safely be terminated. Furthermore, the workflow system also exposes a number of Offers which are concrete tasks that can be executed by users. For example, from the study model shown in Fig. 2 initially the possible offers for instance 1 are HighSchool and Work as seen as tokens on the Offers place. A user can pick an offer and inform the workflow system that is has Selected that work item. The workflow system then either approves or rejects the request. If the request is Rejected, it is dropped. Otherwise if the request is Approved, the user starts working, for example, from Fig. 4 HighSchool has been approved by the workflow system and is seen as a token on the Approved place. At some point, the user either cancels work or completes it, i.e., the workflow system is informed using Cancelled and Completed places. When an instance is consistent, i.e., all the required work items offered to the user have been completed then the user can Close the instance.

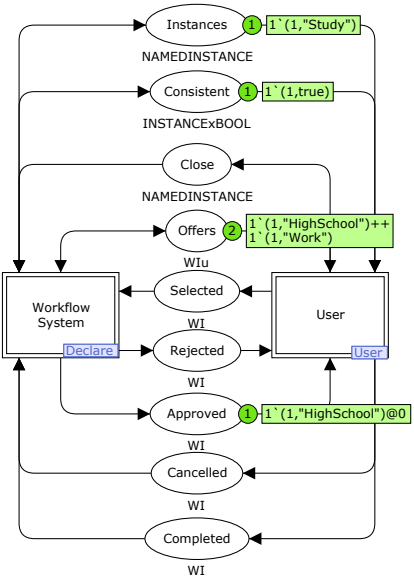

Fig. 4. Workflow system and user

3.1 User Model

A user is modeled as shown in Fig. 5. Users are generated on-demand (depending on the instances that have sent from the workflow system) by the Assign page, which generates users as needed up to a certain threshold (Vacancies). A user starts off being Idle. An idle user can select to Pick Item and start working. This can be done using the aid of Operational Support. After a task has been Requested, it can be executed (or cancelled), making sure to inform the workflow system accordingly. Alternatively, an idle user can Close a consistent instance (i.e., an instance where all the required work items have been completed).

The operational support service can handle recommendation requests (Recommend) and provide Responses. Furthermore, the OSS can be notified when the user has decided to End Session, which is useful for clean-up. After a user has executed an event it is sent to Add Event to allow the operational support service to construct an execution trace. The operational support service is implemented

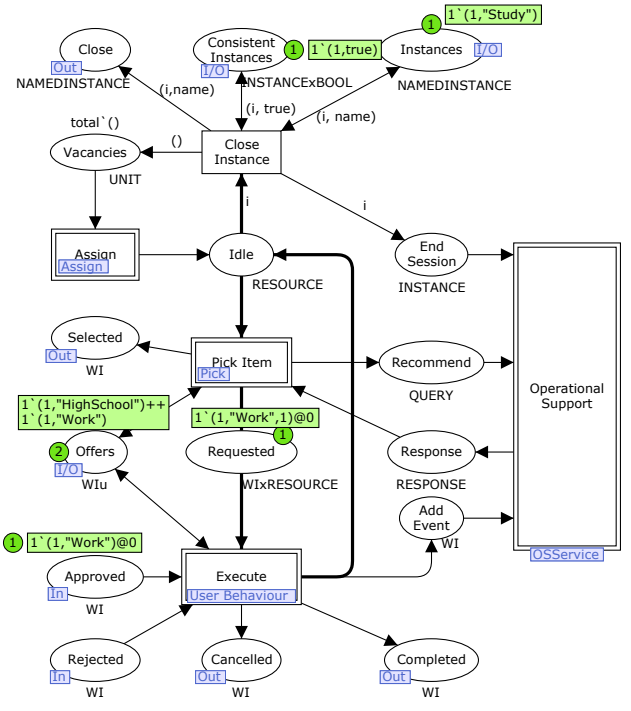

Fig. 5. User model

as a Java module using Access/CPN 2.0 [18]. It comprises 683 lines of code, 227 of which handles a very flexible interface to CPN models allowing formatting of queries in many ways and 143 lines of GUI integration code, making the actual interface just over 300 lines.

Pick Item Model. In Fig. 6, we see that an idle user first needs to decide whether to use support or not. We do this before actually asking for support due to efficiency of simulation. While a more realistic scenario would be to ask for support and use that as a guide, we can see this as just another algorithm for support, and we use the No Support to model a completely clueless user picking at random. The probability for whether support is used or not is configurable. No matter which branch we pick, we end up with a new work item on Selected and transferring control to Requested. It is only when we use support that we send Recommend requests to operational support and get Responses.

When we use operational support, we need a list of all enabled events for a given instance. We thus do as in Fig. 7: we first Select Instance, i.e., which instance of the process to work on. Then we build a list of all events enabled in that instance (Populate Offers for the Selected Instance). From Fig. 7, Selected Instance will be populated by High School and Work. When we have added all events to the list, we can Perform Query, sending the list of enabled events to

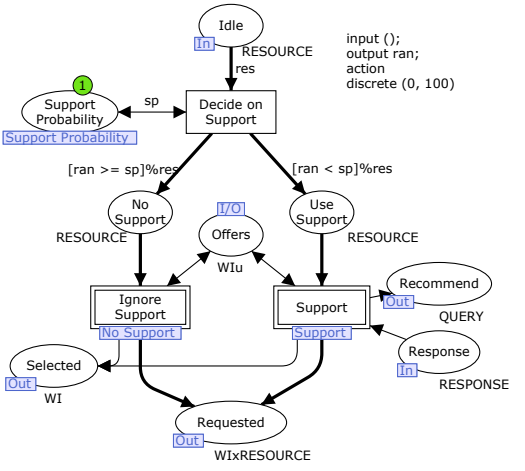

Fig. 6. Pick item model

Recommend of the operational support service (cf. token on the Recommend in Fig. 7). We are then Waiting for a Response, and when it arrives, we blindly Pick Recommended, inform the workflow system of which task we have Selected and transition to the Requested state. For the modelling notations we use here are discussed in [5].

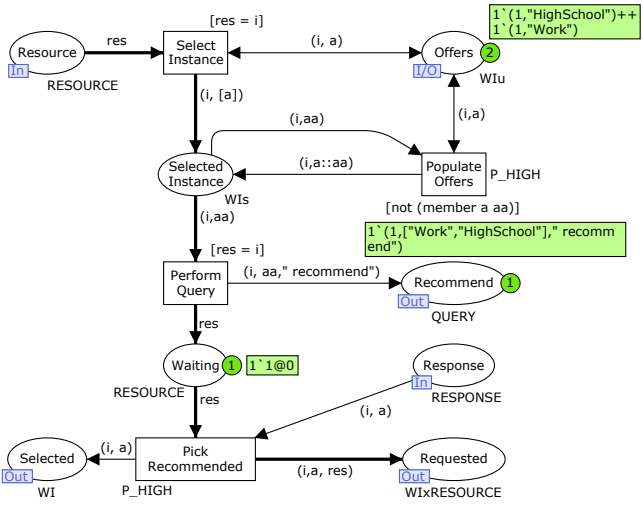

Fig. 7. Implementation of picking using operational support

Task Execution. Task execution (Fig. 8) at the abstract level starts when a task is Requested. If the workflow system Rejected the request, it is Aborted and the user returns to the Idle state. If the request is Approved, it is Executed, the workflow system is informed of success (Completed) or failure (Cancelled), and the operational support service is notified if a new task was executed (Add Event). We have four different implementations of Execute: one with constant execution times, one where execution times are sampled from a probability distribution, one where execution time is dependent on the previously executed task, and one where execution time is dependent on the stress level of the user.

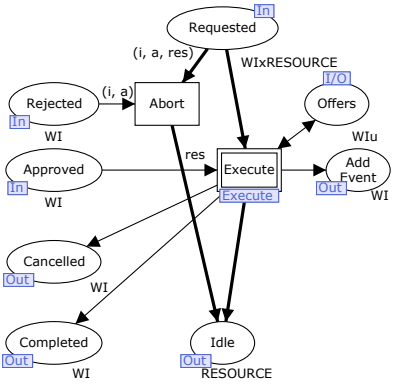

Fig. 8. Abstract task execution

Constant Time or Sample from Probability Distribution. The first two implementations can be treated together as constant time can be viewed as a specific probability distribution. Here, we assume that the execution time for tasks is independent of what was done before and the stress level of the user. Thus, we start in the Requested state in Fig. 9. Due to the timing model of CPN models, we need to make the decision of whether to Cancel Work or to Complete Work as soon as we Start Work. Thus, we pick a random number and if it is below a configurable threshold (Cancel Rate) then Cancel is selected, otherwise In Progress (cf. user 1 is executing Work in Fig. 9). This module is only enabled if the Timing Model is PROB. We get the timing information from shared Time Database and Cancel Database, which contain the timing information for successfully executing and the penalty for cancelling an activity. If we Cancel Work, we inform the workflow system and go to the Idle state, and when we Complete Work, we inform the operational support service as well as the workflow system before transitioning to the Idle state. The transitions all have a guard binding transtype, which indicated when a work item is started, and either cancelled or completed. This is used to subsequently import a simulation log into ProM for further analysis.

Batch Processing. The idea of this timing model is that if one executes similar tasks one after another, then one becomes faster due to step-up time reduction and learning effects. To model this "conveyor belt effect", we need to keep track of the last executed task and how many times we have executed the same task. In our example, we consider all the practical courses to be similar enough to use batch processing and all theoretical courses as well. In Fig. 10 we see the start of the module for batch processing (the remainder is the same as in Fig. 9). Instead of just one start transition, we now have two: one to Start New Work and

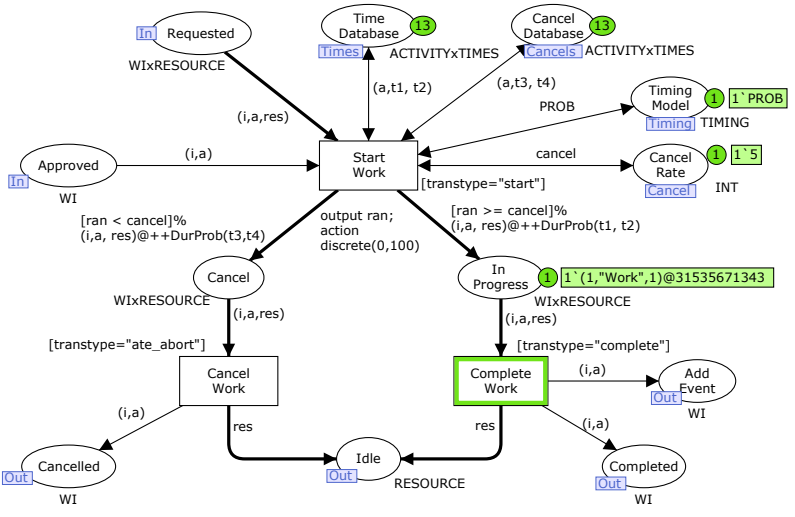

Fig. 9. Task execution model based on a probability distribution

one to Start Batch Work. The apparent complexity is due to the two transitions doing almost the same. Both have access to the two configuration options and two databases as the Start Work transition in Fig. 9. The place Last keeps track of which task we executed last and how many times we have executed a similar task. We read the last task from Last and compare it to the current task to execute. If they are the same, we Start Batch Work and if they are not the same, we Start New Work. In both cases, we update Last accordingly and if we execute batch work, we let the timing be dependent on how many times we have executed the same task.

Execution Time influenced by Workload. According to "Yerkes-Dodson Law of Arousal", the execution speed increases as the stress increases up to a certain optimal level beyond which the performance decreases [10, 12, 20]. In our model we let the execution time be dependent on the number of tasks offered in the queue for a user. This can be modeled as in Fig. 11. Like for batch processing, we only see the initial fragment as the remainder is the same as in Fig. 9. The basic idea is that we need to count how many tasks are in Offers. This is the same construction as we used to build a list of all tasks for operational support in Fig. 7, and has therefore been hidden in a substitution transition Count Tasks for legibility. Otherwise, Start Work is the same as in the simple case, except we compute the execution time with an extra parameter, namely the number of available tasks (ct) on the Count place for each instance (i).

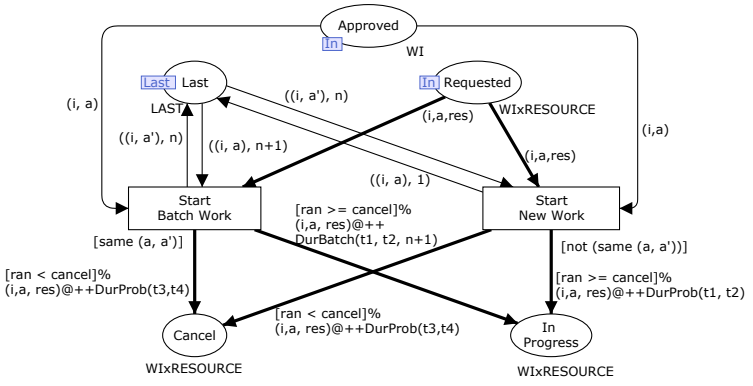

Fig. 10. Task execution model implementing batch processing

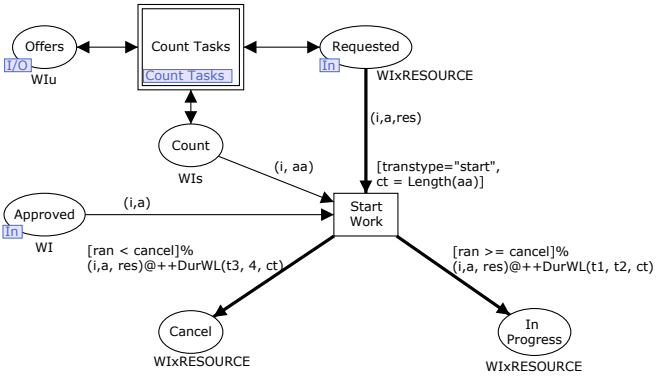

Fig. 11. Task execution taking workload into account

4 Recommendation Algorithms

In this section, we describe the four recommendation algorithms evaluated using simulation. Two of the algorithms are completely general and require no configuration, one is general but requires configuration, and one only works for our running example.

We can implement an algorithm by directly implementing the interface in Listing 1. For simple algorithms, this includes a lot of overhead, which may not be needed for quick testing. For this reason, we have created a generic implementation of the Provider interface using Access/CPN 2.0. This just requires that implementers make a simple CPN model. The generic connector comprises 505 lines of Java code including 206 lines of GUI integration code, making for just under 300 lines of logic.

To make it easy for implementers to get started, we have developed a template model which can be used to very quickly prototype operational support algorithms. The model comprises 7 pages, 30 places and 12 transitions, but a

user typically only has to worry about a single page. In the remainder of this section, we introduce our generic template model, how to implement the two simplest providers using this framework, we give a brief overview of a more advanced provider and show how we developed a provider specific for our running example. The reason for going through the operational support service instead of just incorporating the providers directly in the model is to improve reusability. First of all, we can use any provider from any model and do not have to copy one algorithm from one test model to another. Second, we can use the algorithms immediately and directly from any tool allowing operational support. This decoupling makes it easy to test algorithms on humans interacting with a workflow system if we do not want to just rely on simulation results.

4.1 Provider Model

The provider model quite closely reflects the interface in Listing 1. At the top level (Fig. 12) we handle Accept calls (corresponding to the accept method) and the kind of query (corresponding to recommend). The updateTrace method is not represented explicitly; instead the Traces place contains all currently active partial traces. The destroy method is not necessary as resource management is handled by the plug-in.

Provider setup just decides whether to accept or ignore a session. The default implementation unconditionally accepts sessions. The Query module separates each of the four kinds of queries to their own pages.

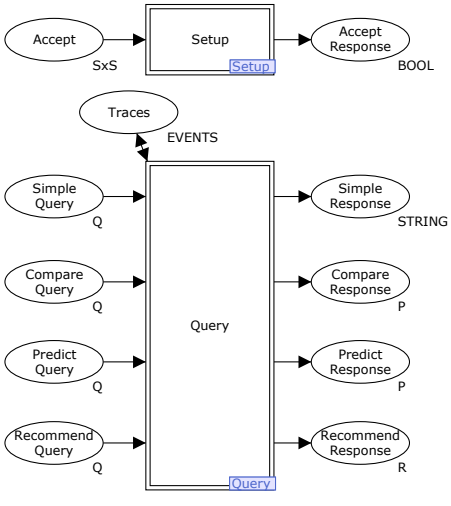

Fig. 12. Top model of a provider

Random Provider. The random provider just recommends a random enabled task from the list of available tasks. Thus, we expect this provider to have the same behavior as using no operational support at all and is put here as a baseline to compare with the more advanced recommendation techniques. The full implementation is shown in Fig. 13. We receive a request on Recommend Query. The request contains evts, a list of available tasks. Our response is just a random event picked using pickRandomEvent if it exists, or a dummy response otherwise (so we always provide an answer).

Batch Provider. The batch recommender always recommends, if possible, the same event as the one executed last. If this is not possible, it just recommends a random event. This is intended to work together with the batch timing, where

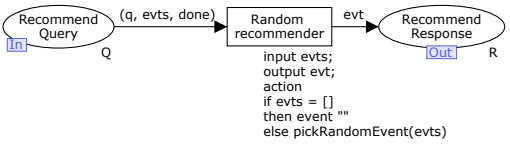

Fig. 13. Random recommender

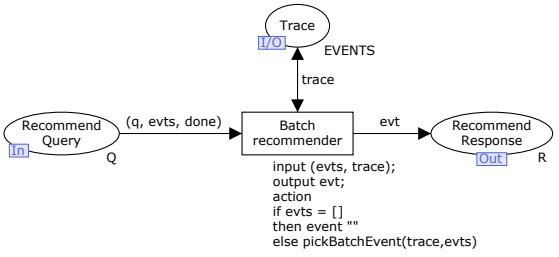

Fig. 14. Batch recommender

executing similar tasks together is faster than executing them interleaved with others. The implementation of the batch provider shown in Fig. 14 is very similar to the random provider, except we now also make use of the execution Trace and use the pickBatchEvent function to pick the event that is similar to the last executed one if one exists, and otherwise a random event.

Model-Specific Provider. The authors are very familiar with the running example, and therefore have an idea of the best way to execute it. We can encode this knowledge in a model-specific recommender. While the previous two recommenders work with any model, they are also not very intelligent or good at finding optimal executions (see also the next section on experiments). By making a model-specific recommender, we can make one that is better, but less generally applicable.

Creating a recommender using our framework is very simple, so we can implement a strategy recommending the academic route directly as in Fig. 15. The implementation has a set of events that should always be recommended with high priority if offered (Preferred). The transition Preferred Activity checks if a preferred event is among the offered ones and if so recommends it. The transition has high priority and will thus be selected before others. If no preferred activity is available, we just return the first (Other Activity). This exploits a known best (at least in some settings) implementation and the fact that all the events are only offered once in this model.

4.2 Log-Based Recommender

We want a recommender that provides better advice than guessing randomly, but at the same time, we would like to avoid models or situation specific

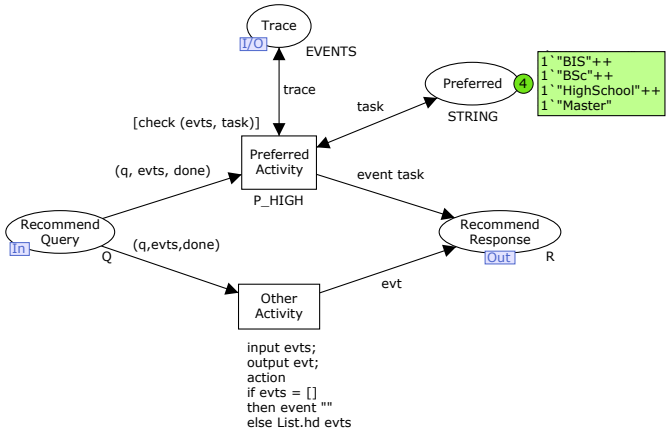

Fig. 15. Model-specific recommender

recommenders. For this purpose the log-based recommender is designed. This provider uses a historical log as guidance for providing recommendations. The idea is that we have a predicate selecting traces from the log we consider to be the same as the current one. We then compute a value on all such traces and return the next event of the trace which yields the best result. In our example, we would consider all traces with the same sequence of completed events similar to the current trace. Our computation would be the complete execution time, and our order would prefer shorter execution times.

The log-based provider is implemented in Java and comprises 513 lines of code, all of which is logic. Furthermore, this provider depends on a complicated library for querying XML documents using XQuery as discussed in [11]. All in all, we do not want to duplicate this code in a CPN model.

5 Experiments

In this section we outline how we have used our testing platform (described in Sect. 3) to test the various providers described in Sect. 4. First of all, we perform tests with the random recommender for all timing models to demonstrate that this yields the same as not using operational support. Also, we use this to eliminate the simplest timing model (constant time for execution) from future tests. Then we evaluate the batch and model-specific recommenders for the remaining timing models. Finally, we evaluate the log-provider using "historical logs" generated by the random provider. We evaluate the providers according to our two goals: shortest execution time and highest success rate, where a trace is successful if the task Master of BPM is executed.

Table 1. Random Provider

	Support	Time Model			
		CONST	PROB	BATCH	WL
Time	0	3998	4029	3941	2206
	50	4024	3971	3905	2207
	90	4014	3994	4015	2220
	100	4034	4071	4043	2205
Success	0	9.0	10.3	10.8	9.2
	50	11.5	11.2	11.1	10.5
	90	11.0	10.8	10.9	11.2
	100	10.7	11.4	11.5	10.2

5.1 Random Provider

Our first test is more a sanity test to test that everything works; if tasks are picked at random with the same probability, it should not matter whether we use support or not. Furthermore, we expect the execution time to be the same for the timing models using constant time as a probability distribution. Finally, we expect the success rate to be nearly the same for all executions, independently of how much support is used and which timing model is used.

In Table 1 we see the results of our first experiments. The table is split in two: the top part shows the average execution time for a trace and the bottom part the success rate as defined above. We show results for each timing model (CONST = execution time for each task is a constant, PROB = execution is independent and identically distributed, BATCH = execution time is dependent on whether you execute the same task more than once, and WL = execution time decreases as stress increases) and for four different values of support. Here, 0% support means that we always chose at random and 100% support means we always ask operational support for advice. All numbers represent 1000 traces.

We see that the support percentage has no effect on either the execution time or the success rate for the tasks with the same time model. We also see that the CONST and PROB time models have the same behavior: the execution time and the success rate are very similar for those. The BATCH time model also has similar though slightly lower execution time whereas the WL time model has a significantly shorter average execution time. The reason for this is that they are allowed to execute tasks faster (if randomly batching or having more tasks in the work list). The qualitative measurement, the success rate does not change (as expected as we only change how long tasks take, not how they are selected).

5.2 Batch Provider

For our second test we want to evaluate the simple batch heuristics. We do not need to perform executions for 0% support (it is the same as the numbers in Table 1). We have also removed the simple time model, CONST as it yields the same results as PROB and is very far from reality. The results are summarized in Table 2.

Table 2. Batch Provider

	Support	Time Model		
		PROB	BATCH	WL
Time	50	4085	3996	2372
	90	4178	4095	2595
	100	4198	4134	2641
Success	50	11.5	11.8	11.5
	90	8.5	13.4	12.5
	100	12.8	15.4	14.4

Table 3. Model-specific Provider

	Support	Time Model		
		PROB	BATCH	WL
Time	50	4047	4001	2376
	90	3866	3766	2797
	100	3793	3711	2946
Success	50	31.0	30.8	32.3
	90	45.8	46.5	46.1
	100	47.3	51.9	51.2

We see generally and sometimes even significantly larger execution times using the batch provider compared to using the random provider (from Table 1). Surprisingly, we also see the execution time increase when the support rate goes up and even for the BATCH time model, which would be expected to benefit from batching. The reason is that while the batch provider recommends batching together similar tasks, it also suggests repeating working as we see in Fig. 2 Work and Master of BPM are the only tasks that can be executed more than once. Thus, this provider does not force progress, and may even prevent it, leading to longer execution times even though individual tasks are executed faster.

5.3 Model-Specific Provider

This is the same test as the one performed for the batch provider, except we now evaluate the provider specially tailored to our model. The results are summarized in Table 3.

We see that for the simple timing models, this provider significantly outperforms the previous, as it successfully picks the shortest path to an education. We also see that when timing is workload-dependent, this provider performs worse. The reason is that while the expected time to get a BSc is 3 years and the sum of the time to take the 6 courses (half a year each) and get a qualifying job (1 year) is 4 years, it is faster to take the 6 courses as the concurrent workload is faster. Thus our smart solution is suboptimal in this case. Regarding the qualitative results, we see that following recommendations leads to higher success rates. The reason the success rate approaches 50 and not 100 is that we do not control when an execution ends, so after executing MSc, BIS (or MSc, ES if we do not listen to operational support) we have a 50% chance of terminating the execution and a 50% chance of continuing.

5.4 Log Provider

Here we need some historical data. As this is a model created for demonstration purposes, we do not have any such data. We can, however, generate an execution log using our model. We simply allow CPN Tools to generate a simulation log and import it into ProM. In Tables 4 and 5 we see the results of using the log

Table 4. Log Provider (Running Time)

	Support	Time Model		
		PROB	BATCH	WL
Time	50	3073	3026	1674
	90	2191	2032	1110
	100	1957	1642	957
Success	50	9.6	5.7	7.0
	90	4.1	10.2	1.3
	100	0.4	13.6	0

Table 5. Log Provider (Success)

	Support	Time Model		
		PROB	BATCH	WL
Time	50	4096	4081	2327
	90	3920	4639	2278
	100	3832	4909	2257
Success	50	24.5	14.8	15.5
	90	45.8	28.7	17.5
	100	48.2	44.5	24.0

provider. We have in all cases used a log generated using random selection, but we make sure to use a log generated using the same time model as the one used to generate the results. We use the log provider to optimize for shortest execution time (Table 4) and for highest chance of obtaining a Master of BPM (Table. 5). This is a good indication of what happens when seeding a recommender with actual historical data from the group of people about to execute the process in a similar situation.

We see that when optimizing for shortest running time we in all cases obtain significantly shorter running time when using support than when we do not. Also, the running time is in all cases much shorter than for all previous providers. The success rate is very low, however. This is because it is possible to make a shorter run by picking MSc, ES as this does not enable Master of BPM. When optimizing for success, we see that the success rate increases and is as good as for the hand-crafted provider. The running time is higher than when we optimize for running time, but the success rate increases and is comparable to the success rate of the hand-crafted provider. The success rate for the workload timing model is surprisingly low, which is because the workload path favors the practical path, which needs to execute more tasks, and hence the historical data may not contain a trace with the same interleaving of the courses.

In Table 6 we have shown the results of runs using logs generated using a different timing model. This is an indication of how well a recommender seeded with randomly generated data using the correct model but with wrong assumptions about the user behavior. In other words, this is an indication of the stability of the simulation results. We have not shown the value of the success rate as it is the same as the one measured in Table 4 as we can become a Master of BPM using both the academic and practical track.

Table 6. Using foreign providers

Source	Time Model		
	PROB	BATCH	WL
PROB	1957	1772	1011
BATCH	1643	1642	1007
WL	1632	1609	957

We see that it does not matter much if the log is generated from data with a BATCH or WL time model. Surprisingly, we get the shortest execution times in all cases when using data generated with a timing model taking the workload into

account. In fact, the log generated using a simple probability measure performs worse even when used for the matching time model. This is because there are other logs favoring a trace that is also beneficial for this timing model.

6 Conclusion and Future Work

In this paper, we have presented a CPN model for testing operational support providers. The model is connected to the Declare workflow system and the operational support service in ProM using Access/CPN 2.0, making it possible to test real systems with a model of a user. Our user model is parameterizable, and can exhibit four different kinds of timing behavior. We have also presented a CPN model which can be used to quickly prototype an operational support provider for integration in ProM and subsequent use in both our testing platform and existing clients of the operational support service. We believe that using this approach is also useful in other settings where an algorithm modifies the domain it is doing computations on.

We have used our test suite to test four different recommendation providers. We see that contrary to what is often observed, simple algorithms fail compared to more sophisticated adaptive algorithms. We also see that a hand-crafted algorithm using domain knowledge does not necessarily outperform a smart general algorithm when the domain knowledge builds on wrong assumptions (here about the user behavior). Even simple algorithms designed to exploit certain traits of models (like speedup in batch processing) may fail if the assumption is correct, if some other aspect is ignored (like the need to work towards termination and not just focus on batching tasks together). We also see that our log provider, which uses historical data, is surprisingly stable and handles situations when input data does not completely correctly reflect reality, making seeding such algorithms with generated data possible. We also see that if the algorithm providing recommendations is not optimal, user deviations can be a good idea. Even for the log provider, which proved very efficient, deviations may benefit execution in the long run as users may reach completely new and more efficient ways of executing the process by chance, making it possible to provide better recommendations in the future.

Our experiments show that experimental results may deviate quite a bit from expectations, making testing invaluable. We of course need to validate that simulated results show the same tendencies as real life and future work includes testing recommendations in real life. We also see that the winner by far was the most sophisticated algorithm, making future research into even better algorithms very interesting. One caveat of the current implementation is that it completely fails to provide recommendations if the historical data does not contain a similar trace (which is quite likely after executing 5-8 tasks). This can be alleviated by using a model annotated with timing information for providing recommendations instead of just a flat log.

References

1. van der Aalst, W.M.P.: Process Mining-Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science - Research and Development 23(2), 99–113 (2009)
3. van der Aalst, W.M.P., Pesic, M., Song, M.S.: Beyond Process Mining: From the Past to Present and Future. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 38–52. Springer, Heidelberg (2010)
4. van der Aalst, W.M.P., Schonenberg, H., Song, M.S.: Time Prediction based on Process Mining. Information Systems 36(2), 450–475 (2011)
5. van der Aalst, W.M.P., Stahl, C., Westergaard, M.: Strategies for Modeling Complex Processes Using Colored Petri Nets. In: Jensen, K., Donatelli, S., Kleijn, J. (eds.) ToPNoC V. LNCS, vol. 6900, pp. 265–291. Springer, Heidelberg (2012)
6. CPN Tools (2012), `cpntools.org`
7. Declare (2012), `http://www.win.tue.nl/declare`
8. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modeling and Validation of Concurrent Systems. Springer (2009)
9. Mans, R.S., van der Aalst, W.M.P., Russell, N.C., Bakker, P.J.M., Moleman, A.J.: Process-Aware Information System Development for the Healthcare Domain - Consistency, Reliability, and Effectiveness. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009. LNBIP, vol. 43, pp. 635–646. Springer, Heidelberg (2010)
10. Nakatumba, J., van der Aalst, W.M.P.: Analyzing Resource Behavior Using Process Mining. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009. LNBIP, vol. 43, pp. 69–80. Springer, Heidelberg (2010)
11. Nakatumba, J., Westergaard, M., van der Aalst, W.M.P.: A Meta-model for Operational Support. BPM Center Report BPM-12-05, BPMcenter.org (2012)
12. Nakatumba, J., Westergaard, M., van der Aalst, W.M.P.: Generating Event Logs with Workload-Dependent Speeds from Simulation Models. In: Enterprise and Organizational Modeling and Simulation. LNBIP. Springer (2012)
13. ProM (2012), `http://www.processmining.org`
14. Resnick, P., Varian, H.R.: Recommender Systems. Comm. of the ACM 40(3), 56–58 (1997)
15. Rozinat, A., Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M., Fidge, C.: Workflow Simulation for Operational Decision Support. Data and Knowledge Engineering 68(9), 834–850 (2009)
16. Schonenberg, H., Weber, B., van Dongen, B.F., van der Aalst, W.M.P.: Supporting Flexible Processes through Recommendations Based on History. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 51–66. Springer, Heidelberg (2008)
17. Weber, B., Wild, W., Breu, R.: CBRFlow: Enabling Adaptive Workflow Management Through Conversational Case-Based Reasoning. In: Funk, P., González Calero, P.A. (eds.) ECCBR 2004. LNCS (LNAI), vol. 3155, pp. 434–448. Springer, Heidelberg (2004)
18. Westergaard, M.: Access/CPN 2.0: A High-level Interface to Coloured Petri Net Models. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 328–337. Springer, Heidelberg (2011)
19. Westergaard, M., Maggi, F.M.: Modeling and Verification of a Protocol for Operational Support Using Coloured Petri Nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 169–188. Springer, Heidelberg (2011)
20. Wickens, C.D.: Engineering Psychology and Human Performance. Harper (1992)