Supporting Dynamic, People-Driven Processes through Self-learning of Message Flows

Christoph Dorn[1,2] and Schahram Dustdar[2]

[1] Institute for Software Research, University of California, Irvine, CA 92697-3455
cdorn@uci.edu

[2] Distributed Systems Group, Vienna University of Technology, 1040 Vienna, Austria
lastname@infosys.tuwien.ac.at

Abstract. Flexibility and automatic learning are key aspects to support users in dynamic business environments such as value chains across SMEs or when organizing a large event. Process centric information systems need to adapt to changing environmental constraints as reflected in the user's behavior in order to provide suitable activity recommendations. This paper addresses the problem of automatically detecting and managing message flows in evolving people-driven processes. We introduce a probabilistic process model and message state model to learn message-activity dependencies, predict message occurrence, and keep the process model in line with real world user behavior. Our probabilistic process engine demonstrates rapid learning of message flow evolution while maintaining the quality of activity recommendations.

Keywords: message prediction, process log mining, people-driven processes, process evolution, message activity dependencies.

1 Introduction

Modern information systems need to enable flexibility and automatic adaptation capabilities in order to cope with continuously evolving environments where a-priori fixed requirements are rarely applicable. Organization of multi-national events such as the Olympic Games or management of value chains across a large set of Small and Medium-sized Enterprises (SMEs) are just two examples where exact work practices cannot be precisely defined and executed. In such environments, users engage in knowledge and coordination intensive workflows that are subject to continuous change. Processes evolve as participants tune their work practice to increase efficiency and effectiveness. Thus, users want to focus on their tasks rather than managing and updating their workflow. Instead the involved information systems should learn from and adapt to the users automatically.

In this paper, we address the case of people-driven dynamic processes. There exists a process model that describes the activities carried out by humans representing their expertise. Users are, however, completely free to deviate from the underlying model which cannot foresee all possible situations. These processes heavily rely on exchanging unstructured or semi-structured messages such as

H. Mouratidis and C. Rolland (Eds.): CAiSE 2011, LNCS 6741, pp. 657–671, 2011.

© Springer-Verlag Berlin Heidelberg 2011

emails. When organizing a large event, these messages are the main artifacts to coordinate between participants. While the necessary activities are relatively clear, the type of messages, their occurrence, and evolution will remain dynamic. We address three major problems (i) learning of message-activity dependencies, (ii) prediction of which messages will arrive, and (iii) automatically reflecting work evolution in the process model. The subsequent challenges are then to distinguish between accidental one-time deviations and desired process improvements, managing the interleaving of messages, and the inability to observe the complete set of user actions.

Our contributions in this paper are (i) a probabilistic process model and execution engine that does not rely on a precise occurrence or absence of messages; (ii) a self-learning (thus unsupervised) message flow algorithm to detect message-activity dependencies and updates thereof; and (iii) a message recommendation mechanism to support the analysis of unstructured and semi-structured messages. The main applied approaches are log sequence mining, and providing — respectively extracting — message-activity correlation information.

Our contributions bring benefit to both process designers and process users. Process designers need not capture all possible deviations nor the exact mapping of input and output messages to activities as this is automatically learned from process users. These in turn see the immediate effect of their applied expertise in form of process model changes without having to include a dedicated process designer.

The remainder of this paper is structured as follows: A motivating scenario sets the scene for our self-learning message flow algorithm (Section 1.1). We discuss related work in Section 2. Section 3 introduces our approach, followed by the probabilistic models in Section 4. Section 5 describes the recommendation and learning mechanism, which we evaluate in Section 6. We provide a short conclusion and outlook on future work in Section 7.

1.1 Motivating Scenario

The example in Figure 1 depicts a flexible people-driven order process. The individual work steps describe a general order of user activities to successfully complete a process instance. The outlined flow, however, does not enforce the exact order of activities, which is up to the user, and covers no exceptions or process adaptations that might arise due to specific customer request, incomplete information, or user specific expertise. Consequently, the listed document types that characterize the exchanged messages specify merely an initial set of expected documents. The process visualization in Figure 1 does not apply any particular process modeling language but rather presents an intuitive view on the involved documents that represent input and output of activities. In this scenario, we encounter various forms of message flow evolution:

Missing Documents : When the *Replenish* step (C) is updated to make use of an automatic restocking system, only user confirmation is required and the *Quote* message (3) no longer occurs.

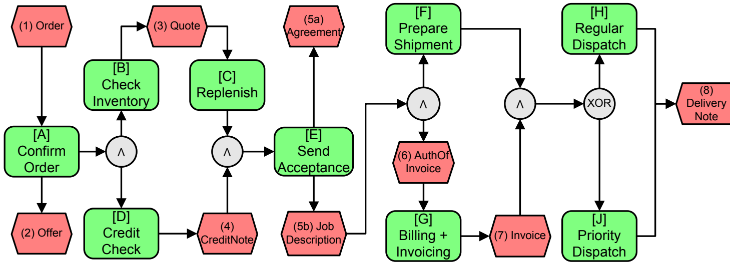

Fig. 1. Generic order process and associated document types

Delayed Documents : A company potentially decides to delay the *Agreement* (5a) message until the *PrepareShipment* (F) activity signals the completion of the production and packaging sub process. Such a company might depend on just-in-time delivery of subcontracted parts and thus cannot guarantee order completion any earlier.

Premature Documents : For premium customers, shipping becomes independent of billing thus the *DeliveryNote* message (8) potentially occurs before *Invoice* (7).

Shifted Documents : To guarantee that the *Invoice* (7) always exactly reflects the packaged goods, the ERP system no longer issues the *Authenticationof Invoice* message (6). Instead the *PrepareShipment* (F) step triggers this message.

2 Related Work

In the last decades considerable research effort was spent on systems for supporting flexible processes. In dynamic environments where business requirements continuously change, processes need to be able to adapt to fluctuating constraints. Processes cannot remain rigidly structured but need to support ad-hoc human control and evolve along alternative execution paths. Depending on the supported level of flexibility, we can distinguish between roughly three types of processes (including some example works):

Ad-hoc processes provide no constraints on the order of process activities and provide the user complete freedom of choice [9,6,20,3].

Semi-structured processes (or case-based processes [17]) contain some structure and capture best practices (e.g., from previous process instances) but are still too complex to be fully specified for automatic execution [18,2,15,19].

Well-structured processes are rigidly configured and determine for each condition the exact flow of control, data, and the involved resources and actors. User involvement is limited to human tasks (if at all) while process management and execution control is fully automated [16,1,14].

On the one hand, our approach incorporates aspects of all three process types. Our notion of people-driven process allows users to determine ad-hoc the process execution order. There exists, however, a well structured process model as a baseline guidance. Recommendations and learning is similar to semi-structured systems as we apply past execution traces to dynamically update the process model.

On the other hand, our approach deviates in several aspects from existing flexible process recommendation systems. We relax the assumption of complete observability of user actions and instead rely on a combination of observed messages and actions. The primary focus in this paper, however, lies on the self-adaptation of message flows, leaving a reordering and adjustment of process activities aside. We addressed this aspect of automatic people-driven process adaptation in our previous work [5]. In addition, we expect the message structure and thus the recognized types to change over time. Subsequently, we determine a dependency between message types and activities dynamically through analysis of execution traces. In a similar effort, Lakshmanan et al. [12] describe how an ant colony optimization algorithm learns dependencies of document contents (e.g., the impact of certain values within a message) to predict the flow and outcome of a process. They also apply exponential aging to keep the decision probabilities up to date. The underlying process model, however, remains unchanged.

Related work that explicitly applies autonomic computing principles [10] for adaptive workflow and process support systems react to system internal events such as workload fluctuations [8], goal changes [7], or service replacements [21] when executing well-structured processes. These approaches, however, target only system elements and offer no support for having the user dynamically adapt the process. In contrast we aim to apply those autonomic principles primarily for achieving unsupervised learning and user recommendation.

3 Approach

Successful activity recommendations in people-driven processes (i.e., the user actually carries out the proposed activity) depend on correct classification of incoming and outgoing messages and their associated activities. Supporting users in dynamically evolving processes consists of two aspects: (i) run-time tracing of process progress including probabilistic message prediction, and (ii) automatic refinement of process model and message probabilities upon process termination.

The main phases of the run-time recommendation support is depicted in Figure 2. The core of the *Probabilistic Process Engine* consists of the process model that describes the general structure and current interdependencies of steps and messages (see Sec. 4.1). The Message State Model captures the likelihood for a message to occur in that process on time, early, late, or repeatedly (see Sec. 4.2).

For each intercepted message and observed user action, the process engine extracts the correlation of messages and activities to determine which activity produced a particular message, and which message served as input to a given activity (1). Next, the engine analyzes whether the observed messages and actions

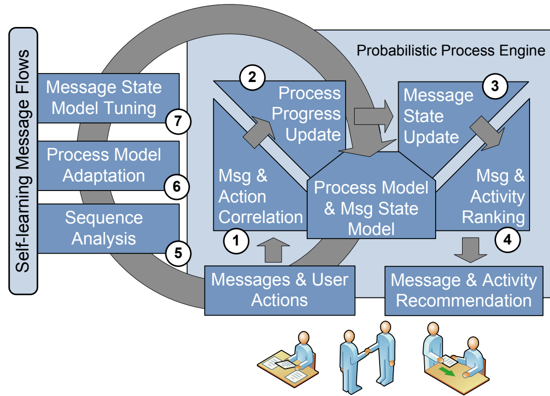

Fig. 2. Supporting message prediction and activity recommendation through self-learning message flows

advance the progress of the process (2). Any completed activity potentially results in an update of one or more messages, respectively their expectation states: messages are activated, become missing, or should no longer occur (3). Finally, the engine ranks messages according to their probability to occur, and activities to be carried out next (4). Activity recommendations support the user in carrying out his/her work. Message recommendations support the classification of unstructured and semi-structured messages that are exchanged between process participants.

After successful completion of a process instance, the mechanism we present in the following sections analyzes the sequence and timing of activities and messages (5) to update the process model (6) and message state model (7) to accurately reflect the changes in the real world.

Our approach specifically targets dynamic business environments that mostly rely on unstructured or semi-structured messaging to communicate and coordinate processes. SMEs and event organization usually coordinate and collaborate via email. Our approach relies on an infrastructure for intercepting those messages, extracting relevant information, and mapping those messages to document types. In our case, the EU FP7 Project Commius provides the necessary framework to extract information from emails and conduct a basic, process-unaware email content analysis. Details on the actual process orchestration as well as message interception, extraction, analysis, and user interfaces are outside the scope of this paper. The interested reader is referred to previous project-related publications [4,13,11]. Our prototype implements the algorithms and techniques introduced in this paper to support the email content analysis by determining the expected document types. The activity recommendations allow the annotation of emails with process-relevant information (as defined in the activity description) before forwarding them to the actual recipient.

4 Models

4.1 Probabilistic Process Model

In an environment where most communication happens via unstructured and semi-structured messages such as email, we cannot expect to have an integrated IT infrastructure (such as traditional process engines require) that allows a unified observation of all user actions. Likewise, we cannot assume a tight coupling of process activities and corresponding message types.

Our process model (see also Figure 3 left side) consists of *ActivityNodes* and *Connectors* which link multiple activities together via *FlowDirection* arcs. Activity nodes, connectors, and arcs form a directed acyclical graph where activity nodes always exhibit exactly one incoming and one outgoing arc. Only connectors potentially have in- and out-degree above 1. The *FlowData* annotations of the *FlowDirection* arcs describe the corresponding activity input and output messages. The *Occurence* attribute describes in the interval $[0, 1]$ the likelihood a message of the stated type will arrive. A connector describes how multiple activities are interlinked applying the traditional flow constructs of AND, OR, XOR splits, respectively joins. An activity consists of one or more *Actions* the user needs to carry out in order to consider that activity as completed. Alternatively, we consider an activity successfully performed when all specified output *Messages* have been transmitted.

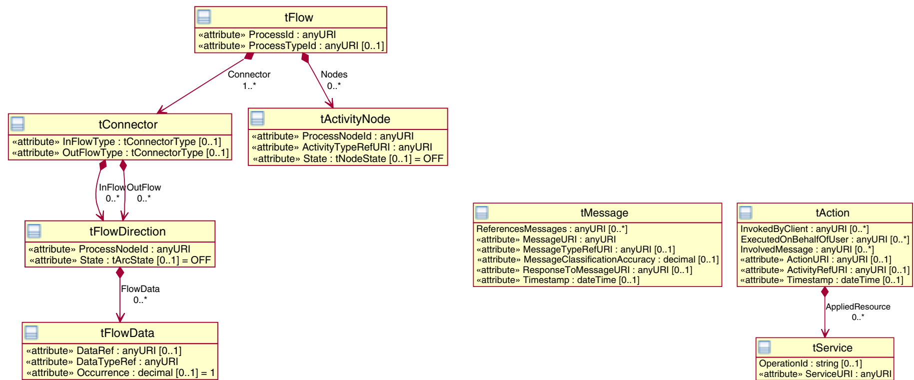

Fig. 3. Flow Model supporting the annotation of transitions with probabilistic message types; Action model and Message meta data model for correlating messages and activities

NodeStates (*Off*, *Active*, *Completed*, *Skipped*) and *ArcStates* (*Off*, *Active*, *Selected*) track the process progress and determine which subsequent arcs and nodes describe upcoming messages to expect and activities to carry out. The flow model describes the involved messages types and their probability to occur on a particular place in the process. The overall probability that a message will arrive at a given point in time, however, is hard to establish. We, therefore, pair the flow model with a *Probabilistic Message State Model* for each involved message type.

4.2 Probabilistic Message State Model

The message state model describes the possible message states (such as scheduled, expected, arrived) and the likelihood for transitions between those states during the process. We track only messages that are defined in the process model, each in a separate state model instance that is valid only for that particular process type and message type.

The state model is defined as a directed graph $G_{State}(V, E)$ where states are represented by vertices ($v \in V$) and labeled edges ($e \in E$) represent transitions. A labeled transition between neighboring states describes the probability that this transition will occur ($w(v_i, v_j) = [0, 1]$). The sum of outgoing transition probabilities is always 1. The state model can be interpreted as a Markov chain that describes the probability of reaching a particular state when a transition occurs. We neglect self-transitions as the duration a message type remains in a certain state is irrelevant for our purpose. We are only interested in the probabilities of reaching each of the subsequent states.

State transitions from *Start* to *Scheduled*, *Expected*, *Missing*, and *NotExpected* are driven by the process progress. Transitions to any *Received* state and to *Repeated* are message driven. The process termination finally triggers the transition to *Occurred* and *NotOccurred*. The initial transition probabilities (depicted in Figure 4) are optimistic: we assume a message to occur exactly once when its *Expected* or *Missing*, and when *Not Expected* to remain absent.

Scheduled. Initially every message is scheduled to take place at some time during the process lifetime.

Expected. A message becomes expected once it is needed to activate an activity or an activity is activated and is expected to produce the message as output.

Missing. Whenever an activity is completed without the required input, we mark its input message as missing. Likewise, we also mark any expected output message of an active preceding activity as missing.

Not Expected. When a user explicitly skips an activity, or an alternative XOR branch is completed — thereby skipping the involved activities — any message type to or from such activity is no longer expected.

Received Early. A message arriving while in state *Scheduled* indicates a potential change in the process model or just a one-time deviation.

Received On Time. A message arrives on time according to the process model.

Received Late. The arrival of a missing message indicates a potential change in the process model or just a one-time deviation.

Received Unexpected. Indicates that a skipped activity needed to be carried out anyway or a parallel branch is incorrectly specified as XOR.

Repeated. A message is received multiple times.

Occurred. The message has been sent or received at process end.

Never Occurred. The message did not occur for whatever reasons.

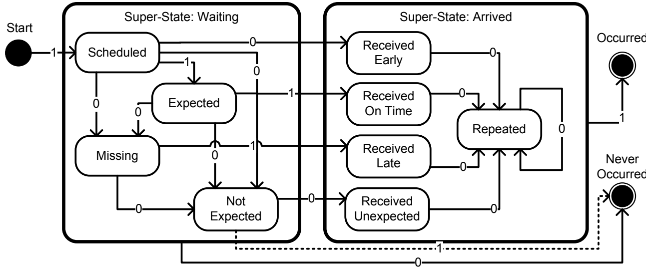

Fig. 4. State Transition model for a single message type. Each model instance provides for a message type and process model the specific transition probabilities

5 Prediction and Self-adjusting Mechanisms

5.1 Probabilistic Process Management

Messages in form of *FlowData* annotations play a central role in the progress tracking and process step activation (and ultimately recommendation). Messages often define the trigger condition for an activity. For example in the scenario: *Quote* (3) is required to continue with *Replenish* (C) but does not arrive for whatever reasons (unobservable communication channel, delayed arrival, or simply no longer necessary) it will halt the process progress. The recommendation mechanism would not continue to suggest to execute inactive process step (C). We, therefore, need a mechanism to decide when the process engine should wait for a message to arrive, and when to continue with analyzing and activating the subsequent *FlowDirection* arcs and process steps.

Whenever the probabilistic flow engine checks a *FlowDirection* arc for activation, it compiles a list of available M_a and missing M_m messages for that arc. For all messages, we calculate the sum of probabilities $p_{total} = \sum p_{occur}(msg \in M_m \vee M_a)$ and additionally the likelihood $p_{miss} = \sum p_{occur}(msg \in M_m)$ that at least one of the missing messages will arrive. We then apply following rules to determine whether the message conditions on an arc can be considered satisfied (GO) or not ($NOGO$):

1. $NOGO$ if $p_{total} == 1 \bigvee |M_a| < 1$: the first condition describes a set of alternative messages and if non of those have yet arrive (second condition) we continue to wait.
2. $NOGO$ if at least one missing message ($msg \in M_m$) always arrives ($p_{occur}(msg) == 1$) we continue to wait.
3. GO if $|M_m| == \emptyset$: no message is missing.
4. GO if $p_{total} == |M_a|$: its sufficient when the arrived messages cover the probability of all documents.
5. GO if $Random(0,1) > p_{miss}$ else $NOGO$: if the set of messages neither constitute XOR alternatives, nor a compulsory AND set, but one (or several) out of many we apply a random value from the interval $[0,1]$. If that random value is larger than the probability that any missing message will

arrive p_{miss}, then we consider the *FlowData* as satisfied and activate the corresponding arc.

Let is consider the incoming arc of *Billing and Invoicing* [G] from the scenario: messages (5b) and (6) always occur thus their probability is 1. Assume (5b) arrives then rule 2 applies, as we are still waiting for (6). Once message (6) arrives, rule 3 would consider the arc conditions satisfied and subsequently enable activity [G].

When the observed messages suffice to activate the underlying arc, we iterate through all non-observed alternative messages (that still reside in state *Scheduled*, *Expected*, or *Missing*) and switch them to *NotExpected* as we no longer require their appearance.

5.2 Message Prediction

Message prediction is purely based on the transition labels between message states. In any of the *Waiting* states the probability is determined based on the transition to the respective received state. From any of the *Arrived* states the probability is determined by the transition weight to the *Repeated* state.

After any change to any of the messages' state model, the messages are ranked according to their probability to occur in their current state. The top ranked message(s) constitute the prediction and are applied during the classification of newly intercepted messages. Suppose we have two messages and their corresponding message state models depicted in Figure 5(a) and (b): the first one currently in state *Expected*, the second in state *Missing*. Hence, we would predict the occurrence of message (b) with $p_{occur}(b) = 0.9$ rather than message (a) with $p_{occur}(a) = 0.7$. We are thus able to address and manage messages that over multiple process instances no longer adhere to the process model but have become delayed.

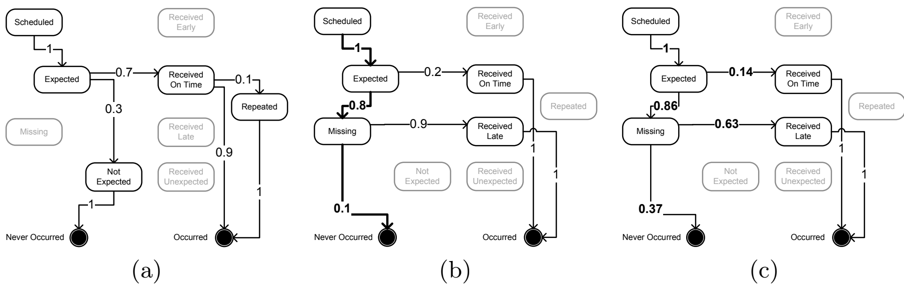

Fig. 5. Examples of message state model instances: only non-zero transitions are included for sake of clarity. Subfigure (a) displays a message that when expected occurs with 70% probability and is repeated in 10% of process cases. Subfigure (b) describes a message that occurs only 20% on time, but still arrives in 90% of all cases when missing. The effect on the transition probabilities of that message never occurring (thick lines and labels) in a single process instance is depicted in subfigure (c).

5.3 Message State Adjustment

After successful process termination, the message state model of each message in the scope of that process is adjusted. During the process execution, each model stores the sequence of message state transition (Seq_{ST}). All transition leaving any state listed in the sequence set are processed so their transition weights reflect the latest process instance. We apply an exponentially weighted moving average (EWMA) to update the transition probabilities.

$$w_{t+1}(v_i, v_j) = \begin{cases} 1 * \alpha + (1 - \alpha) * w_t(v_i, v_j) & \text{if } v_i, v_j \in Seq_{ST}, \\ 0 * \alpha + (1 - \alpha) * w_t(v_i, v_j) & \text{if } v_i, v_j \notin Seq_{ST}. \end{cases}$$

With EWMA, old process sequences have exponentially lower impact on the new probability value than more recent sequences. The coefficient α determines the significance of older values. With α close to 0, new values have next to no impact on the new probability and vice versa for α close to 1. As the updated probability depends only on the previous probability and the current state sequences, little memory is required to store the transition labels. Figure 5 (c) display the transition update result for the underlying message state after the state sequences in subfigure (b) are evaluated.

5.4 Self-learning Message Flows

After each process, we analyze the order in which activities and messages have occurred and compare it to the underlying process model (PM). Any message deviation needs to become reflected in the process model, e.g., a new message type emerged, an activity did not produce the expected message, a message did not serve as input as expected. During process execution, we captured the sequence of all message-activity dependency tuples ($tup(msg, act) \in Seq_{MA}$) by extracting message references from user actions, and activity references from messages. Algorithm 1 describes the technique for updating a process model's *FlowData*. For each tuple, we step through the process model in breadth-first style (lines 4-7, 20-21) and locate the corresponding *FlowDirection* arc and *FlowData* annotation (lines 8-11, 19). Once found, we increase the *FlowData* occurrence value ($occ(fd, mt)$) using exponentially weighted moving average (EWMA) (lines 12-15). At the end, *FlowData* annotations that are not covered by sequence tuples ($tup(msg, act)$) receive a lower occurrence value (lines 22-23):

$$occ_{t+1}(fd, mt) = \begin{cases} 1 * \beta + (1 - \beta) * occ_t(fd, mt) & \text{if } \exists tup(msg, act) \equiv tup(fd, mt) \\ & : tup(msg, act) \in Seq_{MA}, \\ 0 * \beta + (1 - \beta) * occ_t(fd, mt) & \text{otherwise} \end{cases}$$

where the tuple $tup(fd, mt)$ describes a *FlowDirection* fd with a *FlowData* annotation referencing message type mt. Coefficient β determines again how quickly a new message emerges or disappears on an arc.

Message types that show up for the first time, or messages types that arrive early result in a new *FlowData* annotation (lines 16-18). In the latter case, we reduce the original annotation occurrence value in addition. Message types

that take place at a later stage of the process than specified receive the same treatment. We reduce their initial annotation value, and create a new *FlowData* annotation where we actually observed the message. When we have not observed a message type at all, we reduce all instances of the respective *FlowData* annotation. At the end, we clean the process model from rarely used *FlowData* annotations that would otherwise interfere with the message state model as message would be activated too early or remain expected too long (line 24). Experiments have identified a suitable cutoff value of 0.1. Arcs, however, that are inactive at the end of the process (due to explicit user skipping or non executed XOR branches) need no refreshing and are ignored. We insure this in Algorithm 1 by providing only the set of active *FlowDirections* in FD_{active} in the first place.

Algorithm 1. Self-learning Dependencies Algorithm $\mathcal{A}(PM, Seq_{AM}, FD_{active})$.

1: **for all** $tup(msg, act) \in Seq_{AM}$ **do** ▷ For each activity-message tuple.
2: $N \leftarrow PM.startNode$ ▷ List of nodes that we haven't checked yet
3: $found \leftarrow false$
4: **while** $!found \lor !N.empty$ **do** ▷ While not found and not at the process end
5: **for** $i = N.size \rightarrow 0$ **do**
6: $Node\ n \leftarrow N_i$
7: $N \leftarrow N - n$
8: **for all** $FlowDirection\ df \in n.outFlow()$ **do**
9: **if** $fd.getActivity == act$ **then**
10: $actOk \leftarrow true$ ▷ Found the correct activity
11: $FlowData\ data == fd.get(msg)$
12: **if** $data \neq null \lor actOk \lor fd.dir == tup(msg, act).dir$ **then**
13: $increaseByEWMA(data)$
14: $FD_{active} \leftarrow FD_{active} - data$
15: $msgOk \leftarrow true$
16: **if** $actOk \lor !msgOk \lor fd.dir == tup(msg, act).dir$ **then**
17: $fd \leftarrow fd{+}FlowData(msg)$ ▷ Add a new *FlowData* to the arc
18: $msgOk \leftarrow true$
19: $found \leftarrow actOk \lor msgOk$
20: **if** $!found$ **then** ▷ Adding the next set of nodes to the search list
21: $N \leftarrow n.getSuccessorNodes()$
22: **for all** $FlowDatadata \in FD_{active}$ **do**
23: $reduceByEWMA(data)$
24: $removeFlowData(cutoffValue)$ ▷ remove rarely occurring *FlowData*.

6 Evaluation

We evaluate our approach based on the motivating scenario in Section 1.1. Specifically, we are interested in the time it takes to learn an evolved message flow given a fixed process model. In addition, we observe how the updated message states and *FlowData* annotation affect activity recommendations. We simulate

user behavior though prepared log sequences that consist of activities and messages. The sequences serve as input to the Probabilistic Process Engine which is initiated with the original scenario process.

6.1 Experiment Setup and Success Metrics

Two quality metrics measure the success of our learning and adaptation mechanism. The Message Classification Error (MCE in range $[0,1]$) determines for each incoming unknown message during the experiments how close our message prediction algorithm gets. $MCE = 0$ when the actual message type and highest ranked predicted message type are identical, otherwise we extract the matching message type from the ranking list and take the inverse of its expected occurrence probability $1 - p_{occur}$) (i.e., the lower its probability, the higher the MCE). Suppose for an incoming message the algorithm produces following ranking result $[(7) : 0.5], [(8) : 0.4], [(2) : 0.1]$: MCE would yield 0.6 when the message is actually of type (8). For measuring the effect towards the user, we apply the Activity Recommendation Error (ARE in range $[0,1]$) which is analogue to the MCE: when the next element in the log sequence is an activity, we retrieve an activity recommendation and locate the matching activity. We keep the activity recommendation mechanism intentionally simple to focus only on the effect of the message state model and probabilistic flow model. Active activities are ranked based on their time since activation, thus the longer an activity remains unfinished, the higher it will score.

Two activity message log sequences describe an evolution of the initial process model. The *Quote* (3) message is no longer used, the *Agreement* (5a) message is delayed until completion of the *PrepareShipment* (F) activity which also triggers the *Authorization Of Invoice* message (6). The *Invoice* (7) is produced when executing *Priority Dispatch* (J). The *Delivery Note* (8) only applies to *Regular Dispatch* (H). During all experiments, we set EWMA coefficients $\alpha = 0.3$ and $\beta = 0.4$ which is a trade-off between rapid uptake of novel user behavior and robustness against one-time deviations. We set $\beta > \alpha$ as the arcs need to learn and forget message changes quicker than the state model (which tracks the message probabilities across the whole process and not just a single, local arc occurrence).
Log Sequence A: 1, A, 2, B, C, D, 4, E, 5b, F, 6, 5a, G, J, 7
Log Sequence B: 1, A, 2, D, 4, B, C, E, 5b, F, 6, 5a, G, H, 8

6.2 Results

In experiment 1 (Figure 6 a and c), we play each of the log sequences 30 times against the scenario process (dotted lines, green +, and blue x) — as well as alternating each sequence (full lines, red circles) — and determine overall MCE_p and ARE_p (sum of all recommendation errors within one process instance).

MCE_p rapidly drops to zero within a few iterations for individual log sequences A and B. The interleaving sequences take longer to produce a stable process model as sequences A and B display an opposing user behavior (to the

extend allowed by the process model). ARE_p shows similar behavior and settles to normal error rates once the probabilistic message states and flow model updates have settled. The consistent error rates above zero are due to the simple activity recommendation algorithm. Activities at branching points receive almost equal probabilities, and the log sequences happen to prefer the second highest choice.

In experiment 2 (Figure 6 b and d), we apply the same sequences but take an empty process model i.e., all *FlowData* annotations are on the first arc towards activity (A). As expected, MCE_p and ARE_p are high during the first few iterations, but quickly decrease to low error rates and then settle to the same rates as the evolved process model in experiment 1.

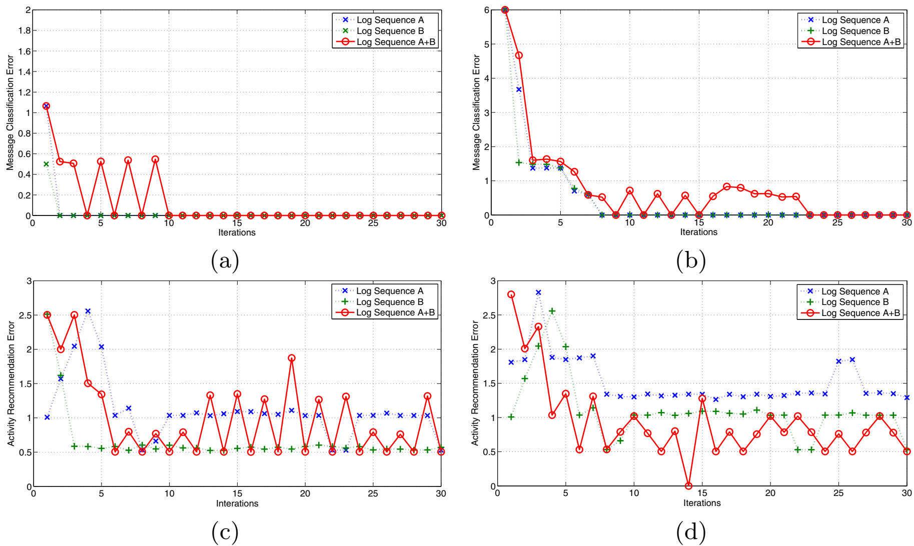

Fig. 6. Overall Message Classification Error MCE_p (a,b) and Activity Classification Error ARE_p (c,d) over 30 process iterations for fixed and alternating log sequences; for existing (a,c) and empty process model (b,d)

6.3 Discussion

Two important characteristics describe our self-learning approach. First, the process model and state management model allow a quick stabilization which reflects in the low prediction and recommendation error rates. As a positive side-effect, the user is hardly affected by incorrect activity recommendation. Second, the described techniques apply not only to cases of process evolution but also efficiently support a grass-roots approach to processes learning. We observe similarly swift adaptation when no message-activity dependencies are a-priori known. The applied 30 iterations are sufficient to demonstrate the learning behavior as our algorithm considers all changes simultaneously. The presented

results are still valid for larger processes where individual segments undergo evolution one at a time. In that case our results describe the behavior for a single segment.

7 Conclusion

Users in collaboration and coordination intensive people-driven processes require self-learning mechanisms to reflect the evolution of message types in the underlying process model. In this paper we presented an approach based on a probabilistic process model and message state model to predict when and which messages will arrive to ultimately give suitable activity recommendations. Analysis of sequence logs containing both messages and activities allows updating the process model automatically. Evaluation based on a motivating scenario demonstrated successfully that our mechanisms work correctly and efficiently.

Future work consists of two main tasks: on the one hand we aim at improving the message prediction algorithm by extracting message patterns such as request-reply from interaction logs, including temporal aspects, and integrating the process learning techniques introduced in our previous work. Pairing the probabilistic algorithms with semantic message analysis is expected to further improve the classification success rate. On the other hand, we plan to conduct additional user studies based on a larger process and log sequences set.

Acknowledgment. This work has been partially supported by the EU STREP project Commius (FP7-213876) and Austrian Science Fund (FWF) J3068-N23.

References

1. Adams, M., Edmond, D., ter Hofstede, A.H.M.: The application of activity theory to dynamic workflow adaptation issues. In: 7th Pacific Asia Conference on Information Systems, pp. 1836–1852 (2003)
2. Adams, M., Hofstede, A., Edmond, D., van der Aalst, W.: Facilitating flexibility and dynamic exception handling in workflows through worklets. In: Proceedings of the CAiSE 2005 Forum, FEUP, pp. 45–50 (2005)
3. Burkhart, T., Loos, P.: Flexible business processes - evaluation of current approaches. Proceedings Multikonferenz Wirtschaftsinformatik, MKWI-2010 (February 2010)
4. Burkhart, T., Werth, D., Loos, P.: Commius – An Email Based Interoperability Solution Tailored For SMEs. Journal Of Digital Information Management 6 (2008)
5. Dorn, C., Burkhart, T., Werth, D., Dustdar, S.: Self-adjusting recommendations for people-driven ad-hoc processes. In: Proceedings of International Conference on Business Process Modelling. Springer, Heidelberg (September 2010)
6. Dustdar, S.: Caramba Process-Aware Collaboration System Supporting Ad hoc and Collaborative Processes in Virtual Teams. Distributed Parallel Databases 15(1), 45–66 (2004)
7. Greenwood, D., Rimassa, G.: Autonomic goal-oriented business process management. In: ICAS 2007: Proceedings of the Third International Conference on Autonomic and Autonomous Systems, p. 43. IEEE Computer Society, Washington, DC, USA (2007)

8. Heinis, T., Pautasso, C., Alonso, G.: Design and evaluation of an autonomic workflow engine. In: Proceedings of the Second International Conference on Automatic Computing, pp. 27–38. IEEE Computer Society, Washington, DC, USA (2005), http://portal.acm.org/citation.cfm?id=1078027.1078524
9. Huth, C., Erdmann, I., Nastansky, L.: Groupprocess: Using process knowledge from the participative design and practical operation of ad hoc processes for the design of structured workflows. In: HICSS (2001)
10. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003), http://dx.doi.org/10.1109/MC.2003.1160055
11. Laclavik, M., Dlugolinsky, S., Seleng, M., Kvassay, M., Gatial, E., Balogh, Z., Hluchy, L.: Email analysis and information extraction for enterprise benefit. Computing and Informatics Journal, Special Issue on Business Collaboration Support for Micro, Small, and Medium-Sized Enterprises 30(1), 57–78 (2011)
12. Lakshmanan, G.T., Duan, S., Keyser, P.T., Khalaf, R., Curbera, F.: A heuristic approach for making predictions for semi- structured case oriented business processes. In: Proceedings of First Workshop on Traceability and Compliance of Semi-Structured Processes @BPM 2010. Springer, Heidelberg (2010)
13. Marín, C.A., Stalker, I.D., Mehandjiev, N.: Engineering business ecosystems using environment-mediated interactions. In: Weyns, D., Brueckner, S.A., Demazeau, Y. (eds.) EEMMAS 2007. LNCS (LNAI), vol. 5049, pp. 240–258. Springer, Heidelberg (2008)
14. Müller, R., Greiner, U., Rahm, E.: $A_{gent}w_{ork}$: a workflow system supporting rule-based workflow adaptation. Data Knowl. Eng. 51(2), 223–256 (2004)
15. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. In: Business Process Management Workshops, pp. 169–180 (2006)
16. Reichert, M., Rinderle, S., Dadam, P.: Adept workflow management system: flexible support for enterprise-wide business processes. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 370–379. Springer, Heidelberg (2003)
17. Reijers, H., Rigter, J., Aalst, W.V.D.: The case handling case. International Journal of Cooperative Information Systems 12, 365–391 (2003)
18. Sadiq, S., Sadiq, W., Orlowska, M.: Pockets of flexibility in workflow specifications. In: Proc. ER 2001 Conf., pp. 513–526 (2001)
19. Schonenberg, H., Weber, B., van Dongen, B., van der Aalst, W.: 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)
20. Stoitsev, T., Scheidl, S., Spahn, M.: A framework for light-weight composition and management of ad-hoc business processes. In: Winckler, M., Johnson, H. (eds.) TAMODIA 2007. LNCS, vol. 4849, pp. 213–226. Springer, Heidelberg (2007)
21. Yu, T., Lin, K.J.: Adaptive algorithms for finding replacement services in autonomic distributed business processes. In: Proceedings of the Autonomous Decentralized Systems, ISADS 2005, pp. 427–434 (April 2005)