Proceedings of I-KNOW ’08 and I-MEDIA '08
Graz, Austria, September 3-5, 2008

KAJAL – An Algorithm for Workflow Navigation in Informal Learning

Eicke Godehardt, Markus Doehring, Andreas Faatz, Manuel Goertz
(SAP AG, Germany, SAP Research CEC Darmstadt, Germany
{eicke.godehardt, markus.doehring, andreas.faatz, manuel.goertz}@sap.com)

Abstract: The more information a knowledge worker has to handle, the more (informal) learning is needed right at the workplace. This paper describes an algorithm that enables users to explore their current workflow, jump and navigate through their own tasks (even future ones), for informal learning or help purposes. This is done by overlaying the real workflow with a cloned one, providing an interface to access additional information, like prerequisites, associated documents or expert contacts. In particular, we developed KAJAL, which is a Petri net formalism to cope with the question of determining the states after a jump to a future task was triggered during the user’s navigation.

Keywords: informal Learning, workflow, workplace-embedded learning
Categories: L.3.2, L.3.4, L.3.6

1 Introduction

Informal e-learning [Ley05] means learning activities without a pre-defined support by electronic curricula or textbooks. Informal learning environments like the prototypes described in [Lin05] can be closely embedded into the working place and can be related to the order of tasks a worker wishes to fulfill on the job. That means that a worker’s workflow guides the training measures, i.e., tasks in that workflow have a connection to electronic learning material. Informal e-learning is at the frontier to context-aware help, which denotes any kind of IT-based help system, that changes due to the state an application is in. This context-awareness can be established/ expressed by the state of a workflow.

This paper aims at navigation in informal learning environments. The strict conduction of a workflow is supplemented by a modus, which allows the exploration of a workflow keeping as most as possible of the current workflow state a worker is in.

Consider for example the following workflow in figure 1 as a possible learn and work status. Each task might have documents attached to it, which explain or train the purpose and the actions of that task.

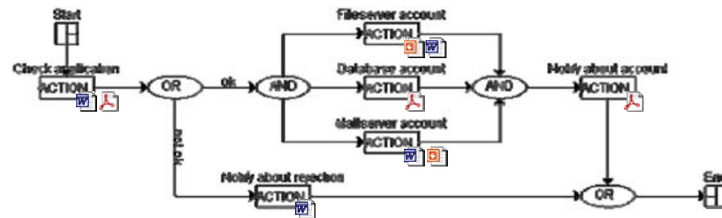

Figure 1: Example workflow with associated documents or learning resources per task (source: http://www.ultraflow.de)

Its middle parts include actions regarding fileserver, database and mail server accounts. These actions are only started, if “check application” returned the state “ok”. Even if a worker is actually in the middle of such a workflow, s/he sometimes requests training material about a future task (and they really do), which might leave the strict conduction of the workflow. The offered training material could still be provided by an LMS (learning management system) and the effect on the user’s skill will be the same, no matter if the user is in working mode or just learning.

KAJAL (**k**eeping states in **a**rbitrary **j**umps **a**nd **l**earn), the system described in this paper, generates a view of the selected task, which is as similar as possible to the worker’s workflow view before the selection. For example, if s/he already checked the fileserver and database account, the preferable situation after the jump to “Notify about account” will be resulted from the assumption, that the mail server account has also been checked – instead of assuming, that the worker will follow the bottom path of the workflow and undertake more different actions. KAJAL presents an approach to formalize this notion exactly and restrict the complexity of the decision, what a state-based similarity of workflows means.

Similarly, if a context-aware help system or informal e-learning material is depending on the working task, the worker might prefer a kind of help, which considers her/his current context but also keeps the logic of the workflow. Thus by jumping into a task for explorative reasons the help under the operating KAJAL would not change disruptively. For instance, if the user never passed the branch in the above illustration, which is about “notify about rejection”, KAJAL would prefer states of the workflow resulting from the passage of the three branches in the middle of the illustration.

This paper describes a formalization of the approach. It is based on state-based representations of workflows. An example for state-based representations of workflows is YAWL, which we will use for illustration. Workflows in any other workflow language, which can be translated to Petri nets, can be subject to KAJAL as well.

The presentation of KAJAL’s results will always be shown by cloning the actual view or screen, i.e., keeping the original graphical user interface and opening a second one, where an interaction under the assumptions and results of KAJAL is possible. This is depicted in figure 2, where the view is cloned from the original view. The target view shows the position of the jump (marked by a dotted frame), while the

initial view shows the initial position (marked by a frame). KAJAL provides the background states for the target view.

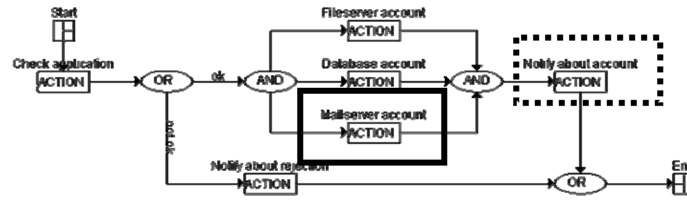

Figure 2: Cloned view on the workflow, to which KAJAL computes states in the background (source: http://www.ultraflow.de)

This paper is organized as follows. We will introduce basic workflow constructs and definitions first. After presenting related work, we will explain our algorithm. At the end we will finish with a conclusion and future work.

2 Workflows: basic concepts and definitions

We first want to define several (workflow) concepts, especially from the YAWL point of view.

Workflow: We explain the concept of a workflow by the example of YAWL [AH05]. Basically, a workflow in YAWL consists of tasks ordered to sequences, parallelism and controlled branches by AND, OR and XOR gates. This set up and the following constructs make YAWL translatable to a Petri Net [Aal02]. Note that we use only the control flow part of workflows (data passed between tasks is not taken into account).

Task: A task is a working step. Tasks can be compared to a transition in Petri-Nets, including the firing behavior (consumption and production of tokens). In YAWL, we can assign an action, e.g., a human, to a task.

Composite Task/Subnet: It is possible in YAWL to structure workflow definitions hierarchically. This is done by means of composite tasks, which act as a placeholder for other autonomous workflow definitions. The autonomous workflow definition in a composite task is also called a subnet.

Condition: Similar to conditions in Petri-Nets, YAWL conditions represent states within a workflow. If two tasks are directly connected with an arc, conditions can be seen as implicitly presented between these two tasks. In YAWL, the extended concept of Colored Petri-Nets (see e.g. [Jen97]) is used. That means a condition may contain multiple tokens that carry different information (e.g., necessary to distinguish between multiple workflow instances).

User context: Any information that characterizes the user’s current situation and that is important for the interaction between the user and the application. Context-aware systems use this context data to offer relevant information and/or services.

3 Related work

There are some related research projects in using workflows for learning. But none of them is workplace embedded or focusing on informal learning based on workflows. Cesarini et. al. [Ces04] uses workflows to model and structure e-learning content. In a similar manner [Kwa05] tries to use the greater flexibility for structuring e-learning content to enhance SCORM based content. Another project [Bru97] is also not workplace-embedded, but uses workflows to model traditional learning content to make it more adaptive. All of these approaches do not have the goal to integrate learning into the workplace and thus still stick to fixed roles for learners and teachers in a dedicated e-learning environment. Also APOSDLE [LGF07, GDF07] as a prototype of an informal learning environment that tries to bring these different roles of work, learn and teach together, has until now no mechanisms like the ones described in this paper, as the strictness of the workflows applied there is rather weak from a control flow perspective. Consequently, no system like KAJAL was designed for these environments.

In the domain of context-sensitive help the works of [DJH04] has proven fruitful, but was rather applied to classical tutorial situations than to workflow-oriented help, i.e., concrete workflow models are not taken into account.

4 Description of the algorithm

4.1 Assumptions and principles

We assume that the user follows a workflow and has a graphical view (i.e., screen) about his/her possibilities. The example graphical view is YAWL, but KAJAL works for any workflow language, which can be formally expressed as a Petri net.

The normal course of following a workflow is that the worker/user processes the enabled tasks, enabling new tasks, which can be processed again. The goal of KAJAL is to offer an alternative modus of operating the workflow: leaving the enabled tasks, i.e., jumping to a task, which is not necessarily enabled. The alternative modus triggered by a jump and computed by KAJAL is displayed on a second screen or view as depicted in figure 2. In the above example, if the user would jump from "ACTION: file server account" to "ACTION: notify about account", a screen for the initial state of the workflow at "ACTION: file server account" would remain and a second screen at "ACTION: notify about account" resulting from KAJAL computations would open. This process walkthrough for learning purposes is intended to provide a feature for the informative examination of workflows (in contrast to live-execution). The user instantiates a kind of "dummy workflow" and navigates through it for a better understanding of the process itself, its internal dependencies and its corresponding work items. For example, s/he could select a task and the system would show up the next possible steps s/he could take subsequently. By manually selecting process steps which seem interesting to him/her, s/he is also able to access the linked material and expert contacts. With the help of the desired "arbitrary jumping" functionality, s/he does not have to run through the whole process if – for instance – s/he is interested in a work item at the end of the workflow. Besides this benefit for jumping once, KAJAL can be exploited for the design of a context-sensitive help: typically, textual

help descriptions give the opportunity to jump in many different tasks the user has not been into. KAJAL will open the respective "dummy workflow" finding the most similar overall picture of the workflow in terms of conditions. All this only holds, if the net never deadlock and do not contain any tasks which are never enabled.

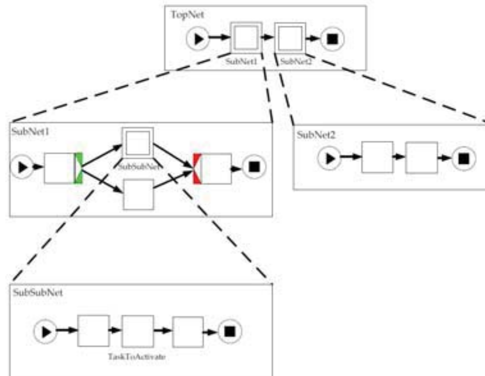

Figure 3: Example hierarchical workflow

4.2 System

We assume that the system starts with no input information except the concerned task ID (indicating the intended task to jump to) and the instance of the workflow. Since we assume that the target task ID only exists once, the reuse of decompositions (i.e., the assignment of single nets to multiple composite tasks) has to be prohibited.

The initial step now is to determine where the task is exactly situated within the workflow hierarchy of YAWL nets. To give a comprehensive example for the reader, figure 3 contains a hierarchical YAWL workflow. The tree consists of three stages, and four nodes. If we want to enable the "task-ToActivate" for instance, this presumes that the composite tasks "Subnet1" and "SubSubNet" are currently executing. If one of those nets is inactive, any attempt to enable the taskToActivate is doomed to failure. That is the reason why it is necessary to gather all involved nets from the root net of the workflow down to the net which contains the target atomic task.

A recursive search is performed to traverse the hierarchy and to determine the path to the desired task. As an example, the determined path for the TaskToActivate of figure 3 would be TopNet → SubNet1 → SubSubNet. The resulting path to the jump task is run through from top to bottom. For each net, three main steps are executed which are described below. The first step is to determine all "allowed" states of the currently processed net. Figure 4 contains a simple YAWL net which serves to demonstrate the characteristics of such a state more specifically. In this figure, A, C and D are OR splits/joins while B and E are AND splits/joins. Please note, that the conditions c1,c2,c3,c4,c5,c6 and c7 do normally not have to be explicitly displayed within a YAWL diagram (see above). For our demonstration purposes it is anyhow inalienable to imply them in the diagram.

The state of a YAWL workflow net can be described in terms of the amount of tokens within each condition. The notation "nci" means, that condition ci contains n

tokens. If for instance c6 and c7 of the YAWL net in figure 4 contain a token (which would enable task E), 1c6;1c7 unambiguously characterizes the current state of the net. We will keep this denotation and from now on speak of a particular marking of a net.

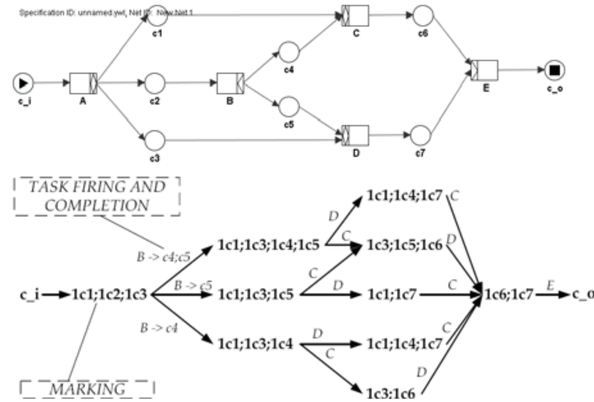

Figure 4: Example YAWL net and the corresponding reachability graph

From all possible markings of a net, only a subset is relevant for our purposes. Solely the reachable markings have to be considered, since all other markings do not stand for a valid state of a YAWL net. The reachability of a marking determines whether there exists an execution order of tasks within a YAWL net that leads to the concerned marking. The whole subset of reachable markings can be gained by generating the finite reachability graph of a net. For our example above, the corresponding reachability graph is shown in the lower part of figure 4. One can see that there are twelve possible markings and six possible execution orders. Details about reachability analysis can be found in [Wyn06].

At this stage of the system, we dispose of a set of markings for each determined net on the path to the jump task. As a second step, the “best” marking for each net that should be realized has to be filtered out. For this purpose, the set of markings for each net is processed by a five-tier ranking system.

Tier 1 – Enabling of obligatory task: Within each net, there is exactly one task which necessarily has to be enabled by the desired marking. Within nets from upper levels of the hierarchy, this concerns the composite task which is decomposed to the net of the next level. Within the leaf net, this corresponds to the atomic task we intend to jump to. In figure 3 for example, the obligatory tasks are SubNet1, SubSubNet and TaskToActivate. All markings that do not enable the obligatory task are filtered out.

Tier 2 – Congruency of enabled tasks: It might be the case that more than one marking has passed the above tier of the ranking system. For this situation, it is intended to select the target marking which is most “similar” to the current marking. Similarity measures can be defined in the following ways:
(i) the size of the enabled task intersection of the initial marking and the target marking. That means markings which preserve most of the enabled tasks are preferred.