

# Re-evaluation of Business Processes

Gero Decker<sup>1</sup>, Marek Kowalkiewicz<sup>2</sup>, Ruopeng Lu<sup>2</sup>, Philipp Maschke<sup>1</sup>, and Mathias Weske<sup>1</sup>

<sup>1</sup> Hasso-Plattner-Institute, University of Potsdam, Germany

(gero.decker,weske)@hpi.uni-potsdam.de

philipp.maschke@student.hpi.uni-potsdam.de

<sup>2</sup> SAP Research Centre, Brisbane, Australia

(marek.kowalkiewicz,ruopeng.lu)@sap.com

**Abstract.** Flexibility and adaptability are important aspects of process aware information systems, and there has been considerable research work done in this area. That work centers around changes of process models during run time and migrating running process instances to modified process models. This paper addresses a different aspect of flexible process management: It takes into account changes of process data while processes run, and re-evaluates the current state of the process instance. Thereby, a new type of flexible workflows is identified, namely those whose structure remains stable, but whose state is re-evaluated based on data that has changed. We motivate the work using a real-world example, introduce a formal model based on Petri nets, and discuss a prototypical implementation of the concepts.

## 1 Introduction

Business process models play an important role to facilitate the understanding of how companies operate and to provide a blueprint for software systems supporting these processes. While well structured processes can be supported with off-the-shelf workflow technology, these systems fall short of providing high flexibility required by modern applications. In this paper we investigate the implications to business processes that result from changes in application data. We do so by providing a framework for business process re-evaluation, consisting of a formal model and a prototypical implementation of the key concepts.

While flexible process management has been a research area in business process management for quite a while, the implications of changing data on the process instance has not been investigated in detail. In ADEPT [12], for instance, structural changes of workflow models at runtime were in the centre of attention, and in the pockets of flexibility approach [17] run time decisions of available alternatives are facilitated. Opposed to this, this paper looks at value changes of application data objects and calculates the implications of these changes on the particular process instance.

This paper is organized as follows: Section 2 provides an example to illustrate the approach. The conceptual basis will be introduced in Section 3, where the

rational of the framework is introduced. A formalization of the concepts is presented in Section 4, before Section 5 introduces a prototypical implementation. A concluding remarks complete this paper.

## 2 Motivating Example

In large organizations, many actions need to be approved before being performed, for instance those related to expenses. For instance, a purchase order may need to be approved by several roles, depending on the value and type of the purchase transaction, and on the ordering agent’s position in the organization’s hierarchy. As a result, the instance of a purchase process may be more or less complex. The more complex the instance is, the more time it may take to go through all of the approval steps. At the same time, the nature of business may require changes to such a purchase order before it has been fully accepted.

Currently it is not a common practice to enable *changes* to application data during workflow execution. The typical solution offered in such cases is an ability to *cancel* the instance and execute a new one with the updated data. This is clearly inefficient due to needlessly re-executing some of the activities.

We introduce here a motivating example that we will use throughout the paper to illustrate the concept of workflow re-evaluation. The scenario is taken from the supplier relationship management (SRM) domain. SRM systems support organizations and their members in acquiring goods and supplies. In this example an employee orders office supplies for his own use. The employee moves to a new office and therefore needs a new desk, a chair, and a new computer. He fills a corresponding shopping cart and submits it to the SRM system. As the shopping cart contains a laptop, he needs to get approval from the IT-administrators, who then have to set up a support plan. Additionally, the cart might need to be approved by a manager, depending on the total cost of the cart. If the value of the shopping cart is higher than \$1,000, the employee’s manager is responsible for the approval. If the total cost of the shopping cart is more than \$5,000, an additional approver has to be involved—the cost centre manager. In real-world SRM scenarios the number of people involved in the approval may be considerably higher.

The example is illustrated in Figure 1 using the Business Process Modeling Notation (BPMN [1]). In this simplified example, the first activity in a process is to fill a shopping cart, which results in updating the SC (Shopping Cart) document. After finishing the action, an email with process information (including links to retrieve purchase process status etc.) is sent to the ordering party. Before the actual approval phase starts, availability of products is ensured, and relevant locks are put in place (so that the products do not become unavailable during the approval phase). That activity requires access to the SC document. At the next step, depending on the value of the order and type of ordered items, a decision on performing each of the available approvals is made. Each of the approvals requires access to the SC document. After the approval phase has finished, the

order is submitted and the ordering party receives an email with complete order information.

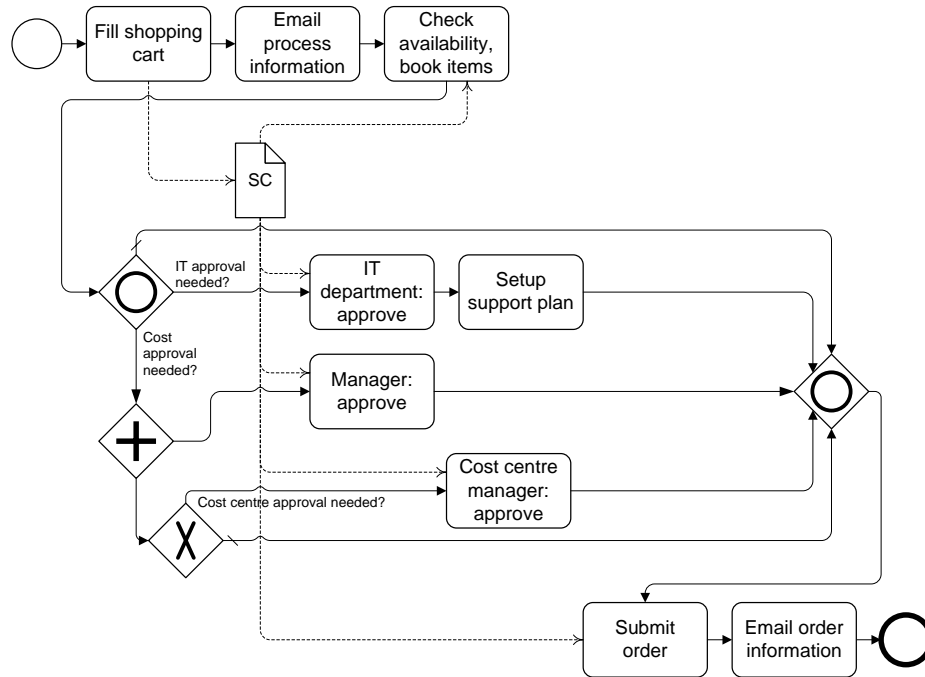


Fig. 1. SRM approval process

The scenario presented in Figure 1 does not take into account a possibility of modifying the SC during process execution. There are many cases where such functionality would be desired. For instance, during the availability check phase, the employee may decide to change the model of the chair. Or, just before submitting the order, the employee may decide to remove the laptop computer from the shopping cart and replace it with a desktop computer. Both examples are very typical and may often happen during execution of purchase processes. Yet, with current BPM systems they require either canceling and re-executing the instance, or preparing very complex models, where in each step it is verified, whether a change to an underlying data object has been made, and depending on the change relevant adaptation is performed.

In this paper we present an approach, where change to underlying data objects is allowed by default, proposing a method for re-evaluation of process instances, should a change to the underlying data objects occur. The approach removes the need for canceling and re-execution of process instances. At the same time it does not add complexity to the graphical representation of process models.

### 3 Re-evaluation

Re-evaluation of business processes is a special case of exception handling. An exception is defined as an execution state that does not comply with the standard execution of the business process [16]. Handling exceptions often requires deviations from normal execution [15]. The exception handlers define additional steps that are executed if the particular exception occurs. The goal is to bring the business process back into a state from which the standard execution can continue.

As we have discussed in Section 2, creating business process models that respond to typical business requirements, such as data change during instance execution, may lead to overly complicated process representations. Trying to avoid that leads in turn to a potentially high cost of process execution, as in many cases the instances may need to be canceled and re-executed.

In our work, we have investigated whether it is possible to allow unexpected changes to data objects during process instance execution without a need to model processes specifically or a need to cancel and re-execute instances. We have also investigated whether it is possible to do it in a way that minimizes costs related to re-execution of affected activities.

If a change occurs to a data object, only these elements of the process that are affected by the change should be considered for re-execution. We illustrate that further in the text using the example provided in Section 2. That way the overall performance of re-evaluated processes can be increased and workload is reduced by avoiding performing redundant activities.

To illustrate the real life situation more accurately, we introduce an additional activity in the example, *Modify shopping cart*. The activity *Modify shopping cart* has access to the SC document, and is not part of the standard control flow.

Since with the updated example a change to the data object may occur in any phase of the instance execution, the following decisions have to be made, before triggering re-evaluation:

**Is the change allowed at this stage?** In our example, the *Submit order* activity is changing the state of the external world, being part of a choreography with an external party<sup>3</sup>. Re-executing the activity, or in other words overwriting an already submitted order, requires proper ability to react by the other party. In this case, the answer depends on specific conditions of the choreography partner. There are however obvious cases, for example when a real world item is destroyed and it is not possible to recreate it.

**How far should the process instance roll back?** The process should roll back to the first node in the process model that is affected by the data object being changed. From that point the decisions on re-executing activities, re-evaluating gateways etc. should be made. There is no need to roll back to the very beginning of the process instance. If there is an *irrevocable* action executed at any time between the earliest affected activity and the

---

<sup>3</sup> To simplify the model, we omit the choreography representation in the diagrams.

current state of the process, the process instance cannot be automatically re-evaluated.

**Which activities should be re-executed?** We distinguish three types of activities in process re-evaluation. (1) *Execute once*: activities that should be executed only once during instance execution (for instance: *Email process information* is sent only once to the employee), (2) *always re-execute*: activities that should be executed every time: during normal execution and re-evaluation (for instance: *Check availability, book items* has to be performed in any case of SC change), (3) *execute conditionally*: activities that are to be re-executed depending on a specified condition (for instance: *IT department approve* should only be re-executed if an IT-related product is modified).

Re-evaluation is carried out in three phases. (1) A decision whether re-evaluation is possible is made, and if possible, the roll-back phase begins. Otherwise, a modification to the data object is rejected (which may trigger human intervention should manual re-evaluation be possible). (2) In the roll-back-phase, the process instance is rolled back to a state where none of the executed process nodes (activities, gateways) is dependent on the data object. Then (3) the roll-forward-phase begins: the process nodes are analyzed and re-executed or skipped according to the rules.

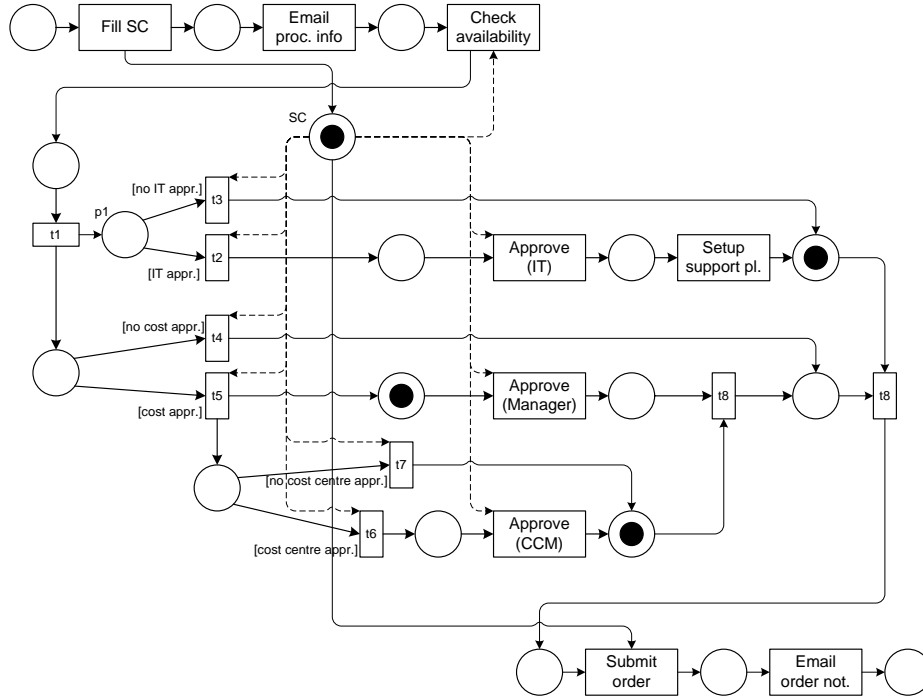
Re-evaluation is triggered by so called re-evaluation events. These events fire in response to changes to data object attributes. A change to a data object attribute defined in the event triggers re-evaluation. Each of process activities may have re-evaluation conditions defined. The re-evaluation conditions specify whether the activity should be re-executed during re-evaluation or not. Other types of activities (*always re-execute, execute only once*) should also be annotated to allow proper re-evaluation. The same holds true for marking *irrevocable* activities.

## 4 Formal Model

This section formally defines re-evaluation on the basis of Petri nets [13], a common formalism for business processes. We define *value-passing nets*, a special class of Petri nets where tokens carry values and transitions have guard conditions.

Figure 2 illustrates the corresponding value-passing net for the example from section 2. Resorting to the well-known notation for Petri nets, circles represent places and rectangles represent transitions. The current marking in Figure 2 represents that the administrators have already approved the shopping cart and set up the support plan. The manager has not approved the shopping cart yet.

We distinguish between two kinds of flow connections between places and transitions. The solid arrows represent classical flow connections. Upon firing of the corresponding transitions, tokens are consumed from the input places and tokens are produced onto the output places. The dashed arrows denote read arcs. Here, tokens on input places are required for the firing of a transition. However, the respective token is not removed from the place.



**Fig. 2.** SRM approval process as value-passing net

Tokens residing on place  $SC$  represent shopping carts from our example. A shopping cart is created upon firing of transition  $Fill\ SC$  and consumed by transition  $Submit\ order$ . Those transitions having a read arc originating in  $SC$  also take a shopping cart token as input. However, the shopping cart is not consumed nor altered upon firing.

The labels in brackets, e.g.  $[noIT\ appr.]$ , are guard conditions for transitions. Such guard conditions restrict the combinations of input tokens which the respective transition is enabled for.

On top of value-passing nets, we define execution logs that store past steps. In order to take full advantage of the concurrent nature of Petri nets, these logs are not strictly sequential but rather partial orders. Figure 3 shows the execution log for the current marking, where log entries are depicted as rectangles and pairs of direct predecessor/successor as arrows.

In this example, re-evaluation applies for altering a token  $v$  on place  $SC$ . We assume this modification to happen from outside the net, resulting in the removal of  $v$  and the introduction of a token  $v'$  on  $SC$ . (i) In the first phase of re-evaluation, the net is rolled back to a marking, where  $v$  was used as input for a transition for the first time. This applies to the approval transitions as well as for the decision transitions  $t2$  to  $t7$ . Rolling back relies on the information provided in the execution log. (ii) In the second phase, rolling forward takes place. The

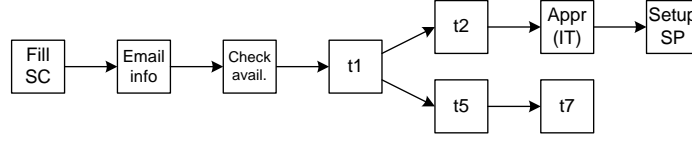


Fig. 3. Execution log

entries previously contained in the log are stored and used as “execution context” during the subsequent execution. In particular this means that those transitions where the input values did not change do not need to be re-executed.

As we have seen in our example, the different approval steps only focus on a certain aspect of the shopping cart. While the administrators only consider IT-related items and ignore the price, the managers mostly focus on the category of items as well as the prices. In order to represent this in the formal model, we introduce filter functions for read arcs.

The following definition introduces value-passing nets. As part of that we assume an infinite set of values *Value* (tokens).

**Definition 1 (Value-passing Net).** A value-passing net is a tuple  $(P, T, F, R, view, g, m)$  where

- $P$  and  $T$  are disjoint sets of places and transitions,
- $F \subseteq (P \times T) \cup (T \times P)$  is the flow relation connecting places and transitions,
- $R \subseteq F \cap (P \times T)$  is a subset of flow connections representing read arcs,
- $view : R \times Value \rightarrow Value$  is a filter function assigning values to tuples of read arcs and values,
- for each  $t \in T$   $g(t) \subseteq \wp(\bullet t \rightarrow Value)$  is the guard relation indicating if a transition  $t$  is enabled for a given combination of input values and
- $m : P \rightarrow MS(Value)$  is a marking function assigning multi sets of values to places.

We define  $\bullet t$  and  $t \bullet$  as auxiliary functions:  $\bullet t := \{p \in P \mid p F t\}$  and  $t \bullet := \{p \in P \mid t F p\}$ . Furthermore, we introduce  $view'((p, t), v) := v$  if  $(p, t) \notin R$  and else  $view'((p, t), v) := view((p, t), v)$ .

The firing semantics of value-passing nets is similar to that of classical place / transition nets in the sense that a transition is enabled only if there is at least one token on each input place. Firing of a transition will lead to consuming one token from each input place (except in the case of read arcs) and producing one token onto each output place. Guard conditions further restrict the enablement of transitions. As tokens carry values, we speak of *transition modes*, i.e. bindings of values to input and output places of a transition.

**Definition 2 (Transition Modes, Enablement and Firing).** Let  $(P, T, F, R, view, g, m)$  be a value-passing net. A transition mode is a tuple  $(\sigma_{in}, t, \sigma_{out})$  where  $\sigma_{in} : \bullet t \rightarrow Value$  assigns values to the input places of  $t \in T$  and  $\sigma_{out} : t \bullet \rightarrow Value$  values to output places.

A transition mode  $tm = (\sigma_{in}, t, \sigma_{out})$  is enabled iff  $\sigma_{in} \in g(t)$  and there exists a function  $\sigma'_{in} : \bullet t \rightarrow \text{Value}$  such that  $\forall p \in \bullet t [\sigma'_{in}(p) \in m(p) \wedge \text{view}'((p, t), \sigma'_{in}(p)) = \sigma_{in}(p)]$ . The reached marking after firing of  $tm$  is  $m'$ , where  $m'(p) := m(p) - \{\sigma'_{in}|_{q \in P|(q, t) \notin R}(p)\} + \{\sigma_{out}(p)\}$ . We denote this as  $m \xrightarrow{tm} m'$ .

The function  $\sigma_{in}$  assigns values to input places as seen through the corresponding view.  $\sigma'_{in}$  denotes those values actually residing on the input places and which will be consumed upon firing.

**Definition 3 (Execution Log).** An execution log is a tuple  $L = (E, <)$  where

- $E$  is the set of log entries where each entry is a transition mode and
- $< \subseteq E \times E$  is a partial order relation on the set of entries.

The values produced upon firing of a transition typically do not have any functional dependency on the input values. This is due to the fact that the Petri nets do not capture the complete world. E.g. the outcome of the *Setup support pl.* transition does not only depend on the input values but also other information not captured in the sample net. Here, value-passing nets are different to classical colored Petri nets [8], where a functional dependency between input and output values exists.

The only case where there exists such a functional dependency in value-passing nets is the following. When rolling forward all those activities are not performed again, where the input values have not changed compared to the previous execution. This is formally represented by producing the same output values again.

Enablement of transition modes therefore depends on an *execution context* already mentioned earlier. This context basically is a subset of the execution log entries, containing only those entries that were subject to a previous rollback.

**Definition 4 (Enablement and Firing with Execution Context).** Let  $(P, T, F, R, \text{view}, g, m)$  be a value-passing net and  $C$  a multi-set of transition modes. A transition mode  $tm = (\sigma_{in}, t, \sigma_{out})$  is enabled for execution context  $C$  iff  $tm$  is enabled and either  $tm \in C$  or  $\nexists \sigma'_{out} ((\sigma_{in}, t, \sigma'_{out}) \in C)$ . The execution context for the reached marking  $m'$  is  $C'$  where  $C' := C - \{tm\}$ .

This definition prevents transition modes from firing that are not contained in the context and where there is another transition mode  $(\sigma_{in}, t, \sigma'_{out})$  in the context. This enforces that the same values are produced – given a transition mode was already executed earlier.

After having defined the behavior for the roll-forward phase, we now define rollbacks. Rollbacks are atomic steps triggered by exchanging a value  $v$  with another value  $v'$ . The basic idea is to undo all transition modes  $tm \in E_{RB}$  that had  $v$  as input or causally depended on such a transition. The log is reduced accordingly and these transition modes are added to the new execution context.

**Definition 5 (Rollback).** Let  $N = (P, T, F, R, view, g, m)$  be a value-passing net,  $L = (E, <)$  its current execution log and  $C$  the execution context. Replacing a value  $v \in m(p)$  with value  $v'$  on place  $p \in P$  results in rolling  $N$  back to marking  $m'$ , execution log  $L'$  and execution context  $C'$  where

- $L' := (E \setminus E_{RB}, <)$  and  $E_{RB} = \{tm \in E \mid \exists tm' = (\sigma_{in}, t, \sigma_{out}) \in E ((tm' = tm \vee tm' < tm) \wedge \sigma_{in}(p) = v)\}$ ,
- $C' := C + E_{RB}$  and
- $m'(q) := m_{back}(q)$  for all  $q \neq p$  and  $m'(p) := m_{back}(p) - \{v\} + \{v'\}$ .  $m_{back}$  is the marking for that holds  $m_{back} \xrightarrow{tm_1} \dots \xrightarrow{tm_n} m$  for any sequence of transition modes  $tm_1, \dots, tm_n$ , where  $\{tm_1, \dots, tm_n\} = E_{RB}$  and  $\forall i, j [tm_i < tm_j \Rightarrow i < j]$ .

Imagine we modify the shopping cart in the state illustrated in Figure 2. Let  $v$  be the original shopping cart and  $v'$  the modified shopping cart. The rollback will include the transition modes involving *Setup support pl.*, *Approve (IT)*,  $t3$ ,  $t5$  and  $t6$ . Imagine the view on the shopping cart has not changed, i.e.  $view((p1, t2), v) = view((p1, t2), v')$  and  $view((p1, t3), v) = view((p1, t3), v')$ , but the shopping cart now requires cost centre approval.

As the modification did not have any effect on  $t2$  and  $t3$ , these transitions will fire with the same output values as in the previous execution. Therefore, *Approve (IT)* and *Setup support pl.* will also have the same effect as before. In contrast to this,  $t6$  will not fire this time. Rather  $t7$  will fire, resulting in the enablement of *Approve (CCM)*. *Approve (Manager)* will be enabled again.

This examples shows how individual activities, e.g. *Approve (IT)* and *Setup support pl.* in this case, do not need to be re-executed by the employees and the activities' outcome can directly be taken from the previous execution. The example also shows that re-evaluation might lead to choosing different branches than in a previous execution: A second cost approval is required for the changed shopping cart.

## 5 Prototypical Implementation

In this section details are given as to how the re-evaluation functionality was prototypically implemented. For this purpose we used an execution engine for distributed Petri nets we developed earlier. It executes nets that are similar to Colored Petri Nets, but with some extended semantics. Embedded inside a webserver, it communicates via HTTP with users, external tools and with other execution engines as well.

The engine supports several interfaces and technologies for manipulating Petri nets:

- PNML (Petri Net Markup Language) - an XML-based standard interchange format for Petri nets. The engine uses it for importing and exporting nets.
- XML (Extensible Markup Language) - tokens have an XML document as their value. Transitions manipulate this XML document when firing.

- XHTML (Extensible Hypertext Markup Language) - used for representing tasks to a user.
- XFORMS - an XML format used here for specifying user interfaces and data bindings for manual transitions
- XSLT (Extensible Stylesheet Language Transformations) - used for specifying XML transformations that manipulate token values when automatic transitions fire.

To support re-evaluation with the given engine, several components had to be modified. Firstly, tokens are not destroyed when they are consumed, but rather set inactive. This way the tokens will be ignored concerning execution semantics, but are still addressable in the engine through an unique identifier. This allows for an efficient rollback/roll-forward technique, since tokens can be toggled active/inactive very fast. Additionally we avoid having to store tokens in the execution log with their complete XML content.

Secondly, each firing of a transition has to be logged, and references to the corresponding input and output tokens have to be attached to the entry.

In order to build the partial order relation of the execution log, described in Definition 3 in Section 4, we need to find out the direct predecessors of firing transitions. For this purpose we take advantage of the fact, that each token is created by one specific transition. We store this information for each token and, using a transition mode's input tokens, we can easily find out the direct predecessors.

When re-evaluation is triggered, a list of all transition modes depending on the modified token is created. During the following roll-back phase, the transition modes stored in the corresponding execution log are rolled back, until none of the remaining modes has a mode from the list as his predecessor. Rolling back a transition mode is done by deactivating all output tokens of a transition mode and activating all input tokens. After rolling back, the transition mode is moved from the log to the respective execution context.

Rolling forward is in itself not a phase like rolling back. Instead, there are extended execution semantics for transitions, when they have an entry in the execution context. If the entry's input values are the same as the current input values, the transition is re-executed automatically. We do this by just deactivating the input tokens and activating the output tokens from the corresponding context entry and moving the transition mode back to the execution log.

It could also prove to be necessary to manually re-execute a task. This is the case when there is no entry with matching input values or when there are several entries with the same input values. The task is put into the inbox of the person that executed it before. That person can then delegate it like any other task, if he wants to do so. Additionally, when re-executing a task manually, we display the previous input and output values and highlight the changes compared to the current values. A screenshot of this feature can be seen in Figure 4.

Furthermore, features such as value filters for read arcs and expressions to determine whether to re-evaluate or not and re-evaluation trigger events had to be added to the engine as well.

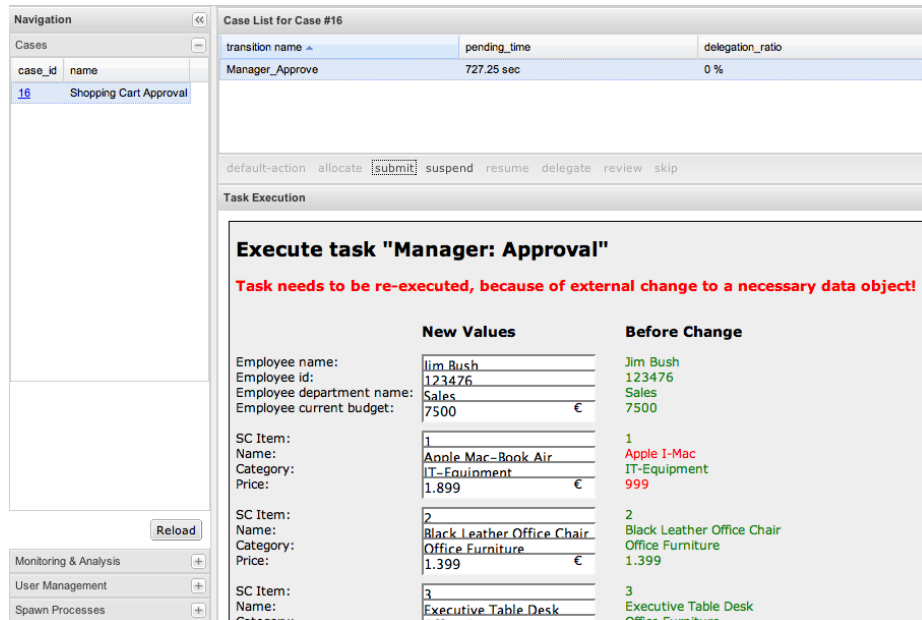


Fig. 4. Screenshot of task to be re-executed

## 6 Related Work

The requirements for providing flexibility in workflow models and execution arose from the need for change in business processes, which have been identified in the early age of workflow research [4, 6]. In the meantime, commercial workflow management systems (WFMS) have been under fire for the lack of runtime flexibility.

The term *dynamic change* or *ad hoc change* have been referred to with slightly different meanings. In general, change is present in some degree and form in almost all domains, and handling dynamically changing workflows is a highly complex issue, which related to multiple workflow aspects including modelling, verification, compliance and enactment [14, 17]. Three dimensions of changes have been identified, namely flexibility, adaptability and dynamism/evolution [17, 19]:

Flexibility is the ability of the business process to execute on the basis of a loosely, or partially specified model, where the full specification is made at runtime. Adaptability is the ability of the workflow processes to react to exceptional circumstances, which may or may not be foreseen, and generally would affect one or a few process instances. Dynamism/Evolution is the ability of the workflow process to change when the business process evolves. This evolution may be slight as for process improvements, or drastic as for process innovation or process reengineering.

There have been a huge influx of research proposals for calibrating WFMS capabilities along one or more abovementioned dimensions. The most prominent ones have been ADEPT [12], AgentWork [10], Pocket of Flexibility [17], Case Handling [18], Worklets [2] and Declare [11]. Among which, case handling is an approach which support case (workflow instance) specific configuration in workflows. Case handling takes a data-driven approach, in which a data object is maintained to contain the state of the case and used for directing case definition (activity routing etc.). Flower [3, 18] is such a system. A domain expert is involved to adapt the entire case, who can browse and/or update the case with relevant information.

The proposed approach in this paper aims at providing automated recovery from runtime workflow change. Recovery is realised by utilising re-evaluation technique that brings the workflow instance back to a state from which the standard execution can continue. This technique is somewhat similar to the recovery techniques (redo, rollback and undo) in transactional databases. In flexible workflows, case handling has conceptualised the notion of *redo* and *skip*. A domain expert can be assigned to the designated redo and/or skip role, besides the *execute* role that is authorised to execute the activity or start a process. Under the redo role, a domain expert is authorised to roll back the execution state of the process before executing the activity, which requires all subsequent activities to be undone. Similarly, the skip role is authorised to skip the execution of one or more activities. An activity with no redo nor skip role assigned is similar to the activities that have to be executed only once in the proposed re-evaluation approach. On the other hand, an activity having both executable and redo role enables the activity to be re-executed for re-evaluation. However it cannot make sure the activity is re-executed for every re-evaluation. A domain expert has to be involved to decide whether re-execution for each such activity is required.

The proposed approach provides a comprehensive theoretical framework for implementation of re-evaluation in WFMS, or Process Aware Information Systems (PAIS). While a special kind Petri nets were used in the framework, BPMN was used for illustrating the example from section 2. Petri nets can be generated from BPMN as described in [5].

Re-evaluation is also related to compensation. In [9] the idea of compensating transactions is formally described. The most prominent language incorporating compensation mechanisms is the Business Process Execution Language [7]. A compensating sphere consists of a set of activities of a process model. The semantics is as follows: Either all activities of the sphere complete successfully or none at all. A problem arises if some activities have already completed, when the sphere needs to be compensated. In this case all activities that have completed successfully need to be compensated for. While in compensation work is lost due to compensating activities, the re-evaluation approach makes sure, work is not lost. For instance, if there is budget allocated and a cheaper laptop is ordered, then the budget allocation is still valid. In the compensation approach, the budget would be de-allocated, only to be allocated later again.

## 7 Conclusion

In this paper we have introduced a new approach for flexible workflow management, based on changing application data during run time. The approach is conceptually underpinned using Petri nets, and a prototypical implementation shows its validity. The approach is, however, based on a number of assumptions that might or might not be fulfilled in real-world application scenarios. We assume that the execution environment facilitates changes to application data. While this is in fact useful, there might be scenarios in which this is not supported.

In real-world settings, the period during which changes to application data can be performed, needs to be defined. This decision is application specific and therefore was not incorporated in the approach. Techniques similar to the well known spheres concept in business process management can be used here. For instance, an order can be changed as long as there is no order sent to a supplier. If the order has already been sent, compensation steps need to be carried out.

It is important that the system is aware of the fact that the process instance is not canceled, but changed. To see this consider the following situation. In the example discussed, after submitting the order, it is granted by the boss by allocating the required budget. If the order is changed and a re-evaluation is performed, it is assumed that the budget is still available and not taken by another order. This is a valid assumption, because the boss allocated the budget and the process instance was not canceled (that would result in de-allocating the budget). Therefore, the budget is available for the modified cart, and the allocation of the budget does not need to be performed a second time.

## References

1. Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), February 2006. <http://www.bpmn.org/>.
2. M. Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. Phd thesis, Queensland University of Technology, May 2007.
3. P. Berens. *Process-Aware Information Systems*, chapter The FLOWer Case-Handling Approach: Beyond Workflow Management, pages 363–395. John Wiley & Sons, Inc., Centre for Information Technology Innovation, Queensland University of Technology, Brisbane, Australia; Department of Technology Management, Eindhoven University of Technology, Eindhoven, The Netherlands, 2005.
4. F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Snchez. Wide workflow model and architecture. Technical report, University of Twente, 1996.
5. R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology (IST)*, 2008.
6. C. A. Ellis, K. Keddera, and G. Rozenberg. Dynamic change within workflow systems. In *COOCS*, pages 10–21, 1995.
7. D. C. Fallside and P. Walmsley. Web Services Business Process Execution Language Version 2.0. Technical report, Oct 2005. <http://www.oasis-open.org/apps/org/workgroup/wsbpel/>.

8. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer, 1996.
9. H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 95–106. Morgan Kaufmann, 1990.
10. R. Müller, U. Greiner, and E. Rahm. Agent work: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.
11. M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst. Constraint-based workflow models: Change made easy. In R. Meersman and Z. Tari, editors, *OTM Conferences (1) CoopIS'07*, volume 4803 of *Lecture Notes in Computer Science*. Springer, 2007.
12. M. Reichert and P. Dadam. ADEPT flex -supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
13. W. Reisig. *Petri nets*. Springer Verlag, 1985.
14. S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data Knowl. Eng.*, 50(1):9–34, 2004.
15. N. Russell, W. van der Aalst, and A. ter Hofstede. Exception handling patterns in process-aware information systems. Technical Report BPM-06-04, BPM Center, 2006.
16. S. Sadiq and M. Orłowska. On capturing exceptions in workflow process models. In *Proceedings of the 4th International Conference on Business Information Systems*, 2000.
17. S. Sadiq, M. Orłowska, and W. Sadiq. Specification and validation of process constraints for flexible workflows. *Information Systems*, 30(5):349–378, 2005.
18. W. M. P. van der Aalst and M. Weske. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.
19. B. Weber, S. Sadiq, and M. Reichert. Lifecycle management for dynamic processes. In *Tutorial in 5th International Conference on Business Process Management (BPM 2007)*, 2007.