Bringing Flexibility into Dynamic Process Change

The Process Re-execution Approach

Lukáš Máčel and Tomáš Hruška
Department of Information Systems
BUT, Faculty of Information Technology
Brno, Czech Republic
e-mails: {imacel, hruska}@fit.vutbr.cz

***Abstract*—The business process must reflect changes in the environment, therefore, the adaptation of the process model or the particular process instance is essential. The state compliance criterion has been introduced to check that dynamic process change is correct and does not lead to soundness problems or run-time errors. In some cases, however, the process instance must immediately be migrated to the changed model or updated itself. Hence, the strategy of coping with the non-compliant process instance must be chosen. This paper presents the process re-execution approach which effectively implements the state compliance test. The re-execution algorithm makes it possible to defer the suitable activities and use them later, thus offering a flexible solution for treating the non-compliant process instances. Moreover, a custom strategy of treating can be used based on the full context of the activity that caused the inconsistency. In many cases, the process re-execution approach enables to treat the non-compliant process instance automatically and thus the total number of instances that are successfully migrated is increased.**

Keywords-flexibility; process change; process evolution; state compliance; re-execution.

I. INTRODUCTION

Quick reaction to changes in the environment (like new technologies, new laws and new market requirements) is one of the crucial abilities of every enterprise. Different business goals, modifications of organizational structure, or legacy changes have influence on procedures and activities within the enterprise. Business processes, which are modeling these procedures and capturing the way that particular goals are achieved, must reflect these changes and adapt to them. As a result, Process-Aware Information Systems (PAIS) must offer tools for such adaptations and ensure that the dynamic changes are applied correctly [1].

We can distinguish different ways of the process change. During long-running processes it is sometimes necessary to deviate from the pre-specified process model and thus the correct behavior of one specific process. In contrast, the evolution of the whole process model may be required in order to accommodate changes or refine the model quality. Changes can be focused on the control logic of the process. For example, some of the process activities are added or deleted. Changes can also affect the data flow of the process by deleting some data edges or inserting new data variables. Control flow adaptation may also be connected with data changes.

In all these cases, the challenge is to avoid errors and inconsistencies caused by dynamic change. Approaches which guarantee a sound and correct process model after adaptation are presented in [1][2][3][4][8]. One well-known criterion used by many approaches is state compliance [4]. If the process instance is non-compliant, it cannot be migrated to the new model (for the case of model evolution). The ad hoc change of specific non-compliant process instance cannot be applied either. One possible solution for the evolution is to leave the non-compliant process instance continuing its execution on the original model. However, this means that the process instance cannot benefit from future model changes. For this reason, it is necessary to find strategies [3][6] to cope with the non-compliant instances.

The goal of this paper is to present a different approach of process execution that efficiently implements the essential state compliance testing, on the one hand, and offers better flexibility in treating non-compliant process instances, on the other hand. The approach is based on the idea that the process instance is always re-executed from the beginning in order to perform the next activity pre-specified in the model. This is useful in the situation in which the process model is changed because the process instance is executed on the adapted model, thus making it possible to verify whether instance migration is possible. Moreover, the first activity causing the process instance to be non-compliant can be found. The activity data collected by the run of the process can be taken into account when choosing a strategy for treating the process instance.

The paper is organized as follows. According to literature, Section 2 includes the state compliance definition as well as an overview of strategies for dealing with non-compliant instances. Section 3 introduces the process re-execution approach that we have developed during our research. The contribution of re-execution approach can be found in Section 4. Section 5 contains our experience with the re-execution approach in practice. Related work is described in Section 6 and Section 7 concludes this paper with a summary and outlook.

 ISBN: 978-1-61208-345-2

II. ENSURING THE CORRECTNESS OF PROCESS CHANGE

A. State compliance

The basic requirement for the process dynamic change is to ensure the soundness property of the changed process model. This can be achieved by the application of reachability analysis on the process model represented by workflow net as described by Weske [2].

If we want to decide whether the process instance can be correctly relinked to the changed model, we must establish a correctness notion. We can distinguish two groups of correctness criteria [1]. The first group includes criteria based on graph equivalence. One example might be the inheritance relation criterion used in WorkFlow nets [11]. The second group contains criteria founded on process execution trace equivalence. One of the well-known criteria from this group is state compliance. For further consideration, we chose this criterion because our approach is based on replaying the process execution trace.

In [8], state compliance is defined as follows. Let $I = (M, \delta)$ be a process instance running on sound model M with execution trace δ. Assume M' is another sound model and M is transformed into M' by the change Δ. Then, I is state compliant with M' if δ is producible on M'. State compliance is based only on a process instance execution trace and presumes no specific process modeling language.

Assume, for example, process model M which defines a sequence of two activities A and B. We change this model into model M' by inserting activity X between activity A and B. Further, we have two process instances I_1 and I_2 based on model M. In instance I_1 activity A is running, thus I_1 is state compliant with model M' because its execution has not entered the changed region yet. In contrast, in instance I_2 activity B is running. However, model M' pre-specified that new activity X must run before B and thus the execution trace of I_2 cannot be produced on M'.

According to the Rinderle-Ma et al. [3], traditional state compliance is too restrictive in connection with loop structures, thus relaxed state compliance is established in order to increase the number of process instances which can be migrated to a changed model. The approach is based on the idea that we logically hide information about activities from previous loop iteration and the modified loop-purged trace of process instance is then used to check state compliance. The approach is also applicable to nested loops [3].

We also need to ensure the correctness of data flow after model adaptation. Compliance conditions for data flow change are defined by Rinderle-Ma [9].

B. Strategies for non-compliant process instances

The state compliance check can uncover process instances which have already progressed too far and their relinking to the changed model must be prohibited because of possible soundness violations or data flow errors. In some cases immediate on-fly migration may be requested, therefore, a solution for the non-compliant process instance must be found. Consider, for example, legacy changes or unexpected situations while treating a patient.

There are three widespread strategies described by Reichert and Weber [8].

1) The partial rollback: This strategy is based on the idea that necessary activities are undone and the process instance is reset into the compliant state. This strategy is closely connected with the execution of compensation activities [4]. Consider, for example, that activity *book a trip to the sea* was completed. The travel agency, however, decided to cancel the trip due to lack of interest. As a consequence, the compensation activity in order to cancel the respective booking is performed.

2) Delayed migration: This strategy assumes that the non-compliant process instance becomes compliant again after a certain time. Consider the changes related to a loop body. Although the current iterration of a loop progresses too far, the next iteration fulfills the state compliance. Hence, the migration will finally be successful.

3) Adjusting change operations: The idea of this strategy is to adjust the intended change itself instead of reseting the process instance state. Consider the insertion of activity A. If we adjust the position of A without violating the data flow correctness or the other semantic constraints defined by the process model, the number of migratable instances is increased.

III. PROCESS RE-EXECUTION APROACH

A. Process re-exectuion algorithm

First, we define an abstract machine which simulates the execution of particular process instance I running on the given process model M and then we describe the way that the process re-execution is performed.

Let A be a set of unique activity labels. Further, let F denote a set of unique activity flow labels which are used to model a situation in which the execution is split into more parallel branches. The flow can be also described as a token in the terminology of Petri nets [14]. Let V be a set of data variable names and D denote a set of possible values of these data variables.

Next, the machine has a memory tape on which data about already performed activities are stored. This tape represents the partial execution trace of simulated process instance I. The tape has one head which can be used both for reading and writing and the current position of the head denotes the data of performed activity which can be used to support re-execution. We define the tape as the sequence $\delta_I = <pa_1, pa_2, \ldots, pa_k>$ where the performed activity is defined as $pa_i = (f, a, DI, DO)$, $f \in F$, $a \in A$, $DI \subseteq V \times D$, $DO \subseteq V \times D$ and $i = 1, \ldots, k$, $k \in N$. DI stands for data inputs and DO denotes data outputs of the performed activity.

The established abstract machine works in two modes of execution. The *real mode* is defined as follows. The machine reads the activity that is pre-specified in process model M and creates a respective work item. A source is chosen and then the activity is performed. The data about current flow, activity label, data inputs and outputs are stored on the machine tape at the position where the head is situated.

 ISBN: 978-1-61208-345-2

The *silent mode*, in contrast, is used during the re-execution of the process instance. The abstract machine reads the activity label from model *M*, although no work item is created. Instead of this, a subsequent test is performed. Assume that the machine is executing activity *a*, the current flow label is *f* and we have a set of current data input variables *di*. If the machine head reads quadruple *(f, a, di, do)* from the tape, the activity data output *do* is used as a result of the activity being executed. This is why we say that activity has been performed "silently".

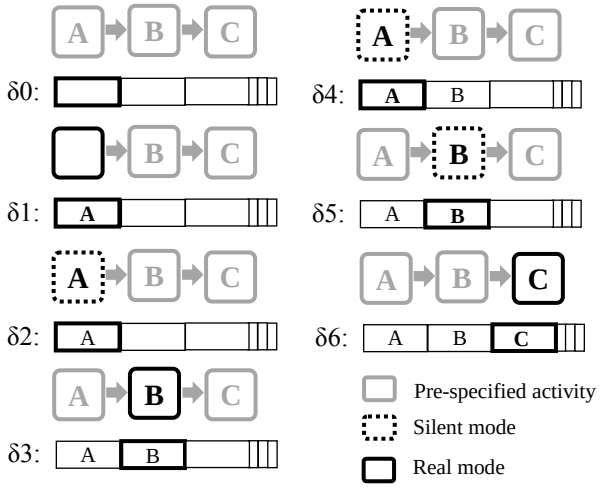

Figure 1. A scheme of the process re-execution.

Finally, we can define the process re-execution algorithm, which is illustrated in Figure 1. At the beginning of process instance execution, the abstract machine is in *real mode* and has an empty tape. Firstly, the machine reads activity *A* according to process model *M*. Once *A* is completed, the machine stores the appropriate data on the end of the tape. The next step of process execution, however, does not focus on the following pre-specified activity *B*. Instead, the process instance re-execution is started. The abstract machine switches into the *silent mode*, the tape is rewound and the execution starts from the beginning again.

The machine acts according to *silent mode* definition. The result of the simulated activity is taken from the current field on the tape and then the head is shifted forward. The machine continues with the next pre-specified activity *B*. We have, however, an empty field under the head. Therefore, the *silent mode* is toggled to the *real mode*. The change of the execution mode means that we re-executed the first performed activity and are now proceeding with the "real" execution. After the completion of activity *B*, the re-execution is repeated. It is important to highlight that the state of the process instance is not held throughout the execution. Instead of this, it is always reconstructed after finishing each individual activity.

B. Process re-exectuion on a changed process model

We will now investigate the process re-execution approach in the context of a changed model. Assume that process model *M* is transformed into model M' and instance *I*, which ran on M, is now relinked to changed model *M'*. Our abstract machine enters *silent mode* and begins to simulate the activities of *I* according to data stored on the tape. The machine detects a difference between the pre-specified activity and its inputs, on the one hand, and the performed activity on the current position of tape, on the other hand. We have already detected that process instance is not compliant with modified model M'. To flexibly cope with this inconsistency, the abstract machine interrupts execution and triggers the event which can be handled in order to treat process instance *I*.

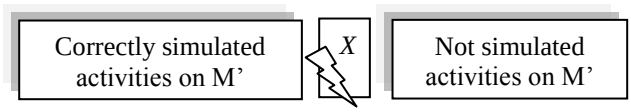

Figure 2. The tape content during interrupted execution.

We can divide the tape into three segments as depicted in Figure 2. The first segment of the tape contains performed activities which are correct in the context of modified model *M'*. The second segment includes activity *X* which caused the inconsistency; the third segment comprises the rest of the performed activities that have not been simulated but may be potentially reused in sequel. The state machine removes the second and third segments from the tape and attaches them to the interruption event. The interruption event is a data structure that contains all the information about the interruption. If there is no treatment specified, the machine can choose a default strategy, or the event is sent to the system administrator to warn that the process migration has failed. Due to the triggered event, we get the full context of the inconsistency and, together with the knowledge about the semantics of the performed model change, are able to flexibly solve this situation.

In some cases, it is helpful to defer suitable activities from the third segment of the tape and use them later, because we can reduce unnecessary loss of work. For this reason, we designed the Store of Deferred Activities (SoDA). The SoDA can be used for these purposes:

- We can search deferred activity and use it during custom-process instance treatment.
- The machine itself may match suitable deferred activity and fill the current empty field on the tape in order to continue in silent mode execution. In other words, the deferred activity may be automatically used later.
- To dynamically modify the result of the activity that has already been completed to perform dynamic data flow change.

Further, we define the SoDA formally. Let PA_I be a set of activities that have already been performed. UA_I denotes a set of performed activities whose outputs have been adjusted by the process participant and DA_I stands for a set of *deferred activities* and $DA_I \subseteq PA_I \cup UA_I$. Then, the SoDA can be defined as triple $S_I = (DA_I, <, m)$ where $<$ is the partial order relation on DA_I and m is a matching function. The matching criteria must be unique, thus the activity label,

flow label and also data inputs of performed activity are taken into account when matching function *m* is scanning the *SoDA*. If deferred activity *ad* is found, the predecessor test is performed. If and only if there is no deferred activity *ad'* < *ad*, then *ad* is matched and returned from function *m*.

We actually need to change the re-execution algorithm in order to integrate the established SoDA as follows. At first, the abstract machine looks into the SoDA and with the help of matching function *m* tries to find suitable deferred activity. If no such deferred activity exists, the machine acts according to the *silent mode* definition. If matching function *m* succeeds and the current tape value is empty, the deferred activity is moved from SoDA to a current position on the tape and the *silent mode* proceeds normally. Finally, if matching function *m* is successful and the head of our machine reads the data that vary from the matched deferred activity in output data only, then the machine triggers the interruption event and the detected data change can thus be handled properly. Moreover, if no handler is provided, the default handler is chosen which moves the found activity from the SoDA to the current field of the tape and the *silent mode* continues.

TABLE I. THE ABSTRACT MACHINE STATES

The abstract machine				
Meaning	***Tape***	***SoDA***	***Event***	***Performed action***
Real mode	Empty	Not found	No	Do activity
Silent mode	(A,i,o)	Not found	No	Use *o*
Activity reuse	Empty	(A, i, o)	No	Use *o*
Data change	(A, i, o)	(A, i, **x**)	Yes	Custom Use **x**
Data change consequence	(A, **x**, o)	Not found	Yes	Custom Search A Do A
Newly added/deleted activity	Not found	Not found	Yes	

Table I clearly shows the possible states of the abstract machine, the meaning of these states and the corresponding action that can be performed. Note that the flow label is omitted because of the place.

As we can see, the first four rows of the table were previously described. The data change consequence (the fifth row) is characterized by the partial compliance on the activity label. However, the values of input data variables differ; therefore, the interruption event is triggered and we can handle it in three different ways:

- It is possible to define a custom handler.
- We can try to find the suitable deferred activity in the SoDA.
- We may allow the activity to perform in the *real mode*.

The last row of the table describes the abstract machine state in the situation where the newly added or deleted activity is detected. Note that to identify whether it is insertion or deletion of the activity, we must know the semantics of the change, because we are only able to find out that there is an inconsistency.

Now, we demonstrate the re-execution approach by means of an example. Assume process model M_2 as depicted in Figure 3 and process instance *I*, which is currently running on M_2. Activities *A, B* and *C* successfully finished.

Further, we will change model M_2 to M_2' by inserting two activities, *X* and *Y*, and one data binding between them. The abstract machine starts re-execution on adapted model M_2' and according to the re-execution algorithm switches into *silent mode*. Activity *A* is found on the tape and that appropriate stored output data are used. Then, we shift to the next newly added activity *X*. The SoDA is empty and the abstract machine detects inconsistency. According to the model, activity *X* should be performed. However, the machine head reads activity *B*. As a result, the abstract machine triggers the interruption event to allow the handling of the unexpected situation.

The re-execution algorithm has actually performed a state compliance check and has detected that process instance *I* is not compliant with adapted model M_2'. Moreover, we have the full context of this situation. The performed activities from the past are on the machine tape. The event contains activity *X*, which caused interruption, as well as the rest of the activities which have been performed and can potentially

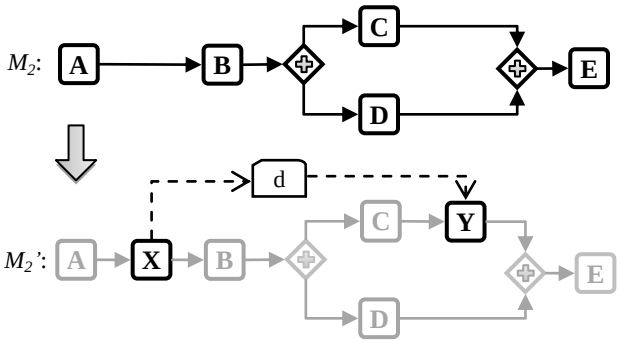

Figure 3. The adaptation of model M_2.

be used to treat non-compliant process instance *I*.

Assume that we have no special strategy to cope with this situation and put the list of already performed activities into the SoDA. Then, we let activity *X* perform in the *real mode*. The respective data are stored on the tape. Then, the abstract machine toggles back to *silent mode* because the current field on the tape is empty. In the next step, activity *B* is matched in the SoDA, therefore, the current content on the tape is filled by a deferred activity item and output data of this activity are used. Activities *C* and *D* are executed in the same way. Then, the abstract machine reads activity *Y*. Activity *X*, which writes the necessary data for *Y*, has been performed as a result of the instance re-execution on the updated model M_2'. Activity *Y* is thus performed successfully and we can proceed to the next activity. Instance *I* can be migrated to adapted model M_2'. In [8], the same example is discussed with the result that *I* cannot be migrated due to the possible deadlock or run-time errors.

 ISBN: 978-1-61208-345-2

The situation described above is an example of the change, including the modification of both control and data flow. Now, we will focus on pure data change, which may also lead to errors [9]. Consider process model M_3 (Figure 4). Process instance *I* has already finished activity *C* which is data-dependent on activity *A*. However, the data output of

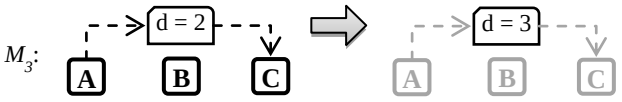

Figure 4. Model M_3 including the data dependency.

activity *A* has to be changed (the value of data variable *d* is 3 instead of 2). This requirement is simulated by inserting a new user activity item into the SoDA which includes the updated value of output variable *d*. This can be practically accomplished by offering form with current data of activity to process participant. After submitting a new value, the corresponding content is added to the SoDA.

Further, the re-execution starts and matching function *m* finds an item for activity *A* in the SoDA. The machine head also reads performed activity *A*, but with different output data, therefore, the interruption event is triggered. Consider that no custom handler is specified. Hence, the default handler to cope with this situation is used. The current content of the tape is replaced with the item found in the SoDA. As a consequence, the output of activity *A* is changed. The next activity *B* is simulated without a problem. However, activity *C* has different input (variable *d* now has a value of 3); thus, the abstract machine triggers the interruption event. A custom strategy to solve this situation takes place. We have the result of the last performance of *C* and under certain situations it may be possible to accept the result of *C*, or we can perform a rollback of this activity and then repeat *C*. It is important to say that we do not needlessly roll back activity *B*.

IV. The Contribution of the Process Re-execution Approach

The main goal of the re-execution approach is to bring more flexibility into process change. The essential requirement is to effectively implement a state compliance test in order to avoid run-time errors [1]. We presume state compliance as a basic corrections notion. The re-execution algorithm satisfied this requirement because re-execution itself always checks whether or not the changed process model or data modifications are correct and the process instance may proceed further. State compliance is also correctly checked regardless of whether there are arbitrary loop constructs in the model. Moreover, the first activity which may cause possible violation is automatically detected.

The second advantage of the re-execution approach is the fact that the process state is always properly reconstructed. Assume, for example, a non-compliant process instance which includes more activities that must be compensated. Additionally, these activities are in a loop body and we need to revert the process state into the second iteration, for example. Performing such partial rollback may be difficult because it is essential to revert all the necessary data variables to their correct values, including the loop control variables. The re-execution algorithm, however, helps us with this complicated situation because the execution is always performed from the beginning. Hence, all data variables (including the loop control variables) are evaluated again and have the correct values.

Partial rollback is not always possible; therefore, we need to choose a different strategy in order to cope with a non-compliant process instance. Some of the previously described strategies can be used. However, in some cases a custom solution according to the semantics of change is necessary. The re-execution approach brings all the essential information for implementing such a custom strategy, because we have the full context of activity which caused inconsistency during execution. The data on the machine tape as well as the content of SoDA can be taken into account to flexibly treat a non-compliant process instance.

Due to the store of deferred activities, we can reuse activities that have already been performed. This is important because the rollback of activities is connected with loss of work which is usually not acceptable for users [3].

V. The Re-execution Approach in Practice

During our research, we developed a prototype of the workflow engine which is based on the re-execution algorithm presented in this paper and successfully interpreted several processes from the area of human resources (the emergence of a new employee, correction of bonus distribution and traveling command).

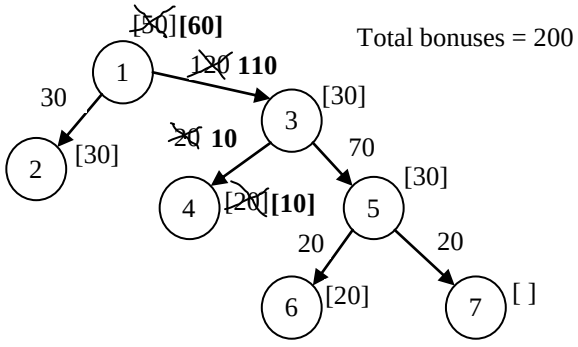

Figure 5. The bonus distribution.

Now, we demonstrate the re-execution approach on the real business process which models the correction of bonus distribution in the enterprise. The bonuses are distributed through the hierarchical organizational structure depicted in Figure 5, where the circles represent the enterprise departments. The number inside the circle identifies the department and indicates the order in which the distribution is made. We define the activity *Divide Bonuses* (*DB*), which has two input parameters – the identification of the department and the total amount of bonus to divide. The result of the activity includes the bonuses for employees within the processed department (the number in square

brackets in the figure) and the value of bonuses for all subordinate departments (the value of the edges leading from the respective department).

Assume that activity *DB* for department 6 is finished and bonuses for department 7 have not been divided yet. Further, department 1 needs more bonuses for its own employees and thus the manager of this department changes the benefits for subordinate department 3. The data output change is shown in Figure 5. The input of *DB* for department 3 has changed to 110. If we apply the extended state compliance test, including the data inputs and outputs of activities described by Rinderle-Ma [9], then the process instance is not state compliant because the input of activity for department 3 has changed. The bonuses for department 4, 5 and 6 have already been divided; thus we must perform rollback of the respective activities to get the process instance into the compliant state.

Assume, further, that the bonuses for subordinate department 4 will be changed from 20 to 10 and bonuses for department 5 will stay the same (70). In this case, the rollback of the activity for department 5 and 6 is useless and the activity *DB* for these departments has to be repeated. In practice, the process participants were informed that the division of bonuses had been cancelled and then they had to repeat activity *DB* with the same value of bonuses.

Now, we investigate the application of the re-execution approach on the presented business process. The re-execution algorithm detects the changed input of activity *DB* for department 3 and interrupts the execution. Activities for departments 4, 5 and 6 are deferred into SoDA. The new division of bonuses for department 3 has to be made again and thus rollback of the particular activity is essential. The activity for department 4 has also changed the input and we use the same strategy as for department 3. The activities for department 5 and 6 are, however, matched in SoDA and thus the rollback is not performed. The algorithm automatically recognized that the new results of *DB* for department 3 do not affect *DB* for department 5 or, consequently, for department 6. Finally, the division for department 7 is performed.

We focused on different kinds of changes related to the process model, such as the insertion, deletion or movement of activity. This tested our ability to cope with process instances which had progressed too far. The result of this test confirmed our expectation that the re-execution approach significantly increases the flexibility of treating such instances. This is because if we know a semantic of the change and use the information from the re-execution algorithm, we can create really customized strategy, thus increasing the probability that the process instance will be successfully relinked to the adapted model.

The limitations of the presented approach lie in the expectation that we assume that the activity acts as a pure function which transforms its inputs to outputs without any side-effects and that the usage of external data must always be encapsulated by the activities. Reference to an outside data structure or data taken from a foreign database, which are used directly in the process model, may be potentially changed during execution, thus, possibly causing inconsistency leading to unexpected results. This limitation does not mean that the process must be run in a completely isolated data environment. We can allow the data output modification of some activity as was described in model M_3 (Figure 4). However, the decision about whether we use the stored result or a new value must be controlled by the particular activity.

Our research also focused on the memory complexity of the re-execution approach. We need to store all relevant activity inputs and outputs, which leads to higher memory consumption. However, it is important to note that we need this data in order to present process instance execution to workflow participants as well as to support administrator intervention, if necessary. Moreover, the content of the tape can be used for further analysis and process mining [10]. In comparison with the approach presented by Rinderle-Ma [9], we do not need to store the complete state of the process instance, because many supporting data variables and conditions are automatically evaluated during re-execution. For large data structure changes, an approach based on saving the differences can be used.

We analyze the time complexity of the re-execution algorithm. The time complexity is defined as the number of steps which the re-execution algorithm must perform in order to execute the process instance. If we have n activities and every activity is performed n-times during re-execution, then the time complexity is quadratic (n^2). Although this may be a potential drawback, we should consider that the overall time spent doing particular asynchronous activities is significantly longer than the total time it takes the re-execution algorithm itself. For example, a manager must read the complete report and other related documents to authorize a decision. As such, the time needed to perform the respective activity in the process can take hours or maybe days. A forward evaluation of process instance is in some cases less time-consuming than a complicated rewind of the process state in order to perform a partial rollback.

We also discovered that the idea of process re-execution enables us to use a common high level programming language. If we create a process model as a program, which is capturing the way that the process is executed, then we must solve the problem with the simulation of long-running processes, because the code of the program finishes immediately. The re-execution approach, however, successfully solves this problem. We can simulate step-by-step the instructions of the program that represent the activities of the process model, therefore, also the asynchronous activities can be executed properly.

The usage of universal high level programming language instead of specialized process modeling language has many advantages. We can easily reuse and extend existing process models and modeling tools. The concepts of object-oriented programming, including encapsulation, inheritance and exceptions, may be applied. We can also use the standard development environment for process modeling, debugging and testing, thus improving the maintenance and flexibility of the created process models.

 ISBN: 978-1-61208-345-2

VI. RELATED WORK

There are many approaches dealing with the correctness of dynamic process change. A detailed description of these approaches and their comparison can be found in [1]. The strict version of state compliance is used by WIDE Graphs [4]. ADEPT WSM-Nets [12] use the relaxed version of state compliance, which is loop-tolerant. The different classes of relaxed state compliance are presented in [3]. On the other hand, WorkFlow Nets [11] are based on the graph equivalence approach. $WASA_2$ Activity Nets use the instance information described in the form of the purged instance graph [13]. The description of other existing frameworks for process flexibility can be found in [7].

Now, we can compare the re-execution approach with given approaches on the basis of the ability to solve typical problems connected with dynamic change. We investigate these problems, which are given in [1]:

1) Changing the Past: If the change affects the region of the process model which has been passed by the process instance, then some control or data inconsistencies may arise. The re-execution algorithm is able to tackle such situations because the activities that have already been performed are deferred, as was presented in several examples. WIDE completely prohibits the changes of the past, as do ADEPT and $WASA_2$. The inheritance transformaiton rules in WF nets enable the changing of the past.

2) Loop Tolerance: Loop constructions can cause that the application of too strict correctness criteria excludes the process instance from migration, although the instance is compliant with the changed model. Re-execution is always performed from the beginning and thus the state of the process instance is properly reconstructed, including the states of the loops. WIDE uses a strict compliance criterion and thus it is not loop-tolerant. $WASA_2$ is also not loop-tolerant due to the usage of acyclic graphs. ADEPT and WF nets are loop-tolerant.

3) Dangling States: If the approach does not distinguish between activated and started activities, the deletion of some activity may be forbidden in some cases, or, conversely, the already running activity is deleted. This is a weakness of the re-execution approach because the state of the activity is not provided for the treatment of the process instance. The problem with dangling states is present for WIDE because its history logs contain only activity entries. The activity state is not contained in WF nets. ADEPT stores information about the state of the activity; $WASA_2$ distinguishes between started and non-started activities.

4) Order Changing: This problem is connected with the swapping of activities, or with the sequentialization/ parallelization of activities. The re-execution approach detects the swapped activity correctly and offers the necessary information to cope with this problem. The putting activity in parallel or removing the parallel branch can also be handled because we store the information about the flow. However, the detailed description of how the matching function works in this situation is beyond the scope of this paper. WIDE avoids this type of problem; ADEPT handles the change of activity order by migration conditions. However, WF nets exclude order changing; $WASA_2$ criteria do not allow order changing.

5) Parallel Insertion: The insertion of a new parallel branch into the process model will be the subject of our future research. WIDE, ADEPT and $WASA_2$ allow parallel insertion. WF nets enable parallel insertion by adding new control tokens to avoid deadlocks.

In [8], the strategies for treating the non-compliant process instance are described. The state compliance graphs are used to return the process instance into the compliant state [5]. The partial rollback of the process instance and the compensation activities are also discussed by Sadiq et al. [5]. More advanced strategies for process migration are given in [6].

VII. CONCLUSION AND FUTURE WORK

In this paper, the process re-execution approach was presented to support better flexibility when coping with problems related to dynamic process change. We showed how this approach efficiently implements the essential state compliance test in order to uncover process instances which cannot be relinked to the new process model. The re-execution algorithm always ensures proper reconstruction of the process state, thus the necessary partial rollback of some activities can be performed safely. If there is inconsistency both in control or data flow caused by the adaptation of the process model, the re-execution approach brings us all the possible information needed to implement a custom strategy for handling a non-compliant process instance. The re-execution approach also makes it possible to use high level programming language to create a flexible and well maintainable process model. We implemented the presented approach as a prototype of the workflow engine and tested all the described features.

In future work, we will focus on the transformation of activity signatures in order to solve problems with different count and types of input and output activity parameters after the application of dynamic change. The model version will also be taken into account during transformation. We will further investigate the possibilities of using high level programming language for process modeling.

ACKNOWLEDGMENT

This research was supported by the grants of MPO Czech Republic TIP FR-TI3 039, the grant FIT-S-10-2, and the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

REFERENCES

[1] S. Rinderle, M. Reichert, and P. Dadam, "Correctness Criteria for Dynamic Changes in Workflow Systems – A Survey," Data and Knowledge Eng., vol. 50, 2004, pp. 9–34, doi: 10.1016/j.datak.2004.01.002.

[2] M. Weske, Business Process Management: Concepts, Languages, Architectures. Springer, 2007, 368 p. ISBN 978-3-540-73521-2.

[3] S. Rinderle-Ma, M. Reichert, and B. Weber, "Relaxed compliance notions in adaptive process management systems," Proc. ER'08. Lecture Notes in Computer Science, vol. 5231, Springer, 2008, pp. 232–247, doi: 10.1007/978-3-540-87877-3_18.

[4] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Workflow evolution," Data and Knowl. Engineering, vol. 24, 1998, pp. 211–238, doi: 10.1016/S0169-023X(97)00033-5.

[5] S. W. Sadiq, O. Marjanovic, and M. E. Orlowska, "Managing Change And Time In Dynamic Workflow Processes," Int. J. Cooperative Inf. Syst, vol. 9, 2000, pp. 93-116.

[6] S. Rinderle-Ma and M. Reichert, "Advanced migration strategies for adaptive process management systems," Proc. 12th IEEE Conference on Commerce and Enterprise Computing (CEC' 10), IEEE Press, Nov. 2010, pp. 56–63, doi: 10.1109/CEC.2010.18.

[7] N. Mulyar, M. Schonenberg, R. Mans, N. Russell, and W. van der Aalst, "Towards a taxonomy of process flexibility," Technical Report BPM-07-11, BPMcenter.org, 2007.

[8] M. Reichert and B. Weber, Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies. Springer, 2012, 515 p. ISBN 978-3-642-30408-8.

[9] S. Rinderle-Ma, "Data Flow Correctness in Adaptive Workflow Systems," EMISA Forum, vol. 29, 2009, pp. 25-35.

[10] M. Pospíšil and T. Hruška, "Business Process Simulation for Predictions," BUSTECH 2012. The Second International Conference on Business Intelligence and Technology, Nice, IARIA, 2012, pp. 14-18, ISBN 978-1-61208-2.

[11] W.M.P.van der Aalst and T. Basten, "Inheritance of workflows: an approach to tackling problems related to change", Theoretical Computer Science, vol. 270, 2002, pp. 125–203, doi: 10.1016/S0304-3975(00)00321-2.

[12] M. Reichert and P. Dadam, "ADEPTflex-supporting dynamic changes of workflows without losing control", Journal of Intelligent Information Systems, vol. 10, 1998, pp. 93–129, doi: 10.1023/A:1008604709862.

[13] W.M.P.van der Aalst, M. Weske, and G. Wirtz, "Advanced topics in workflow management: Issues, requirements, and solutions", Int. J. Integrat. Design Process Sci., vol. 7, 2003, pp. 49–77.

[14] W.M.P.van der Aalst, "The application of Petri nets to workflow management", J. Circuit. Syst. Comp., vol. 8, 1998, pp. 21–66, doi: 10.1142/S0218126698000043.

 ISBN: 978-1-61208-345-2