A Formal Execution Semantics for Sophisticated Dynamic Jumps Within Business Processes

Thomas Bauer
Hochschule Neu-Ulm, University of Applied Sciences, Wileystr. 1, 89231 Neu-Ulm, Germany
thomas.bauer@hnu.de

Keywords: Dynamic Jump, Flexibility, Process Engine, Workflow Engine, Run-Time, Execution State.

Abstract: At business processes (BP) execution, in exceptional cases (e.g. to save time or to correct errors), users must have the possibility to jump forward and backward in the BP. Currently, this topic is hardly respected in scientific literature and only insufficiently realized by commercial BP engines. This paper develops a formal execution semantics for dynamic jumps. It does not only respect simple forward and backward jumps within sequences of activities, but comprehensive requirements as jumps into and out of parallel branches or within loops. Furthermore, the intended behaviour of concerned activities can be modelled, i.e., they may be caught up later or their results (output data) may be preserved and reused at their later repeated execution after a backward jump.

1 INTRODUCTION

In exceptional cases, users of applications that are based on process management systems (PMS), must be able to deviate from the modelled business process (BP) at run-time (Reichert and Weber, 2012). The project CoPMoF (Controllable Pre-Modelled Flexibility) handles deviations that are pre-modelled already at build-time (Bauer, 2019, 2020, 2021). This includes optional edges (Bauer, 2023a) and advanced control-flow dependencies between activities (Bauer, 2023b). A further topic are dynamic jumps. This means that the user detects the exceptional situation and triggers the jump by selecting a target activity. The process execution continues with this activity, i.e. a dynamic change (Weber et al., 2008) of this process instance is performed. To avoid errors and maintain process safety, we consider jumps that are not completely arbitrary. Instead, the possible source and target activities, user rights for triggering the jump, and the intended behaviour were pre-modelled already at build-time. Since, at run-time, the user is free to trigger such a jump at any point in time, this is still a dynamic operation. In (Bauer, 2022) we have introduced dynamic jumps that fulfil sophisticated requirements. That work, however, only describes these requirements and examples from practice, but no formal execution semantics. In the following, we develop such a semantics that enables a process engine to control a BP when dynamic jumps occur. This includes different types of run-time behaviour selectable by the user by defining configuration options, as well as jumps within complex control flow structures, e.g. jumps into and out of parallel branches (after an AND-Split).

A change management process of the automotive domain (cf. Fig. 1a) is used to demonstrate that dynamic jumps are very relevant in practice: An employee requests the change of a vehicle part (activity a). Then, a development engineer details the request (activity b) and rates the costs and effort of this change (activity c). With the composed activity d, the following other business domains rate the request in parallel: After-Sales (d1), Production (d2), Marketing (d3), and Prototyping (d4). If the project leader decides in activity e to accept the request, it is realized in activity f. A certain change may be necessary because of a new law. In this exceptional situation, the ratings of the activities c and d are not relevant since the change must be performed anyway. To save time and effort, after completing activity b, its actor triggers a dynamic forward jump to activity e. Another exception may be that, during the rating of consequences of the change for prototyping (activity d4), an actor detects errors made at the execution of activity b. Therefore, he triggers a backward jump to activity b with the result that the errors can be corrected at its repeated execution. Such jumps are very rele-

vant in practice, but hardly considered in scientific literature. Furthermore, commercial PMS enable jumps only in a very restricted manner (cf. Section 2).

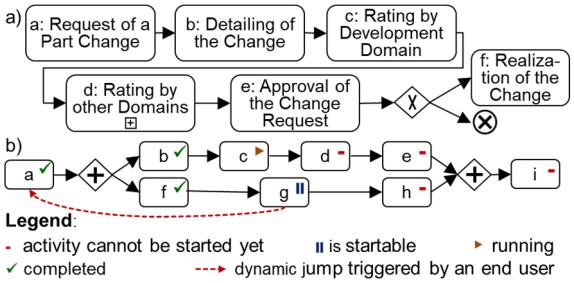

Figure 1: a) Change Management Process (CMP) for Parts b) A Backward Jump Out of a Parallel Region.

The desired behaviour of a jump can be ambiguous: Assume the dynamic backward jump to activity a depicted in Fig. 1b. It is triggered by the actor of activity g before starting it. At this point in time, the activity b of the parallel branch was already finished. After completion of activity a (i.e. at the forward execution occurring after the jump), it is not obvious whether activity b shall be executed again or whether it is possible to keep its originally created output data. In the latter case, the time and effort for its repeated execution can be saved. Activity f belongs to the same branch as the source activity g of the jump. Again, it is not clear whether its execution must be repeated. Furthermore, the execution of the currently running activity c, that belongs to the other branch, may be continued or aborted at the jump. The appropriate behaviour depends on the question, whether the execution of activity c is affected by the jump or by the eventually changed output data of the activities a and b. We present an execution semantics that allows to define the desired behaviour of dynamic jumps in such cases.

Section 2 discusses the state of the art and the resulting research gap. Section 3 summarizes the requirements for jumps. The formal execution semantics is presented in Section 4.

2 STATE OF THE ART

The research gap is identified by analysing literature and relevant functions of a commercial PMS.

Some publications mention dynamic jumps or may serve as work-around for their realization. They are presented in the following since, unfortunately, there does not exist more specific literature concerning sophisticated dynamic jumps.[1] (Russell and Hofstede, 2006) present control-flow patterns, but do not mention dynamic jumps. With the pattern “Arbitrary Cycles”, however, forward and backward edges can be modelled. This allows to pre-model edges for jumps at build-time. The pattern is described only, but no special behaviour for jumps as catching up bypassed activities (cf. Section 1) is mentioned.

At (Reichert et al., 2003), expectable jumps are pre-modelled at build-time and mapped to regular building blocks of the BP meta model. To enable this, priorities for activities and edges are introduced. The result of this mapping is a quite complex process graph. Since no configuration options are respected, it cannot be defined, for instance, whether an activity shall be repeated after a backward jump.

(Reichert and Dadam, 1998) mention dynamic jumps and special requirements, as catching up skipped activities. Furthermore, it shall be definable whether this is possible at any time or only before a given other activity was started. However, no execution semantics is presented for such requirements.

Since dynamic jumps are hardly respected in scientific literature, a commercial PMS is analysed as well: IBM has a long-term experience with such products. Therefore, we assume that their products are at least similar powerful as others. The first IBM product that supported jumps was WebSphere Process Server 6.1.2 (IBM, 2008), but with many restrictions. At the current IBM product Business Automation Workflow (IBM, 2022) jumps are still limited: Jumps into or out of regions with parallel branches are not allowed. A reason for this may be that their intended behaviour can be ambiguous (cf. the example of Fig. 1b). Furthermore, functionality, as catching up bypassed activities, is not supported.

As explained, there does not exist scientific literature that handles sophisticated dynamic jumps and commercial PMS do not support such jumps, as well. The sole exception is our previous work (Bauer, 2022), but it only describes requirements and presents examples from practice to show their necessity. The intended behaviour at BP execution (run-time) is only described informally by examples. That means, there does not exist a formal execution semantics for sophisticated dynamic jumps. This paper reduces this research gap by developing and explaining the required formal execution rules. Before, we explain why such execution rules are the best way to define a formal execution semantics.

[1] Business process, Workflow, Process engine - each in combination with the word Jump - were used when searching for literature, and additionally: Forward jump, Backward jump - each combined with Process.

3 REQUIREMENTS

(Bauer, 2022) introduces requirements for sophisticated jumps including the pre-modelling of such jumps and the visualization of jump edges. The requirements also contain the definition of configuration options by the BP designer at build-time and their modification by the user who triggers the jump at runtime. Now we focus on requirements that are relevant for the execution semantics presented in Section 4. To be able to refer to specific requirements, each has an identifier which indicates its category: Fx = **F**orward jump, Bx = **B**ackward jump, Px = Jump into or out of a **P**arallel branch, Lx = Jump within a **L**oop.

3.1 Forward Jumps

A forward jump may happen before its source activity (the starting point of the jump) is started. The configuration option CatchUpMode allows to define whether this source activity s shall be caught up later on (*Requirement F1*: CatchUpMode(s)=true) or whether it shall be omitted (CatchUpMode(s)=false). A jump shall be even allowed, when its source activity s was already started: For the case that catching up is desired (CatchUpMode(s)=true), the execution of this source activity continues (despite the jump). Otherwise (CatchUpMode(s)=false) it is aborted. Assume for the BP of Fig. 1a that the engineer has already started the execution of activity c when he detects that this change request is very urgent and triggers the jump to activity e (similar as described in Section 1). With CatchUpMode(c)=false, activity c is aborted and the process continues with activity e. That means, the CatchUpMode defines whether the source activity of the jump shall be continued or omitted (and therefore aborted, if it is already running).

For bypassed activities (located between the source and the target activity of the jump) it can be defined as well, whether they shall be caught up (*F2a*). Often these activities shall be omitted (CatchUpMode(x)= false) because this is the "normal intention" of a user when triggering a jump. But it can be necessary to execute missed activities later on (CatchUpMode(x)=true). At the example of Fig. 1a, for each activity that is bypassed by the jump (activities c and d1 to d4) it has to be analysed, whether its output data is required for the further process execution. In such a case, it must be caught up later on. For example, it may be necessary to catch up activity d2 since its output data (the comments of the production domain) are required by activity f to identify the necessary changes of production machines. Therefore, for each activity, it can be defined whether it shall be caught up after a jump (*F2a*). Furthermore, it can be modelled that catching up must happen before a given activity (here: activity f) can be started (*F2b*).

3.2 Backward Jumps

As explained in Section 1, a backward jump may be necessary to correct data that was erroneously captured at a preceding activity t. After the jump, this target activity t is repeated, and the succeeding process steps are executed again (now with correct data); i.e. the backward jump is followed by a "second forward execution". It is possible to define how the original results (output data) of each activity x shall be treated at this forward execution (*Requirement B1*):

1. RepeatMode(x)=Discard: Its original results are discarded and activity x is executed again (normally) as at its first execution (*B1a*).
2. RepeatMode(x)=Control: The original output data of activity x are preserved, but at the later forward execution, it is executed again. At this, a form pre-filled with these original data can be presented to the actor. He has to control these data and may correct it, if necessary (*B1b*).
3. RepeatMode(x)=Keep: The activity x is not executed again, i.e. all output data of this activity stay unchanged (*B1c*).

At a backward jump, it may be meaningful to abort or to continue the execution of its source activity s. The same applies to activities that are located after activity s in the process graph. Such activities can be even started. For instance, at the jump depicted in Fig. 1b, it may be meaningful to complete activity g (the source activity of the jump) if its output data will not change because of the backward jump and the following forward execution. After completion of activity g, the activity h may be started despite this jump if it only depends on the (already correct and in future unchanged) output data of activity g. These activities g and h are executed in a "preponed" manner, i.e. earlier than with a "classic process execution" after a jump. The desired behaviour can be selected with the configuration option ContinueMode(x) (*B2*):

1. ContinueMode(x)=Abort (i.e. no Start and no Completion): The activity x shall be aborted (automatically) if it is currently executed (*B2a*). For instance, it is meaningful to abort activity x if its results will be discarded anyway at the later forward execution because of the RepeatMode(x)=Discard (cf. B1a). It is wasted effort to complete (or even start) the execution of such an activity.
2. ContinueMode(x)=Complete (i.e. no Start): An already started activity x can be completed, but

it must not be started if it is not running yet (*B2b*) in order to avoid the loss of already performed work, that perhaps can be reused at the later forward execution. If ContinueMode(y) =Complete was selected for the successive activity y as well, it cannot be started after the completion of activity x. This can be meaningful since, until now, activity y was not started, i.e. no work can be lost.

3. ContinueMode(x)=Start&Complete: With this ContinueMode, the activity x can be even started, after completion of its preceding activity (*B2c*). This is meaningful if the output data of activity x will be used later on anyway, e.g. because of RepeatMode(x)=Keep (cf. B1c). Then, much time is available for the execution of this activity x, till the forward execution (after the backward jump) reaches activity.

At a backward jump, it can be necessary to compensate an activity x that was already executed before the jump. This is defined with RepeatMode(x)=Compensate (*B3*); e.g., to revoke an order that was already sent to a supplier. For this purpose, a "compensation activity" is modelled and executed at the jump.

3.3 Jumps and Parallel Branches

The presented requirements are especially meaningful for activities of parallel branches: At Fig. 1b, the actor of activity g (of the lower branch) triggers a jump, e.g. because he has detected an error within the process data. If this error does not concern the upper branch at all, the activity c and its successors can be continued without causing any problems.

Forward Jump: As described in Section 3.1, the CatchUpMode defines whether an activity, that was bypassed by a forward jump, shall be caught up. Now, this CatchUpMode becomes relevant for activities of other parallel branches, as well (*Requirement P1*). Assume for the process of Fig. 1b, that the actor of activity c triggers a forward jump to activity i. Furthermore, assume that the execution of activity g has not started at this time. Then, the CatchUpMode of the activities g and h (located in parallel to the jump source activity c) defines whether they shall be caught up after the jump. As already described in Section 3.1, the CatchUpMode of the activities c, d, and e (that belong to the same branch as the source activity c of the jump) defines whether they are caught up.

Backward Jump: As described in Section 3.2, for activities that become executable after a backward jump, again, the RepeatMode defines whether their original output data shall be discarded, controlled, or kept, i.e. whether this activity must be repeated in fact. Now, this RepeatMode becomes relevant for activities of parallel branches, as well (*Requirement P2*). The backward jump of Fig. 1b was triggered by the actor of activity g (belonging to the lower branch), e.g. because of erroneous data created by activity a. If these data are not used by the already completed activity b, it is very meaningful to keep its results.

As already explained, the ContinueMode defines whether activities, that were not completed when executing the jump, can be started or completed despite. Now, ContinueMode is relevant for activities that are located in parallel to the source activity of the jump, as well: At the example of Fig. 1b, activity c is currently executed. Since it is located in a parallel branch, perhaps, it may be not affected by the reason for this jump at all. Therefore, it is especially meaningful that ContinueMode(c)=Complete allows its completion. If the same applies to its succeeding activities d and e, Start&Complete should be used as ContinueMode, to allow their preponed execution.

3.4 Jumps Within Loops

A forward jump into a future or a backward jump into a previous iteration of a loop may be required.

Forward Jump: At a forward jump, it can be meaningful to skip the remaining activities of the current iteration (CatchUpMode=false). But it can be also desired, that they shall be caught up after the forward jump (CatchUpMode=true). That means, the CatchUpMode is relevant for loops as well. Forward jumps into an iteration after the next one are hardly relevant in practice. If they are required despite, this can be realized with multiple forward jumps. Therefore, *Requirement L1* only demands a forward jump to an arbitrary activity of the next loop iteration.

Backward Jumps: Jumps to a previous iteration of a loop may be required as well, for instance when an error was made at activity execution of this iteration. The user selects this target activity instance when triggering the backward jump. Since this error may have happened at any past activity instance, a backward jump to an arbitrary iteration of a loop must be allowed (*Requirement L2*). Again, RepeatMode(x) defines whether an activity x shall be repeated at the subsequent forward execution. ContinueMode(x) defines whether its early execution (at the current and future loop iterations) is allowed.

4 EXECUTION SEMANTICS

Section 4.1 explains state changes of activity instances occurring at normal process execution, i.e.

without dynamic jumps, as for instance described in (Object Management Group, 2011; Reichert and Dadam, 1998; Weske, 2019) . In principle, with our approach, this life-cycle stays unchanged. Section 4.2 presents several additional execution rules that are required to realize dynamic jumps.

The (also possible) alternative approach, to define the execution semantics based on an existing formalism (e.g. Petri-nets), was not chosen since commercial BP engines are often based on BPMN and the corresponding execution rules. For the realization of dynamic jumps, it shall be only necessary to integrate the additional execution rules into such a BP engine. Since they normally are not based on approaches like Petri-nets, it would be much more difficult to integrate such a formalism.

4.1 States of Activity Instances

At classic process execution (i.e. without dynamic jumps), all activity instances have the state Inactive (cf. Fig. 2) when a process instance is started. The state of the start activities of the process (i.e. activities at the beginning of the process graph) is directly changed to Active. In general, activities with this state Active are offered to actors in their worklists.

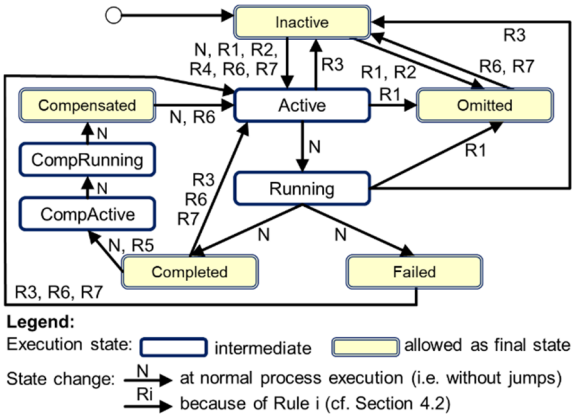

Figure 2: Execution States of Activity Instances.

Whenever an actor selects an activity for execution, its state is changed to Running. Its successful completion results in the state Completed, a completion with a failure in the state Failed. Whenever an activity reaches one of these both states, the state of the succeeding activity is set to Active, with the result that it can be executed now. In the case of Failed and at Join-Gateways, it depends on the modelled behaviour, whether the state of the succeeding activity switches to Active, in fact. The corresponding details are omitted here since this behaviour is not changed by dynamic jumps.

A user may decide to undo a completed activity, e.g. to execute it again, later. For this purpose, a compensation activity is executed. The activity state is changed to CompActive, with the result that the compensation activity is included into the user worklists. Starting the compensation activity results in the State CompRunning and its completion in Compensated. In the case that the regular activity shall be executed again, now, its state switches to Active, again.

4.2 Execution Semantics at Jumps

This section defines the formal execution semantics for sophisticated dynamic jumps covering all requirements of Section 3. The semantics shall be presented in a well understandable way. However, to ensure unambiguousness, the execution rules must be described formally. To increase readability, beforehand, they are explained in detail.

To avoid unnecessary complexity, aspects that stay unchanged compared to the normal process execution (i.e. without dynamic jumps) are omitted, e.g. the behaviour at AND-, XOR-, and OR-Join-Nodes.

4.2.1 Forward Jumps

For the source activity s of a forward jump, it may be desired that it shall be caught up (CatchUpMode(s) =true, cf. Requirement F1). If this applies, State(s) stays unchanged, to allow that its execution is continued. Otherwise, activity s may be aborted (if it was already started) and switches to the state Omitted (cf. Fig. 2). The target activity t of the jump shall become executable. Therefore, its state is changed to Active.

A bypassed activity b is marked with the flag Bypassed(b)=true (initially all flags of all activities have the value false (cf. Table 1 for all flags). The purpose of the flag Bypassed is that, later on, the BP engine is able to detect that this activity shall be caught up (cf. Rule 2). The following execution rule defines this behaviour at a forward jump formally:

Rule 1: A forward jump has the source activity s and the target activity t. The activities B are located between s and t (i.e. $\forall b \in B$ holds: $b \in$ Successor*(q) $\wedge$ $b \in$ Predecessor*(t)[2]). Then, the following changes are made at the jump:

State(s) = Omitted, if CatchUpMode(s) = false

State(t) = Active

$\forall b \in B$: Bypassed(b) = true

[2] Successor*(x) calculates the activities that are located indirectly after activity x (without considering loop edges). Predecessor*(x) calculates the activities that are located indirectly before this activity x.

Table 1: Flags that influence Activity Execution at Jumps.

Name *initial value*	**Meaning of the Value true for the Activity a**
Bypassed *false*	Activity a was bypassed at a forward jump and may be caught up later on (if CatchUpMode(a) = true holds)
Preponed *false*	Act. a was executed in a preponed manner after a backward jump (a is located behind the source activity of the backward jump)
JumpedBackwards *false*	Activity a was bypassed at a backward jump (it may be already executed regularly earlier at normal forward execution)
UseOldResults *false*	Activity a was executed regularly and was located within the area of a backward jump. This flag signals that existing (old) result data must be considered at the repeated forward execution of activity a

In contrast to the flag Bypassed, Omitted is a new activity state. The reason for this decision was that Omitted can be a final state of an activity instance. Introducing this state allows to detect that this activity was omitted, i.e. it was not executed in fact.

The classic execution rules (cf. Section 4.1) must be modified slightly in order to enable catching up bypassed activities (Req. F2a): Whenever an activity p reaches the state Completed, its succeeding activity a switches to Active, if State(a) was Inactive before. The last condition is necessary, because catching up activities will reach the target activity t of the jump (anytime). This activity t already became startable (State(t)=Active) immediately after the jump. Therefore, it does not have the state Inactive anymore. Since it shall not be executed multiple times, the state of an activity a is not changed (again) to Active, if State(a) ≠ Inactive holds (i.e. it was the target of the jump). Condition (2) in Rule 2 handles this case.

When catching up activities, after completing or omitting an act. p (the source activity of the jump or one of its successors), its directly succeeding act. a becomes executable. The corresponding rule, that was already sketched in Section 4.1, must be extended for this purpose: It has to consider that an act. a must be activated even in the case that its preceding activity p was omitted due to its CatchUpMode(p)= false. This can be detected by State(p) =Omitted, what is respected by the condition (2) of Rule 2.

Rule 2: An activity a has the flag Bypassed(a) = true. If the following condition holds
(1) State(a) = Inactive and
(2) the preceding activity p of a has reached a State(p) ∈ {Completed, Omitted),
then its state is changed to

$$\text{State(a)} = \begin{cases} \text{Active}, & \text{if CatchUpMode(a) = true holds} \\ \text{Omitted}, & \text{otherwise} \end{cases}$$

It can be defined for a bypassed activity x that it shall be caught up before the execution of a different activity y starts (F2b). In this case, activity y cannot be started until activity x was finished (successfully or with a failure). To ensure this behaviour, the rule for the (normal, cf. Section 4.1) state change from Inactive to Active is extended by this additional condition:

Rule Extension: Given an activity x with Bypassed(x)=true, which has to be caught up before activity y is started. Then, it is only allowed that the state of activity y changes to State(y) = Active, if State(x) ∈ {Completed, Failed} holds.

4.2.2 Backward Jumps

If a backward jump starts at a source activity s that is currently executable (i.e. State(s)∈ {Active, Running}) its execution is stopped in the case that it has *ContinueMode(s)=Abort* (B2a). Its state changes to State(s) =Inactive. The same state results at *ContinueMode(s)=Complete* if the execution of activity s was not started yet, because this mode does not allow to start an activity (B2b). With this ContinueMode, if the execution of activity s was already started, however, its execution shall be continued. Therefore, State(s) stays unchanged. To enable detecting that this is a preponed activity execution, activity s is marked with the flag Preponed(s)=true. The same (i.e. unchanged State(s) and Preponed(s)=true) results at *ContinueMode(s)=Start&Complete* since then, activity s can be started and executed in a preponed manner, independently of its current state (B2c).

The state of the target activity t of a backward jump is always changed to State(t)=Active. Then it can be executed (again). The flag JumpedBackwards(t)=true is used to signal that it was located within the area of a backward jump.

Rule 3: Activity s is the source activity of a backward jump and activity t is the target activity. Then, the new state and the Flag Preponed(s) of activity s result as follows, depending on its original state and ContinueMode:

Original State	State(s) = Active	State(s) = Running
ContinueMode(s) = Abort (B2a)	State(s) = Inactive Preponed(s) = false	State(s) = Inactive Preponed(s) = false
= Complete (B2b)	State(s) = Inactive Preponed(s) = false	State(s) = Running Preponed(s) = true
=Start&Complete (B2c)	State(s) = Active Preponed(s) = true	State(s) = Running Preponed(s) = true

For the target activity t, these changes are performed: State(t) = Active, JumpedBackwards(t) = true

If the preponed execution of an activity a is completed (State(a)=Complete), its succeeding activity b may become startable. If this applies, its State(b) is changed to Active and its flag Preponed(b) is set to true. However, it is only allowed to start a succeeding activity b if it has ContinueMode(b) =Start&Complete. Otherwise, its state stays Inactive and its flag Preponed(b) stays false.

Rule 4: Activity a reaches State(a) = Completed and Preponed(a) = true holds. Activity b is the succeeding activity of a. For the case that activity b has ContinueMode(b) = Start&Complete, the following changes are made: State(b) = Active, Preponed(b) = true

For an activity a that was located within the area of a backward jump (i.e. between the target activity t and the source activity s), the behaviour at the subsequent forward execution depends on its RepeatMode(a): It may be completely executed once again, its result data may be controlled, or they may be kept unchanged (B1). After the backward jump, it shall be possible to detect whether this activity a, originally, was completed successfully or whether it failed. Therefore, its state (Completed or Failed) is not changed at the backward jump. The flag JumpedBackwards(a)=true is used to signal that a backward jump occurred for this activity. At the later forward execution, this flag allows to detect that there already exist result data that have to be controlled or may even be used directly.

Rule 5: Activity s is the source activity of a backward jump and t is its target activity. Activity a is located between t and s (i.e. $a \in \text{Successor}^*(t) \wedge a \in \text{Predecessor}^*(s)$). Then, JumpedBackwards(a) = true is set.

In case that for an activity a RepeatMode(a) =Compensate was defined, a compensation activity a' has to be executed during the backward jump (B3). For this purpose, State(a) is set to CompActive. Then, the BP engine executes the compensation activity a', what results in the State(a)=CompRunning. After completion of a', State(a) is set to Compensated. To trigger this behaviour, Rule 5 is extended as follows:

Rule 5 (Continuation): If RepeatMode(a)=Compensate applies in addition, the following state change is performed: State(a)=CompActive

For an activity a, that is located within the area of a backward jump, its behaviour at the later forward execution depends on its RepeatMode(a): (1): With RepeatMode(a)=Keep, it is not executed again if it was completed successfully at its original execution, i.e. if State(a)=Completed was reached (B1c). Therefore, this state stays unchanged, with the result that this activity is not offered to actors again, and the succeeding activity can be started now. (2a): For an activity a with RepeatMode(a)=Control, the results, that were created before the backward jump, are kept (B1b). At the subsequent forward execution, this activity a is started again, therefore, its state is set to Active. The old result data shall be re-used and the user is informed that he has to control and adapt these data (if necessary). The BP engine detects that this behaviour is required, because Rule 6 sets UseOldResults= true. (After the completion of each activity, this flag is set to false. Therefore, a further execution, that may occur in future (e.g. within a loop), occurs regularly.) (2b): At RepeatMode(a)=Keep, the same behaviour is desired if activity a was not completed successfully at its original execution (i.e. State(a)= Failed), since already created result data may be relevant, nevertheless. Therefore, they are not discarded, but controlled and adapted by the user. (3): An activity a with RepeatMode(a)=Discard must be repeated completely at forward execution (B1a). This is reached by setting State(a)=Active. All result data, that were created earlier, were discarded because the value of the flag UseOldResults(a)=false is not changed by Rule 6. This represents the default behaviour ("otherwise"), that is also used if activity a was not executed at all, until now, e.g., because it was located in a not chosen XOR-Branch (State(a)=Inactive) or since it was bypassed by a forward jump (cf. Rule 2: State(a) is set to Omitted). The same applies for an activity a that was compensated (State(a)= Compensated). In all presented cases (i.e. 1-3), activity a is treated in a well-defined manner at the forward execution. Therefore, the flag JumpedBackwards(a) is set to false since the (former) backward jump is no longer relevant for the execution of this activity.

Rule 6: For activity a JumpedBackwards(a) = true holds and its preceding activity p changes its state to State(p) = Completed. Then, the new state of activity a results as State(a) =

$$\begin{cases} \text{Completed, (1) if RepeatMode(a)=Keep} \\ \qquad \wedge\ \text{State(a)} = \text{Completed} \\ \text{Active and UseOldResult(a)} = \text{true is set,} \\ \quad \text{(2a) if RepeatMode(a)} = \text{Control} \\ \qquad \wedge\ \text{State(a)} \in \{\text{Completed, Failed}\} \vee \\ \quad \text{(2b) RepeatMode(a)} = \text{Keep} \wedge \text{State(a)} = \text{Failed} \\ \text{Active, (3) otherwise} \end{cases}$$

Furthermore, JumpedBackwards(a) = false is set

Before it is allowed that activity a is started, it may be necessary to wait until its compensation is finished (B3). The state CompActive and CompRunning enable to detect that the compensation was not started or completed, yet. In both cases, the BP engine waits until State(a)=Compensated is reached, i.e. the compensation activity was completed. For this purpose, Rule 6 is extended:

Rule 6 (Continuation): If State(a) ∈ {CompActive, CompRunning} holds, none of these changes are performed until State(s) = Compensation is reached.

After a backward jump, the regular forward execution may reach a preponed executed activity a (i.e. Preponed(a) has the value true). Then, the "gap" between the regular execution of activities (p) and the preponed executed activities (a) is closed. This case can be detected at the completion of the regularly executed preceding activity p (State(p)=Completed) by the fact, that Rule 6 has set JumpedBackwards(p) =false earlier (cf. Condition (1) in Rule 7). Since its succeeding activity a now becomes executable in a regular manner, its flag Preponed(a) is set to false.[3] This has the consequence that even a succeeding activity (with any ContinueMode different to Start&Complete) can be started now.

If for activity p, Preponed(p)=false was set because of Condition (1) of Rule 7, and its execution finishes (i.e. State(p)=Completed), the regular forward execution has reached its succeeding activity a, as well. That means, now, this activity a can be started regularly. Therefore, Condition (2) of Rule 7 enables that Preponed(a) is set to false in this case, as well. The same happens, also because of Condition (2), with the succeeding activity a' of a, as soon as activity a finishes. Therefore, an activity a', that was waiting because of its ContinueMode(a') ∈ {Abort, Complete}, becomes startable; i.e. the whole further process can be executed from now on.[4]

The way how this "execution" is performed (i.e. discard, control, or keep the result data) depends on RepeatMode(a). This mode is respected by Rule 6 and, furthermore, Rule 6 defines the new State(a) of activity a. A difference, that results from the fact that this activity is executed in a preponed manner, is that it may be currently performed by an actor. In this case (i.e. State(a)=Running), the actor is informed that he has the possibility to stop activity execution, and start it afterwards with new input data (created by meanwhile finished preceding activities).

Rule 7: An activity p reaches State(p) = Completed. For its succeeding activity a Preponed(a) = true holds. If (1) JumpedBackwards(p) = false or (2) Preponed(p) = false holds, Preponed(a) = false is set.

[3] At the preponed start of activity p (i.e. before the "regular" process execution reaches this activity), Rule 4 sets Preponed(p)=true. Its succeeding activity a, however, cannot be started until this activity p was finished. Therefore, Preponed(a) is still false. This also applies when activity p is finished, with the result that Rule 7 is not applicable (because of Preponed(a)=true). Later, for the case that activity a was executed in a preponed manner as well, and when the regular process execution reaches this activity, Preponed(a) was already set to true (by Rule 4). Therefore, Rule 7 becomes applicable now.

[4] Rule 7 assigns the initial value Preponed(a)=false to all activities, again. Rule 6 makes the same with JumpedBackwards(a)=false, i.e. all original values are restored. Therefore, a further dynamic jump or a loop iteration can be performed without requiring any special actions.

State(a) is changed as described by Rule 6.

4.2.3 Jumps Involving Parallel Branches

Jumps into and out of regions with parallel branches do not result in fundamental changes of the execution rules. The reason is that even behaviour, that is especially useful for parallel branches, was already respected for "normal" forward and backward jumps. For instance, continuing the execution of the source activity s of a backward jump is also allowed without parallelism. This behaviour, however, is especially meaningful for activities that are located in branches that are executed in parallel to the source activity s of the jump (e.g. activity c in Fig. 1b). Perhaps, these activities are not affected by the problem that caused the backward jump. Therefore, it is meaningful to continue their execution (in a preponed manner).

That means, the Rules 1 to 7 already define the behaviour that is required for parallel branches, in principle. However, it is necessary to extend these rules in such a way that several source and target activities of a jump are respected (i.e. sets S and T of activities). Furthermore, the rules must consider that all activities belong to the area of a jump (e.g. they were bypassed) that are located between an arbitrary source activity s∈ S and an arbitrary target activity t∈ T. This results in changes for the Rules 1, 3, and 5 since they directly refer to the source activity s or the target activity t of a jump.

Rule 1': S is the set of source activities and T the set of target activities of a forward jump. The activities B are located between any activities that belong to S and T (i.e. ∀b∈ B holds: ∃s∈ S, ∃t∈ T with b∈ Successor*(s) ∧ b∈ Predecessor*(t)). Then, the following changes are made at the jump:

∀s∈ S: State(s) = Omitted, if CatchUpMode(s) = false

∀t∈ T: State(t) = Active

∀b∈ B: Bypassed(b) = true

Rule 3': S is the set of source activities and T the set of target activities of a backward jump. Then, the new state and the flag Preponed of all activities s∈ S result as defined by Rule 3.

For the target activities t∈ T, these changes are performed: State(t)=Active, JumpedBackwards(t) =true

Rule 5': S is the set of source and T the set of target activities of a backward jump. Activity a is located between activities that belong to S and T (i.e. for a holds: $\exists s \in S, \exists t \in T$ with $a \in Successor^*(t) \wedge a \in Predecessor^*(s)$). Then, JumpedBackwards(a)= true is set.

4.2.4 Jumps Within Loops

An activity a that is located within a loop may be executed multiple times. Each of these executions represent a separate activity instance that possesses its own activity state (cf. Fig. 2) These instances can be distinguished by using a consecutive number (iteration counter) in addition to the activity name. Such a composite identifier allows to identify an activity instance unambiguously, e.g. in the process history.

Jumps within loops (Requirements L1 and L2) can be realized by modifying the presented rules slightly: In addition to the "normal" control flow edges, the loop edge that is used at this jump, must be respected when the (indirect) predecessors and successors of an activity are calculated. Since the rules stay almost unchanged, in the following, they are not repeated but only the changes are explained.

Rule 1' must also respect loop edges, when calculating the activities that are located between S and T. Therefore, instead of the function Successor*(s), $Successor_L^*(s)$ is used. The latter additionally respects the loop edge that was used for this jump. Likewise, $Predecessor_L^*(t)$ is used instead Predecessor*(t).

When calculating the preceding activity p, Rule 2 has to respect the loop edge, as well.

To realize Requirement L2, similar adaptations result for backward jumps: As already described for Rule 1', Rule 5' uses $Successor_L^*(t)$ and $Predecessor_L^*(s)$. As described for Rule 2, the Rules 4, 6, and 7 must additionally respect the loop edge when calculating the preceding or succeeding activity.

5 SUMMARY AND OUTLOOK

Dynamic jumps, triggered by users at run-time, enable an appropriate reaction in exceptional cases. For instance, it may be necessary to jump forward within a BP to save time by skipping activities. Backward jumps may be required to correct errors made at the execution of previous activities. To keep process safety, instead of arbitrary jumps, the presented approach uses pre-modelled jumps. The behaviour of concerned activities is definable by configuration options. They allow to specify, for instance, whether activities that are bypassed by a forward jump must be caught up later on, e.g. since output data are required by succeeding activities. Because of this flexibility, this approach becomes applicable in many scenarios.

This paper presents a formal execution semantics, based on execution rules for state changes of activity instances. In addition, flags of activity instances were used by these rules, in order to avoid the definition of many additional activity states (one for each combination of relevant flag value and each state).

The execution rules enable the realization of a BP engine that offers sophisticated jumps. In future, usability of the presented approach has to be evaluated based on a (prototypical) implementation of such a BP engine. It shall be used to evaluate whether the users are able to handle the presented concepts, and whether they are really useful and sufficient.

REFERENCES

Bauer, T., 2023a. Behaviour and Execution Semantics of Optional Edges in Business Processes. Proc. Informatik 2023, ZuGPM-Workshop (in German).

Bauer, T., 2023b. Modelling of Advanced Dependencies Between the Start and the End of Activities in Business Processes. Proc. ICEIS 457–465.

Bauer, T., 2022. Requirements for Dynamic Jumps at the Execution of Business Processes. Proc. 12th Int. Symp. on Business Modeling and Software Design 35–53.

Bauer, T., 2021. Pre-modelled Flexibility for the Control-Flow of Business Processes, in: Enterprise Information Systems. Springer, pp. 833–857.

Bauer, T., 2020. Business Processes with Pre-designed Flexibility for the Control-Flow. Proc. ICEIS 631–642.

Bauer, T., 2019. Pre-modelled Flexibility for Business Processes. Proc. ICEIS 547–555.

IBM, 2022. Business Automation Workflow 22.x. https://www.ibm.com/docs/en/baw/22.x (accessed 11.10.23).

IBM, 2008. WebSphere Process Server Knowledge Center. https://www.ibm.com/support/knowledgecenter/SSQH9M_7.0.0 (accessed 7.15.18).

Object Management Group, 2011. Business Process Model and Notation (BPMN) 2.0.

Reichert, M., Dadam, P., 1998. ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control. J. Intelligent Information Systems 10, 93–129.

Reichert, M., Dadam, P., Bauer, T., 2003. Dealing with Forward and Backward Jumps in Workflow Management Systems. Software and Systems Modeling 2, 37–58.

Reichert, M., Weber, B., 2012. Enabling Flexibility in Process-Aware Information Systems. Springer.

Russell, N., Hofstede, A.H.M., 2006. Workflow Control-Flow Patterns. BPM Center Report BPM-06-22.

Weber, B., Reichert, M., Rinderle-Ma, S., 2008. Change Patterns and Change Support Features. Data and Knowledge Engineering 66, 438–466.

Weske, M., 2019. Business Process Management: Concepts, Languages, Architectures, 3rd ed. Springer.