Dynamic Change Propagation for Process Choreography Instances

Conrad Indiono and Stefanie Rinderle-Ma

University of Vienna, Faculty of Computer Science, Vienna, Austria
{firstname.lastname}@univie.ac.at

Abstract. Business process collaborations realize value chains between different partners and can be implemented by so called process choreographies. Change has become a major driver for costly (re-)negotiations between the participants. Static a priori prediction models exist to calculate the feasibility of a change request prior to negotiation. However, the dynamic or behavioral aspect of choreography changes at the choreography instance level has not been investigated yet, i.e., the question whether a process choreography instance is compliant with the change request and hence allows for acceptance of the change request. This work takes the dynamic perspective and analyzes the impact of a single change request from one partner on the entire (distributed) choreography based on the notion of change regions and public check points. Change strategies are elaborated to ensure choreography instance state compliance. One transaction-based approach is specified using rollback regions. It identifies probabilistically the set of activity nodes to be compensated at all levels of the business collaboration to ensure state compliance. The technical evaluation enables observing the properties of the rollback region.

Keywords: collaborative business processes, dynamic change

1 Introduction

Process choreographies implement business process collaborations between different partners in order to reach a joint business goal. Examples stem from the manufacturing or logistics domain. “dynamism is the basis for agility” [10] presents an omnipresent challenge to handling change in process choreographies. Though some work on the static perspective of process choreography change exists, e.g., [12,4], approaches to deal with the dynamic perspective of choreography change are almost entirely missing [17]. This paper tackles this research gap by considering the actual execution state of each participating partner’s process instance to estimate the impact of a change request on the global business choreography. The distributed nature of process choreographies poses particular challenges as it introduces privacy elements, disallowing full insight into direct or indirect partners’ execution environments. This means that on the static model level, partners do not know what exact activities are scheduled to be executed in between the interaction activities. Similarly on the dynamic level, the concrete current execution state as well as the historic execution log of each process instance is not fully known for partners.

This work extends already established work in the area of static change in process choreographies [5]. Here choreography change is described as a process consisting of several steps such as checking change correctness (static and dynamic) and negotiating the change request with the partners. The reason for the latter is that in a fully distributed setting changes cannot be imposed on partners, but have to be agreed on. Hence it can be beneficiary to estimate the costs of such a negotiation beforehand [8], particularly as negotiation might become a costly multi-step process [6]. Cost estimation is realized within the so called change prediction step that is executed before the negotiation takes place. The impact of the initial change request is estimated in terms of a normalized score value that allows comparison between the set of available change requests.

So far, merely the static costs of a change have been considered. The dynamic costs have only been considered in an abstracted manner, i.e., based on the choreography model level using execution probabilities [7]. In this work, the goal is to determine the *change region* of a change for different partners, i.e., the region in which a change might be critical with respect to the choreography instance state. Change regions have been proposed for business processes, but not for process choreographies. Based on the change region it can be determined whether or not a partner can apply the change right away. Further on, it can be considered whether or not the application of change compensation actions such as rollback might be meaningful in order to realize the change. In particular, the costs for such actions can be taken into consideration when estimating the change impact. This research goal is reflected in the following research questions:

- How to identify change regions for choreography changes, i.e., those regions where changing running choreography instances could lead to an inconsistent state?
- Once the change regions are identified, how to estimate the total impact of a change request?

The questions are approached following the design science method (cf. [20]). The relevance of the topic is underpinned by literature [17,8]. The created artifacts comprise definitions of fundamental concepts such as change regions as well as algorithms, i.e., algorithms that determine the rollback regions. The approach is evaluated based on a proof-of-concept implementation. Thereby the paper is structured as follows: In Sect. 2, basic terminology is introduced: Section 3 provides new concepts on choreography instance change. Section 4 provides change propagation strategies including compensation actions based on rollback. The evaluation is presented in Sect. 5, followed by related work in Sect. 6 and a conclusion in Sect. 7.

2 Motivating Example and Fundamentals

As an illustrative example, we study a pc case manufacturing use case consisting of these roles: pc case manufacturer, buyer, metal supplier and coloring supplier. The choreography model of this business collaboration is depicted in Fig. 1 and can be divided into the following areas: choosing a product type (F_1), checking

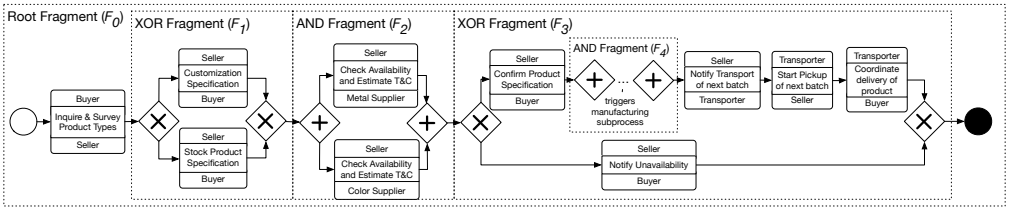


Fig. 1. Motivating Example: PC Case Manufacturing Use Case

for raw resource availability (F_2), the abstracted manufacturing subprocess (F_4) and finally the delivery process (F_3).

Consider a change introduced by the manufacturer in (F_1), which adds an option for laser etching custom designs. An associated change is also implemented in Fragment F_4, which performs the activity for the laser etching. How can we estimate the impact these changes have on the other partners? From previous work [7] we are able to estimate this impact using execution probabilities and an abstract adaptation cost. In this work, we improve the change region to determine the beginning of the change and define rollback regions to mark all possible maximal activities process instances are able to progress to. A concrete transaction based adaptation is used, which informs the cost calculation.

Definition 1. *[**Choreography**] Adapted from [5], we define a choreography $\mathcal{C}$ as a tuple ($\mathcal{G}$, $\mathcal{P}$, Π, $\mathcal{L}$), where*

- $\mathcal{G}$ is the choreography model (i.e., Fig. 1).
- $\mathcal{P}$ is the set of all participating partners.
- $\Pi = \{\pi_p\}_{p \in \mathcal{P}}$ is the set of all private models.
- $\mathcal{L} = \{l_p\}_{p \in \mathcal{P}}$ is the set of all public models.

The Refined Process Structure Tree (RPST) [19] is a structured approach for representing business process models. It divides the model into several fragments, each fulfilling the single entry, single exit property (SESE). A fragment is either a trivial one (a leaf in the tree), representing a single interaction inside a sequence, or a complex one that can be either an XOR or AND fragment and contain further sub fragments. Fig. 2 shows the pc case manufacturing use case as a collapsed RPST. It only shows the sequence under the root fragment (F_0), consisting of leaf nodes (trivial fragments, e.g. interactions) and sub trees representing either XOR or AND fragments (e.g., F_1, F_2, F_3). Note that the actual manufacturing process F_4 is not shown in the root sequence due to it being embedded under Fragment F_3.

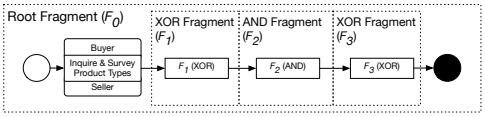


Fig. 2. Top-level RPST Tree of the PC Case Manufacturing Use Case

Definition 2. *[**Change Patterns**] from [5], we define the following change patterns:*

$$\begin{aligned} ChangePattern ::= & \; REPLACE(oldFragment, newFragment) \\ & \mid DELETE(fragment) \\ & \mid INSERT(fragment, how, pred, succ) \\ how ::= & \; Parallel \mid Choice \mid Sequence \end{aligned}$$

Example: To implement the laser etching scenario in Fig. 1, the $REPLACE(F_0, F_{new})$ change pattern would be used, where F_{new} is based on F_0, with the following changes: Fragment F_1 has a new interaction between seller and buyer for offering the laser etching customization. Fragment F_4 is modified, where laser etching specific interactions (e.g, with the corresponding supplier) are added.

Definition 3. *[**Business Process Instance**]*
A business process instance is a tuple (id, m, r, as), where the following holds: $m \in \Pi \wedge r \in \mathcal{P}$. as is a set of tuples each of the form (a, s), where $a \in m \wedge s \in \{inactive, activated, running, aborted, completed\}$ and id is a unique identifier that is mapped to the busines process instance.

We extend the definition of *Choreography* from Def. 1 to be: $\mathcal{C} = (\mathcal{G}, \mathcal{P}, \Pi, \mathcal{L}, \Pi^{\mathcal{I}})$, where $\Pi^{\mathcal{I}}$ is a set of business process instances (see Def. 3), which are running *instantiations* of a corresponding business process model and each activity comprising the process model having an associated state.

3 Dynamic Change Propagation Concepts

To determine the dynamic impacts of a change operation, two components are required: (1) identify the specific nodes that mark the beginning of the change: the *change region* and (2) identify the relative position of all business process instances ($\Pi^{\mathcal{I}}$) to that *change region*: *state compliance*. Having that relative position gives an initial indication whether there could be disproportional costs involved for the proposed change. Those process instances having their state before the *change region* are state compliant, and are thus generally non-problematic in regards to implementing the change. Those process instances whose running activities are already past the *change region* violate state compliance, and change implementation may potentially become problematic.

3.1 Change Region

As the basis for determining the *change region*, we can start with the *smallest fragment* (see Def. 4), which returns the surrounding RPST fragment containing all supplied nodes. Example: the *smallest fragment* that contains { *Inquire & Survey Product Types* } would be the interaction itself, because a leaf node inside a RPST fragment is a fragment. The *smallest fragment* containing { *Customization Specification*, *Notify Transport of next batch* } is the root fragment F_0.

Algorithm 1: Change Region Algorithm

```
Input:
    δ - a change pattern (see Def. 2)
Begin
if δ.type ∈ {INSERT, DELETE} then
    return δ.fragment
else
    ins ← δ.newFragment \ δ.oldFragment
    del ← δ.oldFragment \ δ.newFragment
    F_root ← α(ins + del); F_min ← ∅
    foreach node in ins + del do
        if F_min = ∅ ∨ distance(F_root, node) < distance(F_root, F_min) then
            F_min ← node
    return F_min
```

Definition 4. ***[Smallest Fragment]*** *(from [5])*

Let σ be a public model and S be a set of nodes corresponding to σ. Then: $\alpha_\sigma(S)$ returns the smallest fragment in model σ that contains all nodes from S. Formally: $\alpha_\sigma(S) = \underset{size(F)}{\arg\min}\{F \in \sigma \mid \forall n \in S, n \in F\}$

While the *smallest fragment* is sufficient to determine the beginning of a change in the cases where the change pattern $\in \{INSERT, DELETE\}$, it does not hold for $REPLACE$. In the case of $INSERT$, we know a new RPST fragment is being inserted between *pred* and *succ* (see Def. 2). Thus taking the fragment itself to mark the beginning of the change is feasible. The same concept holds for $DELETE$ change patterns. However, in the case where we have a $REPLACE$ change pattern as in the laser etching example, the *smallest fragment* would return the root fragment F_0 as the surrounding RPST fragment. The beginning of that root fragment does not mark the real change, which is critical for determining the necessity of adaptation before implementing the change. Generally, adaptation is required once the first node marking the change enters a state past *activated* (i.e., *running*, *aborted*, *completed*). Recall that in the laser etching example we have two RPST fragments being modified inside the $REPLACE$ operation: F_1 by adding an interaction *offer laser etching* and F_4 which adds new interactions with suppliers related to the new offer. To identify the concrete change region a simple *smallest fragment* call is not sufficient. We need to further identify the fragment nearest to the start node that has been changed and set that as the beginning of the change region. To do so, we first need the fragments that have been inserted (in both F_1 and F_4) and then find the one with the shortest path from the start of the surrounding fragment: this fragment (F_1) marks the beginning of the change region. Algorithm 1 specifies such a *change region*.

3.2 State Compliance

The second problem with determining dynamic change impact is related to the following: while each partner is able to calculate the relative position of their

process instances from the change region, partners are only able to estimate this location due to privacy issues. Using their own private execution log, partners are able to discern state compliance on their own [13]. For partners to estimate direct partners' state compliance, public interactions can be used as checkpoints. We know that once we have finished an interaction with another partner, the other partner has equally finished the same interaction in their own private process. Execution paths from that point on cannot not be accurately determined by direct partners as it is possible that the partner progresses in such a way that further interaction with the same partner never happens. A comprehensive monitoring approach at the cost of privacy would be required if indirect partner state tracking is required, as these would involve active state reports due to lack of direct interaction points. Estimating state compliance for direct partners can be achieved by taking a private execution log and abstracting by public activities or interactions where the direct partner is involved. The last activity can be seen as the *current state*, even though from the perspective of the private model, the actual progress might be more advanced. Only through passing public checkpoints can direct partners track state compliance.

Definition 5. ***[Choreography Instance]***
We extend the definition of *Choreography* from Def. 3 to include *choreography instances*: $\mathcal{C} = (\mathcal{G}, \mathcal{P}, \Pi, \mathcal{L}, \Pi^{\mathcal{I}}, \kappa, \kappa', \mathcal{G}^{\mathcal{I}})$, where
- $\kappa : corr_{id} \rightarrow \{(id, m, r, as)\}$ is a total function that maps a unique correlation identifier to a set of business process instances working on the same business case (see Def. 3), each assigned a unique private case id, with $corr_{id} \in \mathcal{G}^{\mathcal{I}}$.
- $\kappa' : (id, m, r, as) \rightarrow corr_{id}$, a non-injective surjective function that maps a unique business process instance to a correlation identifier.
- $\mathcal{G}^{\mathcal{I}}$ is the set of all active unique correlation identifiers, each representing a single choreography instance.

In our example, a business case is started by the buyer who intends to have a product manufactured. That buyer process starts with its own case id to track the product. The seller creates a unique case id once a product buying confirmation comes in and uses the same id when contacting suppliers. The two suppliers create their individual private case ids. Finally, the transporter is contacted by the seller, who may create a separate case id to handle the shipment. Even though many different case ids exists, the same product is being handled and referenced. The correlation id groups these unique case ids to the same business case. Thus a choreography instance groups together all partners and their private process instances working on the same business case. The known public markings are used to set the states of each public interaction activity. However, for privacy reasons, the partner who has direct connections with the most partners has the most accurate view. The choreography instance from the perspective of the buyer is more limited compared to the one of the buyer.

An *aggregated choreography instance* can be created (illustrated in Fig. 3) by taking all choreography instances (in $\mathcal{G}^{\mathcal{I}}$), applying κ on each to retrieve all the relevant business process instances and abstracting on public activities of the directly involved partners. The last known public activity can be registered as

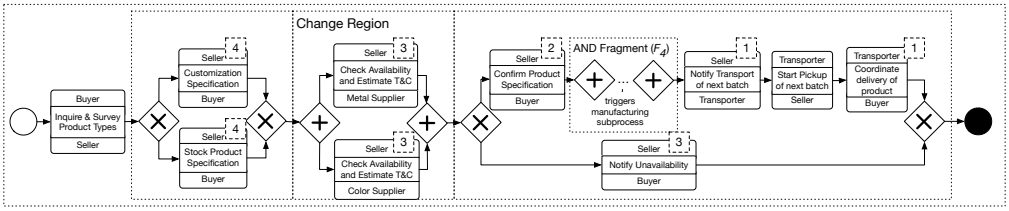


Fig. 3. Example of an Aggregated Choreography Instance

the *current active node*. Such nodes can be counted and the frequency marked on the choreography model ($\mathcal{G}$). Each count represents a single choreography instance. It is now possible to determine which choreography instances violate state compliance: in Fig. 3 these are the three instances inside the change region (WITHIN), and the seven instances after the change region (AFTER).

4 Dynamic Change Propagation Strategies

Having defined the necessary concepts, we will focus our attention on the mechanics of propagating changes to partners, focusing on the question of consistency from the dynamic point of view. At the core, every process instance needs to be stopped to avoid having process instances in the BEFORE state (compliant) entering the WITHIN or AFTER state, which makes these process instances non-compliant in terms of the change requested. This is the pessimistic approach because we assume that regardless of the future path actually taken, it might lead to structural conflicts due to the changes to be committed. Activities that are still before the *change region* are free to be executed according to the previous model. The *change region* represents a critical section, and partners still need to decide on a common understanding for this region (see Change Negotiation [6]). Thus any partners proceeding the execution past the *change region* have at that point already made a choice with which change alternative to proceed. Note that choosing the old version is an alternative as well. The outcome of the change negotiation needs to match this implicitly chosen change alternative for those process instances to remain structurally compliant. Anyone diverging from the common understanding means that any work resulting from that becomes unusable and has to be discarded. Partners are tied to each other with their decisions on which change alternative to pick.

There is a significant disadvantage with this approach: all choreography instances being affected by the change are stopped and no work can be executed that falls beyond the first node in the *change region*, which means there is an opportunity cost: no productive work can be performed due to the waiting time until a change alternative has been agreed on for the contested *change region*. This waiting time ensures the collaboration stays consistent. An alternative approach is the *optimistic change strategy*, which assumes that paying the opportunity cost (of waiting) is higher than just proceeding with execution and pay the cost only if actual state violations occur, meaning a different change alternative has been agreed on than the one partners have implicitly chosen to proceed execution with. In order to accomplish this, we need an approach to estimate this actual *repair* cost, which is the cost of transforming non-compliant

process instances to become compliant again. Weighing these two different costs together (pessimistic cost vs optimistic cost), allows a fair estimation of effort before change propagation is proposed to partners, and bargaining leverage once change negotiations start. In this work, the focus is placed on the technique for determining the *repair* cost, which occurs for making non-compliant process instances compliant.

4.1 Transaction-based Optimistic Change Strategy

In this work we will focus on an optimistic change strategy that is based on transaction support and calculates the repair cost based on performing rollback, starting with the assumptions on the transaction model.

Transaction Model Assumptions Several papers exist that explore the necessary properties a workflow transaction model should entail ([3]). Here we summarize these properties and assume they are supported, independent of the concrete transaction model being used. The work in this paper builds on these assumptions.

A workflow transaction is either (i) a single activity or (ii) a sequence of activities and takes as input a consistent state transforming it into another consistent state. Furthermore, workflow transactions are hierarchically nested with the presence of subprocesses and thus sub activities. Regarding atomicity, workflow transactions relax the *nothing* property of the all-or-nothing concept of traditional transactions [3]. Whereas traditional transactions expect an opaque transformation step from one consistent state to another, workflow transactions are made transparent, where each intermediate consistent state is *opened up*. This transparency allows a more fine-grained ability to perform rollbacks to past consistent states. A rollback path can be defined, which marks the backward sequence of past activities from the current activity up to the desired past consistent state. This paper takes these eligable states as rollback target in the context of inter-organizational business processes. We assume that the past consistent states, which have been traversed so far are represented and available in the form of private and public execution logs. The visibility of which is dependent on the partner accessing it. These execution logs are one of the required inputs for performing selective rollback. Note that for all private activities, the owner of the process instance is able to decide, without coordination with the associated partners, to which past state to rollback to. But once a public activity occurs inside the rollback path, the directly associated partner is bound to the same rollback operation. A transitive effect can be observed as this partner directing its partners to rollback to the relevant public activity, due to their interaction activities in the same rollback path. A decision to rollback then, by the nature of public interactions and the message dependency, affects other partners directly as well as indirectly. Another assumption is the ability to assign compensation tasks to activities, which are executed in the case an activity needs to be rolled back. If a compensation task cannot be semantically mapped, then the activity is marked as a *critical activity*. The effects of *critical activities* are further discussed in section 4.2.

Regarding consistency, we assume that each committed transaction after the execution of a single activity results in a consistent state. Accordingly, each rollback operation and thus the completion of a compensation task results in a consistent state as well.

Regarding isolation, rolling back activities should not affect concurrently running workflow transactions (assuming they are not part of the same choreography instance). In the case of sub activities we have a parent-child dependency between the parent activities and the called sub activities. Due to this parent-child relationship, whenever a decision is made to rollback the parent activity, all of the executed sub activities need to be rolled back as well. Inversely, a rollback on a sub activity does not necessarily require a rollback on the parent activity.

Regarding durability, we assume together with consistency that all committed transactions, as well as any compensation tasks executed, are made persistent.

4.2 Rollback Region

Based on the assumptions on the transaction model mentioned in the previous section, we introduce *rollback regions*. *Rollback region* is a technique that can be classified as a transaction-based optimistic change strategy. It is optimistic because it allows partners to proceed the execution even while the change request has not been committed yet. Whether or not the change request will be committed, the optimistic approach assumes that not every work result needs to be discarded. It is transaction-based due to the use of rollback, specifically compensation tasks, to bring non-compliant process instances back to compliance. *Rollback regions* determine the upper bound up to which a corresponding partner may proceed execution, and based on that calculates a cost estimation that would be required in case compliance transformation occurs. Stopping the execution of a process instance at an interaction gives us certain information about the corresponding partner we are interacting with. When we are waiting for a message then we know that the corresponding partner has not yet reached its corresponding send activity. Conversely, when there is an upcoming receive message activity with a partner and we haven't yet sent the required message for that partner to proceed, we are sure that at the worst case, that partner is waiting for us. These *checkpoints* form the basis for *rollback regions* to estimate the farthest activity direct partners are able to proceed. All possible paths that can be drawn between the current activity node up to the *checkpoints* represent the rollback paths to be traversed (i.e., by running the sequence of compensation activities). The cost of a rollback path is defined in Def. 6. The act of committing a change request is itself conducted within the context of a transaction: either all choreography instances are transformed into a consistent state, or the change commit fails and a rollback occurs which results in the consistent state before the change request was proposed. The *rollback region* captures the worst case cost in the event of a rollback. It is important to note that not all rollback paths need to be traversed, as only the actually traversed path represented by the private and public execution log needs to be compensated. In the following

sections we will discuss the influencing factors that may determine the farthest *checkpoint* that builds the terminal node of a *rollback region*.

Critical Activities The first influencing factor in determining the farthest activity is the presence of a *critical activity*. Recall that *critical activities* are those activities not associated with compensation tasks (c.f. Section 4.1), and thus cannot be compensated in the event of a failed transaction (e.g. rollback). Since the *rollback region* requires the presence of compensation tasks to work, *critical activities* constrain the possible execution paths for process instances in the context of a transaction-based optimistic change strategy. Concretely, from the perspective of a single process instance, if the next activity to be executed is a *critical activity*, then the optimistic change strategy becomes impossible due to not having the prerequisite compensation task to perform a rollback. This results in a hybrid change strategy where the change strategy can be optimistic up to the point of the first *critical activity*, and switching to a pessimistic change strategy from that point on. Of course, some process instances might emit a different execution log which does not entail such *critical activities*. In that case, the optimistic change strategy persists. A *critical activity* thus affects how far a process instance, under an optimistic change strategy regime, may proceed and in the same way limits the choices for the farthest activity.

Sync vs Async Message Passing in Interactions The W3C WS Choreograhy Model defines two distinctive types of interaction activities[1]: The (a) *one-way interaction*, for sending a single message and (b) the *request-response interaction*. In the latter case, the sender expects a response from the receiver of the initial message. In this work we define interactions to be either *async* or *sync*, corresponding to (a) and (b) respectively. The main difference between the two

[1] https://www.w3.org/TR/ws-chor-model/

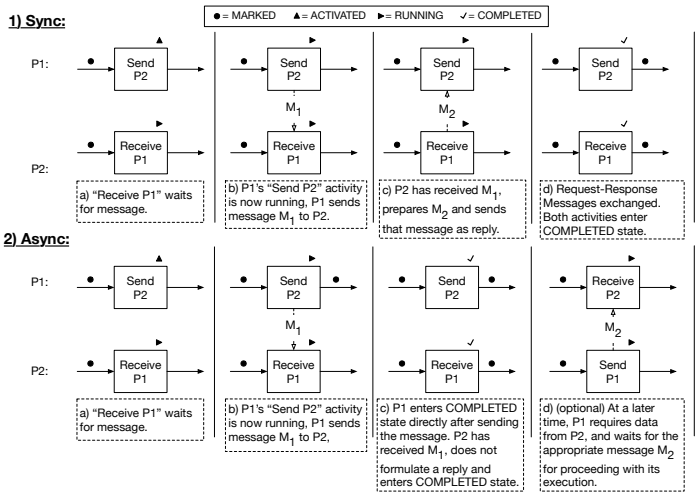


Fig. 4. Execution semantics of sync vs async interactions

interaction types is the following: in the case of the asynchronous message passing style, partners are not obliged to wait for a response after sending a message to their partner (cf. Figure 4). This sending semantic affects the boundary of the *rollback region* (i.e. the farthest activity). In a *sync* interaction, the two partners are in lock-step with each other due to the necessary message coordination in a request-response fashion. From the perspective of the sending partner p_1 in a *sync* interaction, the farthest activity for the corresponding partner p_2 is clear as long as the interaction has not completed: the current interaction node. The initial sender has two states to execute inside a *sync* interaction: the first sending of the initial message, and an implicit receive state for receiving the reply to the initial message. The latter prevents the initial sender to progress the execution of the process instance, and thus also prevent the *rollback region* from diverging with the corresponding partner.

In an *async* interaction it is not as transparent, due to the lack of a blocking receive activity that would wait for a reply. After an *async* interaction completes, the sender will come to a waiting state only if the sender has an explicit receive activity after the initial *async* send activity. Note that after a *sync* interaction completes (i.e., the receiving partner has received the reply and proceeds execution), the same non-transparency issue exists for *sync* interactions. *Rollback region* are then non-deterministic, as well as dynamic. It is non-deterministic due to not knowing which actual execution path will be taken and which blocking node reached. It is also dynamic, because the *rollback region* changes as soon as partners pass checkpoints. A *rollback region* becomes a cost estimate based on a temporary snapshot of the current state of the choreography instance, to estimate the effort and cost required to propagate a change request.

Definition 6. ***[Cost of a rollback path]***

Let RP be the set of private/public activities constituting a rollback path, which does not contain a critical activity. Assume further a function $cost_{comp}(a)$ that returns the cost of performing the compensation task associated with an activity a. Then the cost of a rollback path is the sum of the cost of compensation tasks to be executed: $cost_{rbp}(RP) = \sum_{a \in RP} cost_{comp}(a)$.

Definition 7. ***[Blocking Node]***

A node is a *blocking node* if it is either a *sync* interaction (which includes an implicit receive activity), an explicit receive activity, a criticial activity (an activity without a mapped compensation task), or an end node marking the end of the process instance.

$$BlockingNode ::= SyncInteractionActivity \mid ReceiveActivity \mid CriticalActivity \mid EndNode$$

Rollback Region Algorithm Depending on the perspective of the partner applying the *rollback region*, it can have different purposes. On the one hand, from the perspective of the change initiator, the *rollback region* is used to calculate the effort required for executing compensation tasks in order to make process instances compliant according to the old process model. After initiating

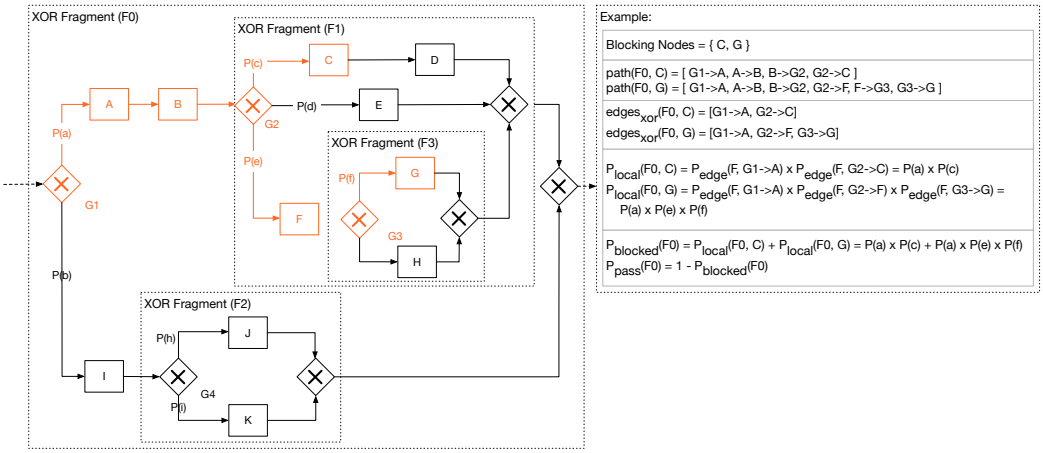


Fig. 5. Fragment-Local Blocking Node Probabilities

the change request, in an optimistic change strategy, the change initiator assumes the change request becomes accepted and proceeds execution, while the decision to commit to the change is not yet made. On the other hand, from the perspective of the change acceptor, the *rollback region* is used to calculate the effort required in order to make process instances compliant according to the new process model. This is due to the process execution maintaining the old version for running the current process instances. The effort then is to make the process instances compliant to the new process models. The *rollback region* is used as the core mechanism to determine the effort required to make process instances compliant, whether for the old process model or the new one. In both of the above cases, the *rollback region* algorithm requires the following inputs: (1) the *change region* marking the beginning of the change, (2) the private process model of the partner for which the dynamic impact is calculated, and (3) the private execution log of the same partner. Since the purpose of the *rollback region* is to calculate the cost of transforming non-compliant process instances to become compliant, we define the start node to be the first node within the *change region* (i.e., input (1) *change region*). The end node to be determined is the farthest activity as discussed earlier. In this section we will focus on the steps required to determine a *rollback region* from the perspective of a single partner. The full specification can be found in Algorithm 2.

Step 1: Determine Fragment-Local Blocking Node Probabilities The first step of the *rollback region* algorithm is to determine for each occuring *blocking node* (cf. Def. 7) its local probability of becoming activated. The *rollback region* is not deterministic, as the actual path to be executed from the current activity is still not yet known. In the simplest case we have a single blocking node on the level of the top-most RPST fragment (inside a sequence). In this case we can determine that in all cases, landing on this node causes the execution to stop. In the case of inside an RPST fragment (XOR gateways), we require branching probabilities (e.g. based on actual historical execution data) to determine the probability of the execution engine reaching that blocking node. Thus we add

another assumption: we have access to a table of branch probabilities for the private model of the partner whose impact we want to estimate. Blocking nodes inside deeply nested subfragments are determined by multiplying the branch probabilities of reaching that subfragment. Since we are only interested in the reachability of blocking nodes through branching probabilities, multiple blocking nodes inside the same subfragments are considered equal and only the first is considered. By summing up these local probabilities of reaching each blocking node, we can determine the probability of this RPST fragment becoming a terminal node: $P_{blocking}(F)$ (cf. Def. 9). The inverse probability of the execution passing through the same fragment without encountering a *blocking node* would then be $P_{pass}(F) = 1 - P_{blocking}(F)$.

Definition 8. ***[Reachability of Local Blocking Node]*** Given
- $path(F, node)$, a function that returns the shortest sequence of edges starting from the beginning of Fragment F leading to *node*.
- $edges_{xor}(F, node)$, a function that returns the edges preceeded by an XOR node from the path from the beginning of Fragment F leading to *node*, defined as $\alpha_{preceeded\_by\_xor}(path(F, node))$.
- $P_{edge}(edge)$, a function that returns the local (XOR) branching probability of that edge being traversed.

The *reachability of a local blocking node* is defined as

$$P_{local}(F, node) = \begin{cases} \prod_{x \in edges_{xor}(F,node)} P_{edge}(F, x) & \text{if is\_xor\_fragment(F)} \\ 1.0 & \text{otherwise} \end{cases}$$

Figure 5 shows an example which calculates the probabilities of reaching blocking nodes $\{C, D\}$ inside a XOR RPST Fragment $F0$.

Definition 9. ***[Probability of a blocking RPST Fragment]***

Given the local probability of reaching a *blocking node* inside a RPST fragment F: $P_{local}(F, node)$, we can define the probability of the whole RPST fragment blocking the execution:

$$P_{blocking}(F) = \sum_{\forall x \in nodes(F): is\_blocking\_node(x)} P_{local}(F, x)$$

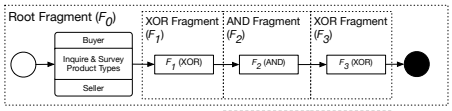
Step 2: Determine Terminal Node A terminal node is the farthest activity a process instance may reach. In the simplest case, the terminal node is the *end* node of the process model. This case happens when there is no more blocking nodes to suspend the execution for a process instance, except the last node (i.e., the *end* node). Another case is a blocking node in the main sequence of the top-most RPST fragment. This becomes a terminal node due to the certainty of this node becoming activated in future executions. It could be either due to partners having to synchronize messaging (receive activity) or reaching a critical activity. The last case is an RPST fragment one level below the top-most RPST fragment, which has $P_{blocking}(F) = 1.0$ (c.f., Def. 9), meaning an absolute certainty of activating a *blocking node* once entered. In contrast, any $P_{blocking}(F) < 1.0$ will have the possibility of executions avoiding *blocking nodes*

Algorithm 2: Local Rollback Region (LRR) Algorithm

```
   Input:
1        Q - change region
2        π_i - Private Process Model of partner i
3        L_i - Private Execution Log of partner i
4  Begin
5  Δ ← Q(π_i); S ← head(Δ); cur ← last(L_i)
6  if S ∉ L_i ∧ S.state ∉ {running, stopped, completed} then
7      return 0
8  else
9      // Step 1: Determine terminal node
10     foreach f in nodespath_rpst(cur, π_i) do
11         if P_blocking(f) = 1.0 then
12             node_terminal ← f
13             break;
14     // Step 2: Adjust local probabilities
15     residual ← 1.0; probs ← ∅; subfrags ← ∅
16     foreach f in nodespath_rpst(cur, π_i) until f = node_terminal do
17         foreach n in nodes(f) do
18             if is_blocking_node(n) ∧ n ∉ subfrags then
19                 probs ← probs + {(f, P_local(f, n) * residual)}
20                 subfrags ← subfrags + {f}
21         residual ← (1 − P_blocking(f)) * residual
22     assert( ∑_{(bn,p)∈probs} p = 1.0 )
23     // Step 3: Calculate expected cost of rollback region
24     cost_expected ← 0.0
25     foreach (bn, p) in probs do
26         cost_expected ← cost_expected + cost_rbp(nodespath(cur, bn)) * p
27     return cost_rbp(nodespath(S, cur)) + cost_expected
```

inside this RPST fragment. To generically determine the terminal node we take the top-most RPST fragment and in that sequence take each node's probability of it becoming a *blocking node*. The first $P_{blocking}(F) = 1.0$ is marked as the terminal node. An example can be seen in Figure 6, where the end node is marked as the *terminal node*, due to F_2 being the *change region* and F_3 having a $P_{blocking}(F_3) = 0.6$.

Step 3: Determine Absolute Probabilities of Rollback Paths inside the Rollback Region Having determined the *terminal node*, we can now calculate the absolute probability of reaching each *rollback path* inside the *rollback region*. The only *blocking nodes* we consider are those starting from the current activity up to the terminal node (c.f., example in Figure 6). The *rollback paths* are built by pairing each candidate *blocking node* as the end activity with the current active node as the starting activity. For the adjustment of the probabilities we again take the sequence of the top-most RPST fragment. For each node, we accumulate the residual, which is the probability of not being blocked by the current node (starting with 1.0) and adjust the local probabilities by multiplying it with the



			Change Region	Current Active	
# of Blocking Nodes:	1	2	2	5	1
$P_{blocked}$:	1.0	1.0	1.0	0.6	1.0
$P_{adjusted}$:	-	-	1.0	0.6	0.4
$Cost_{rbp}$:	-	-	0.5	4.0	0.0
$Cost_{expected}$:	-	-	1.0 x 0.5 = 0.5	0.6 x 4.0 = 2.4	0.0
Total $Cost_{expected}$:	0.5 + 2.4 + 0.0 = 2.9				

Fig. 6. Example of a *rollback region* calculation.

residual. This step stops at the terminal node. The invariant to be upheld is the sum of all absolute probabilities = 1.0.

Step 4: Calculate Expected Cost of a Rollback Region Since *rollback regions* are probabilistic and having calculated the absolute probabilities of each *rollback path*, we can calculate a final value that represents the expected cost of that *rollback region*. Using Def. 6 we can determine the cost of a single *rollback path*. This *rollback path* cost is multiplied by the absolute probability of that *rollback path* actually occuring (from the previous step). The sum of all these individual *rollback path* costs represents the expected cost of the *rollback region*. A fixed cost part exists, which is the *rollback path* cost that is accrued due to the already traversed path starting from the change region up to the current active node.

Using Rollback Region for Change Impact Analysis Recall that the goal of this work is to have the ability to evaluate several change alternatives in terms of change impact on the whole choreography. We have tackled that challenge through the lens of the dynamic state of the individual process instances that together realize the collaboration, in relation to the position of the proposed change alternatives individually. By using *rollback regions*, it is possible to answer these questions. While the specified *rollback region* algorithm LRR works from the perspective of a single partner, and in that context for a single process instance, we can now aggregate the individual expected costs of each *rollback region* for a single choreography instance:

$$ARR(corr_{id}, q) = \sum\nolimits_{i \in \kappa(corr_{id})} LRR(q, i.m, i.as)$$

The expected cost of a change alternative over the complete collaboration would then become:

$$CRR(q) = \sum\nolimits_{corr_{id} \in \mathcal{G}^{\mathcal{I}}} ARR(corr_{id}, q)$$

With $CRR(q)$, it is now possible to compare change alternatives in terms of their impacts by estimating compensation task costs, where the individual

rollback paths dynamically evolve as execution progresses. Note however, that the LRR expects the private model π_i of the partner for whom to calculate the impact for. The change initiator does not have access to the private models of the other partners in the collaboration. This means the aggregation conducted in $ARR(k, q)$ is a fan-out process, where the change initiator asks each partner for the results of applying the local *rollback region* algorithm (LRR) and in the end aggregates the returned individual expected costs. One approach exists to avoid the communication overhead. The change initiator could estimate the change impact of a change alternative by substituting the private model of the intended partner with the corresponding accessible public model. The required execution log can be either (i) derived through abstraction on the public nodes of the intended partner or (ii) directly requested. The result of appliying the LRR on this adjusted input cannot be accurate, as no private activities are accessible and thus the complete compensation cost of those *rollback paths* unknown. What is known are the distances of these *rollback paths* through the number of *checkpoints* past the *change region*. The expected cost of change alternatives based on these inputs are only approximations. One way to increase the accuracy would be by all partners making public several metrics: branching probabilities, average number of private nodes inside RPST Fragments and average compensation cost of private activities.

5 Evaluation

As a technical evaluation we have implemented the concepts introduced in this work, mainly *rollback regions* and the dependent concepts, as a proof-of-concept. Furthermore we evaluate the output of the *rollback region* variations and ensure the critical invariants are upheld[2]. The evaluation setup follows this methodology: (1) Load a pre-defined choreography specified with BPMN 2.0 XML Format. (2) Specify a change region. (3) Randomly scale private activities for each role in the choreography with the following set as the number of private activities per fragment: $\{2, 5, 10, 30, 50\}$. (4) Randomly generate business process instances (by assigning activity states) and creating the associated choreography instances, grouping these business process instances together. (5) Calculate the expected cost using the different *rollback region* variations: (a) default LRR, (b) public *rollback region*, (c) public *rollback region* with added information (public branching probabilities, number of private activities inside each fragment, average compensation task cost of private activities inside each fragment). (6) Determine the error rate between (a) and (b) as well as (a) and (c), defined as the difference between the final costs of each respective algorithms.

The evaluation shows that the error rate is positively correlated with the number of private activities: as the number of private activities inside a fragment is scaled up, so does the error rate. Error rate reduction through the *rollback region* variant (c) can be observed. Thus it is possible to choose between variants of the *rollback region* algorithm depending on the readiness of communication

[2] The implementation can be retrieved under https://github.com/indygemma/rollback-regions

overhead for determining dynamic change impact and sensitivity to the error rate.

6 Related Work

The survey presented in [17] sets out the related areas for this work. One dimension is static versus dynamic change and the other dimension process orchestrations versus process choreographies where this work sits at the intersection of dynamic change and process choreography. A plethora of approaches exists for static and dynamic change in process orchestrations [12]. Propagation strategies for process evolution have been at first defined in [2] including abort, flush, and migrate. For an efficient decision on migrating process instances change regions have been proposed in [1]. Depending on the instance state relatively to the change region the possibility to migrate this instance can be quickly made. The most interesting case are instances that are within the change regions - this holds also true for this work. Static change in process choreographies has been investigated by different approaches. The survey in [17] only names DYCHOR [14] and C^{3}Pro [5] to deal with change propagation, i.e., other approaches focus on structural correctness of the choreography and consistency between public and private processes. So far only [18] has provided a first approach addressing dynamic change in process choreographies according to [17]. [18] addresses the problem of migrating instances after a choreography change by proposing strategies for handling the migration in an ordered way, i.e., by avoiding concurrent changes and by a protocol for the instances to accept or decline the migration. As opposed to [18], this work focuses on the instance states and the costs of the migration. Both approaches seem to be complementary.

Several transactional models for processes have been proposed (for an overview see [15]). Particular focus was put on how to deal with long-running transactions (i.e., instances) such as SAGAS [9] and Spheres [11]. Rolling back instances in order to reach a compliant state again was proposed for process orchestrations by [16]. This approach exploits selected ideas from transactional process management and transfers them to the context of dynamic choreography change.

7 Summary

In this work we have defined *rollback regions*, an algorithm to determine the expected cost of a change request in the context of process choreography instances. The algorithm is based on a transactional model that supports compensation tasks, the messaging semantics of interaction activities (sync vs async), as well as the actual states each single business process instance are situated in relation to the *change region*. There are several variations to the algorithm, and the evaluation shows that it is possible to choose the most appropriate one depending on sensitivity to communication overhead as well as error rate. In future work we would like to tackle the data perspective of applying change propagation requests, both in how it affects dynamic change impact analysis and state compliance, as well in the context of the change propagation algorithms themselves. *Rollback regions* can be further extended to support loops as well as studying alternative adaptations (e.g., versioning) to ensure state compliance.

Acknowledgment This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-072.

References

1. van der Aalst, W.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers 3(3), 297–317 (2001)
2. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data Knowl. Eng. 24(3), 211–238 (1998)
3. Eder, J., Liebhart, W.: Workflow transactions. In: Workflow Handbook 1997, pp. 195–202 (January 1997), handbook of the Workflow Management Coalition (WfMC)
4. Eshuis, R., Norta, A., Roulaux, R.: Evolving process views. Information and Software Technology 80, 20 – 35 (2016)
5. Fdhila, W., Indiono, C., Rinderle-Ma, S., Reichert, M.: Dealing with change in process choreographies: Design and implementation of propagation algorithms. Inf. Syst. 49, 1–24 (2015)
6. Fdhila, W., Indiono, C., Rinderle-Ma, S., Vetschera, R.: Finding collective decisions: Change negotiation in collaborative business processes. In: On the Move to Meaningful Internet Systems. pp. 90–108 (2015)
7. Fdhila, W., Rinderle-Ma, S.: Predicting change propagation impacts in collaborative business processes. In: Symposium on Applied Comp. pp. 1378–1385 (2014)
8. Fdhila, W., Rinderle-Ma, S., Indiono, C.: Change propagation analysis and prediction in process choreographies. Int. J. Cooperative Inf. Syst. 24(3) (2015)
9. Garcia-Molina, H., Salem, K.: Sagas. In: Special Interest Group on Management of Data. pp. 249–259 (1987)
10. Grefen, P., Rinderle-Ma, S., Dustdar, S., Fdhila, W., Mendling, J., Schulte, S.: Charting process-based collaboration support in agile business networks. IEEE Internet Computing (2017)
11. Guabtni, A., Charoy, F., Godart, C.: Spheres of isolation: Adaptation of isolation levels to transactional workflow. In: Business Process Management. pp. 458–463 (2005)
12. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer, Heidelberg New York Dordrecht London (2012)
13. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems: A survey. Data Knowl. Eng. 50(1), 9–34 (Jul 2004)
14. Rinderle, S., Wombacher, A., Reichert, M.: Evolution of process choreographies in DYCHOR. In: On the Move to Meaningful Internet Systems. pp. 273–290 (2006)
15. Rinderle-Ma, S., Grefen, P.W.P.J.: Towards flexibility in transactional service compositions. In: International Conference on Web Services. pp. 479–486 (2014)
16. Sadiq, S.W.: Handling dynamic schema change in process models. In: Australasian Database Conference. pp. 120–126 (2000)
17. Song, W., Jacobsen, H.A.: Static and dynamic process change. IEEE Transactions on Services Computing (2015)
18. Song, W., Zhang, G., Zou, Y., Yang, Q., Ma, X.: Towards dynamic evolution of service choreographies. In: Asia-Pacific Services Computing Conference. pp. 225–232 (2012)
19. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In: Business Process Management. pp. 100–115 (2008)
20. Wieringa, R.: Design Science Methodology for Information Systems and Software Engineering. Springer (2014)