Information Systems 49 (2015) 1–24

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Dealing with change in process choreographies: Design and implementation of propagation algorithms ☆

Walid Fdhila [a,*], Conrad Indiono [a], Stefanie Rinderle-Ma [a], Manfred Reichert [b]

[a] Faculty of Computer Science, University of Vienna, Austria
[b] Institute of Databases and Information Systems, University of Ulm, Germany

ARTICLE INFO

Article history:
Received 25 February 2014
Received in revised form
23 October 2014
Accepted 27 October 2014
Available online 5 November 2014

Keywords:
Process-aware information system
Process choreography
Change propagation
Process change
Business collaboration

ABSTRACT

Enabling process changes constitutes a major challenge for any process-aware information system. This not only holds for processes running within a single enterprise, but also for collaborative scenarios involving distributed and autonomous partners. In particular, if one partner adapts its private process, the change might affect the processes of the other partners as well. Accordingly, it might have to be propagated to concerned partners in a transitive way. A fundamental challenge in this context is to find ways of propagating the changes in a decentralized manner. Existing approaches are limited with respect to the change operations considered as well as their dependency on a particular process specification language. This paper presents a generic change propagation approach that is based on the Refined Process Structure Tree, i.e., the approach is independent of a specific process specification language. Further, it considers a comprehensive set of change patterns. For all these change patterns, it is shown that the provided change propagation algorithms preserve consistency and compatibility of the process choreography. Finally, a proof-of-concept prototype of a change propagation framework for process choreographies is presented. Overall, comprehensive change support in process choreographies will foster the implementation and operational support of agile collaborative process scenarios.

1. Introduction

The optimal design and implementation of their business processes is crucial for enterprises. This not only applies to internal business processes, but also to collaborative processes whose execution involves different partner enterprises. Examples include cross-organizational manufacturing [1] and tourism [2]. The system-based support of such collaborative processes is realized by *process choreographies* [3]. In particular, a *choreography model* describes the interactions between the partner processes through message exchanges. In a supply chain process, for example, the `Supplier` interacts with the `Manufacturer` and the `Manufacturer` with the `Customer`. The `Customer`, for example, may place an order with the `Manufacturer` by sending a corresponding order message.

In general, the interactions among the partners are visible to the outside and described by so called *public process models* (*public model* for short). In turn, the public models constitute views on the underlying internal partner processes, the so-called *private processes*. In particular, the models of the latter (i.e., private models) are not visible to the other partners due to confidentiality reasons. Altogether, a choreography model

☆ The presented work has been conducted within the C³Pro project I743 funded by the Austrian Science Fund (FWF) and the Deutsche Forschungsgemeinschaft (DFG).
* Corresponding author.
E-mail addresses: walid.fdhila@univie.ac.at (W. Fdhila), conrad.indiono@univie.ac.at (C. Indiono), stefanie.rinderle-ma@univie.ac.at (S. Rinderle-Ma), manfred.reichert@uni-ulm.de (M. Reichert).

http://dx.doi.org/10.1016/j.is.2014.10.004
0306-4379/

consists of the participating partners, a *global* view on all partner interactions, and the public as well as private models of the partners.

1.1. Research challenges

Process change has been identified as crucial in most application domains [4–6,58]. The demand for changing business processes arises due to various reasons such as the advent of new regulations or the emergence of new competitors at the market. Research on this topic has been extensive and led to flexible process management technology realized as mature commercial (e.g., AristaFlow[1]) and prototypical systems (e.g., CPEE[2]). So far, however, approaches dealing with process changes have focused on scenarios in which a business process is entirely run within a single enterprise. In turn, little attention has been paid to changes of process choreographies, even though the latter demand for agility and flexibility as well [7–9].

When applying changes to the processes supported by an information systems, in general, it must be ensured that neither structural nor behavioral soundness of the process is violated after the change [6]. For process choreographies, additional properties must be guaranteed due to the complexity introduced by the involvement of autonomous partners as well as the interactions between them. For example, assume that a particular partner applies a change to its private process. In addition to ensuring structural and behavioral soundness of this private process, it must be determined whether its change affects other partners in the choreography as well. Amongst others, this means that it must be checked whether the change of the private model affects the corresponding public model. In this case, it must be further ensured that the private model remains *consistent* with the public model. Note that this might require adaptations of the public model as well.

In turn, changing the public model of a particular partner might affect its interactions with other public models, e.g., when deleting an activity that sends a particular message another partner is waiting for. In order to ensure *compatibility* between the public models of the involved partners, therefore, one may have to *propagate* the changes from one partner and its public model to the other partners and their public models. After adapting the public models of the partners, in turn, the consistency with the underlying private models must be re-checked to ensure overall consistency. Note that change propagation cannot always be restricted to direct partners, but might spread *transitively* over the entire process choreography.

In general, propagating changes must not infringe the autonomy of the partners. In fact, adaptations becoming necessary to maintain the consistency and compatibility of the choreography should be suggested to partners, but the decision whether or not to adopt these adaptations must be left to them and may be subject to negotiations. In general, such negotiations can be costly and time-consuming, particularly in case of failure. This paper focuses on the fundamentals of change propagations in process choreographies whereas negotiation issues are discussed in [60,59]. Another challenge concerns the non-availability of information about the private processes of the partners. Hence, determining the adaptations required for the public and private models of the partners during change propagation is a difficult task.

Altogether, an approach enabling change propagation in process choreographies must tackle the following research challenges:

1. It must provide change propagation algorithms that ensure consistency and compatibility for all affected partners.
2. It must handle transitive change propagation across multiple partners.

In order to obtain an operational change propagation framework, we must further deal with implementation concepts required for realizing the change propagation algorithms for process choreographies.

1.2. Contribution

This paper provides an extended and revised version of the work we presented in [8]. First of all, [8] introduced fundamental notions as well as design decisions such as representing choreography processes as Refined Process Structure Trees (RPST) [12] and restricting the set of change operations to the insertion, replacement and deletion of process fragments (as described in [13]). This paper adopts these design decisions. Further on, [8] addressed Research Challenges 1 and 2 by providing propagation algorithms for change operations REPLACE and UPDATE, whereas other change operations were not considered. Finally, [8] discussed how the propagation algorithms ensure consistency and compatibility of the choreography model and highlighted the problem of transitive change propagation.

Compared to [8], this paper provides significant revisions and extensions of the results related to Research Challenges 1 and 2. This includes (i) a fundamental revision of the previous definitions using the mapping functions between the different choreography models and – in the sequel – the propagation algorithms; (ii) the propagation algorithms for additional change operations (i.e., Insert and Delete), (iii) extensive illustrations of the algorithms, (iv) a revision of the Replace algorithm, (v) an extended discussion on transitivity when propagating changes in process choreographies, and (vi) an extension of the formal evaluation of consistency and compatibility in the context of respective change. Furthermore, this paper provides novel results regarding the technical evaluation of our approach. We propose an architecture for implementing a sophisticated change propagation framework for process choreographies. This architecture consists of three layers for defining, executing and changing processes. The core component of the change layer, which is realized as a proof-of-concept prototype, is the C[3]Editor. The latter allows for the import of private, public and choreography models from tools such as Signavio and jBPM. The C[3]Pro Editor visualizes the different models and enables the definition and application of changes to the private models. Furthermore, it determines and visualizes the partners

[1] www.aristaflow.com
[2] cpee.org

affected by a change and the updates required for change propagation. To the best of our knowledge, this is the first prototype enabling change and change propagation in process choreographies. This paper is organized as follows: Section 2 introduces a motivating example, followed by fundamental definitions in Section 3. Section 4 then presents the change propagation algorithms we developed. Section 5 discusses the handling of transitivity when propagating changes in process choreographies. Our approach is evaluated in Section 6 regarding the consistency and compatibility of the choreography after change propagation. Section 7 provides the details on the architecture and proof-of-concept implementation. In Section 8, we discuss related work. Section 9 summarizes the paper.

2. Running example and model representation

From the perspective of a single partner, three different, but overlapping viewpoints form a collaboration: the private model, public model, and choreography model [16].

- The *private model* describes the internal business logic as well as the message exchanges this partner is engaged in; i.e., the private model corresponds to the executable process of this partner. In general, the internal logic is not visible to other partners.
- The *public model* sketches the message exchanges from the perspective of this single partner as well as their sequencing; i.e., it represents an abstraction of the private activities corresponding to the private model. Compared to the public model, the private model contains the business process logic not visible to the other partners.
- The *choreography model* provides a global view on the interactions of a collaboration; i.e., it captures all interactions among the partners as well as the dependencies between these interactions.

2.1. Running example

We illustrate change propagation issues along the *booking trip* choreography example depicted in Fig. 1. This example is part of the choreography model described in [2]. It has been modeled using the choreography diagram elements of BPMN 2.0 and the Signavio tool [14]. The example describes a collaboration among four partners, i.e., *traveler*, *travel agency*, *acquirer*, and *airline*. The *traveler* sends booking information to the *travel agency* that, in turn, contacts the *acquirer* to request a credit check. If the *traveler* does not have enough credit, failure notifications are sent to the *travel agency* and *airline*, which inform the *traveler* about the reservation failure and purchase cancellation, respectively. Otherwise, an approval is sent to the *travel agency* and the *airline* is triggered to send the ticket and the purchase confirmation.

Fig. 2 depicts a BPMN collaboration diagram listing the public models of all partners involved in the choreo-

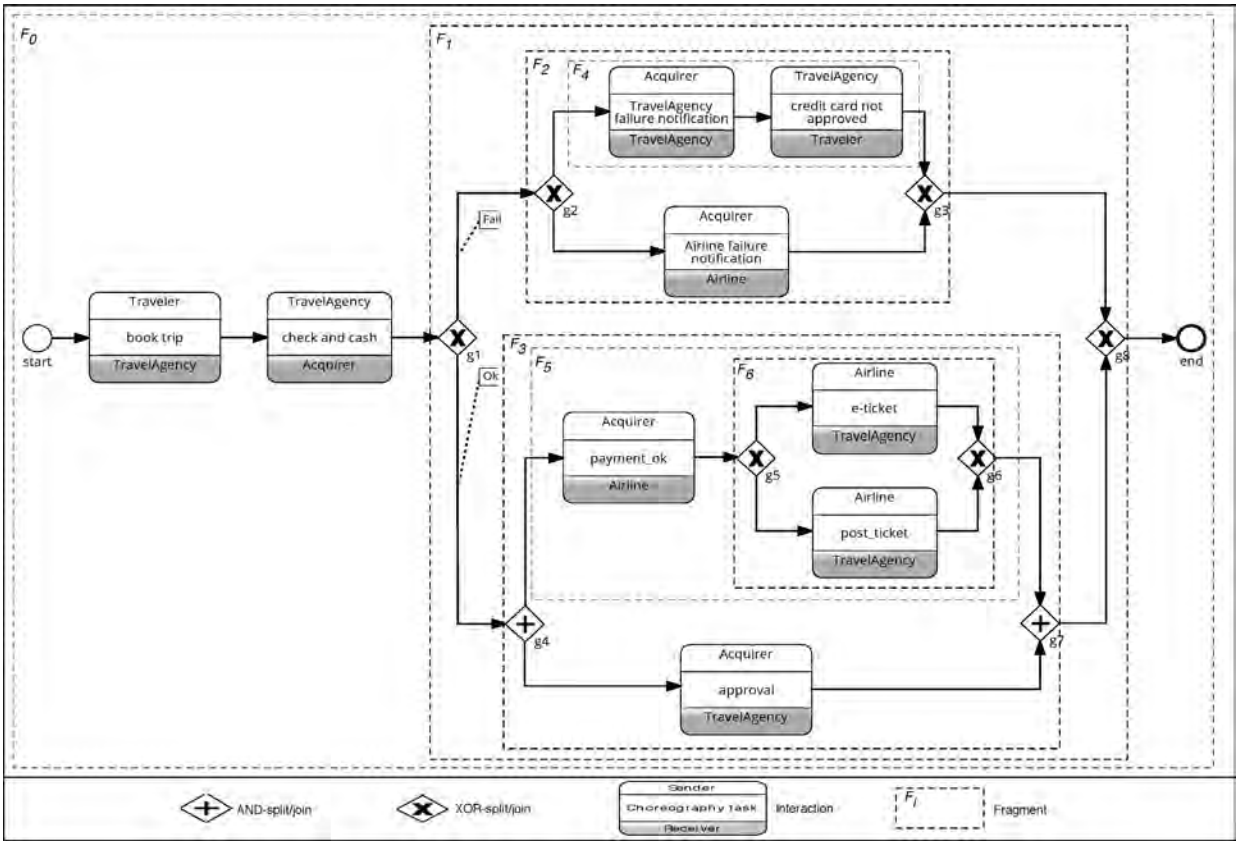

Fig. 1. Choreography model: simplified book trip process [2].

graphy. Each public model includes the interactions the corresponding partner is involved in as well as the control flow between them. Note that Fig. 2 does not show the private models of the partners, which contain their internal activities (cf. Fig. 3). Finally, merged together, the public models lead to the choreography model.

As motivated, in many application scenarios, the partners of a collaboration should be allowed to change their private processes. The specific challenge compared to local changes of a single process is to *propagate* change effects from one partner to the others [8,9] if required. For example, the `TravelAgency` might want to send a questionnaire about customer satisfaction to the `Traveler` after booking the ticket. This could be accomplished by inserting corresponding activities into the private model of the `TravelAgency`; e.g., `DevelopQuestionnaire`, which constitutes an internal activity not visible to other partners, and `SendQuestionnaire`, which constitutes the public activity to be added as well. Furthermore, a respective *change request* needs to be sent to the `Traveler` who should be able to receive the corresponding message and respond to it.

In general, change propagation in process choreographies might become quite complex [8]. Consider the above example and assume that it is not the `TravelAgency` which initiates the collection of customer feedback, but the `Airline` through the `Acquirer` and `TravelAgency`. In this case, the initial change will cause *transitive* effects across multiple partners. To overcome this problem, the effects of this local change in the private model of one partner need to be propagated to the concerned partners. As a consequence, the interactions must be restructured accordingly.

2.2. Model representation

As a prerequisite for precisely defining the notions of private, public and choreography model, we need to be able to represent the control-flow relations between activities and interactions. With the *Refined Process Structure Tree* (*RPST*) [12], this paper adopts a structured representation for this. An RPST model corresponds to a decomposition of a process model into a set of single-entry, single-exit (SESE) fragments. Thereby, each node of an RPST represents a SESE fragment of the underlying process model. Consequently, the root node corresponds to the entire process model, whereas the child nodes of a node N correspond to the SESE fragments directly contained under N; i.e., the RPST parent–child relation corresponds to the containment relation between SESE fragments. As a key characteristic, the RPST can be constructed for any process model captured in a graph-oriented notation [17].

We choose the RPST for various reasons. Besides being generic and language-independent, the RPST is indeed a structured tree representation of a given model. Note that *structured process models* are close to BPEL and are simpler to analyze and easier to comprehend than unstructured models. However, recent work has shown that most unstructured process models can be automatically translated into structured ones [18]. Additionally, computing and propagating changes for unstructured processes is rather complex and might violate the soundness of the choreography. Transforming unstructured processes into structured ones, therefore, eases the propagation of changes and ensures a more sound propagation. Furthermore, using tree structures instead of usual graph

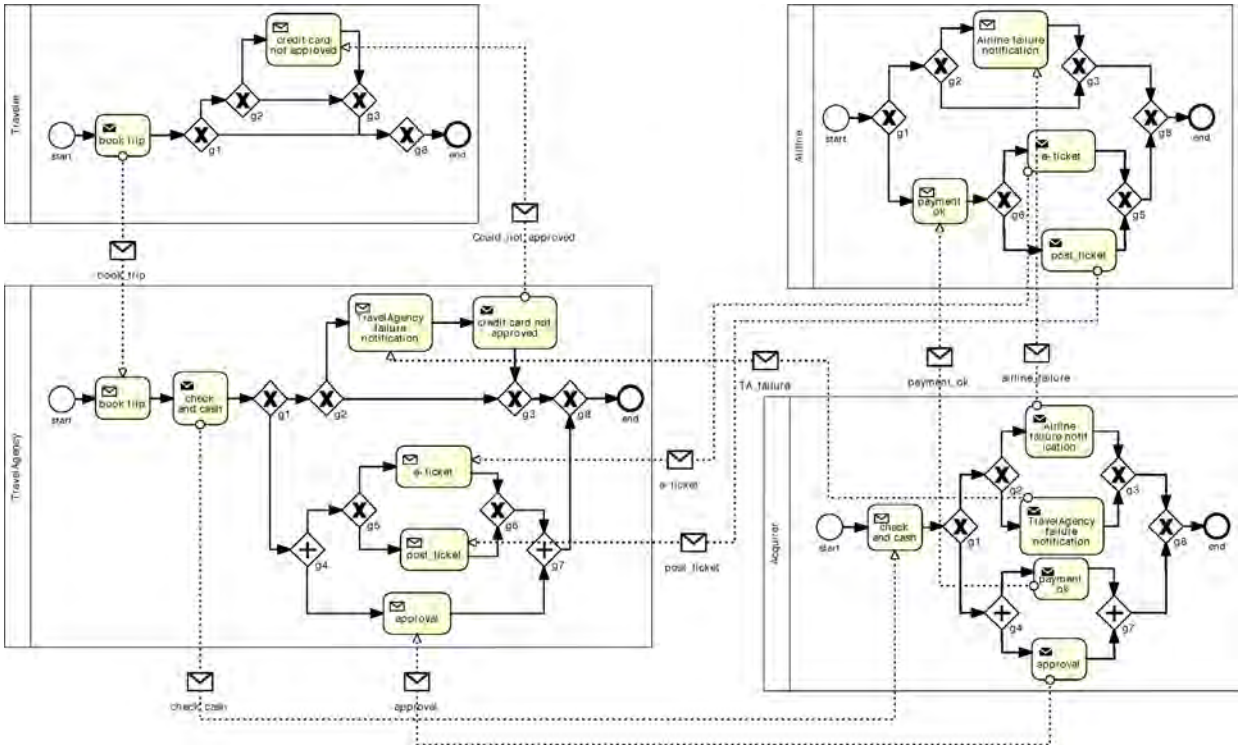

Fig. 2. Book trip process: public models.

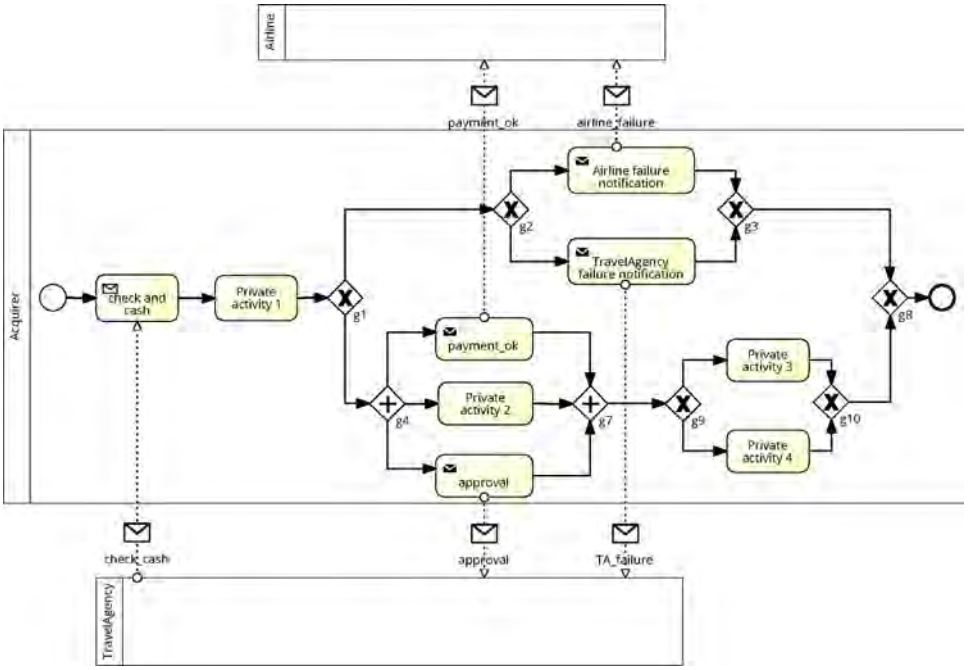

Fig. 3. Book trip process: acquirer private model.

representations significantly reduces the complexity for calculating the impacts of a change (e.g. parsing, identifying fragments). Indeed, high-level change operations (cf. Section 3) refer to entire process fragments (i.e., sets of activities and gateways) instead of single nodes. As process models are block structured in RPSTs, in turn, this makes it easier to identify the fragments to be modified in the processes of the partners involved in a change. Finally, in [12] it was proven that the translation of the process models to block-structured languages (e.g. BPEL) becomes easier through their decomposition into RPST. The transformation of graph models to RPST is linear, idempotent and modular [12].

Fig. 4 depicts the tree model of the choreography scenario from Fig. 1. In essence, the interaction nodes of the original graph are mapped to leaves in the tree model and represent the *Trivial nodes*, whereas the control nodes (i.e., sequence (SEQ), choice (CHC), parallel (PAR), or loop (RPT)) are mapped to internal nodes (for more details see [12]).

3. Fundamental definitions

This section introduces the main definitions used throughout the paper. Section 3.1 provides the formal definitions related to the various models a choreography is composed of. In turn, Section 3.2 presents basic definitions related to change, change propagation, and change operations.

3.1. Process choreography

A choreography includes three types of models: (i) the private model representing the executable process and including private activities as well as interactions with other partners, (ii) the public model (also called the interface of the process) highlighting solely the interactions of a given partner, and (iii) the choreography model giving a global view on the interactions between all partners. In the following we sketch the corresponding definitions.

Definition 1 (*(Structured) Private Model*). A private model π_p of partner p corresponds to a tree with the following structure[3]:

$$\begin{aligned} \mathit{Process} &::= \mathit{PNode} \\ \mathit{PNode} &::= \mathit{Activity} | \mathit{ControlNode} | \mathit{Event} \\ \mathit{Activity} &::= \mathit{PrivateActivity} | \mathit{InteractionActivity} \\ \mathit{InteractionActivity} &::= \mathit{Send}(\mathit{Message}, \mathit{Receiver}) | \\ &\quad \mathit{Receive}(\mathit{Message}, \mathit{Sender}) \\ \mathit{ControlNode} &::= \mathit{SEQ}(\{\mathit{PNode}\}) | \mathit{CHC}(\{\mathit{PNode}\}) | \\ &\quad \mathit{PAR}(\{\mathit{PNode}\}) | \mathit{RPT}(\mathit{PNode}) \\ \mathit{Event} &::= \mathit{Start} | \mathit{End} \end{aligned}$$

SEQ corresponds to a sequence of fragments, CHC to a choice between two or more fragments, PAR to a parallel execution of several fragments, and RPT to an iteration over a fragment.

Example 1. In the private process model depicted in Fig. 3, fragment $\mathcal{F}$ is represented as follows:

$$\mathit{SEQ}(\mathit{PAR}(\mathit{Send}(\mathit{payment\_ok}, \mathit{airline}), \mathit{pr\_activ2}, \mathit{Send}(\mathit{approval}, \mathit{travelAge\_ncy})), \mathit{XOR}(\mathit{pr\_activ3}, \mathit{pr\_activ4}))$$

Definition 2 (*(Structured) Public Model*). The public model $\wp_p$ of a partner p reflects the external behavior of p; i.e., it includes the interactions with other partners as well as the

[3] We use the type definition syntax of the ML language [19].

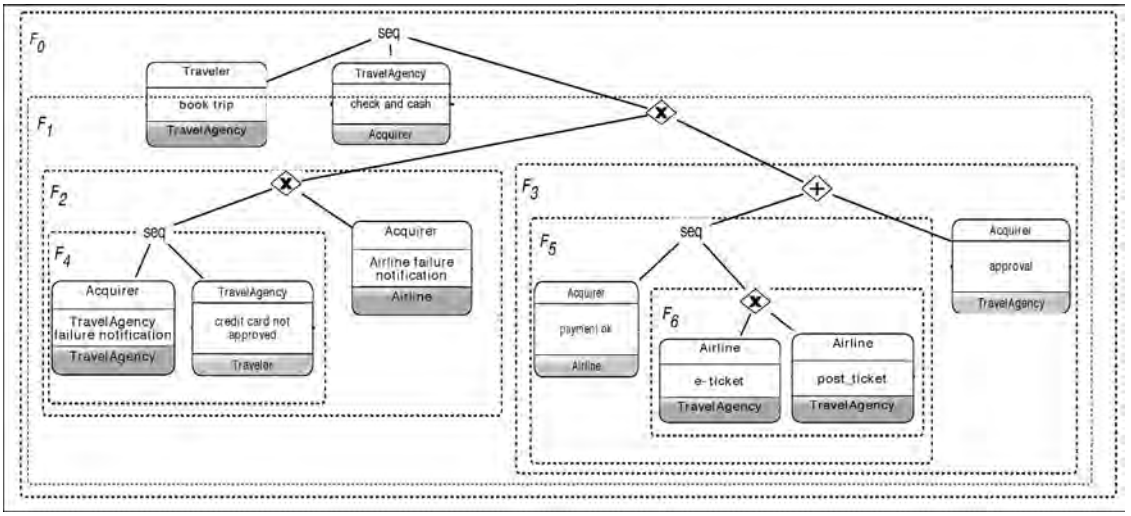

Fig. 4. Process structure tree of the book trip operation.

constraints between them from the viewpoint of p:

$$\begin{aligned} \mathit{LocalModel} &::= \mathit{LNode} \\ \mathit{LNode} &::= \mathit{InteractionActivity} \mid \mathit{ControlNode} \mid \mathit{Event} \\ \mathit{InteractionActivity} &::= \mathit{Send}(\mathit{Message}, \mathit{Receiver}) \\ &\quad \mid \mathit{Receive}(\mathit{Message}, \mathit{Sender}) \\ \mathit{ControlNode} &::= \mathit{SEQ}(\{\mathit{LNode}\}) \mid \mathit{CHC}(\{\mathit{LNode}\}) \mid \\ &\quad \mathit{PAR}(\{\mathit{LNode}\}) \mid \mathit{RPT}(\mathit{LNode}) \\ \mathit{Event} &::= \mathit{Start} \mid \mathit{End} \end{aligned}$$

Fig. 2 represents a collaboration diagram that illustrates the different public models of the book trip choreography example. Note that each panel defines the public model of one single partner.

Definition 3 (*(Structured) Choreography Model*). A global choreography model $\mathcal{G}$ represents a global view on the interactions between collaborating partners.

$$\begin{aligned} \mathit{ChoreographyModel} &::= \mathit{CNode} \\ \mathit{CNode} &::= I(\mathit{Sender}, \mathit{Receiver}, \mathit{Message}) \mid \\ &\quad \mathit{ControlNode} \mid \mathit{Event} \\ \mathit{ControlNode} &::= \mathit{SEQ}(\{\mathit{CNode}\}) \mid \mathit{CHC}(\{\mathit{CNode}\}) \mid \\ &\quad \mathit{PAR}(\{\mathit{CNode}\}) \mid \mathit{RPT}(\mathit{CNode}) \\ \mathit{Event} &::= \mathit{Start} \mid \mathit{End} \end{aligned}$$

I corresponds to an interaction between partners *Source* and *Destination* (i.e., the exchange of message *Message*).

An example of a choreography model is illustrated in Fig. 1. We define a *fragment* $\mathcal{F}$ as a non-empty subtree of a private model, public model or choreography model with single entry and single exit edge (SESE). Regarding Definitions 1–3, a tree model fragment is represented by elements *PNode*, *LNode* and *CNode*, respectively. Next, we define a choreography as the aggregation of all elements necessary for ensuring a sound collaboration between the participating partners.

Definition 4 (*Choreography*). We define a choreography $\mathcal{C}$ as a tuple $(\mathcal{G}, \mathcal{P}, \Pi, \mathcal{L}, \psi, \varphi, \xi)$ where,

- $\mathcal{G}$ is the choreography model (cf. Definition 3).
- $\mathcal{P}$ is the set of all participating partners.
- $\Pi = \{\pi_p\}_{p \in \mathcal{P}}$ is the set of all private models (cf. Definition 3).
- $\mathcal{L} = \{l_p\}_{p \in \mathcal{P}}$ is the set of all public models (cf. Definition 2).
- $\psi = \{\psi_p : l_p \leftrightarrow \pi_p\}_{p \in \mathcal{P}}$ is a partial mapping function between nodes of the public and private models.
- $\varphi : l \leftrightarrow l'$ is a partial mapping function between nodes of different public models.
- $\xi : \mathcal{G} \leftrightarrow l$ is a partial mapping function between nodes of the choreography model and the public models.

Functions ψ and φ can be used to check the consistency between public and private models (i.e., each private model must be consistent with the respective public model) as well as the compatibility between public models.

3.2. Change and change propagation

In order to represent changes of a choreography, we consider four basic change patterns: REPLACE, DELETE, INSERT, and UPDATE (cf. Fig. 5).

Definition 5 (*Change Patterns*).

$$\begin{aligned} \mathit{ChangePattern} &::= \mathit{REPLACE}(\mathit{oldFragment}, \mathit{newFragment}) \mid \\ &\quad \mathit{DELETE}(\mathit{fragment}) \mid \\ &\quad \mathit{INSERT}(\mathit{fragment}, \mathit{how}, \mathit{pred}, \mathit{succ}) \mid \\ &\quad \mathit{UPDATE}(\mathit{activity}, \mathit{attribute}, \mathit{newValue}) \\ \mathit{how} &::= \mathit{Parallel} \mid \mathit{Choice} \mid \mathit{Sequence} \\ \mathit{attribute} &::= \mathit{partner} \mid \mathit{role} \mid \mathit{Input} \mid \mathit{Output} \end{aligned}$$

REPLACE allows replacing an existing fragment with a new one. DELETE removes an existing fragment, whereas INSERT adds a new fragment to the process model between a predecessor node *pred* and a successor node *succ*. Finally, UPDATE allows modifying an attribute of a single activity as illustrated in Fig. 5. Note that more

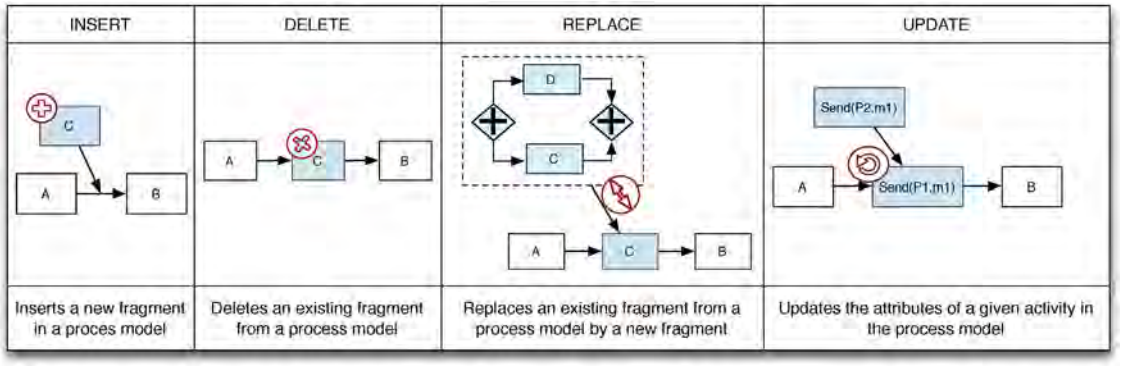

Fig. 5. Change patterns.

complex changes can be expressed by combining these four patterns. Change patterns are defined as follows:

Definition 6 (*Change Operation*). A change operation is a tuple (δ,σ) where σ is either the private, public or choreography model to be changed, and δ:$\sigma \mapsto \sigma'$ corresponds to the change that transforms σ into σ'.

Example 2. Consider Fig. 3: *DELETE*(*check_and_cash*, $\pi_{Acquirer}$) deletes the activity *check_and_cash* from the private model of the Acquirer.

Definition 7 (*Abstraction Function*). An abstraction function $abstr_{\lambda}: \sigma \mapsto \sigma'$ is a projection of a model σ according to criterion λ. The following holds:

- $\forall n \in \sigma$ with n satisfies λ, $\Longrightarrow n \in \sigma'$ (n refers to node).
- $\forall n, n' \in \sigma$ with n, n' satisfying λ and n precedes n' in σ, $\Longrightarrow n, n' \in \sigma' \wedge n$ precedes n' in σ'.

Function $abstr_{\lambda}(\sigma)$ transforms a source model σ into a target model σ' that solely contains activities satisfying λ; e.g., a public model corresponds to an abstraction of a private model with respect to *interactionactivities* (cf. Definitions 1 and 2). The abstraction of a private model may further contain structures not contributing to process execution (e.g., "empty" branches in a parallel branching). In this case, refactorings may be applied [20]. Next, we assume that $\lambda = p'$ refers to the interactions with p'. Hence, $abstr_{p'}(l_p)$ corresponds to the abstraction of l_p according to the interactions of p with p'. As result, we obtain a view on all interactions p has with p'. The abstraction function allows calculating the propagation effects; e.g., by identifying the effects a change of a private model has on its corresponding public model.

Example 3. The result of abstracting fragment $\mathcal{F}$ (cf. Example 1) according to its interactions is as follows:

$PAR(Send(payment\_ok, airline), Send(approval, travelAge\_ncy))$

Definition 8 (*Complement Function*). Assume that $a \in p$ corresponds to an interaction activity with a partner p'. Then: The complement of a, which is denoted as $\overline{a} \in p'$, corresponds to the opposite of a, i.e.,

- $\overline{send(message, p')} = receive(message, p)$.
- $\overline{receive(message, p')} = send(message, p)$.

If $\mathcal{F}$ corresponds to a fragment solely consisting of interaction activities, $\overline{\mathcal{F}}$ corresponds to a fragment having the same structure as $\mathcal{F}$ and replacing each activity of $\mathcal{F}$ with its complement. This function is required to maintain the compatibility between process partners when propagating changes.

Given an arbitrary set of nodes of a model σ, we define α as the function returning the smallest fragment in σ containing all these nodes. This function allows keeping the effects of a change as local as possible.

Definition 9 (*Smallest fragment* α). Let σ be a model and $\mathcal{S}$ be a set of corresponding nodes. Then: $\alpha_{\sigma}(\mathcal{S})$ returns the smallest fragment in σ containing all nodes from $\mathcal{S}$. Formally: $\alpha_{\sigma}(\mathcal{S}) = argmin_{size(\mathcal{F})}\{\mathcal{F} \in \sigma | \forall n \in \mathcal{S}, n \in \mathcal{F}\}$.

Example 4. In Fig. 1, $\alpha_{\mathcal{G}}(\{payment\_ok, approval\}) = F_3$ holds.

Usually, determining the changes to be propagated to the partner processes requires knowledge about the activities executed before or after the changed fragment. In this context, the following definitions are useful.

Definition 10 (*Preset (Postset)*). The *preset* (*postset*) of a node n in model σ corresponds to the set of nodes in σ that can be executed directly before (after) n. Formally:

- $preset(n, \sigma) = \{n' \in \sigma | \exists SEQ(n', n) \in \sigma\}$
- $postset(n, \sigma) = \{n' \in \sigma | \exists SEQ(n, n') \in \sigma\}$

We further define the *preset* (*postset*) of a fragment $\mathcal{F}$ in a given model σ as the fragment of σ that can be executed directly before (after) $\mathcal{F}$.

Example 5. Consider Fig. 1. We obtain $preset(check\_and\_cash, \mathcal{G}) = \{book\_trip\}$ and $postset(g_1, \mathcal{G}) = \{g_2, g_4\}$.

Definition 11 (T_preset_{λ} ($T\_postset_{\lambda}$)). In a model σ, the transitive preset (postset) of a node n, according to criterion λ, represents the set of nodes that satisfy λ and can be executed directly before (after) n.

- $T\_preset_{\lambda}(n,\sigma) = preset(n, abstr_{\lambda}(\sigma))$
- $T\_postset_{\lambda}(n,\sigma) = postset(n, abstr_{\lambda}(\sigma))$

The transitive preset (postset) T_preset_{λ} ($T\_postset_{\lambda}$) of fragment $\mathcal{F}$, according to criterion λ, represents the smallest fragment $\mathcal{F}'$ of σ that contains all nodes satisfying λ and being executable directly before (after) $\mathcal{F}$.

Example 6. Consider Fig. 1. We obtain

- $T\_preset_{Acquirer}(check\_and\_cash, \mathcal{G}) = \{start\}$, *and*
- $T\_postset_{Traveler}(g_2, \mathcal{G}) = \{credit\_card\_not\_approved\}$.

4. Change propagation

Our goal is to enable change propagation in choreographies with multiple interacting partner processes. Our approach is based on six major steps: (i) checking whether a change needs to be propagated or is isolated (i.e., the change is local), (ii) computing the private-to-public effects; i.e., propagating changes from the private model of the change initiator to its public model, (iii) computing the public-to-public effects; i.e., propagating changes to the partners involved, (iv) negotiating the changes with the concerned partners, (v) computing public-to-private effects (if negotiations have succeeded); i.e., each partner calculates internally the effects of the public changes on its private model, and finally (vi) checking the compatibility and consistency of the choreography and implementing the changes. Fig. 6 details these steps and outlines the different actions required to achieve a sound propagation.

When applying a change operation to a partner's private model, we extract all interaction activities concerned by the change–interaction activities are message exchanges with other partners (i.e., sending and receiving messages). If the list is empty (i.e., the change is restricted to the internal behavior), the other partners are not affected by the change. Hence, there is no need for any new agreement on the global choreography. Otherwise, the list of affected interactions is analyzed to identify all partners involved. Then, for each of these partners, a relative change computation is accomplished to determine the changes to be propagated. The latter are computed according to the change operation type. Then, a negotiation phase is launched with each affected partner. If all negotiations succeed, we apply consistency and compatibility checks to ensure the soundness of the obtained models. In turn, if these models are sound, we update the public models affected by the change as well as the choreography model and, if necessary, adapt concerned private models to their new public models. If negotiations do not succeed, either the change is canceled or it is tried to circumvent those partners with whom negotiations failed in the past. Note that this propagation strongly depends on the change pattern applied (i.e., INSERT, DELETE, REPLACE, or UPDATE). We sketch the different algorithms needed for propagating changes

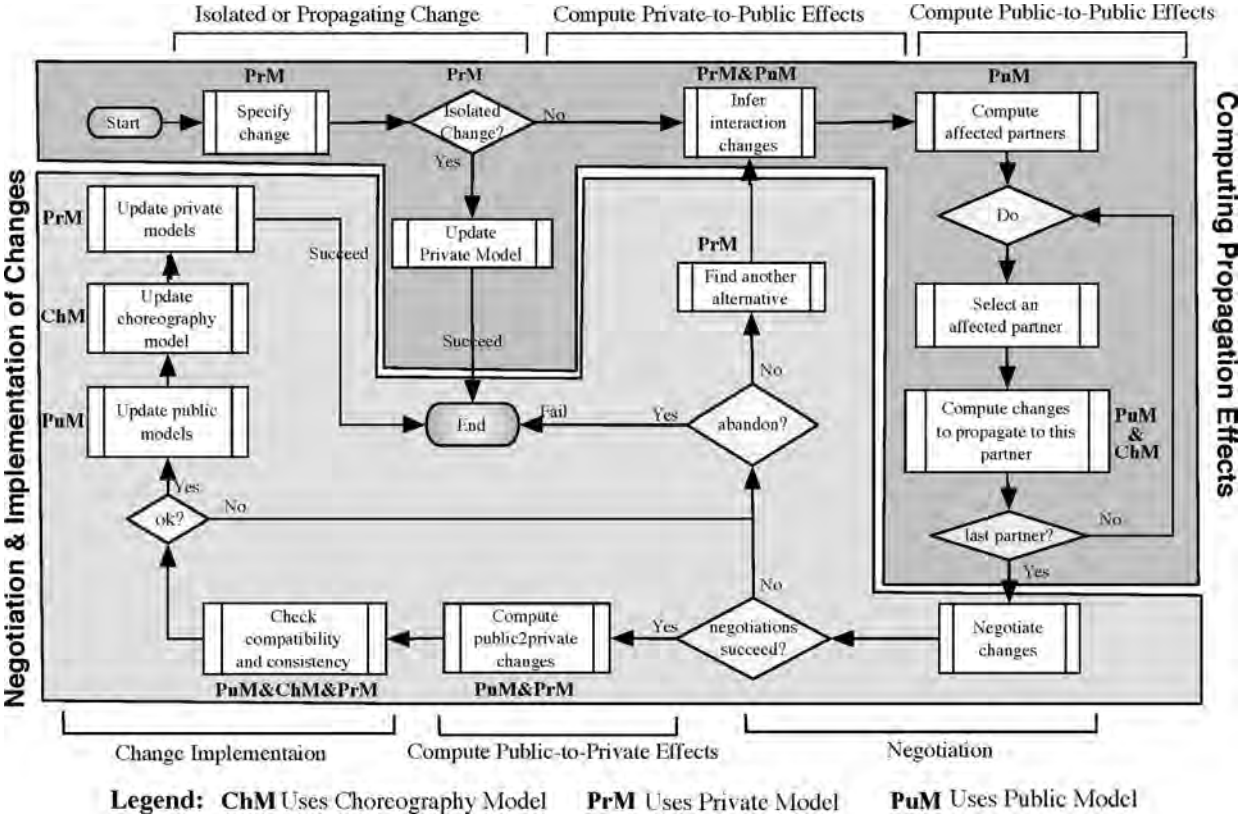

Fig. 6. Change propagation—major steps (adopted from [8]).

depending on the change pattern used. The propagation of change pattern UPDATE is not considered in this paper, but can be found in [8]. According to Fig. 6, the focus is on determining the public propagation effects of a single change (i.e., the parts in dark gray). In particular, we want to identify whether or not a change is isolated, and compute private-to-public and public-to-public effects. Negotiation and computing public-to-private-effects (i.e., effects of changing a partner's public model on its private model) are out of the scope of this paper

4.1. Propagation of fragment insertions

The INSERT pattern is used to add a new fragment $\mathcal{F}$ to the private model π_p of a partner p between two consecutive nodes *pred* and succ. In the following, we use the example depicted in Fig. 7 to explain and illustrate the main propagation steps for propagating fragment insertions. Given a change operation δ of type INSERT applied to a partner's private model π_p, and $\mathcal{F}$ being the fragment to be inserted in π_p between two nodes *pred* and *succ* (cf. Step 1 in Fig. 7), the ripple effects of δ can be computed asfollows:

1. *Isolated or propagating changes*: We first check whether $\mathcal{F}$ contains additional interactions or solely private activities by abstracting $\mathcal{F}$ with respect to interactions (cf. Step 2 in Fig. 7). If $\mathcal{F}' = abstr_{interaction}(\mathcal{F})$ is not empty (i.e., $\mathcal{F}$ contains at least one node), new interactions have been added and a change propagation becomes necessary. Otherwise, the change is considered as isolated; i.e., no propagation is needed.
2. *Private-to-Public effects*: In order to compute the impacts on the public model of the change initiator, we proceed as follows:
   - If $\mathcal{F}'$ is not empty, we calculate the corresponding fragment to be added to the public model l_p of change initiator p. The latter is the partner that initiated the change propagation. For this purpose, we use private-to-public mapping function ψ that transforms the elements of $\mathcal{F}'$ into elements of l_p. Note that this will be crucial if the private and public models are defined in terms of different modeling languages; e.g., in Fig. 7 the elements of π_p and $\mathcal{F}$ might be defined with BPEL, whereas the ones of $\mathcal{F}''$ and l_p are defined in BPMN (cf. Step 3 in Fig. 7).
   - When inserting $\mathcal{F}$ between *pred* and *succ* in π_p, this results in an insertion of $\mathcal{F}''$ in l_p. To maintain the consistency between π_p and l_p, $\mathcal{F}''$ should maintain the precedence relationship with *pred* and *succ*. Since *pred* and *succ* in π_p may be private activities without corresponding elements in l_p, however, it becomes challenging to identify the insertion positions *pred*′ and *succ*′ of $\mathcal{F}''$ in l_p (cf. Step 4 in Fig. 7). Therefore, we first check whether *pred* and *succ* constitute interaction activities or have corresponding elements in l_p ($\psi(pred) \neq \varnothing$). In this case, we consider the corresponding positions in l_p; i.e., $pred' = \psi(pred)$ and $succ' = \psi(succ)$ respectively. Otherwise, we look at the elements of π_p having corresponding elements in l_p and directly preceding *pred* (i.e., $T\_preset_{interaction}(pred)$) and following *succ* (i.e., $T\_postset_{interaction}(succ)$). Then, $\mathcal{F}''$ is inserted

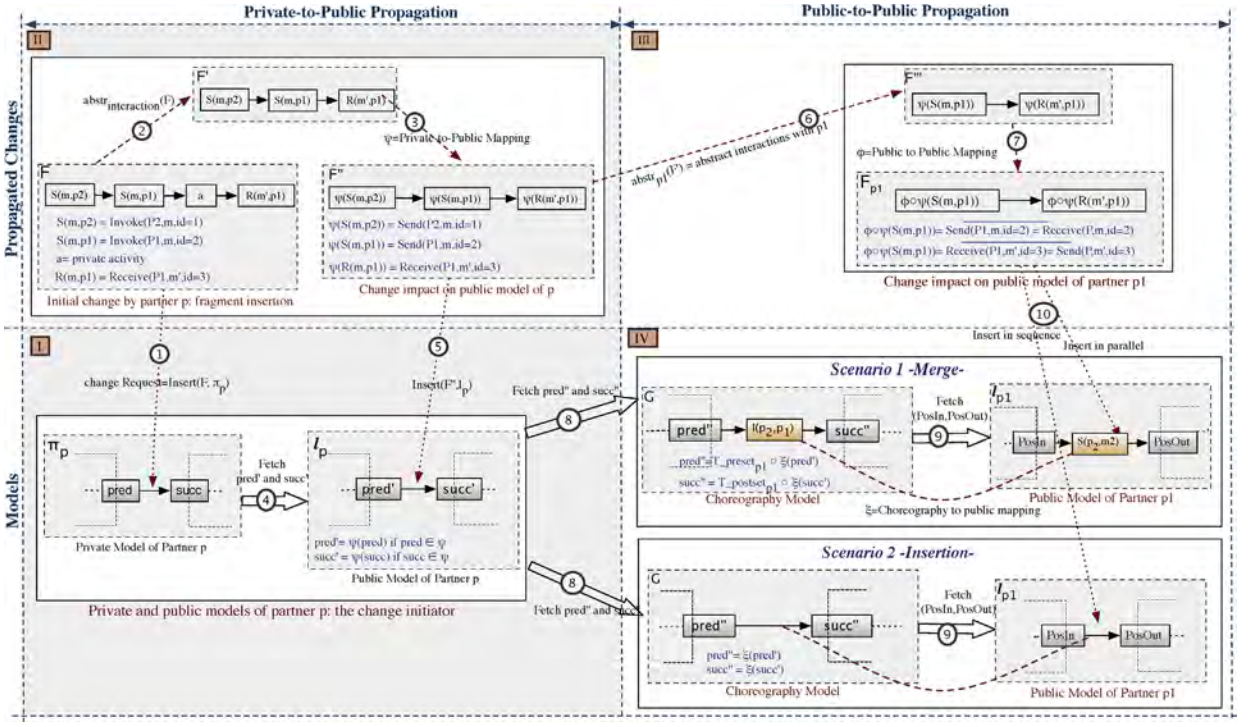

Fig. 7. Major steps for propagating an insert operation.

between the corresponding elements in l_p as follows (cf. Step 5 in Fig. 7):
   - $pred' = \psi \circ T\_preset_{interaction}(pred)$
   - $succ' = \psi \circ T\_postset_{interaction} succ)$
3. *Public-to-Public effects*: In order to calculate the change impacts on the other partners, we proceed as follows:
   - We analyze $\mathcal{F}''$ in respect to the list of partners involved in the change. For each of these partners, we identify the interactions this partner is involved in. Given a partner p_1, this induces the calculation of $abstr_{p_1}(\mathcal{F}'')$ (cf. Step 6 in Fig. 7). In turn, abstraction function $abstr_{p_1}$ returns a connected component including all interactions with p_1; i.e., $\mathcal{F}'''$.
   - $\mathcal{F}'''$ represents the fragment to be inserted into the public model l_{p_1} of p_1. To preserve the compatibility of the collaboration, however, we must calculate the complement of $\mathcal{F}'''$ (i.e., $\mathcal{F}_{p_1} = \overline{\mathcal{F}'''}$) and update the public-to-public mapping function φ (cf. Step 7 in Fig. 7). The latter maintains the correlation between nodes of different public models.
   - Given the fragment $\mathcal{F}''$ to be inserted between $pred'$ and $succ'$ in l_p (cf. Step 5 in Fig. 7), how can we identify the insertion positions of the corresponding fragment $\mathcal{F}_{p_1}$ in l_{p_1}; i.e., *PosIn* and *PosOut* (cf. Step 10 in Fig. 7). This becomes challenging if $pred'$ and $succ'$ of l_p have no corresponding elements in l_{p_1}, or p has no interactions with p_1. Utilizing the choreography model $\mathcal{G}$ then becomes primordial since it provides a global view on the interactions of all partners. Further, it contributes to identify the relationships between the elements $pred'$ and $succ'$ of p, and the interaction activities of p_1. The problem is shifted to finding the corresponding elements of $pred'$ and $succ'$ in $\mathcal{G}$ (i.e., $\xi(pred')$ and $\xi(succ')$ respectively), using the public-to-choreography mapping ξ. Then, we analyze the interactions in $\mathcal{G}$, p_1 is involved in, and which precede $\xi(pred')$ and follow $\xi(succ')$; i.e., $pred'' = T\_preset_{p_1}(\xi(pred'))$ and $succ'' = T\_postset_{p_1}(\xi(succ'))$ respectively. Again, using the public-to-choreography mapping $\xi: \mathcal{G} \rightarrow l_{p1}$, we identify the insertion positions in l_{p_1} with $PosIn = \xi(pred'')$ and $PosOut = \xi(succ'')$. We distinguish two possible scenarios when inserting $\mathcal{F}_{p_1}$:
     - Scenario 1: There exist no interaction activities between *PosIn* and *PosOut* in l_{p_1}. In this case, $\mathcal{F}_{p_1}$ should simply be inserted between *PosIn* and *PosOut*.
     - Scenario 2: There exists a set S of interaction activities *PosIn* and *PosOut* in l_{p_1}. In this case, $\mathcal{F}_{p_1}$ is merged with all elements of S.

4.2. Propagation of fragment deletions

The DELETE change pattern allows removing an existing fragment from a process model. This becomes challenging if the fragment contains interaction activities referring to other partners. If we do not update the processes of these partners when deleting the interaction activities, incompatibilities in the choreography are introduced. For example, a partner might then wait for a message that will never arrive or send a message that will never be consumed. To avoid such errors, a propagation mechanism should be adopted that keeps the processes (i.e., the public models of the partners) compatible with each other. Further note that the deletion of an interaction might have transitive (i.e. indirect) effects that cannot be solely handled based on the process structure; i.e., knowledge about semantics is required.

Example 7. We consider a supply chain scenario. Assume that a local city council starts a new construction project and hence collaborates with a city planner being in charge of the project execution. In turn, the city planner interacts with several third party partners responsible for designing, supplying and building tasks. Therefore, if the city council cancels the project, the city planner must cancel his contracts with the other partners as well.

This section does not consider the transitive effects of an interaction activity deletion, but only its direct structural effects. A non-exhaustive list of transitive scenarios as well as corresponding solutions are presented in Section 5. In the following, we use the example from Fig. 8 to illustrate the most important steps for propagating activity deletions. Given a partner process π_p and the fragment $\mathcal{F} \in \pi_p$ to be deleted, we proceed as follows:

1. *Isolated or propagating changes*: We check whether $\mathcal{F}$ solely consists of private activities. In this case, the change can be considered as isolated and there is no need for any change propagation. If fragment $\mathcal{F}$ contains interaction activities, in turn, change propagation becomes necessary.
2. *Private-to-Public effects*: To determine the impact the deletion of the interaction activity has on the public model of the change initiator, we apply the following steps:
   - We identify all interaction activities to be deleted by abstracting $\mathcal{F}$ with respect to interactions; i.e. $\mathcal{F}' = abstr_{interaction}(\mathcal{F})$ (cf. Step 2 in Fig. 8).
   - We identify the corresponding elements of $\mathcal{F}'$ in the respective public model l_p of p. To this end, we use private-to-public mapping function ψ and delete all elements of $\mathcal{F}'' = \psi(\mathcal{F})$ in l_p (cf. Steps 3–4 in Fig. 8).
3. *Public-to-Public effects*: To determine the change effects on the public models of the other partners, the following is applied: For each partner p_1 involved in $\mathcal{F}''$, we identify all interactions this partner is involved in by applying abstraction function $\mathcal{F}''' = abstr_{p_1}(\mathcal{F}'')$. For each element of $\mathcal{F}'''$, we determine the corresponding element in l_{p_1} using the public-to-public mapping function φ (i.e., $\mathcal{F}_{p_1} = \varphi(\mathcal{F}''')$). Note that the elements of $\mathcal{F}_{p_1}$ are not necessarily directly connected in l_{p_1}; they could be separated by other activities or gateways instead. To handle this case, for each of these elements we generate a separate delete operation. Finally, after each deletion, model refactorings may be applied (cf. Steps 5–7 in Fig. 8).

It is noteworthy that the interactions between two partners are often accomplished synchronously in the sense that the partner process who sends a message to another partner process may wait for a response from the

latter before proceeding with its execution. In certain scenarios, it might happen that the response is deleted due to a transitive effect of the first deletion. We will discuss these transitivity issues in Section 5.

4.3. Propagation of fragment replacement

Change pattern REPLACE modifies the structure and elements of a given fragment in a process model. This pattern is particularly useful when the redesign of the entire process or a part of it becomes necessary; e.g. to optimize the flow between the activities; e.g., in the *book trip* example from Fig. 1, one might want to replace fragment `PAR(Airline_notification_failure, TravelAgency_notification_failure)` by changing this parallel branching into a choice `CHC(Airline_notification_failure, TravelAgency_notification_failure)`. In the following, we refer to the change scenario from Fig. 9 to illustrate the most relevant steps towards the propagation of the resulting changes. Given a fragment $\mathcal{F} \in \pi_p$ to be replaced by a new fragment $\mathcal{F}'$, change propagation can be accomplished as follows:

1. *Isolated or propagating changes*: We first need to determine whether the fragment replacement constitutes a local change or needs to be propagated to other partners as well. For this purpose, we check whether $\mathcal{F}$ or $\mathcal{F}'$ contain any interaction activities. In this case, a propagation becomes necessary to maintain the compatibility between the partner processes. When replacing a fragment by another one, new interaction activities may be added, existing ones be removed, or the sequencing between interaction activities be changed. Accordingly, the partners directly affected by the fragment replacement are those interacting with p in the scope of both $\mathcal{F}$ and $\mathcal{F}'$ (i.e., p_1, p_2 and p_3 in the scenario from Fig. 9).
2. *Private-to-Public effects*: To identify the effects of a fragment replacement on the public model l_p of p, we first abstract fragments $\mathcal{F}$ and $\mathcal{F}'$ with respect to interaction activities. Then, we identify the corresponding elements to be replaced in l_p using the private-to-public mapping function ψ; i.e. $\mathcal{F}_1 = \psi \circ abstr_{interaction}(\mathcal{F})$ and $\mathcal{F}_2 = \psi \circ abstr_{interaction}(\mathcal{F}')$. The initial replace request is then transformed into a $REPLACE_{l_p}(\mathcal{F}_1, \mathcal{F}_2)$ operation that, in turn, needs to be propagated since it affects the interactions with other partners (cf. Steps 2–4 in Fig. 9).
3. *Public-to-Public effects*: To determine the change effects on the public models of the other partners, we do the following:
   - When replacing $\mathcal{F}_1$ by $\mathcal{F}_2$ (cf. Step 5 in Fig. 9), three scenarios are possible: (i) a partner involved in the original fragment $\mathcal{F}_1$ is no longer present in the new fragment $\mathcal{F}_2$ (i.e., the interaction with this partner is deleted), (ii) a partner involved in $\mathcal{F}_2$ was not present in $\mathcal{F}_1$ (i.e., a new interaction activity is added), and (iii) a partner is present in both fragments $\mathcal{F}_1$ and $\mathcal{F}_2$, but with different structure. Note that for one and the same replacement, we may have to deal with various scenarios of which each is related to a particular partner. Accordingly, we abstract both the new and the old fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ with respect to each partner involved in the change. Accordingly, the REPLACE pattern is translated into a concatenation Δ of change patterns to be propagated to the concerned partners.
     (i) *Deletion scenario*: If a partner p' interacts with partner p in the context of the original fragment $\mathcal{F}_1$ and is not engaged in any interaction with p in the new fragment $\mathcal{F}_2$, we delete the respective interaction activities from the public model of p' (cf. Step 6a in Fig. 9). The deletion scenario is handled similarly as described in Section 4.2. (ii) *Insertion scenario*: If a particular partner p' has no interactions with p in

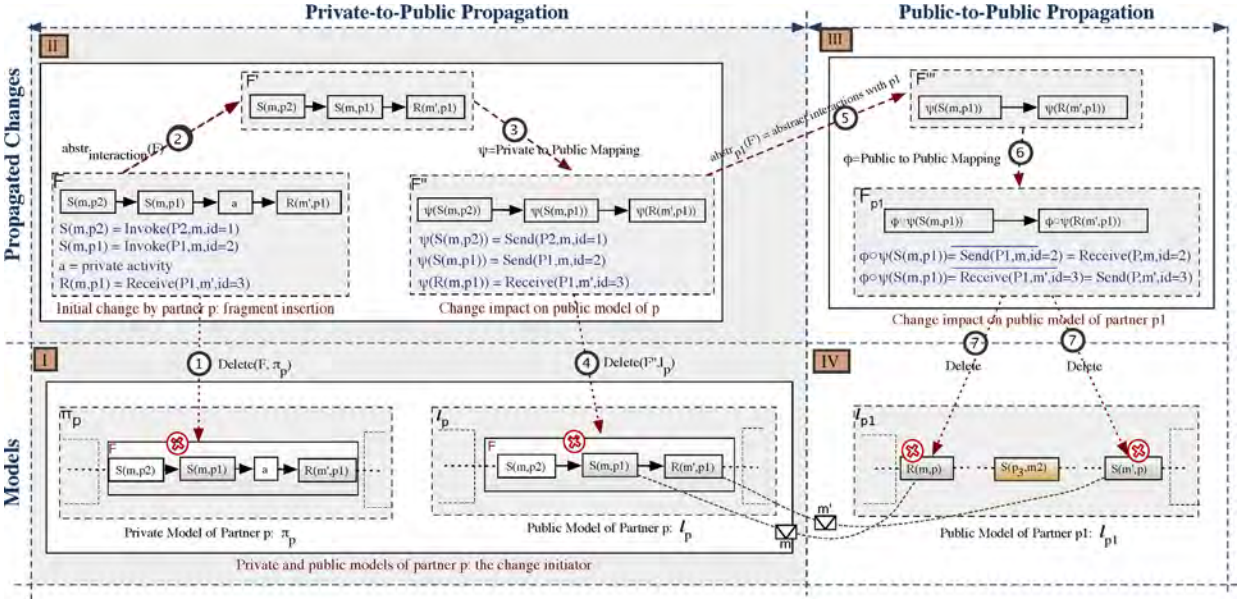

Fig. 8. Major steps of propagating a fragment deletion.

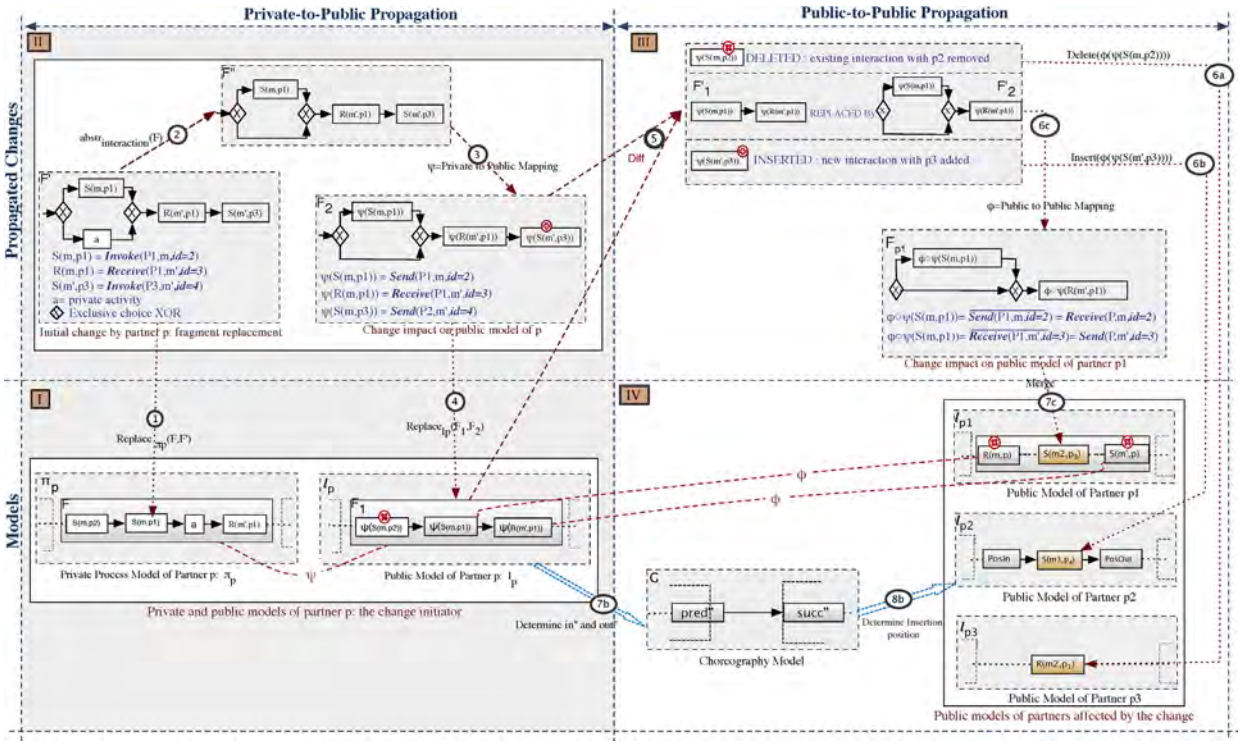

Fig. 9. Major steps of propagating a fragment replacement.

the context of old fragment $\mathcal{F}_1$, but interacts with p in the new fragment $\mathcal{F}_2$, we must insert the new interactions in the public model of p' (cf. Steps 6b–8b in Fig. 9). The insertion scenario is propagated similarly as described in Section 4.1.

(iii) *Replacement scenario*: The last scenario we consider is as follows: both fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ involve interactions with partner p', but with different structure. The latter means that the control flow dependencies between the interactions have changed and, therefore, the public model of p' shall be updated to preserve compatibility between all public models (cf. Steps 6c–7c in Fig. 9). For example, in the change scenario from Fig. 9 and in comparison with $\mathcal{F}_1$, $\mathcal{F}_2$ keeps the same interaction activities with partner p_1 for sending and receiving the messages m and m', but with different structure; i.e., message m is not always sent due to the exclusive choice. Consequently, the public model of p_1 should be updated and, in turn, the private model of p_1 be adapted to the latter if needed. Formally, given $\mathcal{F}_1$ and $\mathcal{F}_2$, we apply an abstraction with respect to each partner involved in the change and compare both results. If $abstr_{p'}(\mathcal{F}_1) = abstr_{p'}(\mathcal{F}_2)$ holds, no propagation to p' is needed since the interactions with p' remain invariant. Otherwise, a propagation is needed and the current interactions in $l_{p'}$ must be changed to ensure compatibility with the new fragment $\mathcal{F}_2$.

When propagating the changes to p', first of all, we need to fetch the matching elements of $\mathcal{F}_1$ in $l_{p'}$. In general, the interactions between p and p' in the scope of the old fragment $\mathcal{F}_1 \in l_p$ do not always have the same structure or distribution in $l_{p'}$ (but the same behavior instead). This is due to the applied refactorings as well as the different interactions p' has with the other partners; i.e., two interaction activities, which are directly connected in sequence in l_p, are not necessarily directly connected in sequence in $l_{p'}$, but could be separated by an interaction activity not involving p instead. The same holds for an interaction activity surrounded by a parallel branch (i.e., AND) in $l_{p'}$, which could be refactored to a sequence in l_p.

Consider Fig. 9. If we look at $\mathcal{F}'_1$ and $\mathcal{F}'_2$ as the abstractions of $\mathcal{F}_1$ and $\mathcal{F}_2$ in respect to p_1, matching activities of $\mathcal{F}'_1 = Seq(\psi(S(m,p_1)), \psi(R(m',p_1))) \in l_p$ are $R(m,p)$ and $S(m',p) \in l_{p_1}$. Note that these are separated by another interaction activity referring to p_2. To integrate the change we must transform $\mathcal{F}'_2$, using the public-to-public mapping $\mathcal{F}_{p_1} = \varphi(\mathcal{F}'_2)$, and merge it with the smallest fragment containing $R(m,p)$ and $S(m',p) \in l_{p_1}$ (i.e., the gray box in l_{p_1} in Fig. 9).

In general, we must consider the smallest fragment containing all interaction activities of $\mathcal{F}'_1$ in l'_p. Then, we must merge it with the corresponding elements of $\mathcal{F}'_2$. For this, we must adopt an algorithm that merges two process models or fragments. Note that merging process models has been widely studied in literature [22,23]. The key idea is to merge different (and overlapping) process models into a single model without restricting the behavior represented in the original models. Formally, if we consider γas a merge function, the problem can be solved by merging $\mathcal{F}_{p_1} = \varphi(\mathcal{F}'_2)$ with $\alpha_{l_{p_1}}(\varphi(\mathcal{F}'_1))$. It is noteworthy that such a merge might result in different scenarios among which the

corresponding partner should chose the most appropriate one.

4.4. Further steps and discussion

This section discusses the change propagation approach and highlights the main steps that follow the public-to-public change propagation. Note that the following steps are outside the focus of this paper, but can be considered as complementary to our work.

Negotiation: Computing change effects on the public models of the partners is automatic, relying on the presented algorithms. As shown in Fig. 6, after this step, a negotiation phase is required to approve or reject the intended changes. In general, such a negotiation cannot be fully automated, but requires an agreement among the partners. In particular, negotiations may involve human actors, e.g., through phone, e-mail, or meetings. Various approaches [60,59] exist that have dealt with negotiations in the context of process choreographies (e.g., based on service level agreements). Finally, note that negotiations might result in a redefinition of the initial change.

Public-to-private propagation: The propagation of a process change to the partners' public models might require adaptations of their private models as well. In general, these adaptations cannot be determined by the partner that initiated the change. Accordingly, once all partners involved in the change have agreed on the public changes, each of them must determine the required changes of its private model. In particular, the new private model must be consistent with the changed public model. Note that changes of the partners' private processes, in turn, might lead to new changes that need to be propagated to other partners (i.e., transitivity). Since a change initiator must not access the private process of other partners, the partners affected by the change themselves are responsible for adapting their private processes to the requested change. In turn, this might lead to cascading effects or even the multiple involvement of a partner during change propagation.

Change implementation: After all public and private changes are determined and agreed on, the soundness of the corresponding models is checked, the changes are implemented, and the public, private and choreography models are updated.

In [35], a multitude of composite change operations are described of which not all are considered in this paper. In general, most change operations can be realized using the basic DELETE and INSERT operations; e.g., REPLACE can be considered as a combination of a DELETE followed by an INSERT. However, change propagation complexity varies significantly. Worst case, for example, the complexity of directly propagating a REPLACE is equal to the one of a DELETE followed by an INSERT. Indeed, the REPLACE operation refers to a fragment instead of a single node. Accordingly, replacing a fragment by a new one not means that all nodes of the old fragment are changed. Taking the nodes that remain unchanged into account significantly improves the propagation process and reduces the number of operations to be propagated. Regarding the REPLACE algorithm (cf. Algorithm 2), the three possible scenarios (i.e., deletion, insertion and replacement) are solely generated for parts that have changed. By contrast, unchanged parts do not require any propagation. However, a DELETE followed by an INSERT will first delete those parts, which entails a propagation to concerned partners, and then re-insert the same parts (entailing another propagation).

5. Transitivity of change propagation

This section presents a non-exhaustive list of use cases demonstrating the transitivity effects of the DELETE change pattern and the solutions to cope with them. Note that this is a semantic issue that cannot be resolved based on the propagation algorithms presented so far, which solely focus on structural issues. As example consider a scenario with three partners p_1, p_2 and p_3. Assume that p_1 invokes p_2 and p_2 invokes p_3. The latter returns the intermediary result to p_2, which then applies data transformations before sending the final result to p_1. If now p_1 decides to delete its interaction with p_2, one must further delete the subsequent interaction between p_2 and p_3, which is solely used to deliver the final result. If a partner deletes an interaction, semantically, this means he is unable to afford this service anymore or he does not need the data anymore. Then, the challenge is to determine whether an interaction has transitive effects on other interactions, and if yes, to identify and resolve these transitive effects.

Case 1. Partner p is the *final consumer* of a data element, and it launches an interaction that requires a response. Accordingly, p contains related interaction activities *send* and *receive*. Thereby, *send* is used to request the data from another partner, whereas the corresponding *receive* is used to receive the response to this request from another partner.

- *Case 1.1* p deletes the *send*/*receive* interaction activities; i.e., it does not need the data anymore (since p is the final consumer). Accordingly, we delete *send* and *receive*. In case all subsequent interactions with other partners are *solely* used to deliver this data, these interactions are deleted as well (e.g. supply chain scenarios). Of course, it is also possible that only a subset of the subsequent interactions are used to deliver this data. These interactions are then deleted only if they do not have any other role in the choreography; i.e., they are not required to calculate any other data (cf. Scenario 1 in Fig. 10). If they play another role in the choreography, in turn, the subsequent interactions are kept (cf. Scenario 2 in Fig. 10).

Example 8. Assume that there are two concurrent requests from partners *A* and *B* to partner *C*. Further assume that *C* is involved in subsequent interactions and then replies to *A* and *B*. If *A* deletes its interaction with *C*, we must not delete the subsequent interactions of C since they are still required to reply to *B*.

- *Case 1.2* p solely deletes the *send* pattern. We distinguish two scenarios:
  (i) Another partner starts the communication instead of p. Accordingly, we just update the corresponding *send* with the new partner.

(ii) p is not responsible anymore for triggering the other partner to deliver the response; i.e., the latter is provided automatically or under certain constraints. Hence, we delete the corresponding *send* (cf. Scenario 3 in Fig. 10).

- *Case 1.3* p solely deletes the *receive* pattern. This means either p does not need the data anymore or the latter is transferred to another partner. In the first case, we just delete the corresponding *receive* and look for other interactions correlated with this response (used *solely* for delivering the response, cf. Scenario 4 in Fig. 10). In the second case, we update the corresponding *receive* with the new partner.

Case 2. Partner p corresponds to the *final consumer* of the data, but is not responsible for launching the first interaction; i.e., p has only the *receive*. If p deletes the *receive* (i.e., p does not need the data anymore), we delete the corresponding *receive* as well as all subsequent interactions that are *solely* used to deliver this data. All interactions participating in the delivery of this data, but having another role in the choreography, are kept.

Case 3. Partner p is the *starting point*, responsible for starting an interaction that results in a response to another partner; i.e., p has the *send*. Either (i) another partner is responsible for starting this interaction; then, we update the *send* with the new partner, or (ii) the interaction starts automatically or under other constraints on the target partner; then we delete the corresponding *send*. Subsequent interactions are not deleted since we still need the final data to be delivered to the final consumer.

Case 4. Partner p is an *intermediary partner*, and has correlated interaction activities *receive* and *send*. p receives a request and starts a *subsequent* interaction necessary for delivering the final response to the requester.

- *Case 4.1* Partner p deletes both the *send* and the *receive* interaction activities and is unable to provide the data anymore. However, still the final response to the requestor is needed. In this case, two choices exist: (i) Looking for another partner that can take over the task of p to deliver this response. Then, we update the subsequent interactions as well as the ones of the root partner (i.e., the partner that invoked p and the one to whom p shall send the result) with this new partner. (ii) Deleting *send* and *receive* as well as all subsequent interactions *solely* used in the context of this *intermediary data*, and looking for another partner or set of partners that can provide this data. Then, we update the interactions with the root partners or p. As example consider Scenario 5 in Fig. 10. If *Partner*3 is able to accomplish the data transformation ($d = f(d_1)$) of *Partner*2, the interactions of *Partner*1 with *Partner*2, which serve to deliver data d, are replaced by new ones with *Partner*3.

- Case 4.2 If p solely deletes the *send* interaction activity, another partner is responsible for starting this intermediary interaction or the subsequent interactions start automatically or under other constraints. In the first case, we update the corresponding *receive* with the new partner, otherwise we just delete it.

- Case 4.3 If p solely deletes the *receive* pattern, this means that p cannot take over the tasks necessary to deliver the final data. (i) If other operations are necessary to deliver the final data, we look for another

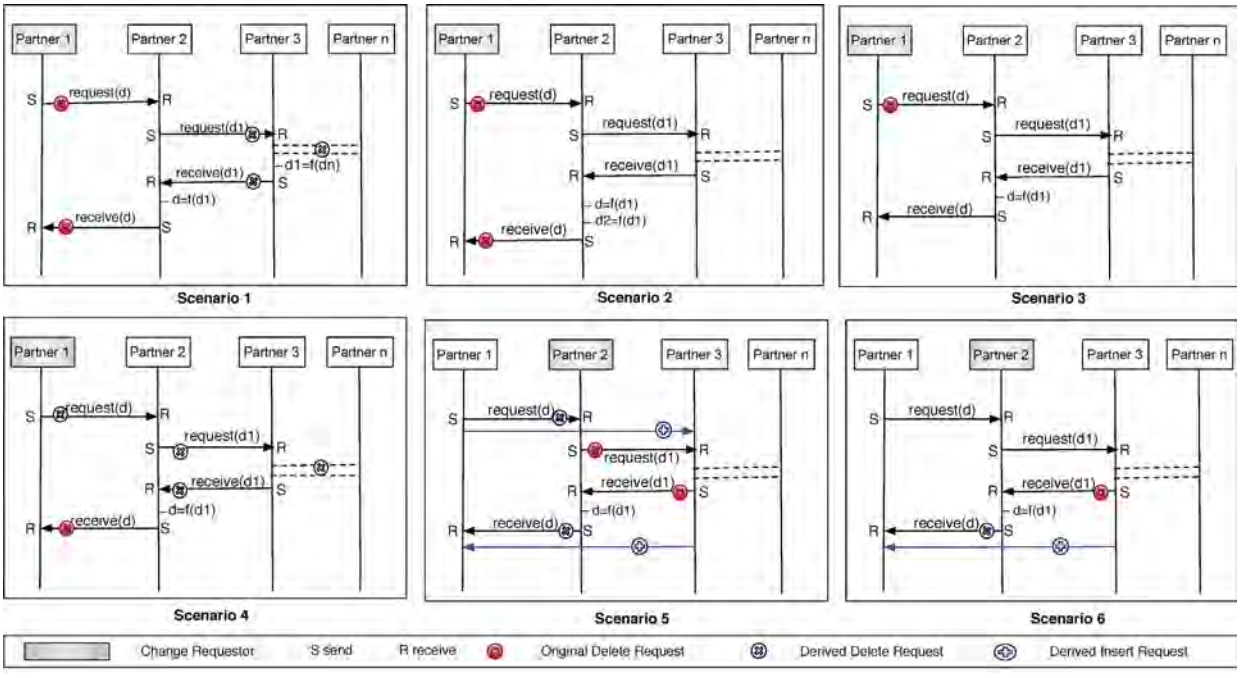

Fig. 10. Transitivity scenarios.

partner that can accomplish the same tasks. (ii) If not, we update the $\overline{Send}$ to link it directly with the root partner (cf. Scenario 6 in Fig. 10).

Conclusion. We presented a non-exhaustive list of possible scenarios of transitivity when dealing with change propagation. Clearly, transitivity is a semantic issue and requires a data model defining the relationships between the exchanged data objects (e.g., an ontology). Due to privacy issues, in addition, not all data correlations are always known, and therefore calculating the transitivity effects remains problematic and cannot be fully automated. Several proposals exist to predict the transitive effects in process choreographies based on prediction metrics (e.g. social graphs) [10].

6. Compatibility and consistency

This section discusses soundness issues of a process choreography in the context of change propagation. In particular, we check whether the compatibility and consistency properties of the collaborating business partners are kept. Accordingly, we assume that the initial public models of the collaborative processes are compatible with each other and that each private model is consistent with its corresponding public model. We further assume well-behavedness of the change operation in terms of structure and semantics. Recently, several proposals were made on checking the soundness of choreographies in terms of compatibility and consistency [15,24–28].

Before discussing the compatibility and consistency of the process choreography in the context of change propagation, first of all, we introduce useful properties. Thereby, Property 1 states that for each node of the public model of a partner p, there should be a matching element in the corresponding private model of p, but not vice versa. Furthermore, Property 2 expresses that for each node of the public model of a partner p, there should be a matching node in a different public model of another partner. Note that this is a necessary, but not yet sufficient condition for ensuring compatibility between public models. Finally, Property 3 states that for each node in a public model, there should be a matching node in the choreography model. In particular, for each interaction in the choreography model, there should be exactly two matching interaction activities in the public models. Formally:

Property 1.
$\forall l\_node \in l_p, \exists p\_node \in \pi_p$ *with* $\psi(l\_node) = p\_node$.

Property 2. $\forall l\_node \in l_p$ *with type* $(l\_node) = InteractionActivity$: $\exists p' \neq p$: $\exists l\_node' \in l_{p'}$ *with type* $(l\_node') = InteractionActivity \wedge \varphi(l\_node) = l\_node'$.

Property 3. $\forall l\_node \in l_p$: $\exists c\_node \in \mathcal{G}$ *such that* $\xi(l\_node) = c\_node$.

Lemma 1. $abstr_{p'}(\mathcal{F}) \in \mathcal{L}_p \Longrightarrow \overline{abstr_{p'}(\mathcal{F})} \in abstr_p(\mathcal{L}_{p'})$. *The complement of the abstraction of a fragment* $\mathcal{F} \in \mathcal{L}_p$ *from the perspective of a participant* p' *is a fragment of the abstraction of* $\mathcal{L}_{p'}$ *according to* p.

Proof. The proof of this lemma can be based on the following compatibility properties of choreographies. (c.f. [15]).

- If $a \in \mathcal{L}_p$ corresponds to an activity that interacts with partner p', the following holds: $\exists b \in \mathcal{L}_{p'}$ with $b = \overline{a}$.
- If $a_i, a_j \in \mathcal{L}_p$ are two activities interacting with the same partner p' and $\beta(a_i, a_j)$ is a function returning the minimal precedence relation (i.e., control flow path) between a_i and a_j [21], the following property (also denoted as bi-simulation property [15,18]) holds:
  $\exists b_i, b_j \in \mathcal{L}_{p'}$ with $b_i = \overline{a_i}$, $b_j = \overline{a_j} \wedge \beta(a_i, a_j) = \beta(b_i, b_j)$ □

6.1. Consistency checking

In our context, consistency means that the implementation of a business process (i.e., a private model) is consistent with its observable behavior (i.e., public model). This ensures that implementations of private processes satisfy the interaction constraints defined in the public models [15]. In our change propagation approach, the public model is defined as an abstraction of the private model by deleting all model elements not related to any interaction (e.g., Property 1). Accordingly, an insertion, deletion or replacement of a fragment in a private model needs to be transformed into an insertion, deletion or replacement of the fragment abstraction in the public model (if required). Since any abstraction preserves the consistency between the original and abstracted model (cf. [29,30]), the propagation from private-to-public does not affect consistency. Regarding deletion or replacement scenarios, refactorings may be applied. In turn, this eliminates unnecessary synchronization elements (e.g. a parallel branching between an activity and an empty branch is reduced to a sequence), but does not affect the consistency between the original and abstracted model. Change propagation might also result in the insertion, replacement or deletion of a fragment from a public model of a partner target. If the change is accepted by the latter, the change requester cannot check for the consistency between the public and private model of that partner since the private model of the latter is not visible. Our approach assumes that any partner affected by the change should update his private model locally if he accepts the change request. In turn, this update must be consistent with the new version of his public model.

6.2. Compatibility checking

Compatibility is a soundness criteria that checks whether the interacting partners are able to communicate with each other in a proper way (e.g., no deadlocks or livelocks will occur). In this context, [15] distinguishes between structural and behavioral compatibility:

Structural compatibility: It requires that for every message that may be sent, the corresponding partner is able to receive it. In turn, for every message that can be received, the corresponding partner must be able to send a respective message. Regarding our propagation mechanism, structural compatibility is always preserved. Depending on the change operation type, for each affected partner we add, update, or remove the complement of what has been changed in the process of the change initiator. In particular, for each interaction activity *send* in one process partner source, we insert or

delete the corresponding *receive* interaction activity with the expected attributes (e.g. message) in the process of the partner target (i.e., affected by the change) and vice versa (cf. Properties 2 and 3, and Lemma 1).

Behavioral compatibility. It considers behavioral dependencies (i.e., control flow) between message exchanges, i.e, it deals with the ordering of the partners' interactions. For example, a *Receive* encapsulated by a *Sequence* in one partner process should not be linked to a *Send* encapsulated by a *Choice* in the process of a different partner. Indeed, this might lead to a deadlock in case the path containing the *Send* in the *Choice* is not executed during runtime.

Assume that (δ, π_p) is the change operation to be applied to process model π_p and $(\delta, \mathcal{L}_p)$ corresponds to the inferred change to be applied to the public model of p. Further, let Δ be the set of changes inferred from $(\delta, \mathcal{L}_p)$ to be propagated to its directly affected partners. For each affected partner p_i, $(\delta_i, \mathcal{L}_i)$ represents the inferred change operation to be propagated to its public model; i.e., $\Delta = \wedge_{i=1..n}(\delta_i, \mathcal{L}_i)$, where n corresponds to the number of affected partners. Note that the number of inferred changes is finite since we only consider propagations to direct partners. In turn, changes that might have structural effects on other partners (due to transitive relations) are propagated to them through their direct partners recursively.

If $(\delta, \mathcal{L}_p)$ is invariant (i.e., it does not affect the public model of p), consistency and compatibility are preserved over the collaborative partners. In addition, since both processes and changed fragments are structured, consistency and compatibility relations can be reduced to those existing between the fragments affected by the change.

- The INSERT pattern augments the process models of the partners affected by the change with new activities and gateways respectively. Further, it does not affect the structural or behavioral dependencies (i.e., control flow) between the existing activities. However, some direct precedence relations between activities may be transformed into transitive ones (due to the insertion of new activities and gateways). The propagation of a change operation of type INSERT results solely in change operations of type INSERT in the public models of the affected partners. According to a particular partner, the insertion is done with respect to the direct and transitive dependencies with the activities of the same partner. As explained in Section 4, if $\mathcal{F}$ corresponds to the fragment to be inserted in $\mathcal{L}_p$, $\overline{abstr_i(\mathcal{F})}$ is the fragment to be inserted in $\mathcal{L}_i$. Note that the latter shows the same behavior (i.e., control flow) as $abstr_i(\mathcal{F})$. In turn, the insertion position is computed based on the transitive *preset* and *postset* of $\mathcal{F}_i$ with respect to partner i. Note that this preserves the order of the fragments and ensures their behavioral compatibility after propagating the INSERT operation. This propagation might result in a merge of the fragment to be inserted with an existing fragment as described in Algorithm 3. In particular, if the partner affected by the change interacts with different partners in the scope of the calculated insertion positions (i.e., there exist others interactions activities between the identified positions *pred* and *succ*), the fragment to be inserted between these two positions must be merged with the existing interaction activities in between. Accordingly, we assume that merge function $gamma: (\mathcal{F}, \mathcal{F}') \rightarrow \mathcal{F}''$ preserves the behavior of $\mathcal{F}$ and $\mathcal{F}'$ in the result $\mathcal{F}''$ of the merge.
- The DELETE operation reduces the process models of the partners affected by the change. This reduction is accomplished in a symmetric way on both sides; i.e., p and the partners affected. The deletion of an activity on one side results in the deletion of the corresponding $\overline{a}$ on the other. Structural and behavioral compatibilities are kept. However, other issues emerge, e.g., an activity might wait for data that will never arrive or send a message that might not be consumed. The solution we proposed in Section 5 deals with typical use cases where the correlated interactions are updated or deleted accordingly. Note that this

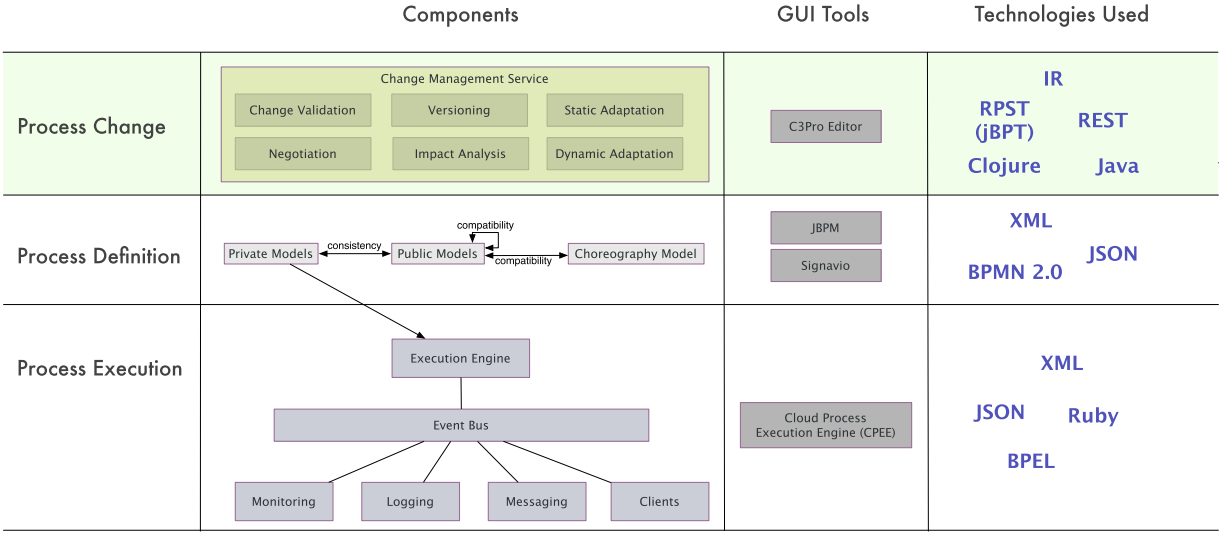

Fig. 11. Architecture of the change propagation framework.

neither affects structural nor behavioral compatibility of the propagation.
- The propagation of the REPLACE operation results in three scenarios: insert, delete, or merge. Assume that the merge function γ is correct and idempotent, preserving the behavior of the merged fragments. We consider $\mathcal{F}_i$ and $\mathcal{F}'_i$ as the fragments to be merged. Then, the behavior of $\mathcal{F}_i$ is reflected by the merge result $\gamma(\mathcal{F}_i, \mathcal{F}'_i)$ (cf. Lemma 1).

The consistency between the public and private models of the partners affected by the change will be checked if negotiations succeed. Each of these partners must then adapt its private model to the change of its public model. Using consistency rules, each partner can check locally whether or not its private model is consistent with its public model. More details about consistency checking and the validation of choreographies can be found in [5,2].

7. Proof-of-concept prototype and validation

This section outlines the architecture of our change propagation framework for process choreographies and presents the prototypical implementation of the C^3Pro Editor as one of its core components.

7.1. Framework architecture

Our architecture must provide functions for defining and executing process choreographies. Further, it must allow specifying, performing and propagating changes. To meet these requirements, we propose a layered architecture as depicted in Fig. 11. It consists of three layers: *Process Definition*, *Process Change*, and *Process Execution*.

The main change propagation functions are realized by the *Process Change* layer. The *C³Pro* Editor is one of the core components of this layer that realizes the change propagation algorithms presented in Section 4. Other functionalities provided by existing tools are delegated (e.g. process modeling and execution); i.e., although we focus on the functionalities of the *Process Change* layer, we communicate with the other two layers as well.

In the *Process Definition* layer, process designers use existing modeling tools (e.g., Signavio or jBPM) to create process as well as choreography models. The latter are serialized as XML or JSON files and serve as input for the *Process Change* layer. In turn, the latter layer defines all components related to change propagation in choreographies. Most prominently, the Change Management Service implements Algorithms 3–2 as well as the internal representation (IR) of private, public and choreography models. The functions of the other components from the *Process Change* layer are follows:

- Versioning capabilities are provided that allow *undoing* as well as *redoing* changes.
- Dynamic adaptation is enabled to deal with the migration of running process instances.
- Model verification is supported to verify the soundness of the models resulting after a change.
- Negotiation becomes necessary if a change is not acceptable for a partner. This component deals with strategies applicable if a negotiation is required (cf. Fig. 6).

All functions provided by the *Process Change* layer are exposed as a RESTful service, which allows for a unified access from any client able to communicate via HTTP. The Change Management Service can be accessed with the *C³Pro* Editor serving as the connector to the *Process Definition* layer. The *C³Pro* Editor provides functions for importing and visualizing choreography models. Moreover, changes may be applied to the models and required change propagations to partners be performed, allowing for the simulation of change propagation. Altogether, the *C³Pro* Editor serves as the front end for all components defined in the *Process Change* layer.

The Change Management Service serves as a pluggable middleware based on which process engines can be integrated. In particular, this integration allows these engines to access all components of the *Process Change* layer. This implies that after a successful change propagation, which includes negotiation and soundness checks (cf. Fig. 6), the updated choreography models are transformed into an executable form being directly passed to the process engine for enactment (*Process Execution* layer). In other words, from the perspective of the *Process Change* layer, the *Process Execution* layer serves as an execution platform for the updated models. The process execution engine we have chosen is the Cloud Process Execution Engine (CPEE).

7.2. Tool support: C³Pro editor

In the following, all occurrences of *nodes* refer to *PNode* (i.e., activities and control nodes) from Definition 3. We implemented the *C³Pro* Editor as the first prototypical client realizing the Change Management Service. In particular, this client component takes the role of a simulation environment for manually stepping through the change propagation process (cf. Fig. 6), which allows testing and verifying change scenarios. Altogether, the *C³Pro* Editor supports the visualization of

1. private, public and choreography models,
2. affected partners' *nodes* and fragments depending on the change type (i.e., INSERT, DELETE or REPLACE), and
3. the models resulting after the application of the calculated changes.

A multitude of process modeling tools exist. For this reason, we delegate the basic modeling functions to these tools. In our case, we have used Signavio [14]. We export the created models to BPMN 2.0 XML format. In turn, the latter is directly supported by our change propagation library for importing models. Once imported, models can be visualized and all changes be performed with the *C³Pro* Editor. The latter is accomplished by propagating the changes to each affected partners. The *C³Pro* Editor not only visualizes process models before and after a change, it also displays auxiliary information such as the affected

partners' nodes and fragments. We utilize the jBPT[4] library for handling the transformation of models (private, public and choreography) to RPST.

We realized the Change Management Service as a Java Library (JAR), which enables any language running on top of the Java Virtual Machine (JVM) to access the underlying public classes and static functions. This allowed us to develop the *C³Pro* Editor in a rapid fashion, still treating the Change Management library as a service. A frequently changing API would have hindered the concurrent development of the service and the editor. After finalizing the API and identifying the required functionalities, we implemented the REST infrastructure for the Change Management Service the *C³Pro* Editor consumes.

We chose Clojure as programming language that is amendable for rapid prototyping and iterative development. Further, it has a rich Read Eval Print Loop (REPL) environment. Its default runtime platform is JVM, enabling seamless interoperability with the Change Management service. Clojure follows a functional programming style and provides concurrent functionalities; the latter are important for GUI application development. Finally, it allows changing the behavior of a running program without restarting it and hence reducing development efforts significantly.

Figs. 12 and 13 depict screenshots of the *C³Pro* Editor. The *Project Explorer* on the left-hand side shows the current choreography model as well as the public models of all partners participating in the collaboration. Double clicking on any one of these items will display the corresponding model as a graph in the *Graph Panel*. In turn, the *Graph Panel* visualizes the selected graph as expected. Left-clicking on a *node* in the displayed graph will bring up its detailed information in the *Detail Panel* and show the related nodes in the *Related Nodes Panel*. The related *node* depends on the currently selected one. For each *send message*, the associated *receive message* is found and displayed, and vice versa. If an interaction activity (i.e., a *node* of the choreography model) is selected, the actual send and receive messages are picked from the appropriate public models and displayed in the *Related Nodes Panel*. If gateways (i.e., *ControlNode* of Definition 3) are selected, the smallest fragment (see Definition 9) that surrounds the selected gateway is displayed in the *Related Nodes Panel*.

Change operations are shown to the user by right-clicking on a node within the *Graph Panel* as well as on the left side below the *Project Explorer* (as buttons). When clicking one of the provided operations, a dialog window pops up prompting the user to specify the change. In the scenario depicted in Fig. 12, the user is asked to load the fragment for the INSERT operation. Afterwards, the *C³Pro* Editor applies the change and triggers the change propagation process required. Fig. 13 shows the screen after applying an INSERT operation. Furthermore, the *Graph Panel* allows for the display of a process model in terms of an RPST. Finally, the *Change Log* shows the output during the processing of a change propagation.

[4] https://code.google.com/p/jbpt/

7.3. Implementation of the trip booking process

As we were unable to find publically available choreography models that can serve as the basis for our simulation, we opted to use the trip booking example (cf. Figs. 1–3). We used the Signavio Process Editor to model both the choreography models and the partner-specific public models of the choreography. Note that it was ensured that all models are structurally as well as behaviorally sound. Further on, we ensured that the process models are block-structured, which, in turn, allowed for their easy transformation into corresponding RPST representations. In case unstructured models shall be imported, the techniques described in [18] can be applied to transform most of these models into structured ones.

Models are exported as XML files and then imported as initial data set into the *C³Pro* Editor. In total, 17,068 change operations of type INSERT, DELETE, and REPLACE were created and tested on the prototype.

8. Related work

Change propagation has been an active research area in software engineering. In particular, the analysis, evaluation and propagation of changes have been widely studied in large complex software systems [31–34,36–38]. However, respective approaches cannot be directly transferred to process choreographies since the latter entail several particularities; e.g., the distributed model structure, partly unavailable information about partner processes, dynamic aspects, and specific requirements (e.g., compliance, privacy and security). Note that these particularities raise additional challenges for change propagation algorithms, which have been partially addressed by only few approaches so far.

In [39], four transfer rules for dealing with dynamic changes of distributed processes are proposed. These rules use projection/protocol and life cycle inheritance relations in order to check whether a changed process corresponds to a subclass of the original one. The suggested method solely allows for changes preserving inheritance transformation rules, i.e., changes having only internal effects. Particularly, there is neither a need for change propagation nor for any new agreements on the global protocols (i.e., the choreography model) since only inheritance-conforming changes are allowed. By contrast, our approach also supports changes that affect the external behavior of a process by computing and propagating them to the affected partners.

In [40,41], change propagation techniques for partitioned processes are proposed, where a process model is split into several distributed partitions. This approach propagates changes applied to the original model to the respective partitions. It uses a decentralization function to compute the affected partitions and to infer the changes to be propagated as well. Moreover, it is one organization controlling the original as well as the derived partitions. Hence, it becomes easier to exactly compute the affected regions and the changes to be applied. Note that this differs from our approach since changes are applied by one partner participating in the choreography and are then propagated to the others. In particular, our work considers fully distributed processes; i.e., no partner holds information about another partner's private model. Each partner

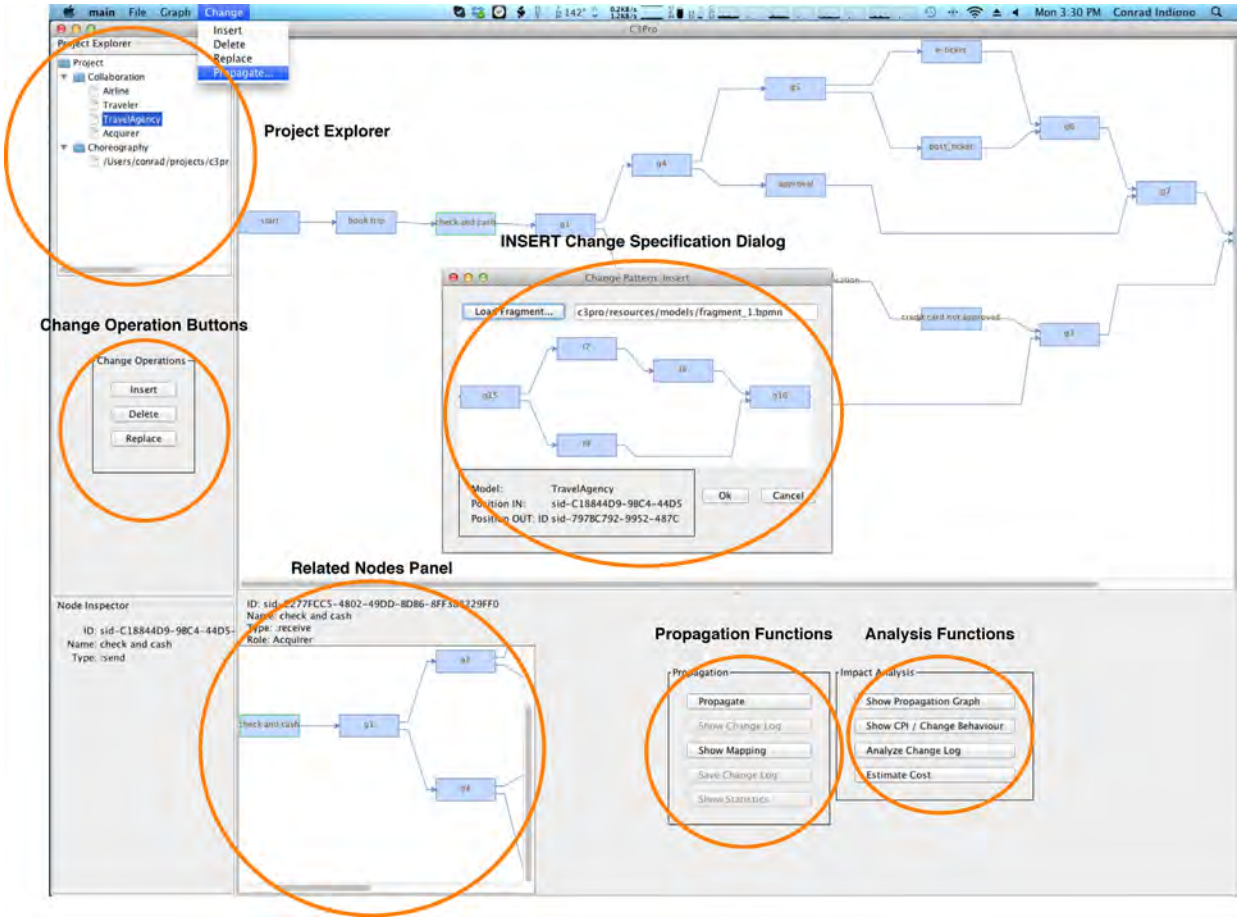

Fig. 12. *C*[3]*Pro* Editor—screenshot showing an INSERT operation being performed.

can only view the public models of the other partners. In the context of changes, this requires a negotiation phase between the affected partners and could have transitive structural and semantical effects on other partners recursively. As opposed to [40,41], our work considers transitivity issues as well. Other approaches similar to [40] are presented in [42,43].

The DYCHOR framework [9] addresses the challenge of propagating changes in process choreographies as well. Thereby, changes are classified into additive and subtractive changes, which may have variant or invariant impact on the interactions. DYCHOR uses annotated finite state automata to model choreographies and employs a set of operators to compute the changes to be propagated. In our work, we propose four change patterns that deal with more complex fragments instead of single activities solely. This leads to particular challenges concerning semantical and structural transitivity effects, and also requires negotiations with the partners affected by the change. We sketched semantic transitive effects as well as the solutions to deal with them. The approach adopted in this paper makes change propagation easier since process models are structured and only changed regions are affected. Hence, there is no need for completely re-computing the public models of affected partners entirely. Instead only the affected regions need to be adapted.

In [44], the problem of dynamic changes and versioning of process models is addressed. The same challenge is tackled in [45], where an ontology-based framework for decentralized workflow change management is presented and different migration rules for dynamic change adaptation are defined. In [46], change propagation between semantically overlapping process models, whose elementary as well as complex correspondences have been identified, is proposed. All these approaches are complementary to our work.

In [47], a method for propagating changes applied to a given software model is presented. In particular, it computes the additional changes required to meet an emerging change requirement. The approach proposes techniques to deal with consistency constraints violation. Different repair solutions (i.e., customizations) are introduced using cost models. In this approach, UML (Unified Modeling Language) is employed to specify the software model; further OCL (Object Constraint Language) is used to define constraints. [48] represents an extension of [47] to support SOA (Service Oriented Architecture); it is shown how changes can be propagated across a number of models using the Service-oriented Modeling language (SoaML). The cost calculation is substituted by a minimal modification strategy that helps selecting change options in such a way that it accommodates both the structural and semantic dimensions of SOA models.

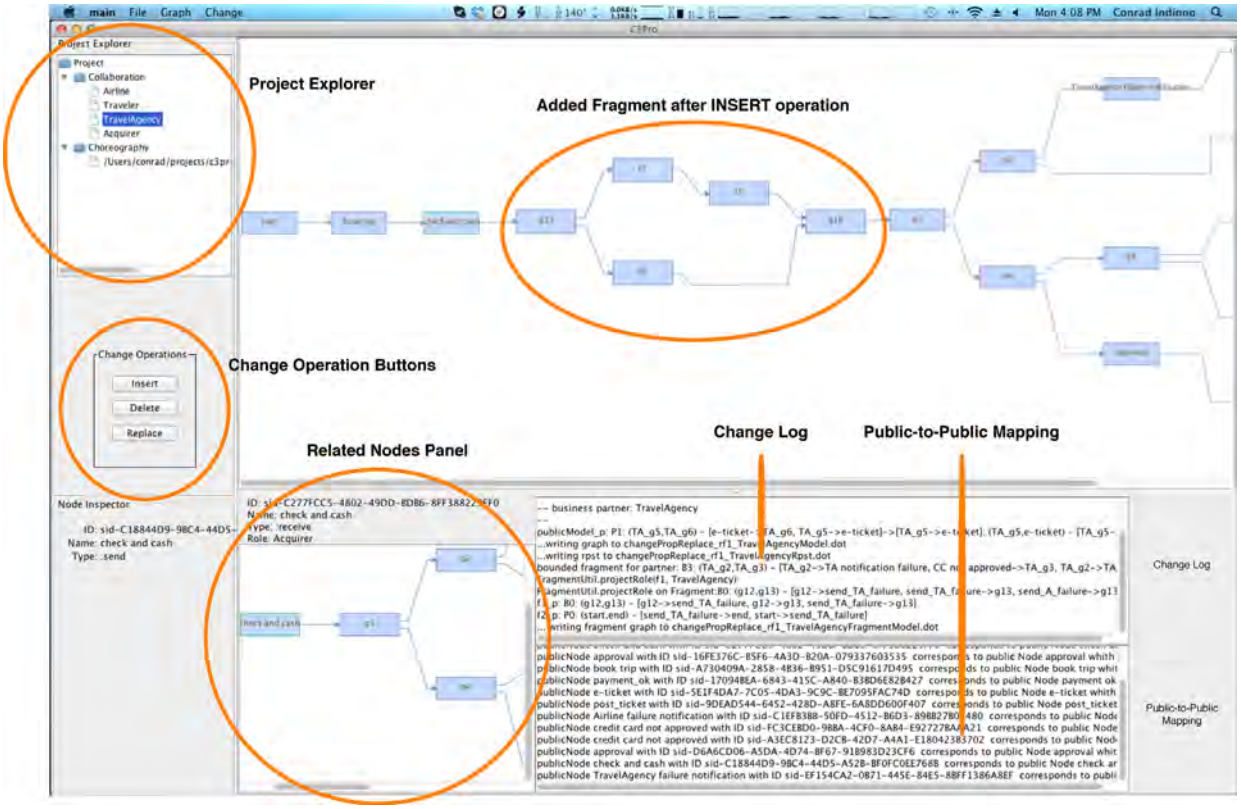

Fig. 13. *C³Pro* Editor—screenshot showing a public model after an INSERT operation.

Mafazi et al. [49] present an approach to propagate changes between *process views*. This approach considers a *reference model* from which several process views are derived. Further, it uses Petri nets to represent the different models, as well as means to check consistency after change propagation. A similar approach is provided in [50]. The models adopted by these approaches are different from the one described in this paper, where we distinguish between the private, public and choreography models. Accordingly, we distinguish between the compatibility public-to-public and the consistency private-to-public. Further, [49] does not consider the transitive effects of propagation. A similar approach is presented in [46], which computes change propagation between process views. However, the relationships between the activities of the different process views and the corresponding reference model are not explicit. This approach uses behavioral profiles to identify changed regions.

Mahfouz et al. [51] present an approach towards the customization of interactions in choreographies. It adopts TROPOS [52] to represent organizational business requirements. A new business requirement of a partner leads to a customization of the choreography model, which, in turn, results in a customization of the public models of these partners. Though [51] describes the general conceptual approach of propagation, it is unclear how the affected regions and changes to be propagated are determined. Finally, neither the change patterns nor the transitivity effects to be handled are discussed in [51].

In [53], an approach for aligning and propagating changes between the business process model and the corresponding service-component configuration model (SCA) is presented. The purpose of this approach is different from ours since it does not consider the change propagation between different process models, but between the business logic and its supporting software architecture logic instead (i.e., its implementation).

Recently, approaches started to analyze propagation effects when applying changes in process choreographies. As after a propagation changes cannot be imposed on affected partners, but are often subject to negotiations, propagation failures might become expensive. First steps towards the understanding of the ripple effects of change propagation in choreographies are taken in [10,11], where [10] operates on the choreography model structure and [11] on change log information by applying memetic mining.

Solutions to check the realizability of the choreography in case a specific reconfiguration or a change is needed are described in [54]. In particular, this allows avoiding changes that affect the realizability of the choreography. For this purpose, choreography models are translated into the FSP process algebra. In [55,56], an approach to model and validate *compliant choreographies* is presented, and techniques to check *compliance rules* in the context of process choreographies are defined. Note that these approaches, combined with change propagation algorithms, can be complementary to our work for ensuring sound propagations.

9. Summary and outlook

While business process management has reached a mature level in respect to enterprise-wide processes, the operational support of cross-organizational processes still constitutes a big challenge. In many application domains, however, any technology support will not be accepted if it is unable to cope with process changes and the evolutionary nature of business processes. This was confirmed in several case studies we conducted in the automotive domain (e.g., cross-organizational processes for product change management [61] and product release management [62]) as well as in the healthcare domain (e.g., cross-organizational processes coordinating the various healthcare partners involved in the preparation and enactment of a complex surgery [63]). Nevertheless, these case studies have also revealed the high need for a flexible support of cross-organizational processes.

This paper provides algorithms for propagating process changes in collaborative scenarios that involve multiple partners. In order to stay independent from a particular process specification language, RPST is used for defining public and private models of the involved partners as well as the choreography model. The proposed propagation algorithms consider typical process change patterns such as INSERT, DELETE, and REPLACE, and are evaluated based on their structural as well as behavioral compatibility.

Certain assumptions are made in this paper. First, the proposed algorithms consider the application of one change operation at a time. However, in practical scenarios, several change operations might be applied in a combined manner within a change transaction. To incorporate such complex changes, optimizations on the change transactions as suggested in [57] may be utilized. These allow calculating the actual effects of the change transaction. Second, change propagation might become necessary in a transitive way, i.e., multiple partners might be affected. This can be handled by applying the change propagation procedure depicted in Fig. 6 iteratively. However, it must be considered whether the transitive propagation becomes cyclic. In this case, mechanisms such as upper bounds on the number of iterations of propagating changes including rollback mechanisms are conceivable.

Currently, we are integrating the change propagation algorithms proposed into our cloud-based process execution engine CPEE. Further, we aim to test and apply these algorithms in future case studies with our partners from the automotive domain. As future work, we will deal with negotiation and public-to-private change propagation issues as well. Although our approach is able to determine the effects changes of a private model have on the public models of the involved partners, the dynamic effects on the running instances have not been considered yet. Therefore, as a next step, we aim to explore the effects of dynamic changes in the context of choreographies, as well as their impact on already running instances. Furthermore, the presented approach mainly deals with structural changes of choreographies and the resulting effects. However, it will be also interesting to extend the choreography models with data semantics (e.g., an ontology of the used data objects) to better cope with the transitive effects of changes. Finally, choreography version management as well as semantic constraints for choreography changes (i.e., to preserve global compliance rules in the context of changes [55]) will be investigated.

Appendix A. Propagation of fragment deletions

This appendix describes Algorithm 1, which shows the steps required to determine the effects of a change request of type DELETE has on a choreography.

1. Isolated or propagating changes (*cf. Lines* 4–5 *in Algorithm* 1).
2. Private-to-Public effects (*cf. Line* 9 *in Algorithm* 1).
3. Public-to-Public effects (*cf. Lines* 10–16 *in Algorithm* 1).

Algorithm 1. Delete operation propagation: $Delete_{Propag_{\pi_p}}(\mathcal{F})$.

```
1  Input: - A Choreography C
2         - The fragment F∈π_p to be deleted
3  begin
4  | if (abstr_(interaction)(F)≠∅) then
5  | |  update π_p          //      local change– no propagation is needed
6  | else
7  | |  Δ←∅                 //      decomposition result
8  | |  P_Δ← List of all business partners involved in F
9  | |  F''←ψ∘abstr_(interaction)(F)
10 | |  for each p'∈P_Δ do
11 | |  |  F_p'←abstr_p'(F'')
12 | |  |  for each n∈F_p' do
13 | |  |  |Δ←Δ∧(Delete(φ(n)), l_p')
14 | |  |  end
15 | |  end
16 |    Output: Δ
17
18 end
```

Appendix B. Propagation of fragment replacements This appendix describes Algorithm 2, which shows the steps to determine the propagation effects of a fragment replacement in choreographies.

1. Isolated or propagating changes (*cf. Lines* 7–8 *in Algorithm* 2).
2. Private-to-Public effects (*cf. Lines* 10–11 *in Algorithm* 2).
3. Public-to-Public effects (*cf. Lines* 12–33 *in Algorithm* 2).

Algorithm 2. Replace operation propagation: $Replace\_\ Propag_{\pi_p}(\mathcal{F},\ \mathcal{F}')$.

```
1  Input: - Choreography C
2         - The old fragment to be replaced F ∈ π_p
3         - The new fragment F'
4  begin
5    Δ ← ∅          //     decomposition result
6    P_Δ ← List of all partners involved in F ∪ F'
7    if (F = ∅ and F' = ∅) then
8    |  invariant change– no propagation needed
9    else
10   | F_1 ← ψ ∘ abstr_(interaction)(F)
11   | F_2 ← ψ ∘ abstr_(interaction)(F')
12   | for each (partner p' ∈ P_Δ) do
13   |  if abstr_p'(F_1) ≠ abstr_p'(F_2) then
14   |   if abstr_p'(F) = ∅ then
15   |   | /* Insertion Scenario: call for insert propagation algorithm*/
16   |   | Δ ← Δ ∧ Insert_Propag_π_p(abstr_p'(F'), Preset(F), Postset(F))
17   |   else
18   |   | if abstr_p'(F') = ∅ then
19   |   |   | /* Deletion Scenario: call for delete propagation algorithm*/
20   |   |   | Δ ← Δ ∧ Delete_Propag_π_p(abstr_p'(F))
21   |   |   else
22   |   |   | /* Replacement Scenario*/
23   |   |   | F'_1 ← abstr_p'(F_1)
24   |   |   | F'_2 ← abstr_p'(F_2)
25   |   |   | F_p' ← γ(α_l_p'(φ(F'_1)), φ(F'_2))/* γ is a merge function*/
26   |   |   | Δ ← Δ ∧ REPLACE(F'_1, F_p', l_p')
27   |   |   | for each (activity a ∈ F'_1 such that a ∉ F') do
28   |   |   | |Δ ← Δ ∧ DELETE_p(a)/* model refactoring may be applied*/
29   |   |   | end
30   |   |   end
31   |   | end
32   |  end
33   | end
34   end
35   Output Δ;
36 end
```

Appendix C. Propagation of fragment insertions

In this appendix, Algorithm 3 is presented, which summarizes the steps required for propagating a change operation of type INSERT to a particular process partner.

1. Isolated or propagating changes(*cf. Lines* 5–6 *in Algorithm* 3):
2. Private-to-Public effects (*cf. Lines* 11–20 *in Algorithm* 3):
3. Public-to-Public effects (*cf. Lines* 23–46 *in Algorithm* 3):

Algorithm 3. Insert operation propagation: $\text{Insert\_Propag}_{\pi_p}(\mathcal{F}, \text{pred}, \text{succ})$.

```
1 Input:  - A Choreography C (cf. Definition 4)
2         - The fragment F to be inserted in π_p
3         - The insertion position pred and succ ∈ π_p
```

4 begin
5 **if** $(abstr_{(interaction)}(\mathcal{F}) \neq \emptyset)$ **then**
6 update π_p // local change–no propagation is needed
7 **else**
8 $\Delta \leftarrow \emptyset$ // The set of changes to be propagated
9 $\mathcal{P}_\Delta \leftarrow$ {Set of all business partners involved in $\mathcal{F}$}
10 /*Calculating the insertion position in the public model of the change initiator*/
11 **if** $\psi(pred) \neq \emptyset$ **then**
12 $pred' \leftarrow \psi(pred)$
13 **else**
14 $pred' \leftarrow \{\psi(n)/n \in T\_preset_{(interaction)}(pred)\}$
15 **end**
16 **if** $\psi(succ) \neq \emptyset$ **then**
17 $succ' \leftarrow \psi(succ)$
18 **else**
19 $succ' \leftarrow \{\psi(n)/n \in T\_postset_{(interaction)}(succ)$
20 **end**
21 $\Delta \leftarrow \Delta \wedge (Insert(\psi \circ abstr_{(interaction)}(\mathcal{F}), pred', succ'), l_p)$
22 /*Calculating the insertion position in the public model of each partner involved in the change*/
23 **for** *each* $p' \in \mathcal{P}_\Delta$ **do**
24 **for** *each* $n \in pred'$ **do**
25 **if** $\varphi(n) \neq \emptyset$ **then**
26 $pred'' \leftarrow pred'' \cup \{\xi(n)\}$
27 **else**
28 $pred'' \leftarrow pred'' \cup \{\xi(n')/n' \in T\_preset_{p'}(n)\}$
29 **end**
30 **end**
31 **for** *each* $n \in succ'$ **do**
32 **if** $\varphi(n) \neq \emptyset$ **then**
33 $succ'' \leftarrow succ'' \cup \{\xi(n)\}$
34 **else**
35 $succ'' \leftarrow succ'' \cup \{\xi(n')/n' \in T\_postset_{p'}(n)\}$
36 **end**
37 **end**
38 $\mathcal{F}_{p'} \leftarrow \varphi \circ abstr_{p'} \circ \psi \circ abstr_{(interaction)}(\mathcal{F})$
39 $posIn \leftarrow \xi(pred'')$
40 $posOut \leftarrow \xi(succ'')$
41 **if**($\exists n \in \mathcal{L}_{p'}$ *between posIn and posOut*) **then**
42 $\Delta \leftarrow \Delta \wedge (Insert(\overline{\mathcal{F}}_{p'}, Parallel, posIn, posOut), l_p)$
43 **else**
44 $\Delta \leftarrow \Delta \wedge (Insert(\overline{\mathcal{F}}_{p'}, Sequence, posIn, posOut), l_p)$
45 **end**
46 **end**
47 **end**
48 **Output**: Δ/***List of changes to be propagated***/
49 end

References

[1] S. Schulte, D. Schuller, R. Steinmetz, S. Abels, Plug-and-play virtual factories, IEEE Internet Comput. 16 (5) (2012) 78–82.
[2] F.M. Besson, P.M. Leal, F. Kon, Towards Verification and Validation of Choreographies. Technical Report, Department of Computer Science, University of Sao Paulo, 2011.
[3] C. Peltz, Web services orchestration and choreography, Computer 36 (10) (2003) 46–52.
[4] W. van der Aalst, A decade of business process management conferences: personal reflections on a developing discipline, in: International Conference on Business Process Management, 2012, pp. 1–16.
[5] F. Casati, S. Ceri, B. Pernici, G. Pozzi, Workflow evolution, Data Knowl. Eng. 24 (3) (1998) 211–238.
[6] S. Rinderle, M. Reichert, P. Dadam, Correctness criteria for dynamic changes in workflow systems: a survey, Data Knowl. Eng. 50 (1) (2004) 9–34.
[7] R. Breu, et al., Towards living inter-organizational processes, in: IEEE International Conference on Business Informatics, 2013, pp. 363–366.
[8] W. Fdhila, S. Rinderle-Ma, M. Reichert, Change propagation in collaborative processes scenarios, in: International Conference on Collaborative Computing: Networking, Applications and Worksharing, 2012, pp. 452–461.
[9] S. Rinderle, A. Wombacher, M. Reichert, Evolution of process choreographies in DYCHOR, in: International Conference on Cooperative Information Systems, 2006, pp. 273–290.
[10] W. Fdhila, S. Rinderle-Ma, Predicting change propagation impacts in collaborative business processes, in: ACM Symposium on Applied Computing (SAC), 2014, pp. 1378–1385.
[11] W. Fdhila, S. Rinderle-Ma, C. Indiono, Memetic algorithms for mining change logs in process choreographies, in: International Conference on Service-Oriented Computing (ICSOC), 2014, pp. 47–62.
[12] J. Vanhatalo, H. Völzer, J. Koehler, The refined process structure tree, Bus. Process Manag. (2008) 100–115.
[13] B. Weber, M. Reichert, S. Rinderle-Ma, Change patterns and change support features—enhancing flexibility in process-aware information systems, Data Knowl. Eng. 66 (3) (2008) 438–466.
[14] Signavio: Signavio process editor. ⟨http://academic.signavio.com/⟩ Accessed 2013-10-24.

[15] G. Decker, M. Weske, Behavioral consistency for B2B process integration, in: International Conference on Advanced Information Systems Engineering, 2007, pp. 81–95.
[16] A. Barros, M. Dumas, P. Oaks, Standards for web service choreography and orchestration: status and perspectives, Bus. Process Manag. Workshops (2006) 61–74.
[17] C.C. Ekanayake, M. Dumas, L. García-Bañuelos, M.L. Rosa, Slice, mine and dice: complexity-aware automated discovery of business process models, in: International Conference on Business Process Management, 2013, pp. 49–64.
[18] A. Polyvyanyy, L. Garcia-Banuelos, M. Dumas, Structuring acyclic process models, Inf. Syst. 37 (6) (2012) 518–538.
[19] R. Milner, M. Tofte, D. Macqueen, The Definition of Standard ML, MIT Press, Cambridge, MA, USA, 1997.
[20] B. van Dongen, W. van der Aalst, H. Verbeek, Verification of EPCs: using reduction rules and petri nets, in: International Conference on Advanced Information Systems Engineering, 2005, pp. 372–386.
[21] W. Fdhila, U. Yildiz, C. Godart, A flexible approach for automatic process decentralization using dependency tables, in: International Conference on Web Services, 2009, pp. 847–855.
[22] M.L. Rosa, M. Dumas, R. Uba, R.M. Dijkman, Business process model merging: an approach to business process consolidation, in: ACM Transactions on Software Engineering and Methodology, 2012.
[23] F. Gottschalk, W. van der Aalst, M.H. Jansen-Vullers, Merging event-driven process chains, On the Move to Meaningful Internet Systems (2008) 418–426.
[24] F. Puhlmann, M. Weske, Interaction soundness for service orchestrations, in: International Conference on Service-Oriented Computing, 2006, pp. 302–313.
[25] W. Fdhila, M. Rouached, C. Godart, Communications semantics for WSBPEL processes, in: International Conference on Web Services, 2008, pp. 185–194.
[26] H. Foster, S. Uchitel, J. Magee, J. Kramer, Compatibility verification for web service choreography, in: International Conference on Web Services, 2004, pp. 738–741.
[27] M. Rouached, W. Fdhila, C. Godart, Web services compositions modelling and choreographies analysis, Int. J. Web Service Res. 7 (2) (2010) 87–110.
[28] M. Rouached, W. Fdhila, C. Godart, A semantical framework to engineering WSBPEL processes, Inf. Syst. e-Bus. Manag. 7 (2) (2009) 223–250.
[29] W. van der Aalst, M. Weske, The P2P approach to interorganizational workflows, in: International Conference on Advanced Information Systems Engineering, 2001, pp. 140–156.
[30] W. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, K. Wolf, From public views to private views—correctness-by-design for services, Web Services Formal Methods (2008) 139–153.
[31] S.A. Bohner, R.S. Arnold, Software Change Impact Analysis, IEEE Computer Society, 1996.
[32] M. Giffin, O. de Weck, G. Bounova, R. Keller, C. Eckert, P.J. Clarkson, Change propagation analysis in complex technical systems, J. Mech. Des. 131 (8) (2009).
[33] G.A. Oliva, G. de Maio Nogueira, L.F. Leite, M.A. Gerosa, Choreography Dynamic Adaptation Prototype, Technical Report, Universidade de Sao Paulo, 2012.
[34] P.J. Clarkson, C. Simons, C. Eckert, Predicting change propagation in complex design, J. Mech. Des. 126 (5) (2004) 788–797.
[35] B. Weber, S. Rinderle-Ma, M. Reichert, Change patterns and change support features in process-aware information systems, in: International Conference on Advanced Information Systems Engineering, 2007, pp. 574–588.
[36] L. Steffen, A Review of Software Change Impact Analysis, Technical Report, Universitätsbibliothek Ilmenau, 2011.
[37] C.M. Eckert, R. Keller, C. Earl, P.J. Clarkson, Supporting change processes in design: complexity, prediction and reliability, Reliab. Eng. Syst. Saf. 91 (12) (2006) 1521–1534.
[38] C. Eckert, W. Zanker, P.J. Clarkson, Aspects of a Better Understanding of Changes, vol. 1, ICED, 2001.
[39] W. van der Aalst, T. Basten, Inheritance of workflows: an approach to tackling problems related to change, Theor. Comput. Sci. 270 (1–2) (2002) 125–203.
[40] W. Fdhila, A. Baouab, K. Dahman, C. Godart, O. Perrin, F. Charoy, Change propagation in decentralized composite web services, in: International Conference on Collaborative Computing, 2011, pp. 508–511.
[41] W. Fdhila, S. Rinderle-Ma, A. Baouab, O. Perrin, C. Godart, On evolving partitioned web service orchestrations, in: International Conference on Service-Oriented Computing and Applications, 2012, pp. 1–6.
[42] M. Reichert, T. Bauer, Supporting ad-hoc changes in distributed workflow management systems, in: International Conference on Cooperative Information Systems, 2007, pp. 150–168.
[43] P. Hens, M. Snoeck, M. De Backer, G. Poels, Verification of change in a fragmented event-based process coordination environment, IEEE Trans. Services Comput. 7 (3) (2014) 501–514.
[44] J.M. Küster, C. Gerth, G. Engels, Dynamic computation of change operations in version management of business process models, Model. Found. Appl. (2010) 201–216.
[45] V. Atluri, S.A. Chun, Handling dynamic changes in decentralized workflow execution environments, Database Expert Syst. Appl. (2003) 813–825.
[46] M. Weidlich, J. Mendling, M. Weske, Propagating changes between aligned process models, J. Syst. Softw. 85 (8) (2012) 1885–1898.
[47] K.H. Dam, M. Winikoff, Cost-based BDI plan selection for change propagation, in: International Joint Conference on Autonomous Agents and Multiagent Systems, 2008, pp. 217–224.
[48] H.K. Dam, A. Ghose, Supporting change propagation in the maintenance and evolution of service-oriented architectures, in: Asia Pacific Software Engineering Conference, 2010, pp. 156–165.
[49] S. Mafazi, G. Grossmann, W. Mayer, M. Stumptner, On-the-fly change propagation for the co-evolution of business processes, On the Move to Meaningful Internet Systems (2013) 75–93.
[50] J. Kolb, K. Kammerer, M. Reichert, Updatable process views for user-centered adaption of large process models, in: International Conference on Service Oriented Computing, 2012, pp. 484–498.
[51] A. Mahfouz, L. Barroca, R. Laney, B. Nuseibeh, Requirements-driven collaborative choreography customization, in: International Conference on Service-Oriented Computing, 2009, pp. 144–158.
[52] A.U. Mallya, M.P. Singh, Incorporating commitment protocols into tropos, Agent-Oriented Software Engineering VI (2006) 69–80.
[53] K. Dahman, F. Charoy, C. Godart, Alignment and change propagation between business processes and service-oriented architectures, in: International Conference on Service Computing, 2013, pp. 168–175.
[54] N. Roohi, G. Salaün, V. France, Realizability and dynamic reconfiguration of chor specifications, Informatica 35 (1) (2011) 39–49.
[55] D. Knuplesch, M. Reichert, R. Pryss, W. Fdhila, S. Rinderle-Ma, Ensuring compliance of distributed and collaborative workflows, in: International Conference on Collaborative Computing, 2013, pp. 133–142.
[56] D. Knuplesch, M. Reichert, W. Fdhila, S. Rinderle-Ma, On enabling compliance of cross-organizational business processes, in: International Conference on Business Process Management, 2013, pp. 146–154.
[57] S. Rinderle, M. Reichert, M. Jurisch, U. Kreher, On representing, purging, and utilizing change logs in process management systems, in: International Conference on Business Process Management, 2006, pp. 241–256.
[58] R. Lenz, M. Reichert, IT support for healthcare processes—premises, challenges, perspectives, Data Knowl. Eng. 61 (1) (2007) 39–58.
[59] S. Paurobally, V. Tamma, M. Wooldrdige, A framework for web service negotiation, ACM Trans. Auton. Adap. Syst. 2 (4) (2007) 14.
[60] R. Vigne, J. Mangler, E. Schikuta, S. Rinderle-Ma, WS-agreement based service negotiation in a heterogeneous service environment, in: International Conference on Service-Oriented Computer and Applications, 2012, pp. 1–8.
[61] VDA Recommendation 4965 T1: Engineering Change Management (ECM)—Part 1: Engineering Change Request (ECR) Version 1.1, 2005.
[62] D. Müller, J. Herbst, M. Hammori, M. Reichert, IT support for release management processes in the automotive industry, in: International Conference on Business Process Management, 2006, pp. 368–377.
[63] B. Schultheiss, J. Meyer, R. Mangold, T. Zemmler, M. Reichert, Designing the Processes for Ovarian Cancer Surgery, Technical Report DBIS-6, University of Ulm, 1996.