Generation and Transformation of Compliant Process Collaboration Models to BPMN

Frederik Bischoff[1], Walid Fdhila[2], and Stefanie Rinderle-Ma[3]

[1] Cronn GmbH, Germany, [2] SBA-Research, Austria, [3] University of Vienna, Austria

Abstract. Collaboration is a key factor to successful businesses. To face massive competition in which SMEs compete with well established corporates, organizations tend to focus on their core businesses while delegating other tasks to their partners. Lately, Blockchain technology has yet furthered and eased the way companies collaborate in a trust-less environment. As such, interest in researching process collaborations models and techniques has been growing. However, in contrast to BPM research for intra-organizational processes, where a multitude of process models repositories exist as a support for simulation and work evaluation, the lack of such repositories in the context of inter-organizational processes has become an inconvenience. The aim of this paper is to build a repository of collaborative process models that will assist the research in this area. A top-down approach is used to automatically generate constrained and compliant choreography models, from which public and private process models are derived. Though the generation is partly random, it complies to a predefined set of compliance rules and parameters specified by the user.

Keywords: Process Collaboration · Process Models · Compliance Rules.

1 Introduction

Digitalization, blockchain and Industry 4.0 have created an environment in which organizations cooperate with more ease and efficiency. This open environment enabled Small and Medium Enterprises (SMEs) to collaborate more efficiently and compete with more established organizations. Therefore, research in process collaborations has become primordial [9]. Academic and industrial research include standards, infrastructure, and solutions that range from management and modeling of process collaborations, to enabling monitoring and improving security, compliance and privacy in such constellation [7]. However, in contrast to Business Process Management (BPM) research for intra-organizational processes where repositories of process models exist to support simulation and evaluation of research techniques [13], such repositories (synthetic and real-world models) are entirely missing in the context of inter-organizational processes.

This paper provides a parametric framework to build a repository of process collaboration models which would serve for testing and evaluating research approaches. As real world models are hard to obtain due to privacy issues, this framework generates synthetic models, whose execution can result in distributed logs of synthetic data useful for mining techniques. The generation must ensure the consistency, compatibility and compliability of such models [7]. A set of compliance rules that follow specific patterns could be specified along with a set of parameters regarding the number or type of tasks or gateways per model. The approach follows a top-dow approach where a compliable choreography model is generated, from which public and private processes are derived. Such models are internally represented as Refined Process Structure Tree (RPST) [17], which are transformed into BPMN models to ensure their executability. The resulted repository could be used to support research simulation such as change propagation [6], compliance checking [7] or mining [1] of collaborative processes.

The paper is structured as follows. In Section 2 , fundamentals of process collaborations are presented. Section 3 elaborates the conceptual approach that is implemented in Section 4. In Section 5 related approaches are discussed in Section 6 the paper concludes.

2 Fundamentals

In an inter-organization setting, partners combine their core businesses to provide an added value service, dynamically or statically, at runtime or design time respectively [14]. Process collaborations comprise different but overlapping models [6]. A **private model** describes the internal logic of a partner including its *private activities* and *interactions*. *Private activities* are tasks that are not visible to other participants. In the private model of Figure 1(a), *Check Inventory* is a private activity, whereas *Receive Order* involves message exchange with another participant and therefore is an interaction. A **public model** is a restricted view on the private model, and shows all *interactions* of one single partner. Private activities which are not relevant for other partners are omitted deliberately. Public activities might also be non-interaction tasks made visible to partners [5, 4]. In this paper, public activities are solely interactions but can easily be extended through model enriching. Figure 1(b) presents the public model of the *distributor* process. A **collaboration model** is the interconnection of all participants public models. In BPMN, participants are represented as pools that contain their corresponding public models and the message exchanges as arrows that connect them. Figure 1(c) shows an example of a collaboration model. A **Choreography model** represents a high level view on the sequencing of all interactions between the involved partners. Each message exchange is represented as an interaction (i.e, choreography activity) with an initiating partner, a receiving partner (shaded in grey) and the message exchanged (c.f. Figure 1(d)).

In collaborative processes, there are different level of correctness within and across the models [11, 7]. **Consistency** means that the private model is consistent with its corresponding public model. **Compatibility** involves the collaboration model and ensures that the public models are compatible with each others. This means that there exist no flaws in the communication between participants (e.g., deadlocks, livelocks). While behavioral compatibility focuses on the correctness of behavioral dependencies between participants (i.e., control flow), structural compatibility requires that for every message that may be sent, the corresponding participant is able to receive it. We also distinguish between

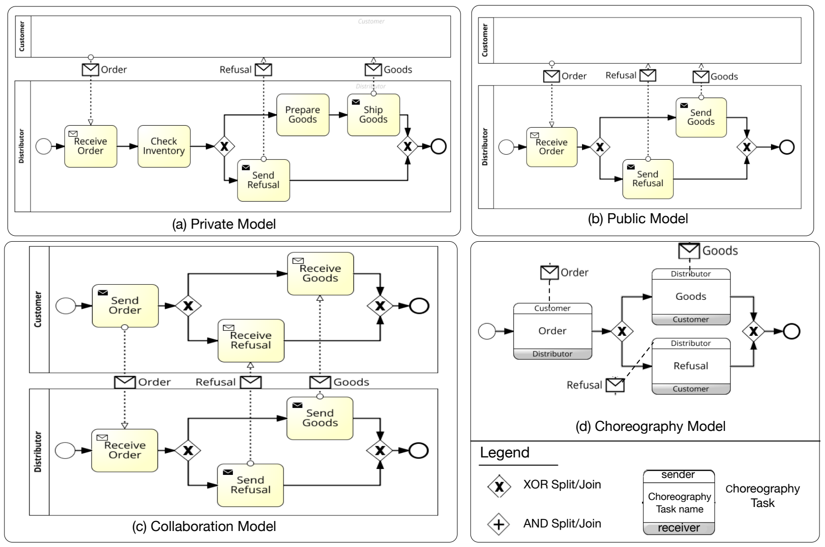

Fig. 1. Choreography Model Example

three types of compliance: (i) global compliance rules (GCR) that constraint the choreography model, (ii) local compliance rules (LCR), which constraint the private model of a particular participant but not visible to other partners, and finally (iii) assertions, which are agreements between two or more partners, where a partner guarantees that its private/public process complies with the constraint[7]. **Compliability** ensures that a choreography model does not conflict with GCRs [7]. As the aim of the paper is to produce collaborative models that comply with pre-specified GCRs, then the generation should ensure the correctness of the resulted models with respect to the aforementioned aspects. Compliance patterns supported in this work will be discussed in section 3.1.

3 Model Generation and transformation to BPMN

3.1 Parametrization and compliance specification

The process collaboration generator conceptualized in this work generates all four different model types and follows a *top-down approach* [7]. In a *top-down approach* (cf. Figure 2), first the choreography model is build, then the public and private models of each partner are derived and defined consistently. Thereby, each interaction (choreography task) of the choreography model is converted into a send and receive tasks in the corresponding public process models. In turn, each private model is derived from its corresponding public model by enriching the latter with abstract private tasks. The collaboration model is build by interconnecting all public models. As depicted in Figure 2, during model generation, all the aforementioned correctness criteria are considered. In particular, the approach ensures that only model specific flow objects are used to build the processes and that they are connected appropriately (structural compatibility). It also guarantees the absence of deadlocks and livelocks (behavioral compatibility)

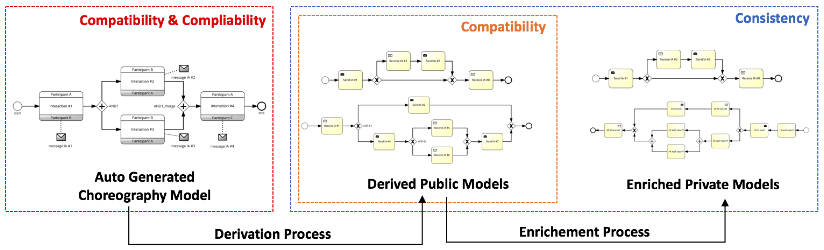

Fig. 2. Top-Down Approach

and offers the possibility to define global compliance rules GCR, to which the generated collaboration should comply (compliability). Deriving public models from a choreography model offers the advantage that if the latter is implemented correctly, the *compatibility* of the derived public models is automatically ensured.

Constraining the Collaboration. Despite the premise that the process collaborations should be generated randomly, it is reasonable to set some boundaries within which the random generation takes place. The implemented generator provides two different ways to influence the resulting choreography model and hence the whole collaboration. The first one provides the possibility to constrain the choreography model in terms of the employed flow objects and their exact quantity by specifying several input parameters. The second one enables the user to impose global compliance rules based on compliance patterns to which the resulting model must comply.

Parametric Constraints. The following input parameters are specified to influence the random generation of the choreography model and subsequently the derived models:

- Number of partners: determines the number of collaboration participants.
- Number of interactions: determines the number of messages exchanges.
- Number of exclusive gateways per model.
- Number of parallel gateways per model.
- Maximum branching: determines the maximum possible number of paths created for each gateway.

Compliance Constraints. To specify GCRs, the pattern-based approach is utilized [16]. In [16], a repository of *process control patterns* is introduced, which are high-level templates used to represent process properties which the process specification must satisfy. In this work, only compliance rules that constrain the sequence and occurrence of interactions are considered. Compliance patterns that involve data, time and resource perspectives are future work. Table 1 summarizes the supported compliance patterns. Note that the *P LeadsTo Q* pattern does not imply immediate succession of interaction Q to interaction P.

3.2 Process Collaboration Generation

The generation of compliant collaboration processes follows the principle *'first build then check'*, which means that after a random choreography model has been generated, it will then be checked whether the interactions defined within

Pattern	Description
P LeadsTo Q	Interaction P must lead to Interaction Q.
P Precedes Q	Interaction Q must be preceded by Interaction P.
P Universal	Interaction P must always occur throughout execution.
P Exists	Interaction P must be specified in process.

Table 1. Overview of supported Compliance Patterns

the compliance rules can be assigned to the already built model in such a way that the resulting interaction sequence complies to the imposed rules. If the interaction allocation is not possible without violating the compliance rules, new random models will be build until a compliant model has been generated. If the checking of the compliance rules fails repeatedly, it's an indicator that the amount of interactions in the model is too small in comparison to those specified within the compliance rules. To overcome this, the number of interactions might be increased by the user. After a successful assignment of the compliance rules, the remaining public and private models are derived out of the generated choreography model. At last, all models will be translated into a valid BMPN/XML. Algorithm 1 illustrates the generation process of process collaborations, with all major steps explained in the following subsections.

Algorithm 1: Overall Collaboration Generation Controller

```
buildSuccess = false;
while buildSuccess ≠ true do
    Generate Random Choreography Model;
    if compliance rules are defined then
        Compliance Rules Assignment;
        if assignment successful then
            buildSuccess = true;
        else if number of interaction mod increase_percentage ≡ 0 then
            increase number of interactions by factor increase_factor;
        end
    else
        buildSuccess = true;
    end
end
Derive Public and Private Models;
Transform Models to BPMN;
```

Random Choreography Model Generation Throughout the generation process, it is essential to keep track of the current model state at every point to ensure structural and behavioral correctness. Therefore a *Model Tracking* component is necessary, which provides control flow logic and a corresponding data model, and uses graph decomposition concept using RPST [17] to ensure structured process models. In [17], a parsing algorithm for *two-terminal graphs*[1] is introduced that results in a unique graph decomposition represented as a hierarchical tree of *modular* and *objective fragments*. In the *Model Tracking* component of this work, the equivalent of a modular and objective fragment is called a *split*. A split is created for each gateway fork that is put into the model. Each split contains several *branches* that represent the different paths created by a parallel or exclusive gateway. Again, each branch holds the set of nodes that are on the

[1] A directed graph that has a unique source node s and a unique sink node $t \neq s$ with all other nodes V are on a path from s to t.

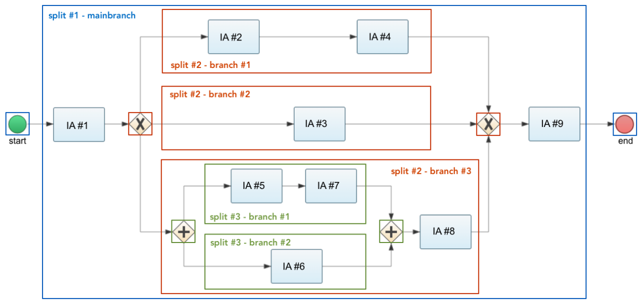

Fig. 3. Model Tracking Component - Concept

path of a particular branch, whereas a path and therefore a split is limited by the merge node of its corresponding fork gateway node. In terms of a choreography model, nodes are limited to interactions and gateways. Figure 3 illustrates the concept of the *Model Tracking* component. There exist three splits with the split nodes: start event (blue), exclusive gateway#1 (red) and parallel gateway #1 (green). The split with the start event as the split node and the end event as the merge node has always only one branch, the root branch. Technically, this is not a split in the sense of the terminology. But because of the underlying control flow logic and data model that defines that every branch must be related to a split, this pseudo-split is necessary to keep track of the root branch. Note that branches and therefore also splits, contain other splits, e.g. split #2 contains split #3 in branch #3. Additionally, each branch has a status, which indicates whether the branch is *open*, *split* or *closed*. *Open* defines that the branch is not yet enclosed by the merge node of its corresponding split node and can further evolve by putting more nodes on its path. *Closed* means that the branch is finalized and can not further evolve. Within the *Model Tracking* component, a branch gets closed by putting the corresponding gateway merge node to the parent's branch and marking the branch as closed. A branch can also be in *split* state if it contains another split and none of this split branches are yet closed, thus there exists no merge node for this split. In this case, a branch cannot evolve until one of it's child split's branches is in state *closed* and a merge node is placed on the branch. Then, the state changes to *open*. When closing a branch, first it is necessary to determine if a branch is allowed to be closed without violating the correctness of the choreography models. This depends on the split node type of the branch. The premise is that if the split node type is a parallel gateway, the branch is determined as *closable* only if there is an interaction on all its enclosed paths. This means, that if a branch has a child split, it does not necessary imply that an interaction is on the parent branch, but might be on the branches of its child split or even on a deeper nested branch.

Tracking branches status is crucial for satisfying structural and behavioral correctness. Because of the parametric limitation on the number of interactions in the build process or even directly at the beginning, interactions are not always allowed to be selected as next node type. Similarly, not every open branch is allowed to be randomly selected for putting the next node into the model as this might violate the correctness of the model or exceeding the number of defined interactions. In order to determine whether this situation applies to a current build state, the *Model Tracking* component monitors the amount of *free* and *reserved interactions*. Reserved interactions are a subset of the remaining interaction that have either predetermined positions in the current model (resInteractionBranches) or will be needed in further paths created by not yet employed gateways (resInteractionsGateways). The exact amount of these reserved interactions depends on the number of non-closable branches of the current model and the number of gateways that are not yet put into model. Free interactions are interactions which are not yet used nor reserved for the model.

Definition 1. *Let x be the number of branches which are open and non-closable, $remainingInteractions$ be all interactions not yet put into the model and $remainingXOR$ and $remainingAND$ the number of gateways not yet put into the model. Then,* $\textbf{\textit{resInteractionBranches}} = x$

$$\textbf{\textit{resInteractionAndGateways}} = remainingAND + 1$$
$$\textbf{\textit{resInteractionXORGateways}} = \textbf{\textit{If}}\ remainingAND > 0\ \textbf{\textit{Then}}\ 0\ \textbf{\textit{Else}}\ 1$$
$$\textbf{\textit{resInteractionGateways}} = resInteractionAndGateways + resInteractionXORGateways$$
$$\textbf{\textit{resInteractionsTotal}} = resInteractionBranches + resInteractionGateways$$
$$\textbf{\textit{freeInteractions}} = remainingInteractions - resInteractionsTotal$$

Regarding the current model, each open and non closable branch increases the amount of *resInteractionBranches* by one. Parallel gateways that are not yet placed into the model will later create at least two new branches, which then again need at least one interaction on each of it's paths. Considering that a gateway node is allowed to be immediately followed by another gateway node without an interaction in between, the minimum amount of *resInteractionAndGateways* is *remainingAndGateways* + 1. This premise also influences the impact of remaining exclusive gateways on the number of *resInteractionsGateways*. Each remaining exclusive gateways only increases the number of *resInteractionsGateways* by 1 if there is no more remaining parallel gateways. Indeed, if there exist a remaining parallel gateway, the exclusive gateway could be put on a branch of the parallel gateway directly after the split, and therefore the one needed interaction of the exclusive gateway is already considered in the calculation of *resInteractionAndGateways*. After the amount of *reserved interactions* is calculated, the number of *free interactions* is determined by the difference between the amount of *remaining interactions* and the number of *reserved interactions*. Based on the values of the specified variables defined in Definition 1, the node type of the next node to be put in the model and the corresponding position can be randomly selected without resulting in an incorrect model. For example, if the amount of *free interactions* is less than 1, the random branch selection (position in the model) for putting the next node is limited to the branches that are not yet closable. On the other hand, if the amount of *free interactions* is superior to 0, then all open branches can be selected for putting the next node. When selecting the next possible node type, interactions are only allowed to be randomly chosen if the amount of *free interactions* is superior to 0 or not all

Algorithm 2: Generate Choreography Model

```
begin
    while $remainingInteractions > 0$ do
        $nextNodeType$ ← getRandomNodeType()
        $selectedBranch$ ← getRandomBranch()
        if $selectedBranch$ is closable then
            close branch by random
            if $closed$ then
                continue
            end
        else
            $nextNode$ ← instantiate node of $nextNodeType$
            if $nextNodeType$ is Gateway then
                $branchCount$ ← getRandomBranchCount()
                $split$ ← instantiate new split
                for $i$ ← 0 to $branchCount$ do
                    $branch$ ← instantiate new branch
                    $split.branches \leftarrow branch$
                    $i \leftarrow i + 1$
                end
            end
            $selectedBranch.nodes \leftarrow selectedBranch.nodes \cup nextNode$
            decrease $remainingNodes$ of $nextNodeType$
        end
    end
    close still open splits
    add end event to root branch
    enrich interactions with reasonable sender and receiver sequence
end
```

remaining interactions are reserved by not yet consumed gateways.
The overall procedure for generating random choreography models is shown in Algorithm 2. Note that the step of random branch closing is necessary to obtain balanced choreography models with respect to nested branches. If there would be no random branch closing mechanism, the resulting models would be very similar. A mechanism that closes branches whenever they are closable would only result in models with lesser nested branches whereas a mechanism that never closes branches would result in models that have highly nested branching. By the time a branch is not randomly closed, a node of the predefined node type gets instantiated. In case of an interaction, only the plain object without any sender, receiver or message gets instantiated. Is the selected node type a gateway, the number of branches is determined by randomly selecting a number between 2 and the current maximum branching amount. The maximum branching amount is generally limited by the user specified max branching parameter. But again, due to the limitation of interactions, the specified maximum amount of branches can not be adducted as the upper border without considering the current amount of free interactions. The possible upper limit is determined dynamically each time a gateway node is put into the model by taking the minimum branching amount, which is always two, and adding the amount of free interactions.

After a random number of branches is determined, the gateway node is added to the assigned branch and the corresponding split and branches are instantiated within *Model Tracking*. Finally, the amount of the selected node type is decreased by one and the loop starts over by selecting a random node type for the next node to be put into the model. To achieve behavioral correctness in choreography models, beside a correct sequence flow, a message flow must be incorporated. Therefore, a sender and receiver must be assigned to each interaction in order to form a valid sender-receiver sequence. Thereby, the sender of a succeeding interaction Q must always be either the sender or receiver of the directly preceding interaction P on the path. If this rule is not considered and the sender of a directly succeeding interaction Q is neither the sending nor the receiving participant of the directly preceding interaction P, a flawless execution of the process is not possible, because the sender of interaction Q will never know if the directly preceding interaction P has been performed yet. For gateways, it is additionally ensured that all branches of that split terminate with interactions that have the same participant in common. This helps to determine a possible sender for the succeeding interaction after the merge. Note that because the sequence flow is first build without considering the corresponding message flow, it is likely that at some points, an additional interaction must be inserted into the model to satisfy the above stated rules of sender-receiver sequences.

Compliance Rules Assignment Instead of considering the imposed compliance rules during the generation, a *first build then check* approach was favored to allow users to specify compliance rules, which can be applied to existing choreography models to check if the latter complies to them. When specifying global compliance rules, it must be checked whether the imposed rules are consistent with one another. In the context of the four supported patterns, this applies only to the the patterns 'LeadsTo' and 'Precedes'. For instance, consider the following set of compliance rules: {**C1**: P LeadsTo Q, **C2**: Q LeadsTo S **C3**: S Precedes P}. In this example, the rules *C1* and *C2* conflict with *C3* because *C1* and *C2* imply that the involved activities must occur in the order P-Q-S, whereas in *C3*, S must occur before P. Algorithm 3 shows the conflict checking procedure. The result of this procedure is a set of conflict free compliance rules, which determines a specific order sequence between the involved interactions. The specific interactions of the compliance rules are then eventually assigned to the existing interactions within the previously generated model in a way that it complies to the interaction order and the compliance rules. Therefore, the first step is to determine all possible positions within the model for each compliance rule. The result is a set of possible position combinations (interactions placed in the model during initial choreography generation) for the compliance rule specified for Interactions P and Q. For each possible position of P there has to be at least one possible position for Q. The rules that determine applicable positions for the four implemented compliance patterns are shown in Definitions 2 - 5.

Definition 2. *Possible position assignments for the interactions P and Q of a compliance pattern P LeadsTo Q are as follows.*

- *Interaction P should have reachable interactions on its subsequent paths.*
- *Interaction Q should be reachable if Interaction P has been reached.*

Definition 3. *Possible position assignments for the interactions P and Q of a compliance pattern P Precedes Q are as follows.*

- *Interaction P is always reached prior to Interaction Q.*

Algorithm 3: Adding Compliance Rules

```
Input  : compliance rule $cr$
dictionary $orderDependencies$ of Interactions $P$ and their succeeding Interactions $S$
begin
    if $cr$ is order pattern then
        $p \leftarrow$ preceding interaction of $cr$
        $s \leftarrow$ succeeding interaction of $cr$
        if !orderConflictCheck($p, s$) then
            add $cr$ to $complianceRules$
            if $p \in P$ of $orderDependencies$ then
                add $s$ to succeeding interactions $S$ of $p$
            else
                add $p$ to $orderDependencies$
                add $s$ to succeeding interactions $S$ of $p$
            end
        else
            add $cr$ to $conflictedRules$
        end
    end
end
Function orderConflictCheck($p, s$)
    if $s \in P$ of $orderDependencies$ then
        foreach $s \in S$ of $p$ do
            if $s == p$ then
                return true
            else if orderConflictCheck($s, p$) then
                return true
        end
    else
        return false
    end
```

– *Interaction Q has interactions on its preceding path that are always reached prior to Interaction Q.*

Definition 4. *Possible position assignments for the interaction P of a compliance pattern P Universal are as follows.*

– *Interaction P = An interaction that will always be reached.*

Definition 5. *Possible position assignments for the interaction P of a compliance pattern P Exists are as follows.*

– *Interaction P = An interaction that can be reached.*

If the interactions used for specifying the rules are disjoint between all the compliance rules, the sets of assignment combinations are already sufficient to assign the involved interactions to positions that result in a model that is compliant with the opposed rules. But if there are particular interactions that are used in more than one compliance rule specification, the intersection of the interaction's possible assignments of all involved compliance rules represents the set of possible assignments for this particular interaction. The assignment procedure iterates over the interaction order and for each interaction, the intersection of the possible assignments of all affected compliance rules is calculated. Is the current interaction specified as the succeeding interaction of an affected order compliance rule, the possible model positions of this rule are limited to the succeeding model positions of the corresponding, already assigned, preceding interaction. Is the resulting intersection of the sets of possible assignments empty, then there is no valid position in the model where the interaction could be assigned to. In this case, the whole assignment process fails and results in a failed choreography build process. Is the intersection of possible model positions not empty, the procedure choses the interaction that has the most interactions on it's succeeding path.This ensures, that the assignment process does not fail because of higher ranked interactions being assigned to positions at the end of the model, so that there are no valid positions left for lower ranked ones.

Deriving the Collaboration Models In the process of deriving the models, each interaction of the choreography model results in a send and receive task in the corresponding public models of the involved partners. For an interaction, in the initiating and receiving participant public models, a send task and a corresponding receiving tasks are inserted respectively. Additionally, for each public model, a reduction of the model's sequence flow is enabled, without violating the choreography model sequence flow. Thereby, each gateway of the choreography model is checked for interactions within its subsequent paths involving the current participant. If there are none, the gateway and it's subsequent paths are not put into the public model of this participant. In order to derive the private models from the public models, the public models are randomly enriched with private tasks as well as some additional sequence flow elements (gateways) without violating the predefined sequence flow. The public models are used as a basis for private models, which then enriched with private tasks.

3.3 BPMN Transformation

In order to translate the RPST representation of the models to BPMN/XML, the internal model elements for events, tasks, gateways, edges and participants must be mapped to the corresponding BPMN elements of the different model types. Therefore the procedure loops recursively through all the graphs edges, extracts all the necessary information from the fragments (source, target) and generates the corresponding BPMN elements. In Algorithm 4, the procedure for transforming private and public models is outlined. As input serves the RPST and the internal collaboration representation, which includes necessary informations about the public task relationships. Collaboration, public and private models share the same XML structure, which is initialized in the first step (initialize BPMN XML collaboration document). In collaboration models all participant public models are described, whereas the public models only contain the described process of one participant and only a black box process for the others, which is necessary for referencing to public activities. When creating public activity elements (send / receive task), the necessary partner references are available in the internal collaboration model representation.

4 Implementation

The presented work was implemented and integrated within the C3Pro framework [2] [6]. The latter provides techniques for defining, propagating and negoti-

[2] Source code available at: http://gruppe.wst.univie.ac.at/c3pro/repo.zip

Algorithm 4: Transform Private and Public Model to BPMN

```
Input  : $edges \leftarrow$ edges of RPST
$collaboration \leftarrow$ internal collaboration representation
begin
    $xmlDoc \leftarrow$ initialize BPMN XML collaboration document
    foreach $edge \in edges$ do
        $nodes \leftarrow$ edge.getSource() $\wedge$ edge.getTarget()
        $sequenceFlows \leftarrow$ create new $sequenceFlow$ XML-Element for edge
        foreach $node \in nodes$ do
            if $node == SendTask$ then
                $processNodes \leftarrow$ create new $sendTask$ XML-Element
                $messages \leftarrow$ create new $message$ XML-Element
                $messageFlows \leftarrow$ create new $messageFlow$ XML-Element
            else if $node == ReceiveTask$ then
                $processNodes \leftarrow$ create new $receiveTask$ XML-Element
                $messages \leftarrow$ create new $message$ XML-Element
                $messageFlows \leftarrow$ create new $messageFlow$ XML-Element
            else if $node == PrivateActivity$ then
                $processNodes \leftarrow$ create new $task$ XML-Element
            else if $node == ParalellGateway$ then
                $processNodes \leftarrow$ create new $paralellGateway$ XML-Element
            else if $node == ExclusiveGateway$ then
                $processNodes \leftarrow$ create new $exclusiveGateway$ XML-Element
            else if $node == Event$ then
                if $node == startEvent$ then
                    $processNodes \leftarrow$ create new $startEvent$ XML-Element
                else
                    $processNodes \leftarrow$ create new $endEvent$ XML-Element
                end
        end
    end
    $xmlDoc \leftarrow$ add $sequenceFlows, processNodes, messages$ and $messageFlows$
    Export $xmlDoc$
end
```

ating changes in the context of collaborative processes. The framework already provides functionalities for importing and transforming BPMN process models into RPST representation (but not vice versa) and calculating change effects on the different models. The current work complements the framework by automatically generating repositories of collaborative models that would serve as a testbed for assessing and simulating change propagation techniques. As this work also supports the specification of compliance rules to which the generated models should comply, the resulted repository is also being used for evaluating approaches for compliance checking in the context of process collaborations. Even though the implementation is integrated within the C3Pro framework, it still represents an independent component that could be used for several research purposes; e.g., faults prediction [2], mining. The component for transforming RPST models to BPMN enables their simulation and executability.

Figure 4 represents a simplified class structure of the implemented components, that are necessary for generating process collaborations, starting with the generation of the Choreography Model that complies to imposed compliance rules, leading to deriving the public, private as well as the collaboration models, and finishing with the translation to BPMN/XML. The numbers indicate the order in which the components are instantiated.

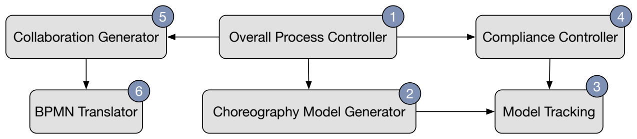

Fig. 4. Prototype Architecture

The logic of coordinating the entire generation process (see Algorithm 1) is implemented in the *Collaboration Generation Controller* component. The *Choreography Model Generator* comprises the algorithm for generating random models (see Algorithm 2). Thereby it utilizes the *Model Tracking* component constantly, which represents the actual model and provides the necessary functionalities to ensure the correctness of the resulting model. The introduced logic of specifying and imposing GCRs is implemented within the *Compliance Controller* component. It also utilizes the same instance of the *Model Tracking* class in order to find possible assignments for the imposed interaction order. The *Collaboration Generator* provides the functionalities of deriving the public and private models from the generated choreography model. At last, the translation of the internal model representation to BPMN, is encapsulated within the *BPMN Translator* component (see Algorithm 4). The prototype has been tested and already served as input for research on change propagation and compliance checking in collaborative processes. The execution of generated BPMN models[3] might be automated to enable logging and process collaborations mining. Models were generated to test the influence of the parameters "number of parallel/exclusive gateways" and "maximum branching" on the effort for model generation (without compliance rules) and the interrelation of number of imposed compliance rules, gateways, and successful generation of models.

Results and lessons learnt: The time to generate the models increases linearly with the number of gateways, independently whether parallel or exclusive gateways are used. The same holds for the "maximum" branching parameter. The reason is that the number of branches in the generated models increase and hence the algorithm has to check for more branches when creating the models. The number of parallel gateways does not influence the success of model generation for any number of compliance rules (1-100 compliance rules were tested). The existence of exclusive gateways greatly influences the model generation success; independently of the number of compliance rules (in the simulation 1-100) 76% of the model generations fail. Another simulation run tested the number of successful model generations depending on the number of exclusive gateways (for 60 compliance rules). For 0 and 1 exclusive gateways 100% of the generations are successful. The number of successful generations then drops in an inverse exponential way; for more than 50 exclusive gateways no generation attempt is successful anymore. Here also the effect of compliance rules that are depending on each other and hence impose strict sequence order on the models kicks in.

[3] Data available at: http://gruppe.wst.univie.ac.at/c3pro/data.zip

5 Related Works

Several research methods have been proposed, which generate process models from natural language text [8, 10]. In [8], BPMN models are produced from natural language texts by utilizing syntax parsing and semantic analyzing mechanism in combination with anaphora resolution. The result of the parsing algorithm is a declarative model that includes the extracted actions, actors and their dependencies, which serves as basis for generating the BPMN model. In [10], BPMN and DMN models are constructed from SBVR vocabulary[4]. Similarly, several transformation approaches have been proposed, which generate BPMN models from existing UML use cases [12, 19] or sequence diagrams [15].
In comparison to this work, the aforementioned approaches require the original specification as text or UML diagram to generate the business models. This is limited by the availability of such resources, and also do not deal with compliance constructs or choreography models.
In [18], process models are generated using semi-structured information about process activities along with their execution conditions. This specification is then formalized as a constraint satisfaction problem (CSP) and fed to a constraint solver that generates synthetic execution logs, which in turn, serve as input for process mining techniques. This again, requires a data collection phase, in which participants have to provide valid specifications to be merged. Also, it does not deal with choreography models nor differentiate between public or private models. In [3], BPMN process models are generated randomly. In contrast to this work, the latter focuses on intra-organizational process models. It also supports user-defined parameters to influence the model outcome in terms of number of node types and degree of branching. In, [14] a bottom-up approach has been proposed, which combines existing private processes to build process collaborations using adaptors. This assumes the availability of such models and requires a preselection of the models that will be composed (e.g, consumer, provider). The approach does not support compliance rules and does not allow much control over the complexity of the output models (e.g., number of exclusive gateways).

6 Conclusion

This work provided an approach that generates repositories of constrained process collaborations while ensuring their correctness in terms of compatibility, consistency and compliability. Such repositories are useful for simulating and evaluating research works in the context of inter-organizational processes. The approach is implemented and the resulted repositories are already exploited for simulating change propagation, compliance checking and faults prediction in collaborative processes. Future work includes generating distributed logs for mining and considering more compliance patterns.

Acknowledgment This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-072 and COMET SBA-K1.

[4] Semantics of Business Vocabulary and Business Rules

References

1. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer Publishing Company, Incorporated, 1st edn. (2011)
2. Borkowski, M., Fdhila, W., Nardelli, M., Rinderle-Ma, S., Schulte, S.: Event-based failure prediction in distributed business processes. Information Systems (2017)
3. Burattin, A.: PLG2: multiperspective process randomization with online and offline simulations. In: BPM Demo Track. pp. 1–6 (2016)
4. Cabanillas, C., Norta, A., Resinas, M., Mendling, J., Ruiz-Cortés, A.: Towards process-aware cross-organizational human resource management. In: Bider, I., Gaaloul, K., Krogstie, J., Nurcan, S., Proper, H.A., Schmidt, R., Soffer, P. (eds.) Enterprise, Business-Process and Information Systems Modeling. pp. 79–93 (2014)
5. Eshuis, R., Norta, A., Kopp, O., Pitknen, E.: Service outsourcing with process views. IEEE Transactions on Services Computing **8**(1), 136–154 (Jan 2015). https://doi.org/10.1109/TSC.2013.51
6. Fdhila, W., Indiono, C., Rinderle-Ma, S., Reichert, M.: Dealing with change in process choreographies: Design and implementation of propagation algorithms. Information Systems **49**, 1 – 24 (2015)
7. Fdhila, W., Rinderle-Ma, S., Knuplesch, D., Reichert, M.: Change and compliance in collaborative processes. In: SCC. pp. 162–169 (2015)
8. Friedrich, F., Mendling, J., Puhlmann, F.: Process model generation from natural language text. In: CAISE (2011)
9. Grefen, P., Rinderle, S., Dustdar, S., Fdhila, W., Mendling, J., Schulte, S.: Charting process-based collaboration support in agile business networks. IEEE Internet Computing pp. 1–1 (2018). https://doi.org/10.1109/MIC.2017.265102547
10. Kluza, K., Honkisz, K.: From sbvr to bpmn and dmn models. proposal of translation from rules to process and decision models. In: ICAISC (2016)
11. Knuplesch, D., Reichert, M., Fdhila, W., Rinderle-Ma, S.: On enabling compliance of cross-organizational business processes. In: BPM. pp. 146–154 (2013)
12. Lubke, D., Schneider, K., Weidlich, M.: Visualizing use case sets as bpmn processes. In: 2008 Requirements Engineering Visualization. pp. 21–25 (2008)
13. Rosa, M.L., Reijers, H.A., van der Aalst, W.M.P., Dijkman, R.M., Mendling, J., Dumas, M., García-Bañuelos, L.: Apromore: An advanced process model repository. Expert Syst. Appl. **38**, 7029–7040 (2011)
14. Seguel, R., Eshuis, R., Grefen, P.W.P.J.: Architecture support for flexible business chain integration using protocol adaptors. Int. J. Cooperative Inf. Syst. **23** (2014)
15. Suchenia, A., Kluza, K., Jobczyk, K., Winiewski, P., Wypych, M., Ligeza, A.: Supporting bpmn process models with uml sequence diagrams for representing time issues and testing models. In: Artificial Intelligence and Soft Computing. pp. 589–598 (2017)
16. Turetken, O., Elgammal, A., van den Heuvel, W.J., Papazoglou, M.P.: Capturing compliance requirements: A pattern-based approach. IEEE Software (2012)
17. Vanhatalo, J., Voelzer, H., Koehler, J.: The refined process structure tree. Data & Knowledge Engineering **68**(9), 793 – 818 (2009)
18. Wisniewski, P., Kluza, K., Ligeza, A.: An approach to participatory business process modeling: Bpmn model generation using constraint programming and graph composition. In. Applied Sciences **8**(9) (2018)
19. Zafar, U., Bhuiyan, M., Prasad, P.W.C., Haque, F.: Integration of use case models and BPMN using goal- oriented requirements engineering. JCP pp. 212–221 (2018)