universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

"Automatic Generation and Translation of Process Collaboration Models to BPMN/XML"

verfasst von / submitted by

Frederik Bischoff BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Math. Dr. Stefanie Rinderle-Ma

Contents

1. Introduction

In the research field of business process models and techniques, researchers can only rely on a repository of centralized, intra-organizational processes to support their work. But regarding decentralized, cross-organizational models, they face the problem that there is a lack of available model examples. Within the scope of this work, this lack is aimed to be tackled. Thereby, the main objective is to build a repository of distributed and collaborative process models by developing and implementing an automatic generation process for business process collaborations. The generation process must thereby ensure soundness, consistency and compatibility of the resulting models. Additionally, it should also allow the generation of models which comply to imposed compliance rules. At last, to ensure the executability of the auto-generated models by process engines, an additional goal of this work is to develop a transformation of the utilized RPST[1] representation to BPMN[2]. These requirements ensure that the repository can then be used for further research, such as change management or process mining of collaborative processes. Based on those objectives, the following research questions are derived:

- *RQ-1*: How to build a repository of collaboration process models that can be used as support for further research in this field?
- *RQ-2*: How to ensure that the resulting process models in this repository are correct in terms of consistency, compatibility and compliability? Which process flow perspectives and compliance rule patterns should be supported regarding compliability?
- *RQ-3*: How to transform a collaborative process represented as an RPST into an executable form?

This work is part of the CRISP[3] project (ICT15-072) funded by the Vienna Science and Technology Fund (WWTF). The implementation of the prototype is integrated in an already existing framework. The main research within the project is to analyze flexibility and adaptivity of collaborative business processes at design time as well as at runtime. Regarding consistency and correctness of collaborative business processes, the propagation of a process change over all process participants is one of the main challenges the project tries to tackle. Furthermore, the impact of compliance rules imposed on collaborative business scenarios is analyzed as well as the

[1]Refined Process Structure Tree
[2]Business Process Model and Notation - http://www.bpmn.org/
[3]http://gruppe.wst.univie.ac.at/projects/crisp/

issue of ensuring business compliance by considering the privacy and autonomy of the involved business partners.

This work is structured as followed. First, a brief overview of related work is given in Chapter 2, followed by an introduction into process collaborations represented in BPMN by explaining the different model views in Chapter 3. Then the concept of the automatic collaboration process generator is introduced in Chapter 4. Thereby the approach, the possibility of influencing the outcome and the therefor necessary components are explained. Afterwards, in Chapter 5, the adaption of the RPST for the internal process model representation is explained as well as the class structure and data models of the implemented components. The performance of the implemented generation process depending on different parameter settings is then examined in Chapter 6. Finally, the work is concluded in Chapter 7.

2. Related Work

This work focuses on the generation of collaborative business process models. Thus, the thereby introduced method of generating processes randomly can be compared with work that also generate business process models but differ from the input source of generation and it's algorithm. In comparison to the purpose of this work, which is building a repository of process collaborations to facilitate further research, they mainly have the goal to simplify the process of business process modeling by reducing the time for process acquisition and also generation. At the end, a similar research is introduced which also follows an approach of random process generation with the goal to facilitate further research with process examples.
There is research on generating processes from natural language [1] [2]. In [1], Friedrich et al. introduced an approach to generate BPMN process models from natural language texts by utilizing syntax parsing and semantic analyzing mechanism in combination with anaphora resolution[1]. The result of the parsing algorithm is a declarative model that includes the extracted actions, actors and their dependencies. This model then serves as basis for the BPMN model composition. Compared to that, Krzysztof et al. introduced an transformation approach for structured natural language in form of SBVR[2] in [2].
There are also approaches which generate BPMN models on the basis of UML[3] use cases [3] [4] or sequence diagrams [5].
Another interesting, recent research focuses on the generation of process models based on constraint programming [6]. Therein, Wisniewski et al. introduced an approach which uses semi-structured information about process activities along with their execution conditions as input for a constraint satisfaction problem (CSP). On

[1]Resolving what a pronoun or noun refers to.
[2]Semantics of Business Vocabulary and Business Rules
[3]Unified Modeling Language

the basis of this CSP, a constraint solver generates synthetic execution logs of all valid execution sequences. At last, the BPMN model is generated with the aid of process mining techniques. Their approach is similar to the one introduced in this work. The conceived model generation algorithm in this work also takes user defined dependencies between tasks (see Chapter 4.2.2) into account.
In [7], Burattin also introduced a tool for generating BPMN process models randomly. In contrast to this work, which focuses on process collaborations, the tool only supports basic process models with the components task, exclusive and parallel gateways. The tool also supports user-defined parameters to influence the model outcome in terms of number of node types and level of branching. It is also possible to import existing models in order to evolve them. But in contrary to this work, it's main purpose is less the generation of random process models than the combination with the possibility to simulate the processes in order to obtain execution logs for testing process mining algorithms.

3. Process Collaborations in BPMN

This chapter provides the basics of business process collaborations, represented in BPMN. In BPMN, collaborative processes are represented from different perspectives, whereas each perspective is represented by a different BPMN model type. In the following, the different model types and their represented perspective are explained with the help of a collaborative business process example.

Private Model

The private model is modeled from the perspective of a single participant of a collaborative process. In a collaborative scenario, it describes the complete internal business logic of one partner as well as the messages exchanged with other partners. Activities which are only for the participant are called *private activities*, whereas activities which involve participation of other partners are called *public activities*.

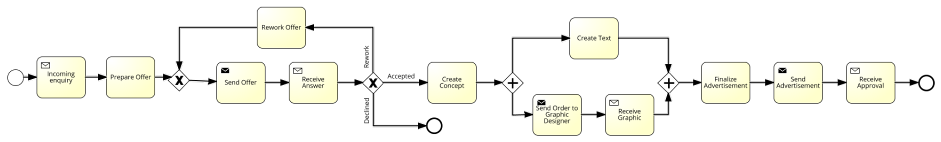

Figure 3.1.: Private Model Example

For example, in the private model shown in Figure 3.1, the task *Create Concept* represents a private activity, whereas the task *Send Offer*, involves message exchange with another participant and therefore represents a public activity. A private model is a process model in the classical sense. It's fundamental modeling objects are unchanged since BPMN 1.0. Is a private process modeled and attributed in detail, it also represents an executable process that can be executed by a process engine.

Public Model

The public model is also modeled from the perspective of a single participant of a collaborative process. It is a reduced view on the private model of a partner. It can also be described as a projection of the whole collaboration process focusing on one participant. It only includes public activities, involving message exchange with other partners. Private activities which are not relevant for other partners and which don't want be to be shared with the partners are omitted deliberately. Figure 3.2 shows the corresponding of the already introduced private model.

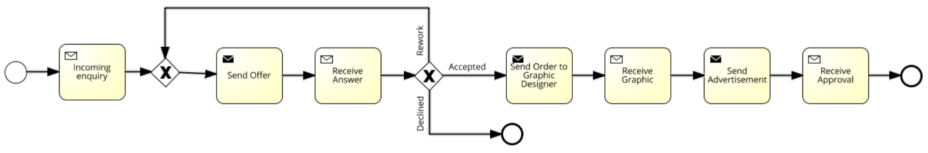

Figure 3.2.: Public Model Example

Collaboration Model

The collaboration model is the interconnection between the public models of all participants. The thereby formed model gives a holistic view on the collaborative process and does not focus on one partner. Each partner is represented as a pool and the message exchange between them is shown by a message flow that connects two public activities or just the pools. Figure 3.3 shows an example collaboration model.

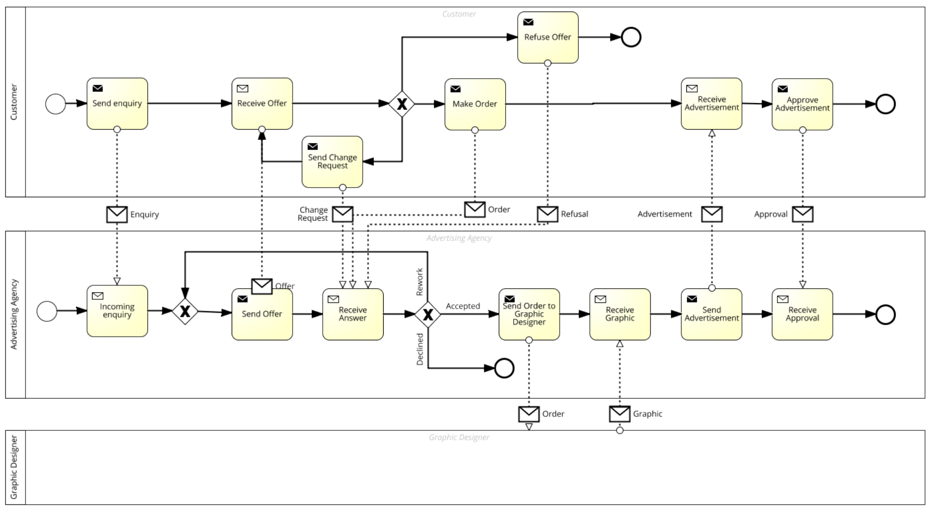

Figure 3.3.: Collaboration Model Example

Each partner's public process is modeled inside a pool. It is also allowed for a process to not include the public model inside an participant's pool. If a pool contains a process, it is called a "white box". If a pool is empty, it's called a "black box". For Example in figure 3.3 the pools of the partners *Customer* and *Advertisement Agency* are modeled as "white box" and the pool of the *Graphic Designer* as a "black box" [8].

Choreography Model

The choreography model is available since BPMN 2.0 and focuses solely on the sequence of message exchanges between the partners. Each message exchange is represented as an interaction with an initiating partner, a receiving partner (shaded in grey) and the message exchanged. In contrast to the collaboration model, which also focuses on the message flow, the choreography model additionally displays the exact sequence flow (i.e. conditional message flows or parallel message flows), which is not always evident form the former (i.e. black box pools).

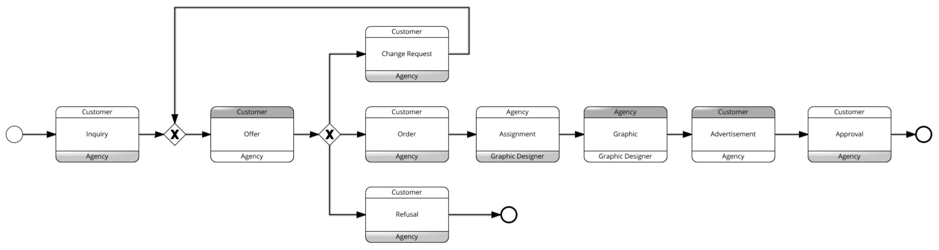

Figure 3.4.: Choreography Model Example

4. Conception

The previous chapter covered the necessary background information on business process collaborations represented in BPMN and their different perspectives. In this chapter, the conception of the automatic process collaboration generator is introduced.

4.1. Approach

As already mentioned in Chapter 3, a process collaboration involving several partners can be modeled from different perspectives (partner or global) through the use of different model types. The process collaboration generator, implemented in the context of this work, generates all three different model types as the output. In general, there exist two different approaches to build a process collaboration with all the described models [9]. In the *top-down approach*, first the choreography model is build, then the public and private models of each partner are derived and defined consistently. In comparison, in the *bottom-up approach*, each partner has already defined a private and public process. Then, the choreography model is constructed by connecting the public models via message exchange.

The automatic generator, presented in this work, follows the *top-down approach*, by first generating the choreography model and then deriving the collaboration, public and private models from it. Thereby, each interaction (choreography task) of the choreography model is converted into a send and receive task and then added to the involved partners processes, to build their public process models. In turn, each private model is derived from its corresponding public model by enriching the public model with abstract private tasks. The collaboration model is then built by composing the partner's public processes into one model.

Generally, when defining business process models, three levels of correctness need to be considered and satisfied by the models:

Syntactical Correctness is defined by the underlying BPMN meta model and refers to the correct use and composition of the corresponding model elements. Syntactical constraints for example include the fact that any model must at least have one start and one end event, as well as that flow objects (e.g. tasks, gateways, events) can only be connected by control flow edges [10].

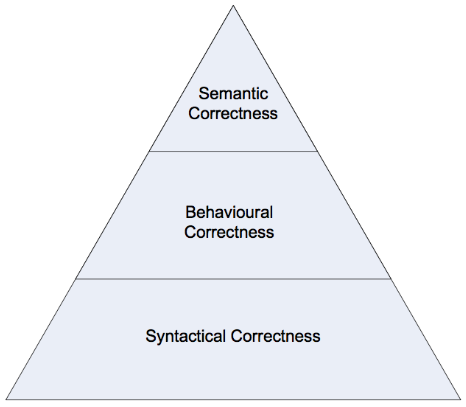

Figure 4.1.: Pyramid of Business Process Model Correctness (Source: [10])

Behavioral Correctness constitutes that a process model must be executable and therefore free of deadlocks or lifelocks. It assumes that the model is syntactically correct, because the behavior of a syntactical incorrect model is undefined. Regarding collaborative, cross-organizational processes, it is also required that the composition of the involved public models is compatible. For example, this means, that for every message that is send, a corresponding partner must be able to receive it [9,10].

Semantic Correctness means that the a process model must comply with imposed compliance rules [10]. Thereby it must be differentiated between *local* and *global compliance rules*. *Local compliance rules* constrain the private process of a partner, whereas *global compliance rules* constrain the interaction between multiple partners [9]. In this work, only *global compliance rules* are considered to constrain the choreography model.

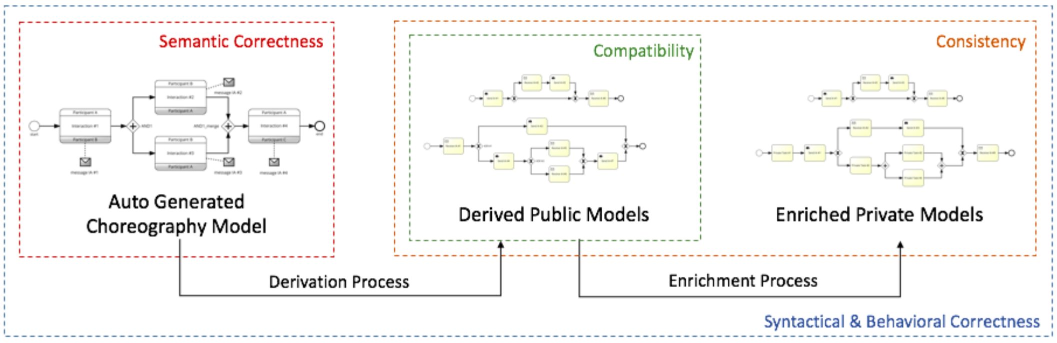

Figure 4.2.: Top-Down Approach

All three correctness levels are considered by the proposed algorithm for implementing an automatic process generator. The algorithm ensures that only model specific flow objects are used to build the processes and that they are connected appropriately (syntactical correctness). It also guarantees the absence of deadlocks and lifelocks (behavioral correctness) and offers the possibility to define global compliance rules (see Chapter 4.2.2), to which the generated collaboration should comply (semantic correctness). Deriving all models from the before generated choreography model offers also the advantage, that if the deriving process is implemented correctly, it already ensures *consistency* and *compatibility* between the different models. In the context of collaborative processes, *consistency* means, that the private model of a partner has to be consistent with the corresponding public model, whereas *compatibility* requires the public models of the collaborating partners to be compatible with one another [11]. This ensures that the executed business process of one partner satisfies the behavior that is communicated to the partners through his public models [9]. Figure 4.2 illustrates the the collaboration generation approach with it's different levels of correctness.

4.2. Constraining the Collaboration Generation

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.

4.2.1. Parametric Constraints

The following input parameters are specified to influence the random generation of the choreography model and hence also the deriving models:

- Number of Partners:
  Determines the number of participants that are involved in the process collaboration.
- Number of Interactions:
  Determines the number of messages that are exchanged between the partners.
- Number of Exclusive Gateways:
  Determines the number of Exclusive Gateways that are put into the model.
- Number of Parallel Gateways:
  Determines the number of Parallel Gateways that are put into the model.
- Max. Branching:
  Determines the maximum possible number of paths created for each gateway.

4.2.2. Compliance Constraints

Generally, compliance rules can be defined for different process flow perspectives of a process. It can be distinguished between compliance rules that constrain the control flow (sequence of activities), the data associated with the activities, the resources (specific user or role) that perform the activities or the time perspective. There exist several languages and approaches on how to define and specify compliance rules, including formal languages [12], [13], visual languages [14], [15] and pattern-based approaches [16], [17]. Both visual and pattern-based approaches aim at hiding formal details (e.g. temporal logic) and therefore simplifying the specification of compliance rules.

For the specification of compliance rules for the automatic process generator, the pattern-based approach of Turetken et. al. [16] is utilized. 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. There are four different groups of patterns:

- *Order* patterns concern the sequencing of activities. For Instance, '*Customer_Inquiry* LeadsTo *Offer*', is used to express that after an inquiry is sent and received, an offer must be sent at some point afterwards.
- *Occurrence* patterns represent rules that address the existence of certain activities. For instance, '*Check_Credit_Worthiness* Universal', is used to express that the activity must always occur when the process is executed.

- *Resource* patterns constrain that certain activities must be performed by a specific user or group of users (role). For instance, '*Approve_Offer* PerformedBy *Role Q*' constrains, that the activity must be assigned to a user who is part of the user group Q.
- *Time* patterns are used to assign temporal rules to Order or Occurrence patterns. For instance, '*Customer_Inquiry* LeadsTo *Offer* Within *7(days)*' constrains, that an Offer has to be replied within seven days after an inquiry has been received.

Because the generated process collaborations neglect the data, time and resource perspective and focus solely on the control flow, only compliance rules that constrain the sequence of activities are possible to impose. Following process control patterns are supported to constrain the automatic generated choreography:

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 4.1.: Overview of supported Compliance Patterns

Note that the *P LeadsTo Q* pattern does not demand an immediate succession of interaction Q on interaction P.

4.3. Random Process Collaboration Generation

In this chapter, the needed components and their functionality for implementing a random process generator are explained. Each component encapsulates a step of the above described approach. Based on this approach, four components with single responsibilities can be derived: one component is responsible for randomly generating a choreography model, one for imposing compliance rules on the resulting process, another for deriving the remaining model types and the last component keeps control over the overall process. In the following the components and their functionalities will be explained in details.

4.3.1. Overall Process Controller

The *Overall Process Controller* represents the orchestration component for building random process collaborations, based on given constraints. The process follows the principle *'first build then check'*, which means that after a random choreography model has been generated (see chapter 4.3.2), it will then be checked if the interactions defined within the compliance rules, can be assigned into 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 imposing of the compliance rules fails repeatedly, it's an indicator that the amount of interactions in the model is too small relative to the unique interactions specified within the compliance rules. To overcome this, the number of interactions is increased by 10 percent every 10th build. 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. The whole process of generating a random choreography by coordinating the different functions is outlined in Algorithm 1.

Algorithm 1: Overall Collaboration Generation Controller

```
buildSuccess = false;
while buildSuccess $\neq$ true do
    build new choreography model;
    if compliance rules are defined then
        assign interactions;
        if assignment successful then
            buildSuccess = true;
        else if number of interaction mod 10 $\equiv$ 0 then
            increase number of interactions by factor 1.1;
        end
    else
        buildSuccess = true;
    end
end
generate whole collaboration;
export models to BPMN;
```

4.3.2. Generating Random Choreography Models

The actual algorithm for generating graphs, representing choreography models, is implemented within the *Choreography Model Generator* component. Throughout the generation process, it is essential to keep track of the current model state at every point, in order to know where it is possible to put a new node without violating the syntactical and behavioral correctness of the resulting BPMN model. Therefore a *Model Tracking* component is necessary, which provides control flow logic and a corresponding data model. The concept of this component is based on the RPST graph decomposition which was introduced by Vanathalo et al. in [18]. In [18], 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*. Modular means that a local change of the graph only impacts the corresponding, decomposed fragment, whereas objective requires that a fragment

[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.

does not overlap with another fragment. Furthermore, fragments can be characterized as *trivial* and *non-trivial*. A fragment is *trivial* if it contains exactly one edge and therefor, in the RPST, trivial fragments are represented by their edge and are always leaf nodes in the RPST. In contrast, each *non-trivial* fragment is represented as non-leaf node. The root fragment (non-trivial) of a RPST decomposition contains all edges of a graph. Therefore it contains all other *trivial* and *non-trivial* fragments and has the source and sink node as it's boundaries. 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* (minimum two and maximum is defined by the max_branching parameter), that represent the different paths created by a parallel or exclusive gateway. Again, each branch holds the set of nodes that are on the path of a particular branch, whereas a path and therefore also 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.

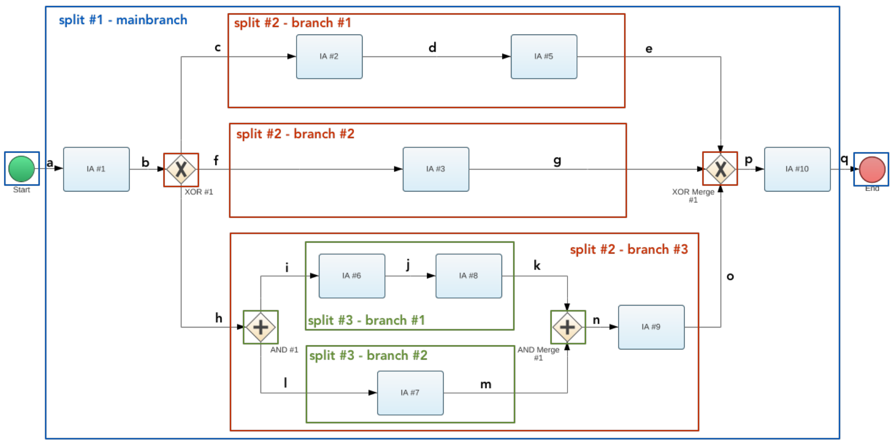

Figure 4.3.: Model Tracking Component - Concept

Figure 4.3 illustrates the concept of this *Model Tracking* component. In this example, there are in total 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, which is shown in figure 4.4, that defines that every branch must be related to a split, this pseudo-split is necessary to keep track of the root branch.

The root branch (split #1 - branch #1) holds the set of ordered nodes: Interaction #1, XOR gateway #1, XOR merge #1 and Interaction #10. Split #2 has three branches. The first branch contains the nodes Interaction #2 and Interaction #5. The second branch consists only of Interaction #3 and the third branch contains AND gateway #1, AND merge #1 and Interaction #9. At last, the third split

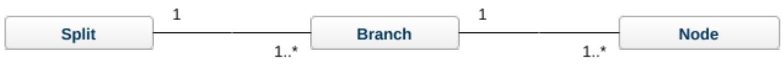

Figure 4.4.: Model Tracking Component - Data Model

consists of two branches, which hold the remaining interactions #6, #8 and #7 respectively. Note that branches and therefore also splits, contain other splits, e.g. split #2 contains split #3 in branch #3. This interleaving determines the hierarchy of the resulting RPST, which is shown in Figure 4.5. The root fragment is split #1 and contains the non-trivial fragment *split #2* as well as the trivial fragments (edges) E = {a, b, p, q}. In turn *split #2* contains the non-trivial fragment *split #3* and the edges E = {c, d, e, f, g, h, n, o}. The last non-trivial fragment *split #3* contains the edges E = {i, j, k, l, m}.

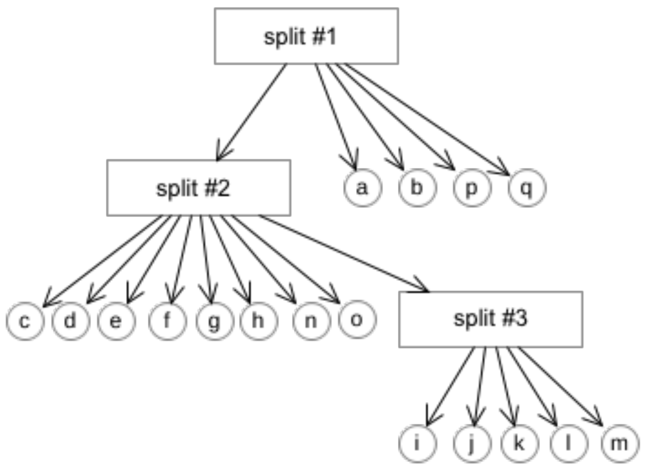

Figure 4.5.: Refined Process Structure Tree

During the generation process, the current model build is maintained within the *Model Tracking* component and simultaneously as a RPST graph. So far, there is no apparent necessity for the *Model Tracking* component. But in order to automatically generate graphs representing choreography models, additional control flow logic is needed to ensure syntactically and behavioral correctness and to supervise the compliance with the build parameters. Therefor, each branch needs a status. This status indicates whether the branch is *open*, *splitted* 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. In turn, *closed* defines 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. Within the RPST graph, the merge node and an edge between the branch's last node with the merge node is added. A branch can also be in *splitted* state, if it contains another split and none of this split's branches are yet closed, thus there exists no merge node for this split. In this case, a branch can not 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* again. For example, in figure 4.7, *branch #1* of *split #2* represents a branch in state *split*, whereas the main branch is in state *open* because *branch #2* of *split #1* is already closed by the merge node of *split #2*. When closing a branch, first it is necessary to determine if a branch is allowed to be closed, without violating the soundness of BPMN choreography models. This is dependent 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, its not necessary that an interaction is on the parent branch itself but on the branches of its child split or even on a deeper nested branch.

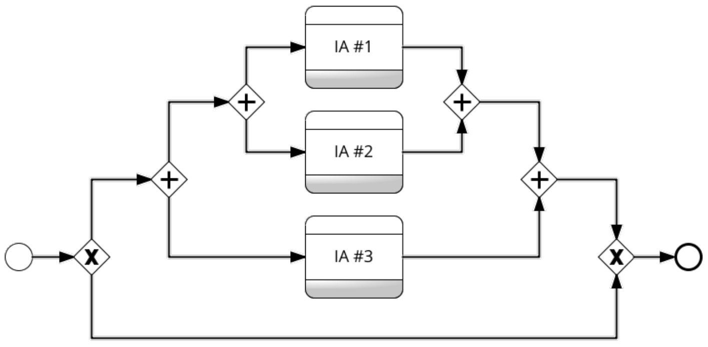

Figure 4.6.: Minimum interactions reserved by remaining gateways

For instance, see the upper branch of the second parallel gateway in Figure 4.6. The branch itself holds only another parallel gateway split node and a corresponding merge node but no interaction. This is valid because both branches of the parallel child split have an interaction and therefore also all enclosed paths of the parent branch. If the split node type is a exclusive gateway, it's allowed that one of the split's branches has no interaction on its path. For instance, see lowest branch of fist exclusive gateway in Figure 4.6. This depicts the circumstance that for an exclusive gateway, it's required to be able to also model a branch, where under certain process conditions no interaction between participants is necessary. This approach of tracking the status of the branches becomes crucial when only few remaining interactions are available and several branches are still open in the model. Because of the parametric limitation of the number of interactions, at a certain point in the build process or even directly in the beginning, if the proportion of specified gateways and interactions is small, interactions are not always allowed to be selected as next node type and not every open branch is allowed to be randomly selected for putting the next node into the model without resulting in a violation of the correctness of the model or exceeding the number of defined interactions. In order to determine whether this situation applies to a current point in a build process, the *Model Tracking* component monitors the amount of *free interactions* and *reserved interactions.*

Reserved interactions, are a subset of the remaining interaction that have either predetermined positions in the current graph (resInteractionsBranches) 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 graph and the number of gateways that are not yet put into model. Regarding the current model, each open and non closable branch increases the amount of *resInteractionsBranches* by one. Parallel gateways that are not yet placed into the graph 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 gateways node without an interaction in between, the minimum amount of *resInteractionsAndGateways* 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 gateway. Because if there is also a remaining parallel gateway, the exclusive gateway could be put on a branch of the parallel gateway directly after the split and therefor the one needed interaction of the exclusive gateway is already considered in the calculation of *resInteractionsAndGateways.* The influence of nested gateways onto the minimum amount of reserved interactions is illustrated in Figure 4.6. Given the parametric constraints *numberOfInteractions = 3*, *numberOfParallelGateways = 2* and *numberOfExclusiveGateways = 1*, the figure represents one out of three possible resulting choreography models, which only differ in the sequence of used gateway types but not in the way of nesting and branching. 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 4.3.1, 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 < 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* > 0, then all open branches can be selected for putting the next node. For the limitation of random branch selection, see also Algorithm 3 and Figure 4.7, which shows an unfinished choreography model at the point during the build process, assuming the parametric constrains numberOfInteractions = 6, numberOfAndGateways = 1 and numberOfExclusiveGateways = 1, where not all open branches are allowed to be selected for placing the node (in this case an interaction). In order to obtain a sound choreography model while not increasing the number of initially specified interactions, the sixth interaction is only allowed to be placed on branch

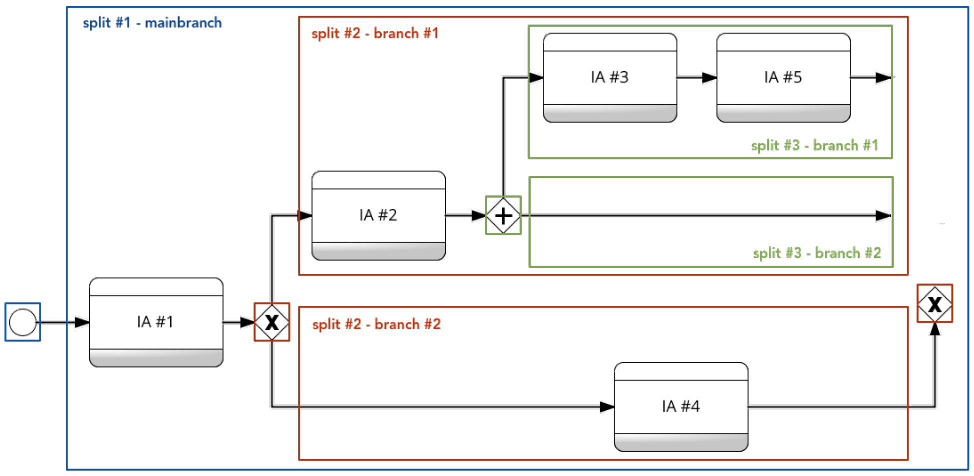

Figure 4.7.: Limitation of random branch selection

#2 of split #3. In this scenario, *reservedInteractionsTotal* equals one and *freeInteractions* therefor equals zero. On the other hand, if assuming the total number of interactions being 7, all branches of split #3 and the root branch are possible candidates for placing the sixth interaction, because at this point, *freeInteractions* would equal one. In case of selecting the next possible node type, interactions are only allowed to be randomly selected, if the the amount of *free interactions* > 0 or not all remaining interactions are reserved by not yet consumed gateways.

Definition 4.3.1 *Let x be the number of branches which are open and non-closable, $remainingInteractions$, all interactions not yet put into the model and $remainingXOR$ and $remainingAND$ the number of gateways not yet put into the model. Then*

$$\begin{gathered}
\boldsymbol{resInteractionsBranches} = x \\
\boldsymbol{resInteractionsAndGateways} = remainingAND + 1 \\
\boldsymbol{resInteractionsXORGateways} = \textbf{\textit{If}}\ remainingAND > 0\ \textbf{\textit{Then}}\ 0\ \textbf{\textit{Else}}\ 1 \\
\boldsymbol{resInteractionsGateways} = resInteractionsAndGateways + \\
resInteractionsXORGateways \\
\boldsymbol{resInteractionsTotal} = resInteractionsBranches + resInteractionsGateways \\
\boldsymbol{freeInteractions} = remainingInteractions - resInteractionsTotal
\end{gathered}$$

The overall procedure of generating random choreography models is shown in Algorithm 6. The input for a random model generation are the parametric constraints introduced in section 4.2.1. At first, it is checked if the specified combination of the amount of interactions and gateways are sufficient for generating a sound model. This is the same evaluation as in determining the reserved interactions by remaining gateways. Therefore, the specified *numberOfInteractions* must be greater or equal *resInteractionsGateways*. If the validation is successful, the *Model Tracking* component gets instantiated. Thereby, a split for the start event and the root branch is created. After this setup, the algorithm loops over the number of remaining interactions until all interactions are put into the model. Within each loop, at first a node type for the next node is randomly selected out of the pool of remaining nodes. This can be an interaction, exclusive or parallel gateway, depending on the number of free and reserved interactions as already explained. The algorithm for random possible node type selection is shown in Algorithm 2.

Algorithm 2: getRandomNodeType()

```
begin
    possibleNodeTypes ← {}
    if $freeInteractions > 0 \vee remainingInteractions$ ¿ $resInteractionsGateways$ then
        possibleNodeTypes ← possibleNodeTypes ∪ Interaction
    end
    if $remainingParallelGateways > 0$ then
        possibleNodeTypes ← possibleNodeTypes ∪ ParallelGateways
    end
    if $remainingExclusiveGateways > 0$ then
        possibleNodeTypes ← possibleNodeTypes ∪ ExclusiveGateways
    end
    return random NodeType of possibleNodeTypes
end
```

After a node type has been randomly selected, a position in the model is determined for placing a node of the previously selected node type by randomly selecting a possible open branch. Which branches are in the pool of possible, selectable branches, is again depending on whether there are free interactions available or not. If there are free interactions left, all open branches are allowed to be selected, independent of the priorly selected node type. Otherwise, only branches that are not closable are allowed to be included in the pool of possible branches. The algorithm for random possible branch selection is shown in Algorithm 3.

Algorithm 3: getRandomBranch()

```
begin
    possibleBranches ← {}
    if $freeInteractions > 0$ then
        possibleBranches ← all open branches
    else
        possibleBranches ← all not closable branches
    end
    return random Branch of possibleBranches
end
```

In the next step, the randomly selected branch is checked whether it's closable or not and if so gets closed by random. This doesn't apply to the root branch to make sure that there is always one branch where the model can further evolve. This step of random branch closing is necessary to obtain balanced choreography models regarding 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 possible to close 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. If the selected branch gets randomly closed, a merge node for the selected branch's split is created and added to the parent branch. Additionally, in the data model of the corresponding split, the merge node gets noted in order to assure that the last nodes of the other branches of this split will get connected with the same merge node as soon as they get closed. Finally, the algorithm jumps back to the beginning of the loop and starts again with randomly selecting a possible node type for the next node to be placed in the model.

Algorithm 4: getRandomBranchAmount()

```
Input :
    – $minBranching \leftarrow 2$
    – $maxBranching \leftarrow$ user specified upper branch amount border
    – $freeInteractions \leftarrow$ number of currently free interactions
1  begin
2      if $freeInteractions \geq 0$ then
3          $currentMaxBranching \leftarrow minBranching + freeInteractions$
4          if $currentMaxBranching > maxBranching$ then
5              $currentMaxBranching \leftarrow maxBranching$
6          end
7      else
8          $currentMaxBranching \leftarrow minBranching$
9      end
10     return random value between $minBranching$ and $currentMaxBranching$
11 end
```

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. If only the user specified upper branching border (max. branching parameter) is adducted, there is a high chance that this would result in an inconsistent model, because the remaining interactions are insufficient for all paths created. In order to prevent this, 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. The algorithm for random possible branch amount selection is shown in Algorithm 4. 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 newly created edge between the preceding and the new node gets added into the RPST graph, before the loop starts over by selecting a random node type for the next node to be put into the model.

Algorithm 5: closeSplit(Split)

```
Input :
    – mainSplit ← split of root branch
begin
    foreach branch of split.branches do
        foreach node of branch.nodes do
            if node is gateway then
                closeSplit(split)
            end
        end
        if branch is open then
            branch.close()
        end
    end
end
Function branch.close()
    split ← split of branch
    if split.mergeNode == null then
        mergeNode ← instantiate merge node of gateway type
        split.mergeNode ← mergeNode
        branch.state ← closed
    else
        branch.state ← closed
    end
```

After all interactions and gateways are put into the model, the loop ends and all still open branches are getting closed. Algorithm 5 displays the function to close all yet open branches by looping the model recursively. At this point, where the model doesn't further evolve, a branch is closed if the belonging split has a merge node assigned. If this is not the case, a merge node is created, added to the corresponding branch and assigned to the split.

At this point the generated model is already syntactically correct because all model elements are used and connected according to BPMN specification. To achieve also behavioral correctness in choreography models, beside a correct sequence flow, a

Algorithm 6: Generate Choreography Model

```
hbt Input :
    – $remainingNodes \leftarrow$ number of nodes by type
    – $participants \leftarrow$ number of participants
    – $loops \leftarrow$ number of loops
    – $graph \leftarrow$ RPST graph
begin
    if $number\ of\ interactions \leq number\ of\ gateways + 1$ then
        model generation not possible
    end
    $modelTracking \leftarrow$ initialize model tracking component
    while $remainingInteractions > 0$ do
        $nextNodeType \leftarrow$ getRandomNodeType()
        $selectedBranch \leftarrow$ getRandomBranch()
        $precedingNode \leftarrow$ last node of $selectedBranch$
        if $precedingNode\ is\ NULL$ then
            $precedingNode \leftarrow$ split node of $selectedBranch$
        end
        if $selectedBranch\ is\ closable$ then
            close branch by random
            if $closed$ then
                continue
            end
        else
            $nextNode \leftarrow$ instantiate node of $nextNodeType$
            if $nextNodeType\ is\ Gateway$ then
                $branchCount \leftarrow$ getRandomBranchCount()
                $split \leftarrow$ instantiate new split
                $modelTracking.splits \leftarrow$ split
                for $i \leftarrow 0\ branchCount$ do
                    $branch \leftarrow$ instantiate new branch
                    $split.branches \leftarrow branch$
                    $i \leftarrow i + 1$
                end
            end
            $selectedBranch.nodes \leftarrow selectedBranch.nodes \cup nextNode$
            decrease $remainingNodes$ of $nextNodeType$
            add edge between $precedingNode$ and $nextNode$ to $graph$
        end
    end
    close still open splits
    add end event to root branch
    enrich interactions with reasonable sender and receiver sequence
end
```

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. In case of gateways, additionally it is ensured that the receiver of the last interaction of each branch of the gateway is the same in order to determine a possible common sender for the succeeding interaction after the merge. Figure 4.8 illustrates a correct message flow within choreography models.

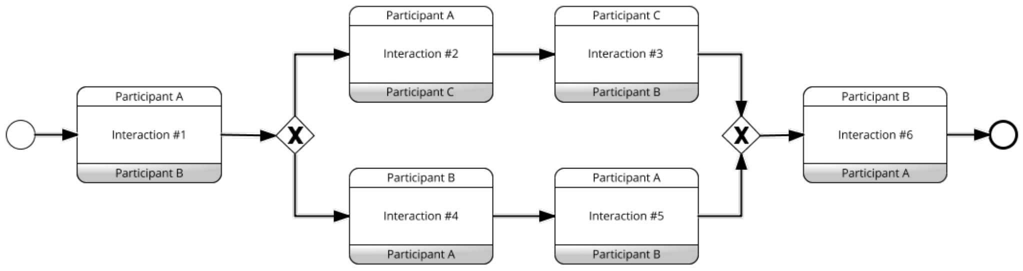

Figure 4.8.: Message Flow - Sender/Receiver sequence

Note that because of the fact that 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.

4.3.3. Compliance Rules Assignment

As pointed out in Chapter 4.3.1, first a model is generated and afterwards it is checked whether the interaction sequence specified within the compliance rules can be applied to the generated model. Instead of considering the imposed compliance rule during the choreography generating, the followed approach was favored to allow users to specify compliance rules, which then can be imposed to also already existing choreography models in order to check if this particular model complies to the specified rules. Until now, the generated model complies only to the parametric constraints. The logic of specifying and imposing compliance rules is implemented within the *Compliance Controller* component.

When specifying compliance rules to which the choreography model must comply, it must be checked whether the imposed rules are consistent with one another. In the context of the four supported patterns (see table 4.1), this applies only to the two order patterns (LeadsTo and Precedes). For instance, consider the following set of compliance rules:

- *CR-1*: P LeadsTo Q
- *CR-2*: Q LeadsTo S
- *CR-3*: S Precedes P

In this example, the rule *CR-1* in combination with *CR-2* is in conflict with *CR-3*, because *CR-1* and *CR-2* determine, that the involved activities must occur in the order P-Q-S, whereas *CR-3* constrains, that activity S must occur before activity P, which is in violation of the order determined by *CR-1* and *CR-2*. Algorithm 7 shows the conflict checking procedure implemented within the *Compliance Controller*.

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 before 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 Interaction P and Interaction Q. For each possible position of Interaction P there has to be at least one possible position for Interaction Q. The rules that determine applicable positions for the four implemented compliance patterns are shown in Definitions 4.3.2 - 4.3.5.

The difference between the determination of possible positions for the patterns *LeadsTo* and *Precedes* is that in case of *P LeadsTo Q* the position for Interaction Q must always be reached after the position of Interaction P has been reached, whereas in case of *P Precedes Q* the Interaction Q must only be possible to reach afterwards. In other words, if Interaction Q is executed, Interaction P must have occurred before. This means that unlike for the *LeadsTo* pattern, in case of a *Precedes* the succeeding interaction Q can also be inside an exclusive branch of the subsequent path of interaction P. For example, let P be assigned to *Interaction IA 2* of the choreography model shown in figure 4.9. In case of a *Precedes* pattern, the set of possible interactions for Q is $Q_{\text{IA2}} = \{$IA 3, IA 4, IA 5, IA 6, IA 7, IA 8, IA 9, IA 10, IA 11, IA 12, IA 13, IA 14$\}$. These are all interactions that are possibly reachable after IA 2 has been reached. On the other hand, in case of *LeadsTo*, the set of possible interaction assignments Q_{IA2} only includes $\{$IA 13, IA 14$\}$, because if IA 2 has been reached, only these two are always possible to be reached afterwards. Additionally to this rules, for a given P assigned within a parallel branch, the succeeding interaction Q is not allowed to be assigned to a position of another parallel branch, because there is no control mechanism that

Algorithm 7: Adding Compliance Rules

Input :

- compliance rule *cr*
- dictionary *orderDependencies* consisting of Interactions P and their succeeding Interactions S

```
begin
    if cr is order pattern then
        p ← preceding interaction of cr
        s ← succeeding interaction of cr
        if !orderConflictCheck(p, s) then
            add cr to complianceRules
            if p ∈ 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 ∈ P of orderDependencies then
        foreach s ∈ S of p do
            if s == p then
                return true
            else if orderConflictCheck(s, p) then
                return true
        end
    else
        return false
    end
```

ensures the correct sequence of parallel interactions. This must be considered for both order patterns.

Definition 4.3.2 *Possible position assignments for the interactions P and Q of a compliance pattern P LeadsTo Q are as following:*

Interaction P *= An interaction that has interactions on its subsequent paths that will always be reached.*
Interaction Q *= An interaction that will always be reached after Interaction P has been reached.*

Definition 4.3.3 *Possible position assignments for the interactions P and Q of a compliance pattern P Precedes Q are as following:*

Interaction P *= An interaction that is always reached prior to Interaction Q.*
Interaction Q *= An interaction that has interactions on its preceding path that were always reached prior to Interaction Q.*

The two occurrence patterns *Universal* and *Exists* differ only in that in for *P Universal*, the position of interaction P must always be reached throughout process execution, whereas for *P Exists* the position of interaction must only be reachable or in other words must be defined within the model.

Definition 4.3.4 *Possible position assignments for the interaction P of a compliance pattern P Universal are as following:*

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

Definition 4.3.5 *Possible position assignments for the interaction P of a compliance pattern P Exists are as following:*

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

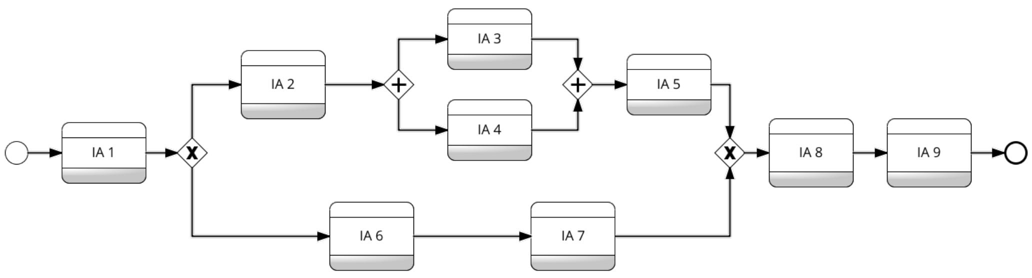

Figure 4.9.: Compliance Rule Assignment

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. For example, consider the choreography model shown in Figure 4.9 and the following compliance rules within which interaction B and C are each specified in the compliance rules *CR-1* and *CR-2*:

- *CR-1*: Interaction A LeadsTo Interaction B
- *CR-2*: Interaction C Precedes Interaction B
- *CR-3*: Universal Interaction C

The three compliance rules form two valid order sequences between the three specified interactions: Interaction A $\rightarrow$ Interaction C $\rightarrow$ Interaction B and Interaction C $\rightarrow$ Interaction A $\rightarrow$ Interaction B. More than one valid order indicates that there is no strict sequence between some defined interactions. For example, within the three stated compliance rules there is no determined sequence between *Interaction A* and *Interaction C*. Thus, that in this case, the two interactions can be also assigned to paths that are parallel to one another. If there is more than one possible order, the implemented procedure choses one by random.

After the interaction order is set, for every specified compliance rule the possible model positions are determined independently, based on the rules defined in Definitions 4.3.2 to 4.3.5. The result of this step is shown in Tables 4.2 to 4.4. These two steps are the preconditions for the actual assignment of the interactions into the existing model, which is shown in Algorithm 8. The assignment procedure iterates over the interaction order and for each interaction the intersection of the possible assignments of all affected compliance rules (commonPossibleAssignments) 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. For instance, let the order of the example compliance rules be C $\rightarrow$ A $\rightarrow$ B and let *Interaction C* be already assigned to *IA 1* and *Interaction A* to *IA 3* of the model shown in Figure 4.9. In order to determine the possible model positions for *Interaction B*, the intersection of the sets of possible positions for *Interaction B* of the affected rules *CR-1* and *CR-2* has to be determined:

- *CR-1*: {IA5, IA8, IA9}
- *CR-2*: {IA2, IA3, IA4, IA5, IA6, IA7, IA8, IA9}
- *CR-1* $\cap$ *CR-2*: {IA5, IA8, IA9}

Algorithm 8: Compliance Rules Assignment

```
   Input  : interactionOrder ← ordered CR interactions
1  begin
2      determinePossibleAssignments()
3      foreach interaction in interactionOrder do
4          if !assignInteraction(interaction) then
5              return false
6          end
7      end
8  end
9  Function assignInteraction(Interaction ia)
10     affectedCRs ← all compliance rules with ia involved
11     crPossibleAssignments ← HashMap<cr, model interactions>
12     modelAssignments ← HashMap<cr interaction, model interaction>
13     foreach cr in affectedCRs do
14         if ia is specified P of cr then
15             crPossibleAssignments ← add all possible model interactions for P
16         else if ia is specified Q of cr then
17             pAssignment ← already assigned model interaction of P
                 crPossibleAssignments ← add all possible model interactions of Q
                 for the given P
18     end
19     commonPossibleAssignments ← common model interactions between all
         crPossibleAssignment entires
20     if commonPossibleAssignments is not empty then
21         selectedInteraction ← get interaction with most succeeding interactions
             out of commonPossibleAssignments
22         modelAssignments ← add ia with selected model interaction
23         return true
24     else
25         return false
26     end
```

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, which triggers a new build process from the beginning (see Algorithm 1 - line 6). Is the intersection of possible model positions not empty, the procedure choses the interaction that has the most interactions on it's succeeding path, or in terms of RPST, the highest ranked trivial fragment on the hierarchy. 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.

Interaction A	Interaction B
IA1	{IA8, IA9}
IA2	{IA3, IA4, IA5, IA8, IA9}
IA3	{IA5, IA8, IA9}
IA4	{IA5, IA8, IA9}
IA5	{IA8, IA9}
IA6	{IA7, IA8, IA9}
IA7	{IA8, IA9}
IA8	{IA9}
IA9	{}

Table 4.2.: Possible assignment combinations for CR-1

Interaction C	Interaction B
IA1	{IA2, IA3, IA4, IA5, IA6, IA7, IA8, IA9}
IA2	{IA3, IA4, IA5}
IA3	{IA5}
IA4	{IA5}
IA5	{}
IA6	{IA7}
IA7	{}
IA8	{IA9}
IA9	{}

Table 4.3.: Possible assignment combinations for CR-2

Interaction C
IA1
IA8
IA9

Table 4.4.: Possible assignments for CR-3

4.3.4. Deriving the Collaboration Models

Following the introduced *top-down approach*, the collaboration model as well as the public and private models of each partner are derived from the generated choreography model. As already mentioned, the public models are projections of the choreography model, which means that they represent the view on the collaboration process from the perspective of each involved participant, while focusing on the process of one participant at a time. They only include activities that are necessary to communicate with other process participants. The private models are then enhanced versions of the public models. Additionally, they include internal process activities that don't involve interaction with other participants and that are not relevant for

other participants to know. At last, the collaboration model is the interconnection between the public models of each participant. Together they form a holistic view on the whole collaboration process and is therefore a different representation of the choreography model, without any information loss.
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. In the public model of the initiating participant of an interaction, a send task is inserted and in the model of the receiving participant a corresponding receive task. Additionally, for each public model, it is tried to reduce the model's sequence flow as much as possible without violating the underlying, predetermined sequence flow of the choreography model. 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.

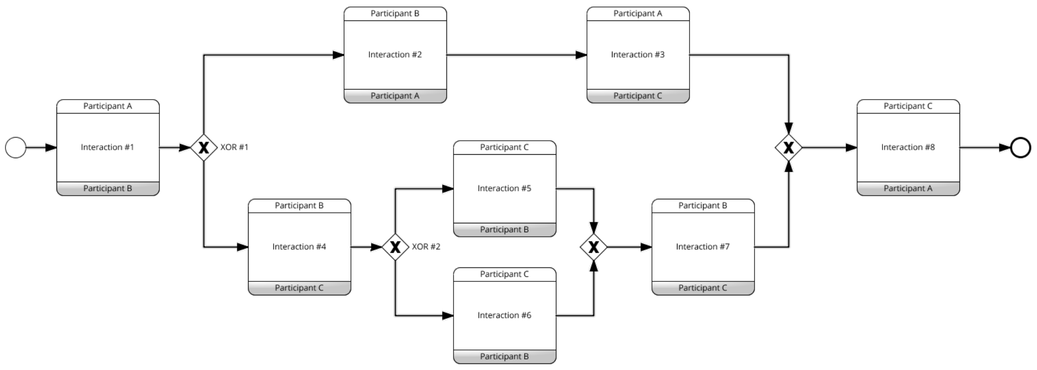

Figure 4.10.: Choreography Model

For instance, the choreography model shown in figure 4.10 would result in the three public models shown in Figures 4.11 to 4.13. The public model of Participant A (figure 4.11) is the only model where the sequence flow can be reduced in comparison to the choreography model. Because the lower path of the exclusive gateway *XOR #1* does not involve interactions that affect participant A, the whole path with all it's subsequent paths are not necessary and therefor not put into the public model.

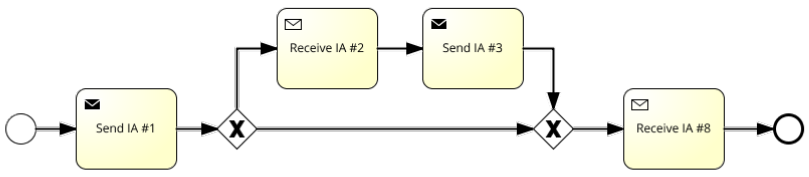

Figure 4.11.: Public Model - Participant A

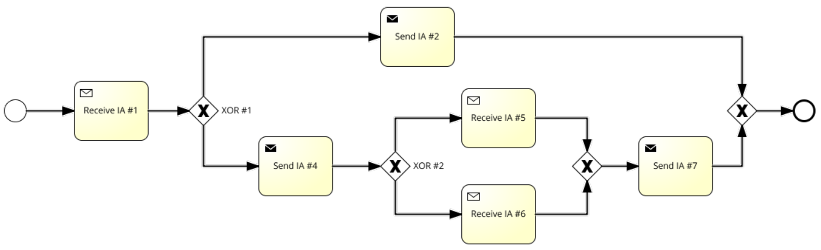

Figure 4.12.: Public Model - Participant B

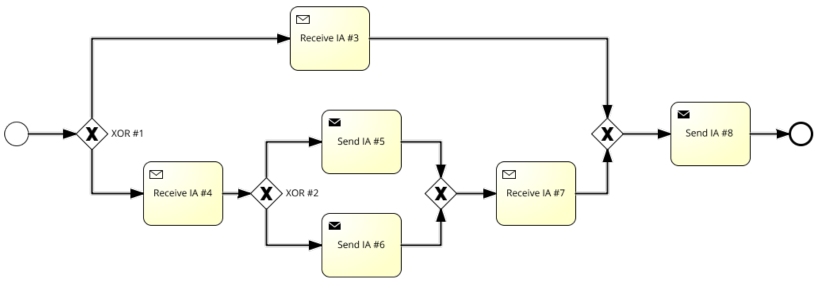

Figure 4.13.: Public Model - Participant C

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 could also be used as private models without the enrichment, but because in real process scenarios, it is not likely that a participant does only perform public, interacting tasks, this enrichment is implemented. Figure 4.14 shows a possible outcome of a private model for participant A.

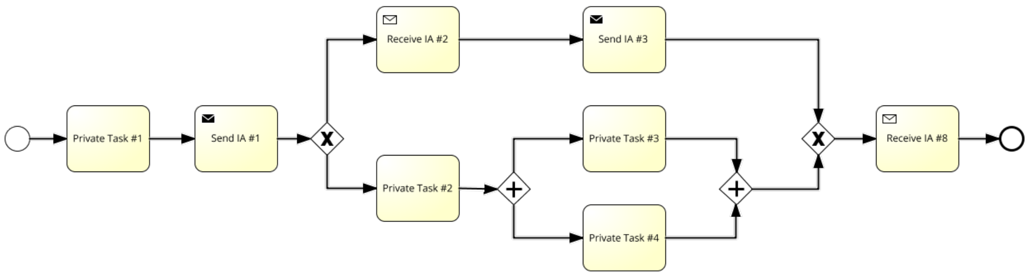

Figure 4.14.: Possible Private Model - Participant A

4.4. Model Translation to BPMN/XML

BPMN is the standard for describing business processes of any kind and therefore it is necessary to translate the generated process collaboration models in order to make them shareable and usable by a broad audience. Also, if some conventions are respected, the translated private models are out of the box executable and therefore the process can be tested by typical process engines. 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 (see Table 5.1) of the different model types and a process for generating a valid BPMN file must be designed. It has to be mentioned that the resulting BPMN/XML files only contain the formal process description. A generation of a the graphical process description will not take place. The following Table 4.5 shows the mapping of model type independent objects. These objects are used in all different model types and are therefore also common in the different BPMN models. Each model object is referenced by or has a reference to one or more other objects. For instance, the edges between flow objects do always refer to the flow objects (tasks and gateways) that are connected through this edge or, in case of an interaction, the message being sent is referenced. For this purpose, each model object has a unique id that also must be generated during the translation process.

Internal Object	**BPMN/XML Element**
Participant	`<participant id="(unique-id)" name="(name)"/>`
Start Event	`<startEvent id="(unique-id)" name="">`
	`  <outgoing>ref-to-sequenceFlow</outgoing>`
	`</startEvent>`
End Event	`<endEvent id="(unique-id)" name="">`
	`  <incoming>ref-to-sequenceFlow</incoming>`
	`</startEvent>`
Parallel Gateway	`<parallelGateway id="(unique-id)" name=""`
	`  gatewayDirection="Diverging">`
	`      <incoming>ref-to-sequenceFlow</incoming>`
	`      <outgoing>ref-to-sequenceFlow</outgoing>`
	`      <outgoing>ref-to-sequenceFlow</outgoing>`
	`</parallelGateway>`
	`<parallelGateway id="(unique-id)" name=""`
	`  gatewayDirection="Converging">`
	`      <incoming>ref-to-sequenceFlow</incoming>`
	`      <incoming>ref-to-sequenceFlow</incoming>`
	`      <outgoing>ref-to-sequenceFlow</outgoing>`
	`</parallelGateway>`
Exclusive Gateway	`<exclusiveGateway id="(unique-id)" name=""`
	`  gatewayDirection="Diverging">`
	`      <incoming>ref-to-sequenceFlow</incoming>`
	`      <outgoing>ref-to-sequenceFlow</outgoing>`
	`      <outgoing>ref-to-sequenceFlow</outgoing>`
	`</exclusiveGateway>`
	`<exclusiveGateway id="(unique-id)" name=""`
	`  gatewayDirection="Converging">`
	`      <incoming>ref-to-sequenceFlow</incoming>`
	`      <outgoing>ref-to-sequenceFlow</outgoing>`
	`      <outgoing>ref-to-sequenceFlow</outgoing>`
	`</exclusiveGateway>`
Edge	`<sequenceFlow id="(unique-id)" name=""`
	`  sourceRef="(ref-to-flow-object)"`
	`    targetRef="(ref-to-flow-object)"/>`
Message	`<message id="(unique-id)" name=""/>`
	`<messageFlow id="(unique-id)" messageRef="(ref-to-message)"`
	`        sourceRef="(ref-to-sendTask/participant)"`
	`        targetRef="(ref-to-receiveTask/participant)"/>`

Table 4.5.: Mapping Common Model Elements to BPMN/XML

In the following, for each model type, the mapping of the internal model elements to the corresponding BPMN elements is explained. Because collaboration, public and private model share the same BPMN structure, they are consolidated in one chapter.

4.4.1. Choreography Model

The flow object interaction is the only unique object in choreography models. It references the connecting edges (sequenceFlows), the messageFlow and the two participants, that are interacting. Table 4.6 shows the structure of the corresponding BPMN/XML object.

Internal Object	BPMN/XML Element

Interaction	`<choreographyTask id="" name="" initiatingParticipantRef="">`
	`<incoming>ref-to-sequenceFlow</incoming>`
	`<outgoing>ref-to-sequenceFlow</outgoing>`
	`<participantRef>ref-to-participant</participantRef>`
	`<participantRef>ref-to-participant</participantRef>`
	`<messageFlowRef>ref-to-messageFlow</messageFlowRef>`
	`</choreographyTask>`

Table 4.6.: Mapping Choreography Model to BPMN/XML

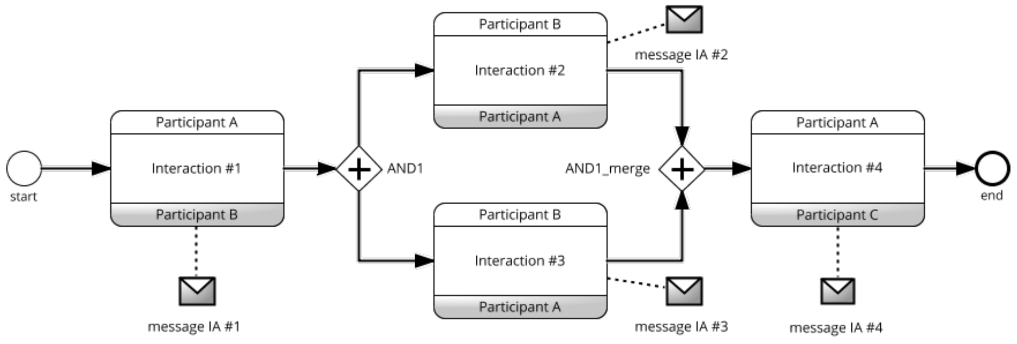

Figure 4.15.: BPMN/XML Translation Example - Choreography Model

The BPMN/XML shown in Example 4.1 represents the valid translation of the example choreography model shown in Figure 4.15. Independent from the model type, every BPMN/XML has the root element *definitions* with the namespace and schema declarations. The actual model objects are located inside the *choreography* element as already explained above. Because all different model types share a common xml structure convention and other models, e.g. collaboration models, can have more than one process described inside one BPMN/XML, data objects (e.g. messages) are located outside the actual process definitions, to be referenced by every including process.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance" typeLanguage="http://www.w3.org/2001/XMLSchema"
    xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL␣http://www.omg.org/
    spec/BPMN/2.0/20100501/BPMN20.xsd">
  <message id="m111" name="message␣IA␣#1"/>
  <message id="m112" name="message␣IA␣#2"/>
  <message id="m113" name="message␣IA␣#3"/>
  <message id="m113" name="message␣IA␣#4"/>
  <choreography id="c111">
    <participant id="p111" name="Participant␣A"/>
    <participant id="p112" name="Participant␣B"/>
    <participant id="p113" name="Participant␣C"/>
    <messageFlow id="mf111" messageRef="m111" sourceRef="p111" targetRef="p112"/>
    <messageFlow id="mf112" messageRef="m112" sourceRef="p112" targetRef="p111"/>
    <messageFlow id="mf113" messageRef="m113" sourceRef="p112" targetRef="p111"/>
    <messageFlow id="mf114" messageRef="m113" sourceRef="p112" targetRef="p111"/>
    <choreographyTask id="ct111" name="Interaction␣#1" initiatingParticipantRef="p111">
      <incoming>sf111</incoming>
      <outgoing>sf112</outgoing>
      <participantRef>p111</participantRef>
      <participantRef>p112</participantRef>
      <messageFlowRef>mf111</messageFlowRef>
    </choreographyTask>
    <choreographyTask id="ct112" name="Interaction␣#2" initiatingParticipantRef="p112">
      <incoming>sf113</incoming>
      <outgoing>sf115</outgoing>
      <participantRef>p112</participantRef>
      <participantRef>p111</participantRef>
      <messageFlowRef>mf112</messageFlowRef>
    </choreographyTask>
    <choreographyTask id="ct113" name="Interaction␣#3" initiatingParticipantRef="p112">
      <incoming>sf114</incoming>
      <outgoing>sf115</outgoing>
      <participantRef>p112</participantRef>
      <participantRef>p111</participantRef>
      <messageFlowRef>mf113</messageFlowRef>
    </choreographyTask>
    <choreographyTask id="ct114" name="Interaction␣#4" initiatingParticipantRef="p111">
      <incoming>sf117</incoming>
      <outgoing>sf118</outgoing>
      <participantRef>p111</participantRef>
      <participantRef>p113</participantRef>
      <messageFlowRef>mf114</messageFlowRef>
    </choreographyTask>
    <startEvent id="e111" name="start">
      <outgoing>sf111</outgoing>
    </startEvent>
    <endEvent id="e112" name="end">
      <incoming>sf117</incoming>
    </endEvent>
    <parallelGateway id="g111" name="AND1" gatewayDirection="Diverging">
      <incoming>sf112</incoming>
      <outgoing>sf113</outgoing>
      <outgoing>sf114</outgoing>
    </parallelGateway>
    <parallelGateway id="g112" name="AND1_merge" gatewayDirection="Converging">
      <incoming>sf115</incoming>
      <incoming>sf116</incoming>
      <outgoing>sf117</outgoing>
    </parallelGateway>
    <sequenceFlow id="sf111" name="" sourceRef="e111" targetRef="ct111"/>
    <sequenceFlow id="sf112" name="" sourceRef="ct111" targetRef="g111"/>
    <sequenceFlow id="sf113" name="" sourceRef="g111" targetRef="ct112"/>
    <sequenceFlow id="sf114" name="" sourceRef="g111" targetRef="ct113"/>
    <sequenceFlow id="sf115" name="" sourceRef="ct112" targetRef="g112"/>
    <sequenceFlow id="sf116" name="" sourceRef="ct113" targetRef="g112"/>
    <sequenceFlow id="sf117" name="" sourceRef="g112" targetRef="ct114"/>
    <sequenceFlow id="sf118" name="" sourceRef="ct114" targetRef="e112"/>
  </choreography>
</definitions>
```

Example 4.1: BPMN Choreography Model Example

4.4.2. Collaboration / Public / Private Models

As mentioned in Chapter 3, in a collaborative scenario, the collaboration, public and private models share the same BPMN/XML structure, because in each of the model types, there is at least one interactive task (send / receive messages) and a corresponding message flow (sender, receiver and message) that must be described within the BPMN/XML. Each partner can also decide if they want to share the private tasks within the public model and therefore also in the collaboration model, that is essentially a combined view of all partners public models. Because of this, the three model types can comprise all the BPMN objects shown in Table 4.7.

Internal Object	**BPMN/XML Element**
Send Task	`<sendTask id="(unique-id)" name="(name)">` `<incoming>ref-to-sequenceFlow</incoming>` `<outgoing>ref-to-sequenceFlow</outgoing>` `</sendTask>`
Receive Task	`<receiveTask id="(unique-id)" name="(name)">` `<incoming>ref-to-sequenceFlow</incoming>` `<outgoing>ref-to-sequenceFlow</outgoing>` `</receiveTask>`
Private Task	`<task id="(unique-id)" name="(name)">` `<incoming>ref-to-sequenceFlow</incoming>` `<outgoing>ref-to-sequenceFlow</outgoing>` `</task>`

Table 4.7.: Mapping Collaboration Model to BPMN/XML

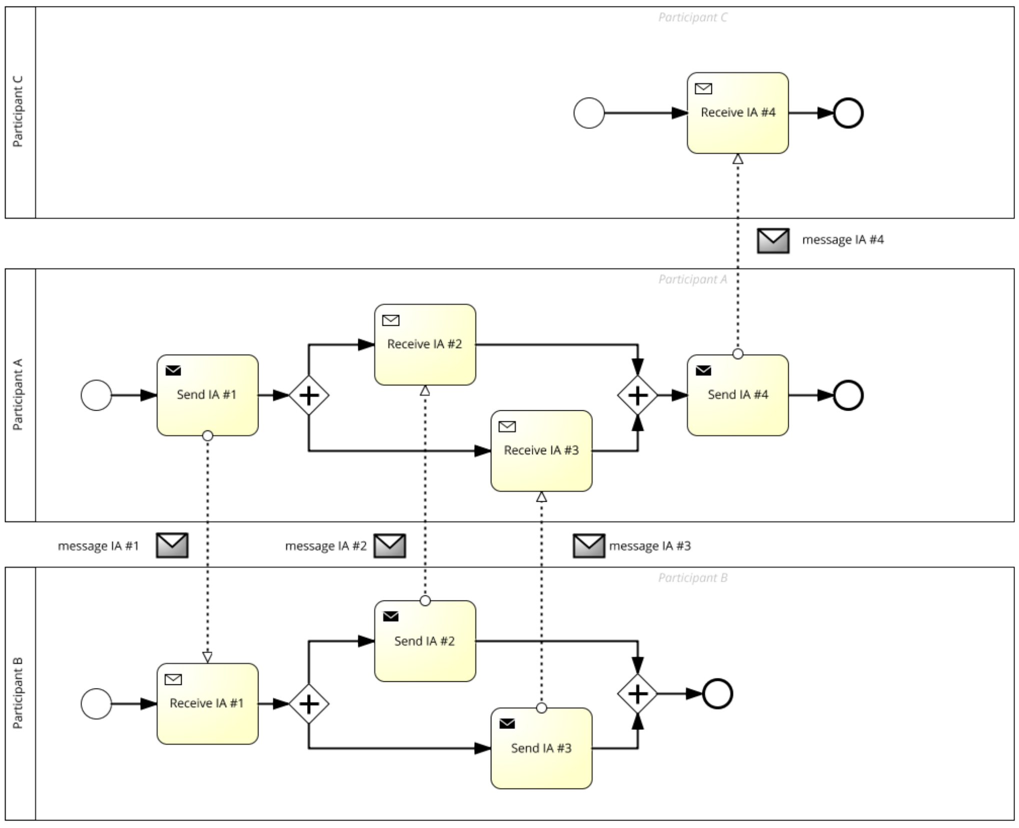

Figure 4.16.: BPMN/XML Translation Example - Collaboration Model

The BPMN/XML shown in Example 4.2 represents the valid translation of the collaboration model shown in Figure 4.16. Following the BPMN/XML convention, the root element is included as well the *definitions* with the namespace and schema declarations. A layer below, the process independent data objects *message*, the collaboration specific objects and a *process* for each participant are defined. Within the *collaboration* element, the participants, with a reference to their corresponding process and the message flow of the involving interacting tasks are declared. The *messageFlow* object references the send and receive tasks that exchange the message. Or, if a process of a participant is represented as a black box, the *messageFlow* object can also reference the process itself. In case of public or private models, where it is also possible that only one process of a specific participant is included, the *messageFlow* object references the participant that sends or receives the message.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance" typeLanguage="http://www.w3.org/2001/XMLSchema"
    xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL␣http://www.omg.org/
    spec/BPMN/2.0/20100501/BPMN20.xsd">
  <message id="m1" name="message␣IA␣#1"/>
  <message id="m2" name="message␣IA␣#2"/>
  <message id="m3" name="message␣IA␣#3"/>
  <message id="m4" name="message␣IA␣#4"/>
  <collaboration id="collab1">
    <participant id="p1" name="Participant␣A" processRef="pr1"/>
    <participant id="p2" name="Participant␣B" processRef="pr2"/>
    <participant id="p3" name="Participant␣C" processRef="pr3"/>
    <messageFlow id="mf1" messageRef="m1" name="" sourceRef="t1" targetRef="t5"/>
    <messageFlow id="mf2" messageRef="m2" name="" sourceRef="t8" targetRef="t2"/>
    <messageFlow id="mf3" messageRef="m3" name="" sourceRef="t9" targetRef="t3"/>
    <messageFlow id="mf4" messageRef="m4" name="" sourceRef="t4" targetRef="pr3"/>
  </collaboration>
  <process id="pr1" name="Participant␣A">
    <sendTask id="t1" name="Send␣IA␣#1">
      <incoming>sf1</incoming>
      <outgoing>sf2</outgoing>
    </sendTask>
    <startEvent id="e1" name="start">
      <outgoing>sf1</outgoing>
    </startEvent>
    <parallelGateway gatewayDirection="Diverging" id="g1" name="AND1">
      <incoming>sf2</incoming>
      <outgoing>sf3</outgoing>
      <outgoing>sf4</outgoing>
    </parallelGateway>
    <receiveTask id="t2" name="Receive␣IA␣#2">
      <incoming>sf3</incoming>
      <outgoing>sf5</outgoing>
    </receiveTask>
    <receiveTask id="t3" name="Receive␣IA␣#3">
      <incoming>sf4</incoming>
      <outgoing>sf6</outgoing>
    </receiveTask>
    <sendTask id="t4" name="Send␣IA␣#4">
      <incoming>sf7</incoming>
      <outgoing>sf8</outgoing>
    </sendTask>
    <parallelGateway gatewayDirection="Converging" id="g2" name="AND1_merge">
      <incoming>sf5</incoming>
      <incoming>sf6</incoming>
      <outgoing>sf7</outgoing>
    </parallelGateway>
    <endEvent id="e2" name="end">
      <incoming>sf8</incoming>
    </endEvent>
    <sequenceFlow id="sf1" name="" sourceRef="e1" targetRef="t1"/>
    <sequenceFlow id="sf2" name="" sourceRef="t1" targetRef="g1"/>
    <sequenceFlow id="sf3" name="" sourceRef="g1" targetRef="t2"/>
    <sequenceFlow id="sf4" name="" sourceRef="g1" targetRef="t3"/>
    <sequenceFlow id="sf5" name="" sourceRef="t2" targetRef="g2"/>
    <sequenceFlow id="sf6" name="" sourceRef="t3" targetRef="g2"/>
    <sequenceFlow id="sf7" name="" sourceRef="g2" targetRef="t4"/>
    <sequenceFlow id="sf8" name="" sourceRef="t4" targetRef="e2"/>
  </process>
  <process id="pr2" name="Participant␣B">
```

```
        <startEvent id="e3" name="start">
            <outgoing>sf9</outgoing>
        </startEvent>
        <receiveTask id="t5" name="Receive_IA_#1">
            <incoming>sf9</incoming>
            <outgoing>sf10</outgoing>
        </receiveTask>
        <parallelGateway gatewayDirection="Diverging" id="g3" name="AND1">
            <incoming>sf10</incoming>
            <outgoing>sf11</outgoing>
            <outgoing>sf12</outgoing>
        </parallelGateway>
        <sendTask id="t6" name="Send_IA_#2">
            <incoming>sf11</incoming>
            <outgoing>sf13</outgoing>
        </sendTask>
        <sendTask id="t7" name="Send_IA_#3">
            <incoming>sf12</incoming>
            <outgoing>sf14</outgoing>
        </sendTask>
        <parallelGateway gatewayDirection="Converging" id="g4" name="AND1_merge">
            <incoming>sf13</incoming>
            <incoming>sf14</incoming>
            <outgoing>sf15</outgoing>
        </parallelGateway>
        <endEvent id="e4" name="end">
            <incoming>sf15</incoming>
        </endEvent>
        <sequenceFlow id="sf9" name="" sourceRef="e3" targetRef="t5"/>
        <sequenceFlow id="sf10" name="" sourceRef="t5" targetRef="g3"/>
        <sequenceFlow id="sf11" name="" sourceRef="g3" targetRef="t6"/>
        <sequenceFlow id="sf12" name="" sourceRef="g3" targetRef="t7"/>
        <sequenceFlow id="sf13" name="" sourceRef="t6" targetRef="t8"/>
        <sequenceFlow id="sf14" name="" sourceRef="t7" targetRef="t9"/>
        <sequenceFlow id="sf15" name="" sourceRef="t8" targetRef="g4"/>
    </process>
    <process id="pr3" name="Participant_C">
          <startEvent id="e5" name="start">
            <outgoing>sf16</outgoing>
          </startEvent>
          <receiveTask id="t8" name="Receive_IA_#4">
               <incoming>sf16</incoming>
            <outgoing>sf17</outgoing>
          </receiveTask>
          <endEvent id="e5" name="end">
            <incoming>sf17</incoming>
          </endEvent>
        <sequenceFlow id="sf16" name="" sourceRef="t9" targetRef="g4"/>
        <sequenceFlow id="sf17" name="" sourceRef="g4" targetRef="e4"/>
    </process>
</definitions>
```

Example 4.2: BPMN Choreography Model Example

4.5. Conclusion

In this chapter, the followed *top-down approach* of how to build a process collaboration, starting with the choreography model and then deriving the public and private models from it, was explained. It was also shown how the random choreography model generation can be influenced by specifying various build parameters as well as by imposing pattern-based compliance rules. Furthermore, all necessary components for generating sound choreography models that satisfy the three levels of correctness were introduced by explaining the involved generation algorithms as well as by pointing out the necessity of the *Model Tracking* component, in which the model is decomposed into splits, branches and nodes. Also, the indispensability of the thereby involved control flow logic, with it's status model for branches and the necessary functionality of calculating the exact proportion of interactions that are at free disposal and those that have already determined places within the evolving model or those that will be needed later by paths that will be created by not yet consumed gateways, were highlighted. The chapter was concluded by explaining how the internal model representation is translated to BPMN/XML. Thereby, the mapping of the internal components to the corresponding BPMN/XML elements was shown. In the next chapter, it will be explained how the introduced components of the random process collaboration generator are implemented within the CRISP framework.

5. Implementation

As part of the CRISP Project, a framework has been developed to test the elaborated change negotiation and propagation algorithms for collaborative processes. The framework already provides functionalities for importing process models in form of BPMN 2.0 XML, deriving public models out of choreography models as well as inserting new fragments into existing models. Within the importing process, the BPMN models are converted into a RPST without loosing any information on the control flow and connection objects of the original model. This resulting graph structure enables model manipulation and analysis techniques. In order to save the effort of implementing the same or similar graph model structure for the automatic process collaboration generator, it was decided to integrate it into the existing framework. Hence, the structure and it's complementary components and services can be utilized with minor adaptations.

The following chapter describes the internal process model representation structure and the implemented class structure of the automatic generator and translator.

5.1. Internal Process Model Representation

The framework internal process model representation utilizes the jBPT[1] library, which was developed by Polyvyanyy et. al [19] and facilitates the modeling of process models as RPST. Figure 5.1 shows an excerpt of the core structure of the library. The structure enables the creation of different types of graphs to support the capturing of various process modeling languages, such as petri nets, EPC[2] or BPMN.

For the purpose of constructing BPMN process models, an implementation of the *AbstractMultiDirectedGraph* class is suitable. A *multi directed graph* represents a graph whose vertices can be connected among themselves through multiple *directed* edges [20]. Figure 5.1 also shows that all graph models are typed with generics. In the context of a *multi directed graph* model, the parameter E is bound to a an instance of *IDirectedEdge*<V>, whereas parameter V is bound to an instance of

[1]Business Process Technologies for Java - https://code.google.com/p/jbpt/
[2]Event-driven process chains

IVertex.

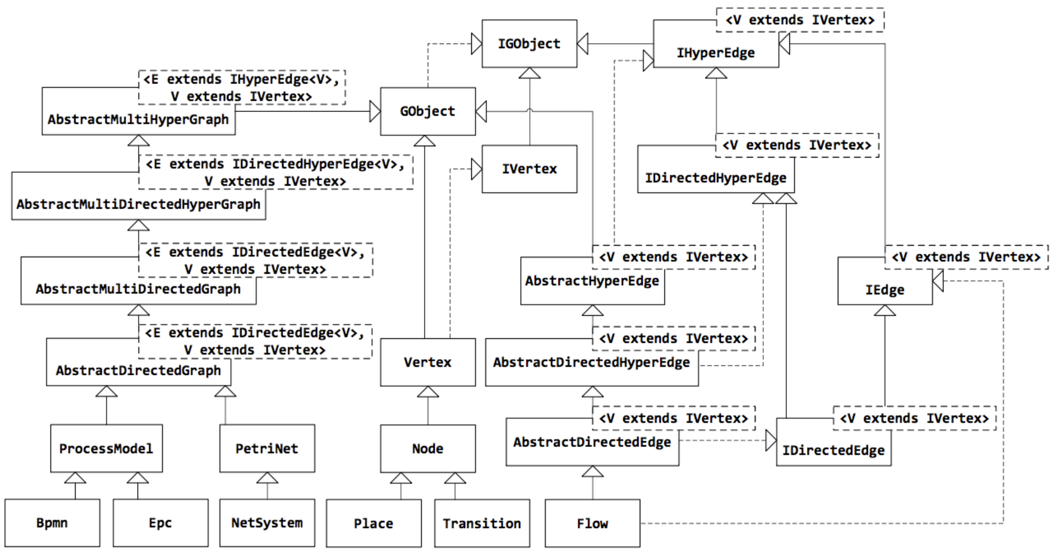

Figure 5.1.: Class and interface hierarchy of jBPT (Source: [19])

In order to utilize the jBPT structure to build any type of BPMN process model, all specific BPMN flow objects must therefor implement the *IVertex* interface. The flow objects, for instance tasks or gateways, are then in turn connected through Edges, which represent the BPMN sequence flow to create the graph and therefor the process model. How the jBPT library is incorporated and utilized by the framework is illustrated in figure 5.2. The intensive use of interfaces enables the reuse of flow objects that are part of different model types. For instance, gateways and events are used in all four model types. For the sake of clarity, Figure 5.2 only shows model specific flow objects that are appropriate for BPMN choreography models. Table 5.1 lists all BPMN flow objects, that are supported by the framework to create process collaborations and their different models.

	Choreography Model	Collaboration Model	Public Model	Private Model
Start Event	■	■	■	■
End Event	■	■	■	■
Parallel Gateway	■	■	■	■
Exclusive Gateway	■	■	■	■
Interaction	■	□	□	□
Task	□	■	■	■
Send Task	□	■	■	■
Receive Task	□	■	■	■

Table 5.1.: Overview of supported BPMN flow objects

Within the framework, all four model types support the flow objects *Start Event, End Event, Parallel Gateway* and *Exclusive Gateway.* These are basic control flow objects that are not specific to any model type in BPMN. Additionally available, for *collaboration, public* and *Private Models*, are the activities *Task, Send Task* and *Receive Task*. According to the BPMN 2.0 specification, the activity *Task* represents an *Abstract Task* [8]. An *Abstract Task* is a task that is not further specified. There are several further specified tasks in BPMN, including the also supported *Send* and *Receive Task*. Since the focus of the CRISP project is on collaborative processes and the message exchange between the participating partners, it is mandatory that these two messaging activities are also specified within the framework. For the remaining activities that are not part of the message flow, it is in this context not relevant if the task is actually a *User, Manual, Service* or *Script task*. The only flow object specific for *Choreography Models* is the *Choreography Task* or *Interaction*, the equivalent of a *Send* and *Receive task* sequence in a collaboration model.

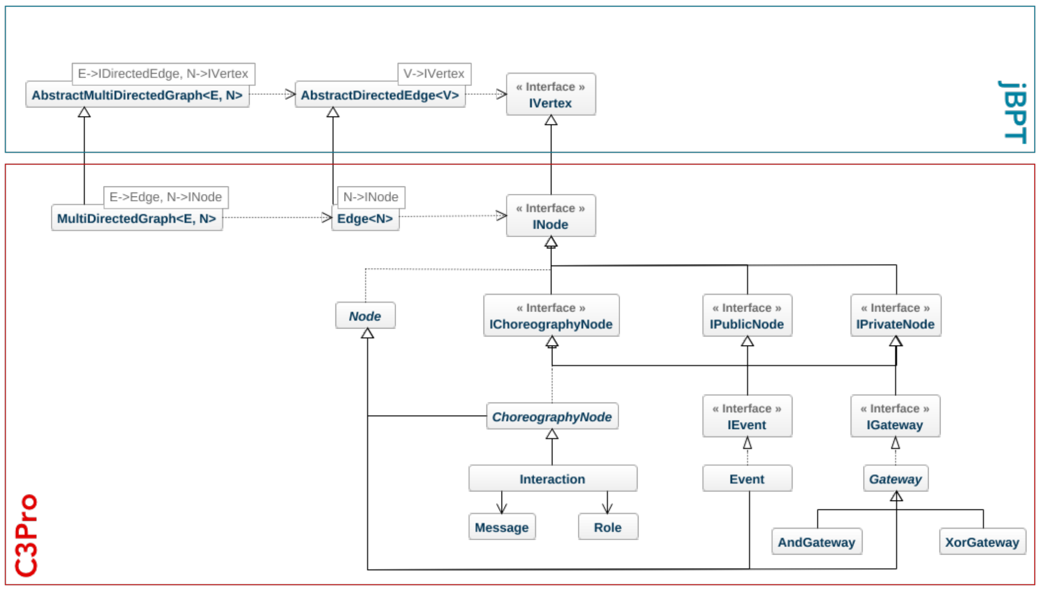

Figure 5.2.: Framework Model Representation Structure

5.2. Class Structure and Data Model

The diagram shown in Figure 5.3 represents the simplified class structure of the implemented components, that are necessary for generating an entire process collaboration, starting with the generation of the Choreography Model that complies to imposed compliance rules, leading to deriving the Public and Private Models out of it and finishing with the translation to BPMN/XML. The numbers indicate the order in which the components are instantiated. In the following, each class, their core functions and their relation with to the algorithms, introduced in the section of conception, will be described.

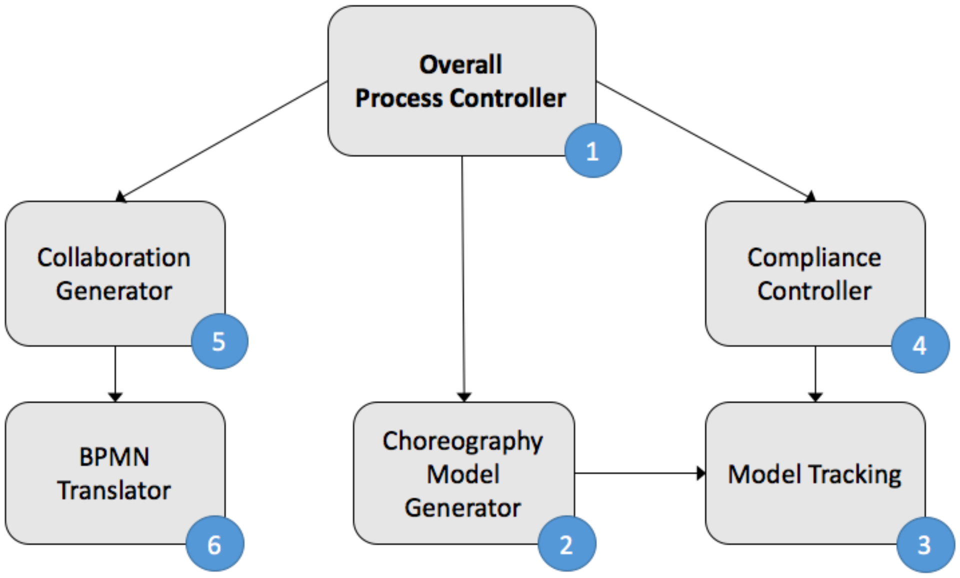

Figure 5.3.: Class Structure Random Collaboration Generator

The logic of coordinating the entire generation process (see Algorithm 1) is implemented in the class *CollaborationGenerationController*. It's purpose is to orchestrate the whole process and functionalities provided by the other components. Also the parametric build constraints and compliance rules, on which the collaboration generation is based, are specified within this class.

The class *ChoreographyModelGenerator* encapsulates the functionality of generating a random choreography model. Figure 5.4 shows the extended class structure of the *Choreography Model Generator* component and the important instance variables and implemented methods of each class. The main class of this component is the *ChoreographyModelGenerator* class. Within this, the algorithm for generating random models, explained in the chapter of conception (see Algorithm 6), is implemented within the *build()* method. The thereby used functions *getRandomNodeType()* (see Algorithm 2), *getRandomBranch()* (see Algorithm 3) and *getRandomBranchAmount()* are also implemented within this class, whereby all utilize methods that are implemented within the *ModelTracking* class. This includes mainly methods for updating the model and determining the amount of free and reserved interactions. Within the *ModelTracking* a set of splits is contained which represents the actual model as branches and associated nodes. The different node types all implement the *IChoreographyNode* interface.

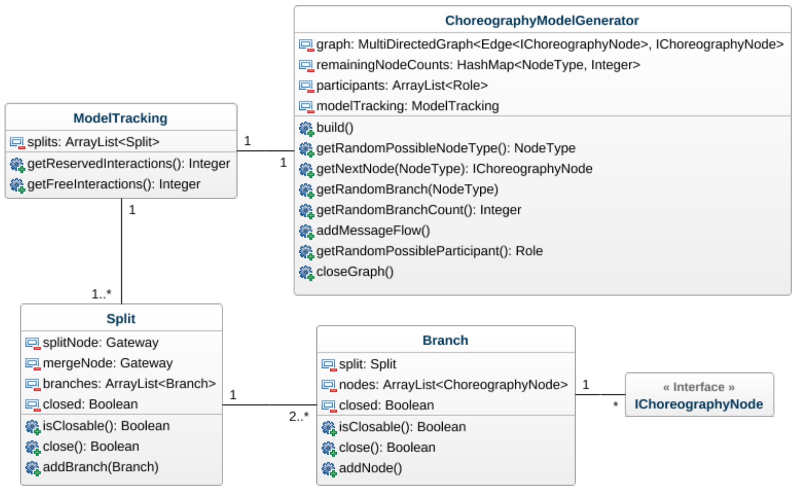
ChoreographyModelGenerator
graph: MultiDirectedGraph<Edge<IChoreographyNode>, IChoreographyNode>
remainingNodeCounts: HashMap<NodeType, Integer>
participants: ArrayList<Role>
modelTracking: ModelTracking
build()
getRandomPossibleNodeType(): NodeType
getNextNode(NodeType): IChoreographyNode
getRandomBranch(NodeType)
getRandomBranchCount(): Integer
addMessageFlow()
getRandomPossibleParticipant(): Role
closeGraph()
ModelTracking
splits: ArrayList<Split>
getReservedInteractions(): Integer
getFreeInteractions(): Integer
1
1
1
1..*
Split
splitNode: Gateway
mergeNode: Gateway
branches: ArrayList<Branch>
closed: Boolean
isClosable(): Boolean
close(): Boolean
addBranch(Branch)
1
2..*
Branch
split: Split
nodes: ArrayList<ChoreographyNode>
closed: Boolean
isClosable(): Boolean
close(): Boolean
addNode()
1
*
« Interface »
IChoreographyNode

Figure 5.4.: Class Structure - ChoreographyModelGenerator

The introduced logic of specifying and imposing global compliance rules on a choreography model is implemented within the class *ComplianceController*. It also utilizes the same instance of the *ModelTracking* class, that represents the finished model in order to find possible assignments for the imposed interaction order.

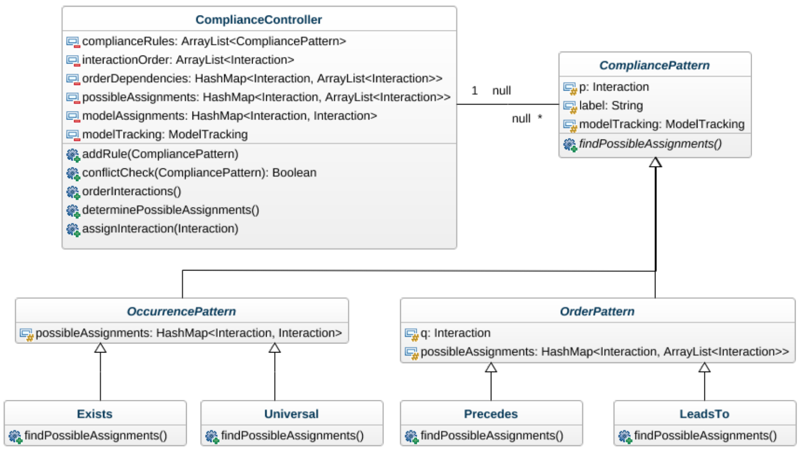
ComplianceController
complianceRules: ArrayList<CompliancePattern>
interactionOrder: ArrayList<Interaction>
orderDependencies: HashMap<Interaction, ArrayList<Interaction>>
possibleAssignments: HashMap<Interaction, ArrayList<Interaction>>
modelAssignments: HashMap<Interaction, Interaction>
modelTracking: ModelTracking
addRule(CompliancePattern)
conflictCheck(CompliancePattern): Boolean
orderInteractions()
determinePossibleAssignments()
assignInteraction(Interaction)
1 null
null *
CompliancePattern
p: Interaction
label: String
modelTracking: ModelTracking
findPossibleAssignments()
OccurrencePattern
possibleAssignments: HashMap<Interaction, Interaction>
OrderPattern
q: Interaction
possibleAssignments: HashMap<Interaction, ArrayList<Interaction>>
Exists
findPossibleAssignments()
Universal
findPossibleAssignments()
Precedes
findPossibleAssignments()
LeadsTo
findPossibleAssignments()

Figure 5.5.: Class Structure - ComplianceController

Figure 5.5 represents the class class diagram of the implemented component. Within the *ComplianceController* class, the procedure for conflict checking between the specified rules (see Algorithm 7) is implemented by the *conflictCheck(CompliancePattern)* method. The *assign()* method orchestrates the whole assignment process by fist triggering the *findPossibleAssigment()* method of each involved compliance pattern by calling the *determinePossibleAssignments()* method. The *findPossibleAssigment()* methods determines the possible model positions for each pattern type individually based on the rules described in Definitions 4.3.2 - 4.3.5. In the next step, the order of involved interactions is determined by the *orderInteractions()* method. Based on the resulting order, the interactions are tried to be assigned into the choreography model by the *assignInteraction(Interaction)* method (see Algorithm 8).

The class *CollaborationGenerator* provides the functionalities of deriving the public and private models from the generated choreography model. This component is already implemented within the framework and is therefor not described as a part of the implementation.

At last, the translation of the internal model representation to BPMN/XML, based on the mappings described in Section 4.4 of the conceptional chapter, is implemented within the *BPMNTranslator* class. The process is a typical XML generation process, utilizing the established jDOM[3] library.

5.3. Conclusion

In this chapter, the internal model representation and the therefore utilized jBPT library was explained. It was shown how the provided *AbstractMultiGraph* and its data model is extended in order to meet the requirements of BPMN choreography models. The chapter was concluded with an overview of how the different components are implemented within the existing framework and by linking the important methods with the algorithms introduced within the chapter of conception. In the next chapter, the implementation of the choreography generation process is examined by analyzing the performance and the resulting models based on different parameter settings.

6. Performance Analysis

In this chapter, the influence of different build parameter settings on the performance of the generation process and the resulting choreography models is examined. As measure for the performance of a choreography generation process that is constraint by parametric constraints as well as imposed compliance rules (see Chapter 4.2), the number of necessary iterations for a successful generation is adducted. The iteration limitation is thereby set to ten. After ten iterations, the choreography generation is marked as failed. For the examination of the generation process that is only constraint by parametric constraints, the necessary duration for a successful build is used as measure, because the generation algorithm ensures that a model generation

[3]Java library for XML manipulation

without imposed compliance rule will always result in a successful model within one iteration. In the following, first, the performance dependent on the parametric constraints is analyzed, and afterwards, the influence of different compliance rule settings, in conjunction with parametric constraints, is examined. In order to obtain representative results, each parameter configuration is run with 100 repetitions, if not stated otherwise.

Figure 6.1 illustrates the influence of the number of parallel gateways onto the duration of a choreography model generation with 100 interactions. It shows that the generation duration is increasing linear to the number of gateway nodes included in the model. It also shows that there is no significant difference between the respective gateway types and also the combination between parallel and exclusive gateways (equal proportion). The generation duration increase is expected, because the more gateways are in the model, the more branches are created, and the more branches are in the model, the longer it takes to loop through the model to determine the number of free or reserved interactions.

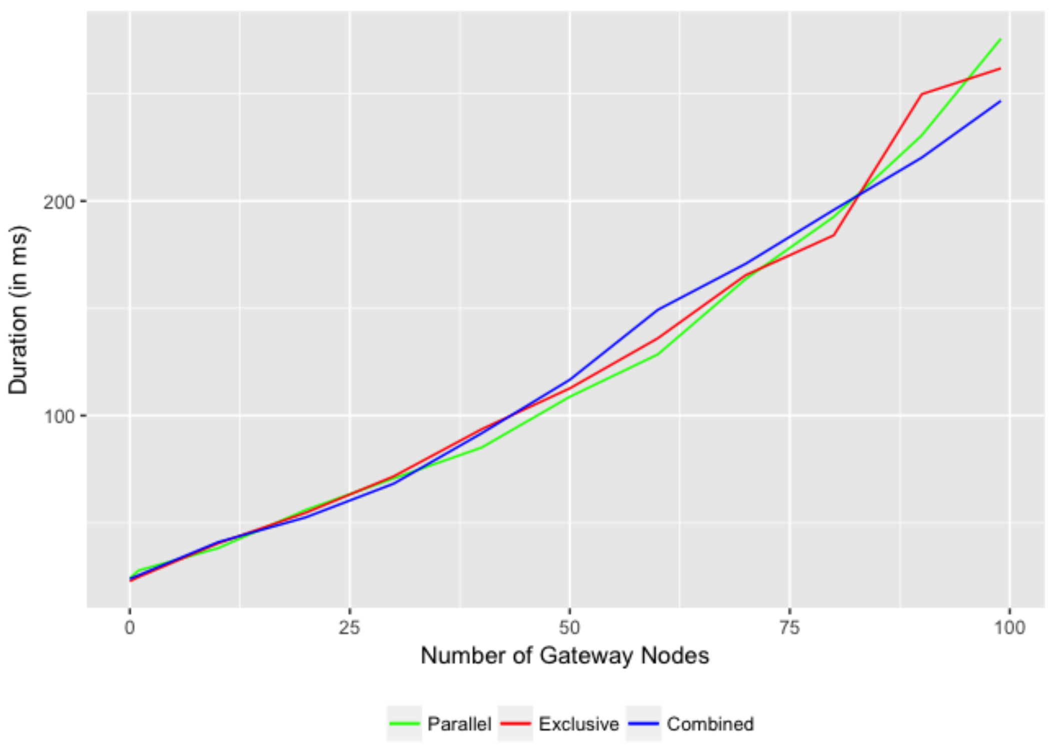

Figure 6.1.: Choreography model generation duration depending on the number of gateway nodes

Figure 6.2 illustrates the influence of the max. branching parameter (see Section 4.2.1) on the duration of a choreography model generation with 400 interactions and

20 gateway nodes. The graph shows that the generation duration is increasing linear to the increase of the max. branching parameter. The explanation for this is the same as for the influence of the number of gateway nodes. With the increase of the max. branching parameter, the number of branches in the build increases (premised that there are enough free interactions), and therefore the algorithms which loop through the branches consume more time.

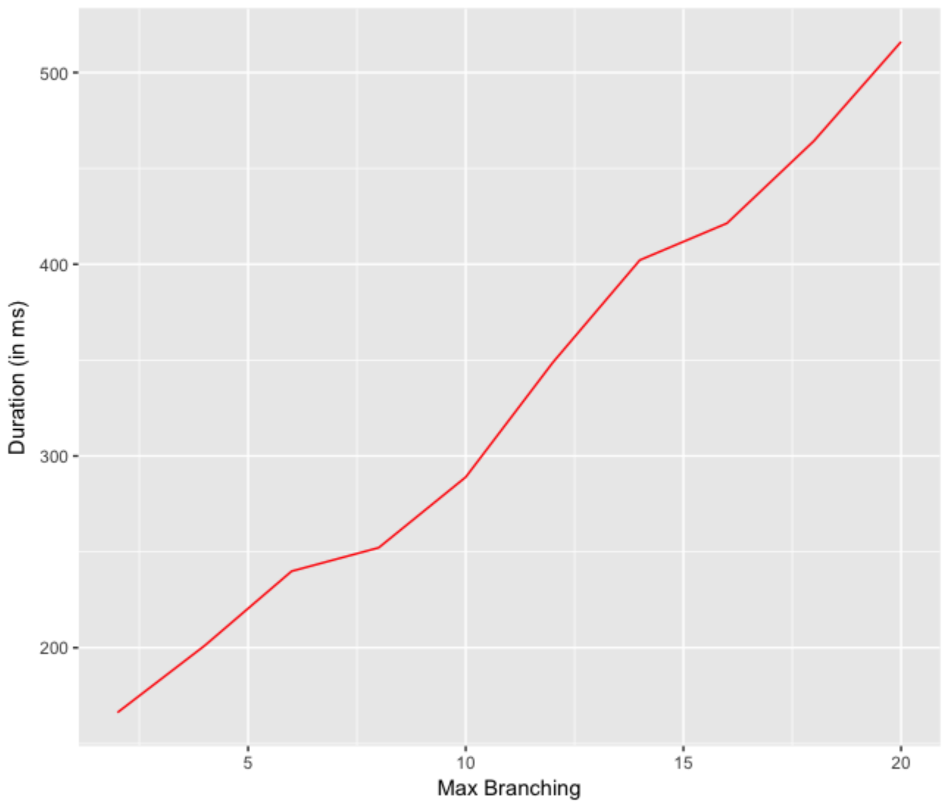

Figure 6.2.: Choreography model generation duration depending on the max. branching parameter

For the analysis of the influence of different compliance rules settings on the model generation process, a random compliance rule generator is used. By specifying a number of interactions and a number of compliance rules, the generator generates supported compliance rules between those interactions randomly, which are then tried to be imposed on the generated models. Figure 6.3 illustrates the influence of the number of compliance rules on a model generation process with 100 interactions, 10 parallel gateways and max. branching set to 2. It shows that the number of compliance rules has no impact on a model generation which only includes parallel gateways (various amounts of parallel gateway nodes were analyzed). Each model generation process was successful within the first iteration. This result is expected, because in a model without exclusive paths, all four supported compliance rule patterns (see Section 4.2.2), taken individually, can be assigned to any position in the model as long as they are one the same path (not parallel to another).

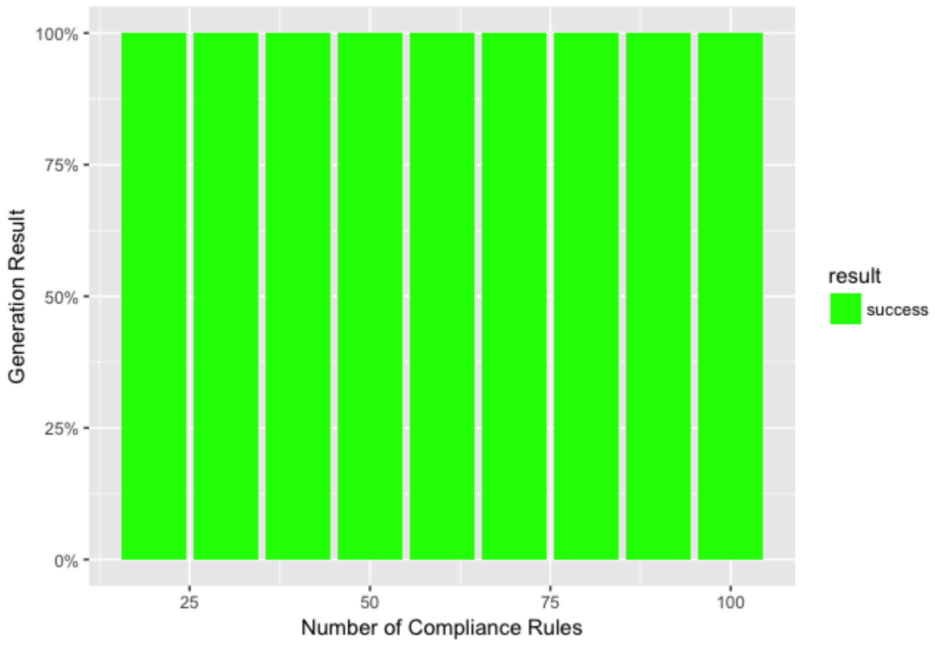

Figure 6.3.: Choreography model generation result depending on the number of compliance rules

This changes if a model generation includes exclusive gateways. Figure 6.4 illustrates the influence of the number of compliance rules on a model generation process with 100 interactions, 10 exclusive gateways and max. branching set to 2. On the one hand, the result shows that the number of imposed compliance rule has no significant influence on the model generation success. On the other hand, on average only approx. 24% of the generation processes are successful within 10 iterations. Figure 6.5 shows the necessary iterations for the successful model generation processes depending on the number of specified compliance rules. This result is also expected, because the supported compliance rule patterns *LeadsTo*, *Precedes* and *Universal* are hard to assign in a model with a lot of exclusive paths (see Definitions 4.3.2 - 4.3.4). This is confirmed by examining the influence of the number of exclusive gateway onto a generation process restricted by 60 compliance rules, which is illustrated in Figure 6.6. The result shows that the more exclusive gateways, hence more exclusive paths, the more often the generation process fails within the specified iteration limitation. A successful generation is thereby also influenced by the level of dependency among the compliance rules. A high dependency among order patterns result in a more strict sequence order between the involved interactions, which makes it more difficult to assign them successfully.

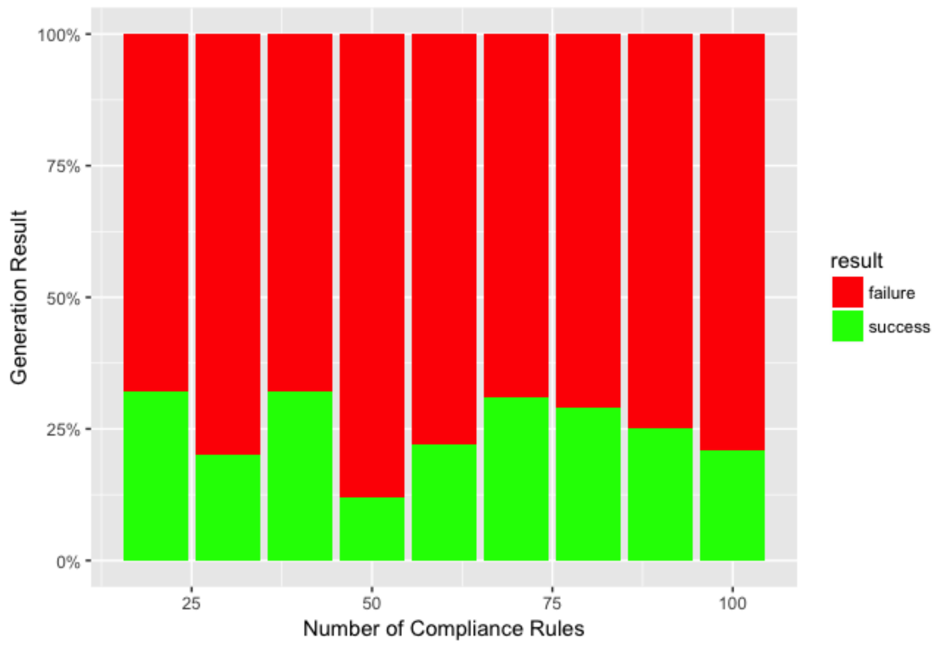

Figure 6.4.: Choreography model generation result depending on the number of compliance rules

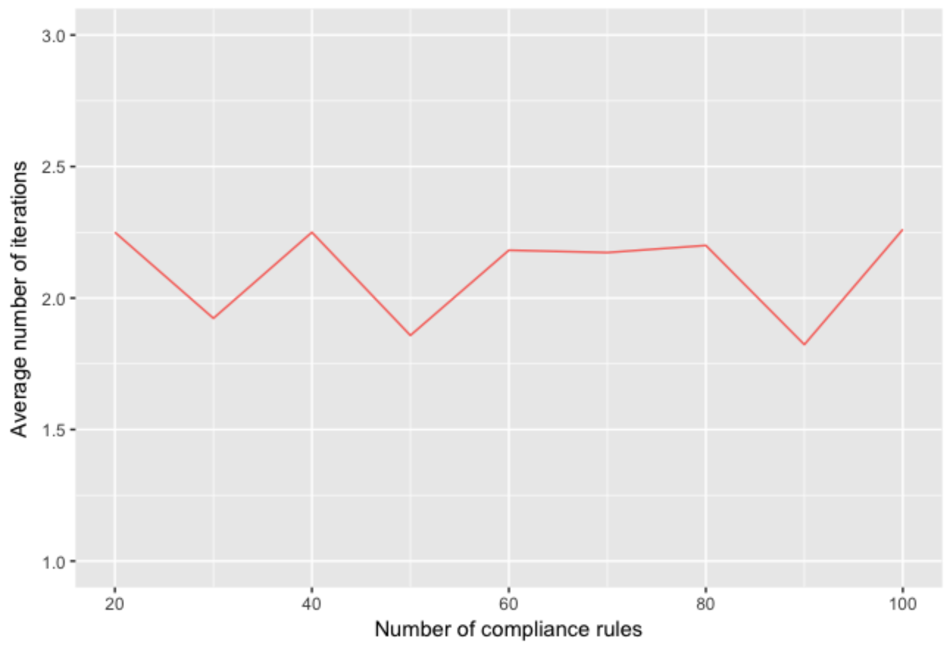

Figure 6.5.: Average iterations for successful generation

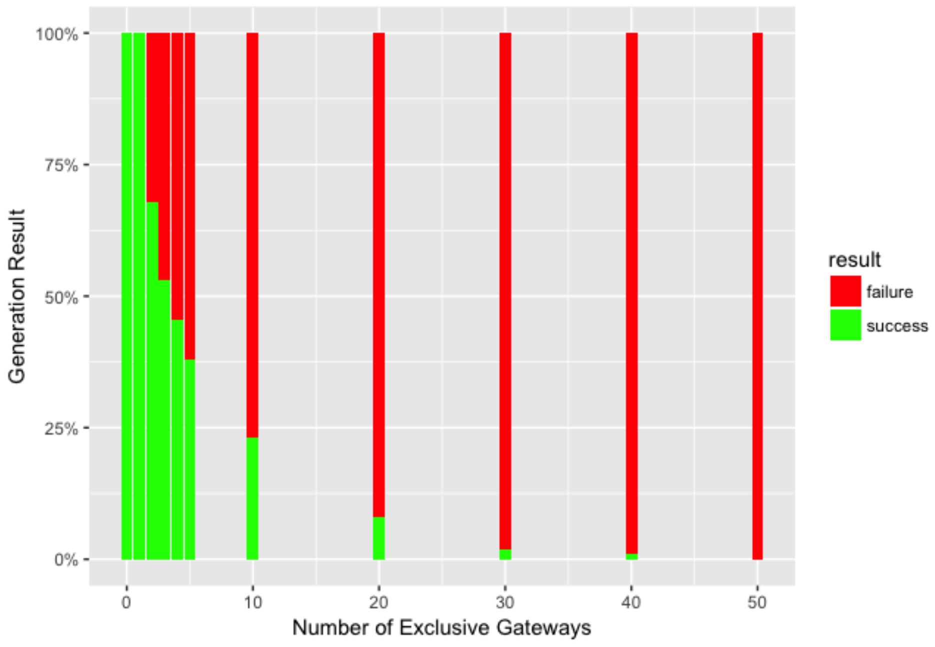

Figure 6.6.: Choreography generation result depending on the number of exclusive gateways and compliance rules

7. Conclusion and Future Work

7.1. Conclusion

In this work, the lack of available collaborative process models represented in BPM-N/XML for different research purposes, such as process mining or change propagation, was addressed. The main contribution within this work is the creation of a model repository by implementing a process generator which generates decentralized, cross-organizational models based on several, user-specified parameters. Thereby, an introduction into the two possible approaches of how to build collaborative processes, with all the models representing the different process perspectives, has been provided. It was argued that the followed *top-down approach*, if implemented correctly, already provides good preconditions for ensuring compatibility between the public models as well as consistency between the public and the private model of one partner by deriving the public from the generated choreography model and enhancing the public model with private tasks to obtain the private model. Additionally to the model-specific parameters, also the possibility of specifying and imposing compliance rules was examined and evaluated by implementing a prototypical compliance rule component which supports a small subset of established

compliance patterns in order to also examine the semantical correctness of the models. The main focus of this work was on the conception of the automatic generator and the therefor necessary components and algorithms. The benefit of expressing the models as a RPST was argued as well as the importance of tracking the model at any point during the generation process in order to comply with the user-specified parameters, which serve as boundaries for the random generation. Therefore, the *Model Tracking* component and the concept of already reserved but not yet consumed interactions was explained in detail. Due to the simplicity of the underlying algorithms, the deriving and enhancement of the public and private model, as well as the translation of the internal model representation to BPMN/XML was merely discussed superficially by explaining the node mapping between the different models as well as the mapping to the BPMN/XML elements.
Finally, a performance analysis was performed, which showed the influence of different build parameters on the performance of the generation process as well as on the result. Thereby, it was shown that the number of specified nodes influences the performance of the generation process linearly. But the analysis also exposed room for improvement of the generation process in general and especially regarding the specification and imposition of compliance rules, which will be addressed in the next chapter.

7.2. Future Work

In the context of the conducted performance analysis, it was exposed that for generation processes with imposed compliance rules, combined with an amount of exclusive gateways greater than five, the outcome was not satisfying. By generating and imposing random compliance rules at a ratio of min. 1 : 5 to the amount of interactions, the build was only successful at approx. 1 out of 4 times. This was not unexpected and is a result of the characteristics of the supported compliance patterns, which are more difficult to assign the more they are interdependent. This problem could be solved by supporting more compliance patterns, especially *P Exclusive Q*, which requires Interaction P to be on a different exclusive path than Interaction Q [16]. In addition to that, the overall performance of assigning the compliance rules onto the model, could be improved by introducing a caching concept for possible positions instead of looping through the whole model for each compliance rule, which can be highly time consuming if the model is large. It would also be interesting for this problem to utilize a constraint solving framework like *CHOCO*[1]. For the general build process, it could be conceivable to implement more parameters that allow the user to influence the model outcome even more. Thereby, for example, a parameter which influences the branch selection dependent on the degree of nested branching by dynamically favoring branches which are already highly nested or those that are not, depending on the parameter setting.

[1]An Open-Source java library for constraint programming - http://http://www.choco-solver.org/

List of Figures

List of Tables

Bibliography

[1] Fabian Friedrich, Jan Mendling, and Frank Puhlmann. Process model generation from natural language text. In Haralambos Mouratidis and Colette Rolland, editors, *Advanced Information Systems Engineering*, pages 482–496, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Krzysztof Kluza and Krzysztof Honkisz. From sbvr to bpmn and dmn models. proposal of translation from rules to process and decision models. In *ICAISC*, 2016.

[3] D. Lubke, K. Schneider, and M. Weidlich. Visualizing use case sets as bpmn processes. In *2008 Requirements Engineering Visualization*, pages 21–25, Sept 2008.

[4] Usman Zafar, Moshiur Bhuiyan, P. W. C. Prasad, and Farzana Haque. Integration of use case models and BPMN using goal- oriented requirements engineering. *JCP*, 13(2):212–221, 2018.

[5] Anna Suchenia, Krzysztof Kluza, Krystian Jobczyk, Piotr Wiśniewski, Michal Wypych, and Antoni Ligeza. Supporting bpmn process models with uml sequence diagrams for representing time issues and testing models, 05 2017.

[6] Piotr Wisniewski, Krzysztof Kluza, and Antoni Ligeza. An approach to participatory business process modeling: Bpmn model generation using constraint programming and graph composition. *Applied Sciences*, 8(9), 2018.

[7] Andrea Burattin. PLG2: multiperspective process randomization with online and offline simulations. In *Proceedings of the BPM Demo Track 2016 Co-located with the 14th International Conference on Business Process Management (BPM 2016), Rio de Janeiro, Brazil, September 21, 2016.*, pages 1–6, 2016.

[8] Object Management Group (OMG). Business process model and notation (bpmn) version 2.0. Technical report, Object Management Group (OMG), January 2011.

[9] Walid Fdhila, Stefanie Rinderle-Ma, David Knuplesch, and Manfred Reichert. Change and compliance in collaborative processes. In *12th IEEE International Conference on Services Computing (SCC 2015)*, pages 162–169. IEEE Computer Society Press, June 2015.

[10] David Knuplesch, Manfred Reichert, Jürgen Mangler, Stefanie Rinderle-Ma, and Walid Fdhila. Towards compliance of cross-organizational processes and their changes. In *1st Int Workshop on Security in Business Processes (SBP?12), BPM'12 Workshops*, number 132 in LNBIP, pages 649–661. Springer, September 2012.

[11] Walid Fdhila, Conrad Indiono, Stefanie Rinderle-Ma, and Manfred Reichert. Dealing with change in process choreographies: Design and implementation of propagation algorithms. *Information Systems*, 49:1 – 24, 2015.

[12] Aditya Ghose and George Koliadis. *Auditing Business Process Compliance*, pages 169–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[13] G. Governatori, Z. Milosevic, and S. Sadiq. Compliance Checking Between Business Processes and Business Contracts. In *10th International Enterprise Distributed Object Computing Conference (EDOC 2006)*, pages 221–232. IEEE Computing Society, 2006.

[14] David Knuplesch, Manfred Reichert, Linh Thao Ly, Akhil Kumar, and Stefanie Rinderle-Ma. Visual modeling of business process compliance rules with the support of multiple perspectives. In *32nd Int'l Conference on Conceptual Modeling (ER 2013)*, number 8217 in LNCS, pages 106–120. Springer, November 2013.

[15] Ahmed Awad, Gero Decker, and Mathias Weske. *Efficient Compliance Checking Using BPMN-Q and Temporal Logic*, pages 326–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[16] O. Turetken, A. Elgammal, W. J. van den Heuvel, and M. P. Papazoglou. Capturing compliance requirements: A pattern-based approach. *IEEE Software*, 29(3):28–36, May 2012.

[17] Elham Ramezani, Dirk Fahland, and Wil M. P. van der Aalst. *Where Did I Misbehave? Diagnostic Information in Compliance Checking*, pages 262–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[18] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The refined process structure tree. *Data & Knowledge Engineering*, 68(9):793 – 818, 2009. Sixth International Conference on Business Process Management (BPM 2008) – Five selected and extended papers.

[19] Artem Polyvyanyy and Matthias Weidlich. Towards a compendium of process technologies : the jbpt library for process model analysis. In Rebecca Deneckere and Henderick Proper, editors, *CAiSE'13 Forum at the 25th International Conference on Advanced Information Systems Engineering*, pages 106–113, Valencia, Spain, 2013. Sun SITE Central Europe.

[20] Robin J Wilson. *Introduction to Graph Theory*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

A. Appendix

A.1. Abstract

In the research field of business process models and techniques, researchers can only rely on a repository of centralized, intra-organizational processes to use as support of their work. But regarding decentralized, cross-organizational models, they face the problem that there is a lack of available model examples. Within this work this lack is tried to be tackled. Thereby, a concept is introduced how to generate collaborative business processes randomly by following a *top-down approach*, which first generates the choreography model and then derives the public and private models of each process participant from it. Additionally, a possibility for specifying and imposing global compliance rules onto the collaboration is elaborated. The conception of this random process generator is prototypically implemented within an existing framework in order to evaluate the process.

A.2. Zusammenfassung

Zur Unterstützung bei der Forschung an Geschäftsprozessmodellen und -techniken stehen lediglich ausreichend zentrale, unternehmensinterne Beispielprozesse zur Verfügung. Bei dezentralen, organisationsübergreifenden Prozessmodellen mangelt es allerdings an ausreichend verfügbaren Modellbeispielen. Im Rahmen dieser Arbeit wird versucht diesen Mangel zu beheben. Es wird ein Konzept vorgestellt, wie kollaborative Geschäftsprozesse randomisiert generiert werden können. Dabei wird ein "Top-Down" Ansatz verfolgt, der zunächst das Choreographie Modell generiert und anschließend die öffentlichen und privaten Modelle jedes Prozessteilnehmers daraus ableitet. Darüber hinaus wurde eine Möglichkeit erarbeitet, globale Compliance-Regeln zu definieren und dem Generierungsprozess als Rahmen aufzuerlegen. Um den erarbeiteten Prozess evaluieren zu können, wurde innerhalb eines bestehenden Frameworks ein Prototyp implementiert.