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 *IVertex*.

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

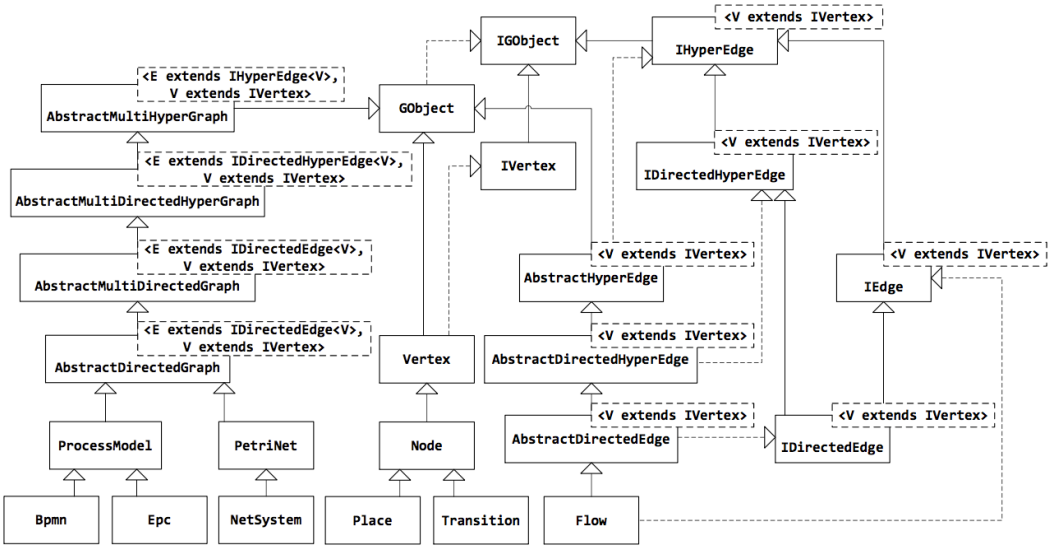

Figure 5.1.: Class and interface hierarchy of jBPT (Source: [19])

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.

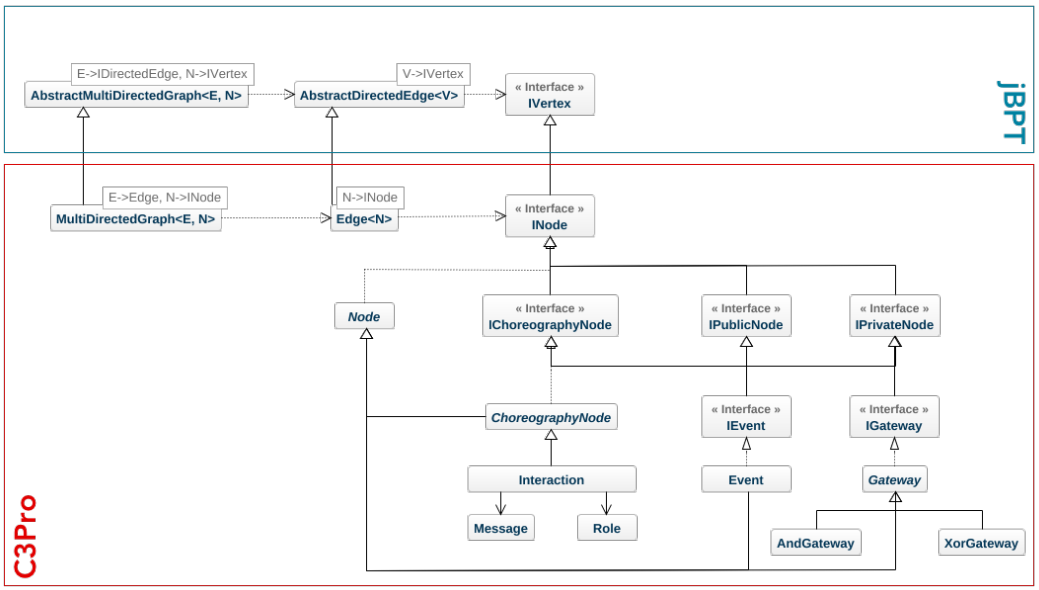

Figure 5.2.: Framework Model Representation Structure

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.

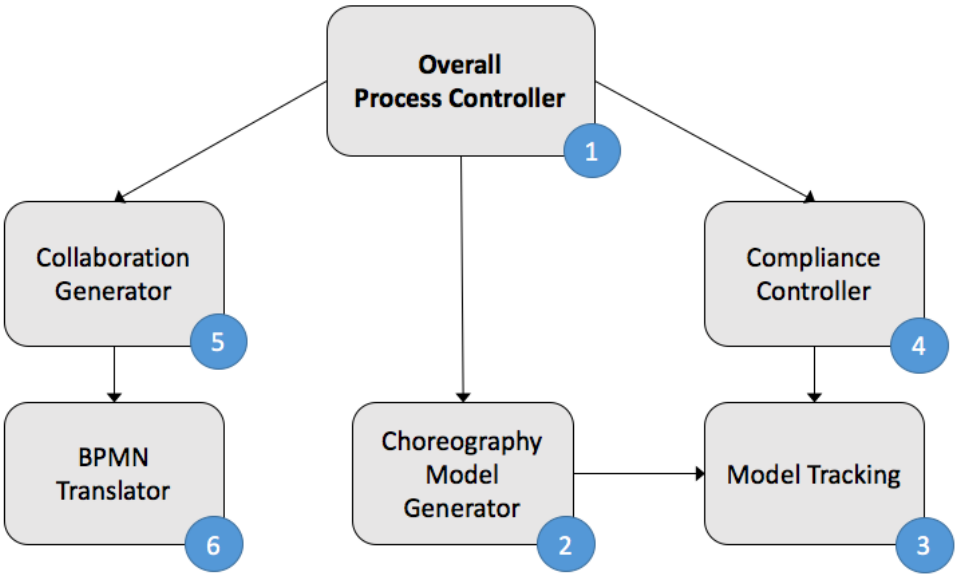

Figure 5.3.: Class Structure Random Collaboration Generator

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.

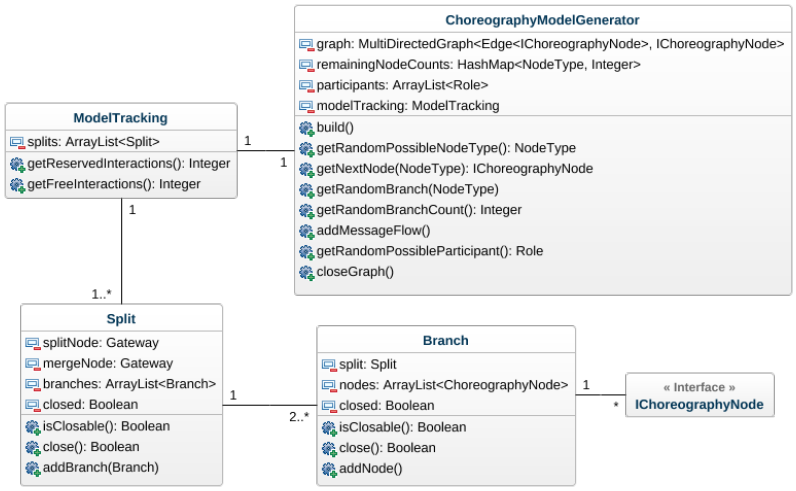

Figure 5.4.: Class Structure - ChoreographyModelGenerator

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.

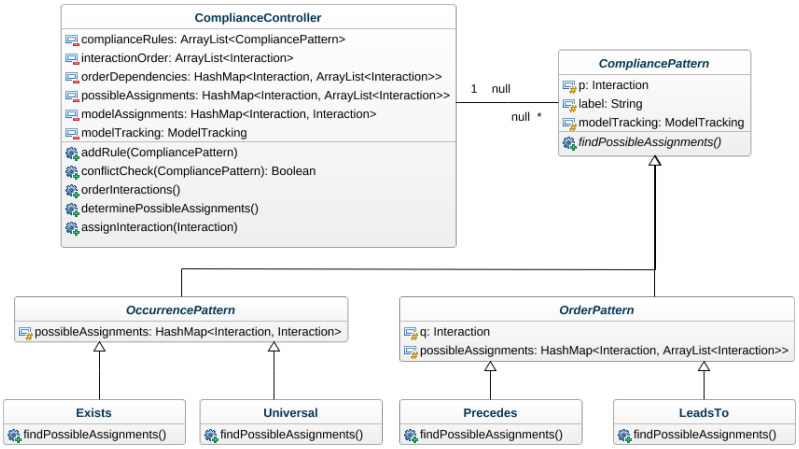

Figure 5.5.: Class Structure - ComplianceController

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.

[3]Java library for XML manipulation