Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Master Thesis Nr. 3574

Choreography-based Business Process Consolidation in One-To-Many interactions

Elkhan Dadashov

Major:	InfoTech
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Sebastian Wagner
commenced:	01.04.2013
completed:	31.10.2013
CR-Classification:	H.4.1

Abstract

In different real world scenarios the big companies can acquire other companies, or the company can insource some of its own organizational units residing abroad to increase security and control on those units, also achieve minimization of transaction costs. In these scenarios business processes of partner companies need to be consolidated with each other. Interaction of the business processes of partner companies can be modeled by choreographies.

The related works contain an approach for consolidation of the business processes which are represented in choreography with only one instance per process type. In other words, the related works only contain consolidation solution for one-to-one interaction scenarios. However, this thesis presents a concept for choreography based business process consolidation in one-to-many interaction scenarios, where one process interacts with multiple instances of another process. In particular, the number of involved instances is unknown at design time, and it only becomes known at run time of choreography. Flight ticket booking choreography is used as a motivation scenario, where it is assumed that number of involved airlines is not known in advance. On the whole, the process consolidation approach is extended for supporting consolidation of multi-instance partner processes into one merged process.

Acknowledgement

I would like to take this opportunity to express my deep gratitude to my master thesis supervisor Sebastian Wagner for his patient guidance, consistent encouragement and advice throughout my thesis time. I would like to thank him for valuable comments, insightful discussions, handmade sketches which helped me in understanding sophisticated concepts.

I would like to express my sincere gratitude to Professor Frank Leymann for giving me the chance to work on this thesis at his institute. I have been inspired by his character, carefulness for all students‘ understanding every detail, friendly, interactive teaching style and encouragement of asking good questions during his lectures.

Finally, I thank my parents and friends for their support and motivation throughout six-month thesis period.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Business process is a collection of related and ordered set of tasks accomplished by company employers and machines in order to produce the target service or product. By using business processes, companies automatize their tasks, increase production throughput, decrease latency time and can easily optimize their production process in systematic way.

1.1 Motivation for Business Process Consolidation

The recent deal of Microsoft acquiring Nokia’s mobile phone business is a bright example of an interaction scenario between companies. After the deal has been signed Nokia’s business processes have to be consolidated with Microsoft’s business processes. In dynamically changing world there are many scenarios (Google acquired Motorola, Volkswagen AG acquired Audi, Bentley, Bugatti, Ducati, Lamborghini, Man, Porsche, Scania, Seat, Škoda) where business process consolidation becomes unavoidable.

IT managers commiserate over the challenges of convincing senior executives that, contrary to popular belief, outsourcing is not always a money-saving option, even though outsourcing can lead to a reduction in IT costs, this reduction often comes at a price: reduced service [HL00]. Insourcing becomes better option when it is cheaper to do same task inside company than outside, or it is too critical share control of business process. With the rise of cloud computing services and the use of personally identifiable information will obligate the most of European companies to insource the critical organization units and data of their business processes from USA (due to The Patriot Act[1]). Insourcing - being example of company-to-company interaction, also requires business process consolidation phase as a backbone of whole operation.

Flight Ticket Booking System (FTBS) will be used as motivating scenario throughout this thesis. There are three parties involved in FTBS scenario: traveler, travel agency and airline. Traveler represents any person who wants to buy ticket from travel agency. Traveler submits the flight details to travel agency. Travel agency contacts the airlines (business processes) with given flight details for available ticket prices. The airlines provide their actual ticket prices to travel agency. Travel agency business process picks the best ticket offer and contacts the airline - which provided that ticket, for ordering. (Travel agency passes traveler‘s contact details to that chosen airline. Then airline issues E-Ticket to that traveler directly. Passing traveler‘s contact details to airline by travel agency is reference passing and will not be discussed throughout this thesis.) Figure 1.1 illustrates FTBS motivation scenario which is used throughout this thesis:

[1] http://www.zdnet.com/blog/igeneration/case-study-how-the-usa-patriot-act-can-be-used-to-access-eu-data/8805

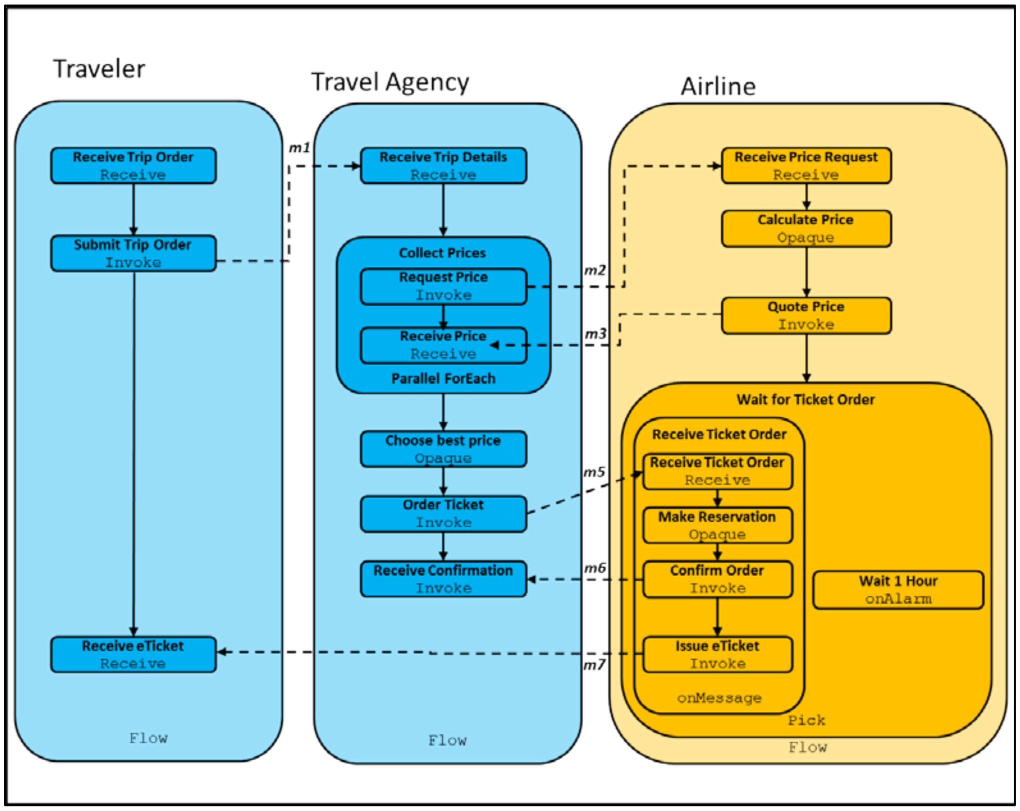


Figure 1.1 Motivation Scenario – Flight Ticket Booking System

The consolidation of single-instances of interaction processes was introduced in work of Wagner et al. [WKL11]. In their work, each process has only one instance created during choreography execution. In this thesis, travel agency business process interacts with multi-instances of airline business process. Another challenge of FTBS scenario is when the number of instances of airline business process - to be contacted by travel agency - cannot be known at choreography design time, but only at run-time.

1.2 Service Composition by Orchestration and Choreography

Often a web service is seen as an application accessible to other applications over the Web [AGA07]. The web services enable companies to expose their functionalities as services. As invocation of services is made by a program, thus requesting and executing a service involves a program calling another program by decreasing human interference in interactions [ACKM04]. The definition of W3C states that web services should be defined, described and discovered. Taking into account platform independence, easy integration and loose coupling properties of web services, companies have combined web services in business processes to automatize tasks accomplished by different applications inside their organization, as well as across their organizations. Business processes presents clear view of almost all operations-provided as web service, of the company.

1.2.1 Orchestration

Orchestration refers to a business process that can interact with Web services, and these interactions occur at the message level [PEL03]. That business process plays role of central management and has full control of business logic and on the order of service invocations. None of the participating services in orchestration need to know the existence of other services, also there is no need for an agreement between participating web services. All the participating services communicate only with a central business process, and the central business process knows all participating services and can interact with all of them. WS-BPEL is the language which can be used for defining business processes. Business processes can be executed on orchestration engines such as Oracle BPEL Process Manager[2], IBM WebSphere Process Server[3], WSO2 Business Process Server[4].

In the classical example given by John Evdemon (Software Architect at Microsoft) orchestration requires "conductor" which is responsible for execution, management, logging of process related data[5]. The "conductor" in orchestration plays same role as conductor in orchestra. The conductor in orchestra must know the entire composition in order to be able to lead all the musicians in orchestra. Musicians in orchestra - play role of different web services participating in orchestration, must only know how to play his own instrument and his part of music, and of course understand the commands of conductor when to start playing music, when to finish, and so on. Figure 1.2 illustrates service composition by orchestration:

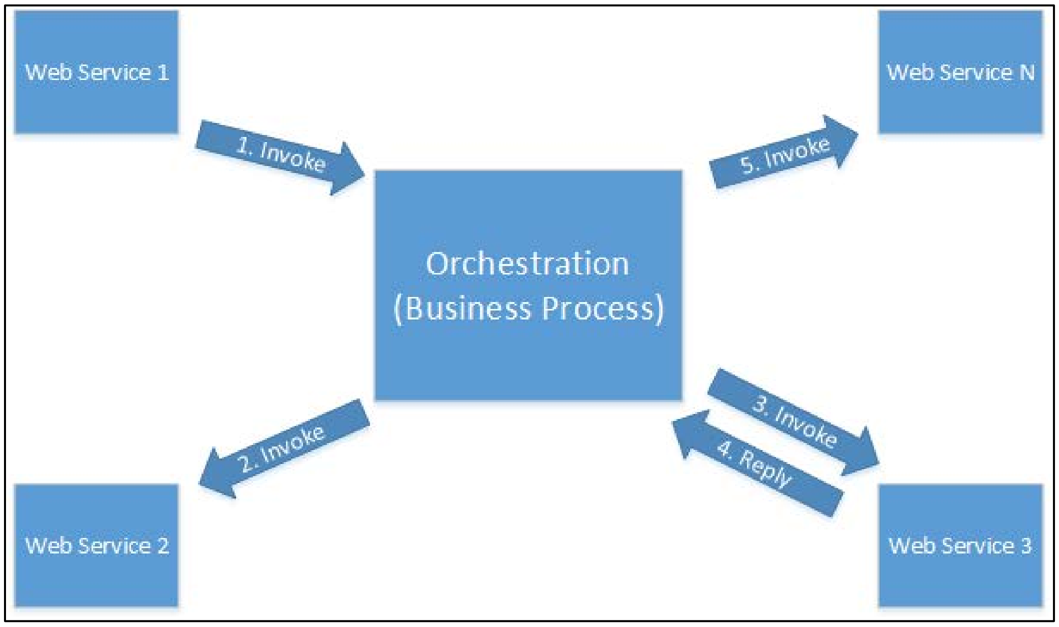


Figure 1.2 Service composition by orchestration, adapted from [JMS06]

Due to central control, orchestration is more robust to faults in the flow of execution. If one of web services fails due to any reason, then by alternating business flow, control can be transferred to another web service having similar business logic. In other words, in orchestration failing web services can be easily and transparently replaced by other web services having same (or similar) business logic. Orchestration mainly focuses on business logic and the order of accomplishment of participating web services.

2 http://www.oracle.com/technetwork/middleware/bpel/overview/index.html
3 http://www-01.ibm.com/software/integration/wps/
4 http://wso2.com/products/enterprise-service-bus/
5 http://msdn.microsoft.com/en-us/library/bb833024.aspx

In orchestration, the whole sequence of process flow can be represented by graphical view and easily mapped to service oriented architecture.

1.2.2 Choreography

In contrast to orchestration, choreography does not have central control process [PEL03]. Choreography focuses on interactions among (sub) set of participating web services – which can be called equally righted participants, and on the message exchanges between them. In other words choreography defines the order of message exchanges among involved participants.

Choreography requires that before implementation of business logic of web services, the interface through which they will interact with each other must be agreed upon. So choreography can be seen as multi-service agreement. Each interacting web service can determine the status of choreography after interpretation of sent and received messages. Figure 1.3 illustrates service composition by choreography:

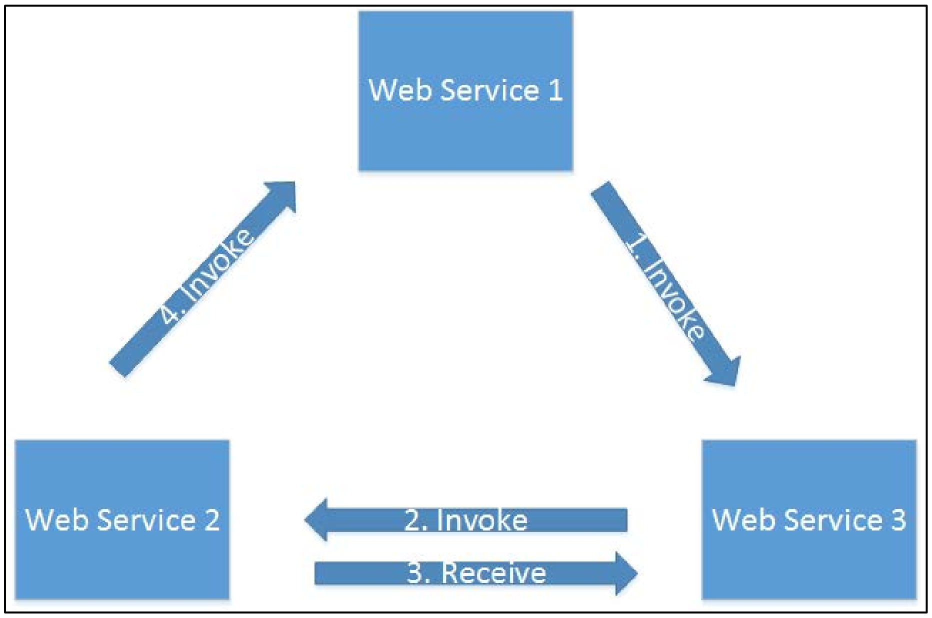


Figure 1.3 Service composition by choreography, adapted from [JMS06]

Participating services – participants know when and which other service they should interact by exchanging messages.

WS-CDL is the language for describing multi-service agreements (contracts) and collaborations between participating web services. According to Weiß et al. [WASKV13] choreographies can be also modeled and represented by graphical view using the BPEL4Chor Designer tool.

Peltz [PEL03] has shown the relation between orchestration and choreography in Figure 1.4, where it can be seen that focus of orchestration is business logic of web services and their calling order, but focus of choreography is tracking message sequences between participating parties:

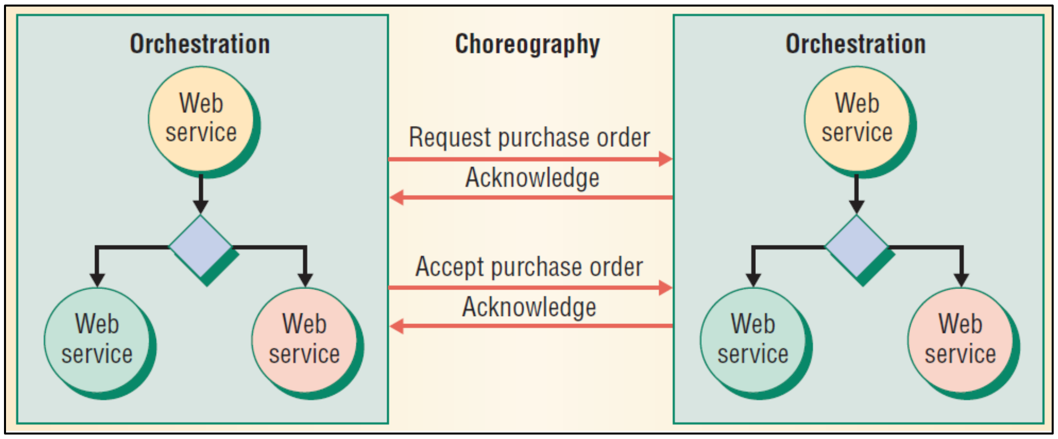


Figure 1.4 Choreography versus orchestration, taken from [PEL03]

Bpel4Chor language will be used for describing choreography throughout this thesis. Interaction and interconnection choreographies are described by examples in Section 2.2.

2 Background on BPEL and BPEL4Chor

This first part of this section describes BPEL and its features for service composition in SOA environment. Then BPEL activities which will be used in BPEL4Chor is presented using Flight Ticket Booking System example. The second part introduces BPEL4Chor by explaining its parts and roles in details. The third part of this section introduces Allen's interval algebra for visualizing the execution order relationships of activities.

2.1 BPEL Activities used in FTBS Motivation Scenario

BPEL is XML-based language to express a business process's event sequence and collaboration logic, whereas the underlying Web Services provide the process functionality [PAS05].

BPEL satisfies SOA environment requirements for service composition [WCLSF05]:

- *Flexible integration* – BPEL provides sufficiently rich and adaptable composition model to adapt changing services and their interactions. Asynchronous message exchanges gives the developer great flexibility when messages are sent, received, or processed [PAS05]. Using abstract processes for describing observable message exchange behavior of participating processes, enables hiding internal data management of process [OAS07][OAS07]. Use of explicit opaque tokens and omissions in abstract processes provide flexibility to change local aspects of the process implementation.

- *Recursive composition* – BPEL process can be offered as Web Services, which enables reuse of existing web services composition.

- *Separation and compose ability of concerns* – BPEL allows decoupling of business logic from technical and platform dependent specifications such as quality of service, messaging frameworks, and coordination protocols.

- *Stateful conversation and lifecycle management* – BPEL allows combining long-running services into a process and provides a clear lifecycle model for the resulting process.

- *Recoverability* – BPEL provides built-in fault handling to cope with expected exceptions during process execution. Compensation handlers provide alternative approach of a two-phase commit distributed transaction support and enable undoing of unwilled but completed actions [PAS05].

Activities are the basic constructs used in BPEL for describing event sequence and logic of process. In the following parts activities used in process consolidation scenario will be described. Specifications of activities are mainly taken from Web Services Business Process Execution Language Version 2.0 [OAS07]

2.1.1 <receive>

`<receive>` activity – is responsible for receiving incoming messages from partner processes. `<receive>` activity has several attributes which helps to understand its role:

- `partnerLink`: contains `myRole` used to receive incoming messages from other processes.

- `portType` (optional): defines the operations supported by web services, and messages which are input and output of supported operations

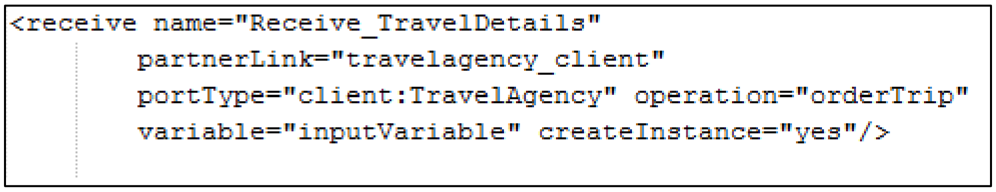
- `operation`: specifies the operation implemented by the process where `<receive>` activity resides. This is the operation partner processes wants to invoke by sending appropriate message to this process.

- `variable`: is used to store the received message data.

- `createInstance`: attribute can have yes or no value. By indicating yes as the value of `createInstance` attribute, the business process in which this `<receive>` activity resides, can be instantiated.

In order `<receive>` activity to complete successfully message with expected type must be received by process instance.

In Listing 2.1 you will find example of receive activity from Flight Ticket Booking System scenario:



```
<receive name="Receive_TravelDetails"
        partnerLink="travelagency_client"
        portType="client:TravelAgency" operation="orderTrip"
        variable="inputVariable" createInstance="yes"/>
```

Listing 2.1 `<receive>` activity example

This receive activity is start activity (the first activity to be invoked) in travel agency process. When message of type *inputVariable* is received, then new instance of travel agency process is instantiated.

2.1.2 <invoke>

`<invoke>` activity – is used to invoke Web services provided by third parties or invoking partner process which is exposed as Web Service. The real purpose is invoking required operation implemented as service. Operations executed by invoke activity can either have return value or be a void function which doesn't have return value at all. If operation invoked is void function, then invoke activity does not have output variable attribute. Invoke activity have similar attributes as receive activity:

partnerLink, *portType*, *operation*, *inputVariable* and *outputVariable*. *InputVariable* attribute stores the data which is input to the called operation. *OutputVariable* attribute will store the result data of invoked operation.

In Listing 2.2 you will find example of invoke activity from motivating scenario:

```
<invoke name="Invoke_Airline_GetPrice"
    inputVariable="Invoke2_getTicketPrice_InputVariable"
    partnerLink="Airline" portType="ns1:Airline"
    operation="getTicketPrice"
    bpelx:invokeAsDetail="no"/>
```

Listing 2.2 `<invoke>` activity example

This `<invoke>` activity resides inside travel agency process, and invokes *getTicketPrice* operation in airline process. As this `<invoke>` activity does not have *outputVariable* attribute, it can be concluded that *getTicketPrice* operation is void operation. In general `<invoke>` activity enables synchronous communication between processes.

2.1.3 <reply>

`<reply>` activity – is used to send response message to a request message received by receive activity in the same process previously. `<reply>` activity enables asynchronous communication for request-response interactions. In addition to same attributes with `<receive>` activity, `<reply>` activity has *faultName*, *messageExchange* (optional) attributes. *Variable* attribute of `<reply>` activity contains the name of variable which holds the data to be sent. *FaultName* attribute is used only when variable attribute indicates a fault. Optional *messageExchange* attribute differentiates the pair of inbound message activity and `<reply>` activity.

2.1.4 <sequence>

`<sequence>` activity – contains several activities (at least one) which must be performed in sequential order. The order of execution of activities inside `<sequence>` activity is determined by their order of declaration inside `<sequence>` activity. The `<sequence>` activity is completed when its last activity completes execution.

```
<sequence name="Sequence1">
    <invoke name="Invoke_Airline_GetPrice" ... />
    <receive name="Receive_Airline_ReceivePrice" ... />
    <assign name="Assign_StoreReceivedPrices">
        ...
    </assign>
</sequence>
```

Listing 2.3 `<sequence>` activity example

In the example of Listing 2.3, three activities are defined inside sequence activity in travel agency process. At first travel agency process invokes airline process to get flight ticket price from airline process. Then travel agency receives price of flight ticket from airline process. The last activity – `<assign>` activity stores received flight price in some variable locally.

2.1.5 <flow>

`<flow>` activity – enables execution of grouped activities concurrently. `<flow>` activity is considered complete when all activities inside it are finished [OAS07], or its enabling condition evaluates to false and none of its activities is executed. `<links>` can be used within flow activities to define explicit control dependencies between nested child activities [OAS07]. `<links>` are described in detail in Section 2.1.14.

```
<flow>
    <invoke name="Invoke1" ... />
    <invoke name="Invoke2" ... />
</flow>
```

Listing 2.4 `<flow>` activity example

In the Listing 2.4 example two `<invoke>` activities are calling operations which are independent from each other and can be executed concurrently, so they are grouped inside `<flow>` activity.

2.1.6 <scope>

`<scope>` activity - bounds visibility and context of enclosed activities. The variables, partner links, message exchanges and correlation sets defined inside scope can only be accessed within the scope.

```
<scope name="Scope1">
    <partnerLinks>
        <partnerLink name="Airline" partnerLinkType="ns1:Airline"
            partnerRole="AirlineProvider" myRole="AirlineRequester"/>
    </partnerLinks>
    <variables>
        <variable name="Receive1_receiveTicketPrice_InputVariable"
            messageType="ns1:AirlineTicketPriceResponseMessage"/>
        <variable name="Receive1_quoteTicketPrice_InputVariable"
            messageType="ns1:AirlineTicketPriceResponseMessage"/>
        <variable name="InvokeOrderTicket_receiveTicketOrder_InputVariable"
            messageType="ns1:AirlineTicketOrderMessage"/>
    </variables>
<scope>
```

Listing 2.5 `<scope>` activity example

In the Listing 2.5 example all the *partnerLinks* and *variables* defined within scope has meaning only in the context of Scope1. Outside Scope1 these *variables* and *partner links* are invisible.

2.1.7 <if>

`<if>` activity (with optional `<elseif>` and `<else>` activities) – provides different execution path depending on condition. If the condition element of `<if>` activity evaluates to true then its contained activity is executed. If condition element of `<if>` activity evaluates to false, then condition element of `<elseif>` activities are checked. The contained activity of first `<elseif>` activity whose condition element evaluates to true is executed. If none of the `<elseif>` activities is taken, then the activity contained by `<else>` activity is executed. In Listing 2.6 you can find example from WS-BPEL specification 2.0 [OAS07]:

```
<if xmlns:inventory="http://supply-chain.org/inventory"
    xmlns:FLT="http://example.com/faults">
    <condition>
    bpel:getVariableProperty('stockResult','inventory:level') > 100
    </condition>
    <flow>
    <!-- perform fulfillment work -->
    </flow>
    <elseif>
        <condition>
        bpel:getVariableProperty('stockResult','inventory:level') >= 0
        </condition>
        <throw faultName="FLT:OutOfStock" variable="RestockEstimate" />
    </elseif>
    <else>
        <throw faultName="FLT:ItemDiscontinued" />
    </else>
</if>
```

Listing 2.6 `<if>` activity example

2.1.8 <partnerLinkType>

`<partnerLinkType>` activity – expresses conversational relationship between two communicating web services. Each service provides its role in this conversation and indicates exactly one port type.

```
<plnk:partnerLinkType name="Airline">
  <plnk:role name="AirlineProvider" portType="client:Airline"/>
  <plnk:role name="AirlineRequester" portType="client:AirlineCallback"/>
</plnk:partnerLinkType>
```

Listing 2.7 `<partnerLinkType>` activity example

In Listing 2.7 example described above the airline `<partnerLinkType>` is defined inside wsdl file of airline process. Airline process declares its *role* as AirlineProvider, and its *partner role* as AirlineRequester.

2.1.9 <partnerLink>

`<partnerLink>` construct – identifies the parties that interact with business process[6].

`<partnerLink>` activity has several attributes:

- *name*: indicates unique name of this partner link within the same immediately enclosing scope
- *partnerLinkType*: indicates partnerLinkType used
- *myRole*: the role of process in which used partnerLinkType is declared
- *partnerRole*: is the role of partner processes which wants to communicate with this process
- *initializeParnterRole* (can be omitted): can have “yes” or “no” value. This attribute lets WS-BPEL processor know either to initialize the endpoint reference of the *partnerRole* before that endpoint reference is first utilized by the WS-BPEL process [OAS07].

The partner links defined within scope are only visible inside that scope.

```
<partnerLink name="Airline"
    partnerLinkType="ns1:Airline"
    partnerRole="AirlineProvider"
    myRole="AirlineRequester"/>
```

Listing 2.8 `<partnerLink>` activity example

The Listing 2.8 example describes Airline partner link defined inside travel agency process.

2.1.10 <assign>

`<assign>` activity – is used for storing data in variables and copying data from one variable into another. Another usage of `<assign>` activity is copying endpoint references to and from partnerLinks. Expressions can be used to perform simple computations in `<assign>` activities.

```
<assign name="Assign_BestPrice">
    <copy>
        <from>oraext:min-value-among-nodeset($arrayOfPricesVariable.parameters/price)</from>
        <to>$bestPriceVariable</to>
    </copy>
</assign>
```

Listing 2.9 `<assign>` activity example

The `<assign>` activity in Listing 2.9, copies minimum price (best price) value from $arrayOfPricesVariable list into $bestPriceVariable.

[6] http://docs.oracle.com/cd/E19182-01/821-0539/6nlj8ms9l/index.html

2.1.11 <wait>

`<wait>` activity – enables delay in the execution of process for a given period of time or until timestamp becomes equal to given deadline.

2.1.12 <pick>

`<pick>` activity – enables handling of timer-based and message receiving events. *OnMessage* element of pick activity waits for the reception of particular type of incoming message. Several *OnMessage* elements can be defined inside pick activity which allows receiving different types of messages. *OnAlarm* element of pick activity triggers contained activity based on the given (duration or deadline) time. *OnAlarm* event allows to bound waiting time for specific type of message to arrive.

```
<pick name="Pick1">
    <onMessage variable="OnMessage_orderTicket_InputVariable"
        partnerLink="airline_client"
        portType="client:Airline"
        operation="receiveTicketOrder">
        <sequence name="Sequence1">
            <assign name="Assign_TIcketOrder">
                ...
            </assign>
            <invoke name="Invoke_Traveler_issueETicket" .../>
        </sequence>
    </onMessage>
    <onAlarm>
        <for>'PT5M'</for>
            <empty name="Empty"/>
    </onAlarm>
</pick>
```

Listing 2.10 `<pick>` activity example

In the above code snippet described in Listing 2.10, `<pick>` activity's *onMessage* element waits for message of type of *onMessage*_orderTicket_InputVariable variable. *onAlarm* element specifies duration time of five minutes for waiting *onMessage*_orderTicket_InputVariable variable type message to arrive, if message doesn't arrive during 5 minutes, then `<empty>` activity is executed. `<empty>` activity can be thought as no operation activity – nothing needs to be done.

2.1.13 <forEach>[7]

`<forEach>` activity – enables executing several activities within its first child `<scope>` activity [*finalCounterValue* - *startCounterValue*] times. `<forEach>` activity's *counterName* attribute defines

[7] Only the elements and attributes of `<forEach>` activity which are used throughout this thesis, are explained in this subsection

variable name for loop counter. *Parallel* attribute allows specifying parallel or sequential/serial execution of scope instance. If *parallel=”no”* then each scope instance will start its execution only after completion of previous instance, in other words scope instances will be executed in sequential order. If *parallel=”yes”* then [*finalCounterValue* - *startCounterValue*] scope instances will be started concurrently. *Completion condition* element of `<forEach>` activity is optional, when it is specified it prevent some of the children from executing in serial `<forEach>` case, or forces early termination of given number of the children in parallel `<forEach>` activity [OAS07].

```
<forEach parallel="yes" counterName="ForEach1Counter"
    name="ForEach">
    <startCounterValue>number(1)</startCounterValue>
    <finalCounterValue>$inputVariable.payload/client:numberOfAirlines</finalCounterValue>
    <scope name="Scope1">
        ...
    </scope>
</forEach>
```

Listing 2.11 `<forEach>` activity example

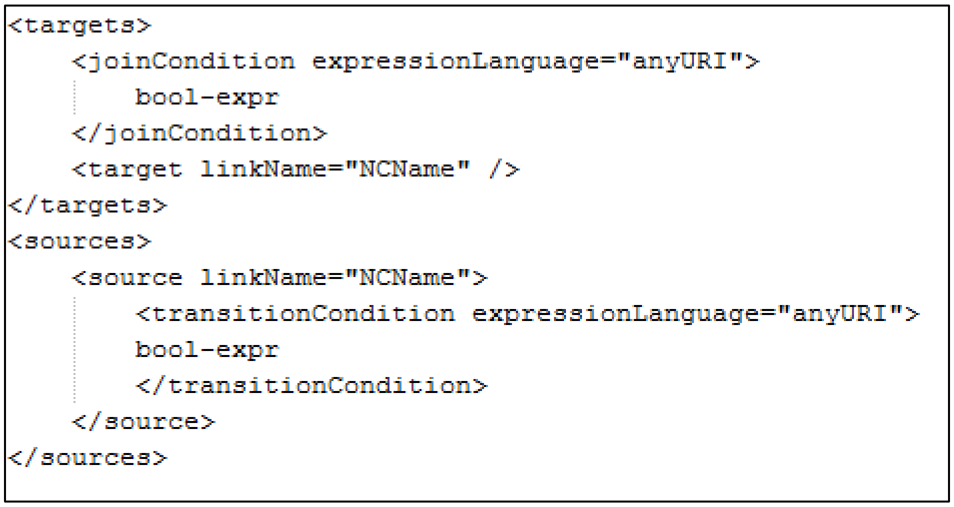
In the Listing 2.11 example of parallel `<forEach>` activity is described from FTBS scenario.

2.1.14 <link>

`<link>` construct is used to maintain synchronization dependencies between activities that are nested within `<flow>` activity. Declarations of `<link>` activities are enclosed by a `<flow>` activity [OAS07]. A `<link>` activity has mandatory *name* attribute, which must be uniquely distinguished from all other links defined in immediately enclosing `<flow>` activity.

2.1.15 <sources> and <targets>

Each WS-BPEL activity can have the optional container `<sources>` (`<targets>`) which contain collection of `<source>` (`<target>`) elements. These elements are used to establish synchronization relationships through a `<link>` activity [OAS07].



```
<targets>
    <joinCondition expressionLanguage="anyURI">
        bool-expr
    </joinCondition>
    <target linkName="NCName" />
</targets>
<sources>
    <source linkName="NCName">
        <transitionCondition expressionLanguage="anyURI">
        bool-expr
        </transitionCondition>
    </source>
</sources>
```

Listing 2.12 `<targets>` container example

linkName attribute of the `<source>` (`<target>`) must have value of a `<link>` declared in an enclosing `<flow>` activity. Two different links MUST NOT share the same `<source>` and `<target>` activities, that is, at most one `<link>` may be used to connect two activities [OAS07]. In other words every activity within `<flow>` activity can be used exactly once as `<source>` (or `<target>`) activity of exactly one `<link>` activity among all `<link>` activities.

Let `<source>` activity *S* of a `<link>` *L* be nested in another *C* activity (at any level) and the `<link>` *L* itself is not declared inside *C* activity at any level. Such a link *L* is called link leaving *S* activity.

Let *T* be a `<target>` activity of a `<link>` *L'* and be nested in another *C'* activity (at any level), but the `<link>` *L'* itself is not declared inside *C'* activity (at any level). Such `<link>` *L'* is called link entering *T* activity.

A link leaving or entering activity is called *cross-boundary* link.

```
<flow>
    <links>
        <link name="AtoB" />
    </links>
    <sequence name="A">
        <sources>
            <source linkName="AtoB" />
        </sources>
        <invoke name="Invoke_1" ... />
        <invoke name="Invoke_2" ... />
    </sequence>
    <sequence name="B">
        <targets>
            <target linkName="AtoB" />
        </targets>
        <receive name="Receive_1" .../>
        <invoke name="Invoke_3" ... />
    </sequence>
</flow>
```

Listing 2.13 `<source>` and `<target>` activity example, adapted from [OAS07]

The XML snippet in Listing 2.13 describes AtoB link as an example link which crosses boundary of two activities: sequence A and sequence B. Hence AtoB is called cross-boundary link.

There exist some constraints for using links inside `<forEach>` activities in BPEL processes. A link which is used within `<forEach>` activity must be declared in a flow which is itself nested inside the `<forEach>` activity. A link must not cross the boundary of `<forEach>` activity, in other words leave or enter from outside the scope activity of `<forEach>` activity.

2.1.16 Standard Attributes and Standard Elements

joinCondition is an optional attribute of *targets* collection (if not specified default join condition is logical OR operation) which describes join condition of all incoming link activities to this activity. An optional *transitionCondition* attribute of source element specifies the transition condition – to follow the outgoing links or not (if this attribute is not specified according to default behavior all outgoing links need to be followed).

Each WS-BPEL activity can have another optional attribute *suppressJoinFailure*. By the help of this attribute, it is stated whether a join fault should be suppressed or not when *bpel:joinFailure* fault is raised.

2.2 BPEL4Chor

At first this section will describe the terminology which will be used for describing concepts of choreography throughout this thesis. Later three artifacts of BPEL4Chor- participant behavior description, participant grounding, participant topology, will be introduced. FTBS scenario will be used for illustrating examples.

Real-world organizations, persons, information systems or software services that interact with other organizations, persons, systems or services are called participants [DEC09]. Participants interact with each other by exchanging messages. In other words participants are instances of participating business processes.

Activities of participant can be separated into two groups: internal activities and communication activities. Internal activities do not interact outside world, but only with local process and infrastructure. For example activity for storing received ticket prices in local database is internal activity. Communication activities are responsible for communication with outside world, by sending (receiving) messages to (from) other participants.

There are two different modeling approaches in choreography: interaction and interconnected models. In interaction model approach elementary interactions such as request and request-response message exchanges are the basic building blocks and behavioral dependencies are defined between them [DKLW07]. WS-CDL and Let's Dance are the languages used for interaction modeling. Figure 2.1 illustrates interaction model example. In this example traveler interacts with travel agency for submitting flight details in order to buy flight tickets. Travel agency interacts with several airlines for requesting ticket price satisfying traveler supplied information. Then each airline interacts with travel agency by quoting the current price of the ticket of interest. Travel agency chooses the best (the cheapest) ticket providing airline, and interacts with it by ordering a ticket for traveler. After receiving ticket order request from travel agency chosen airline interacts with traveler by issuing the eTicket to him.

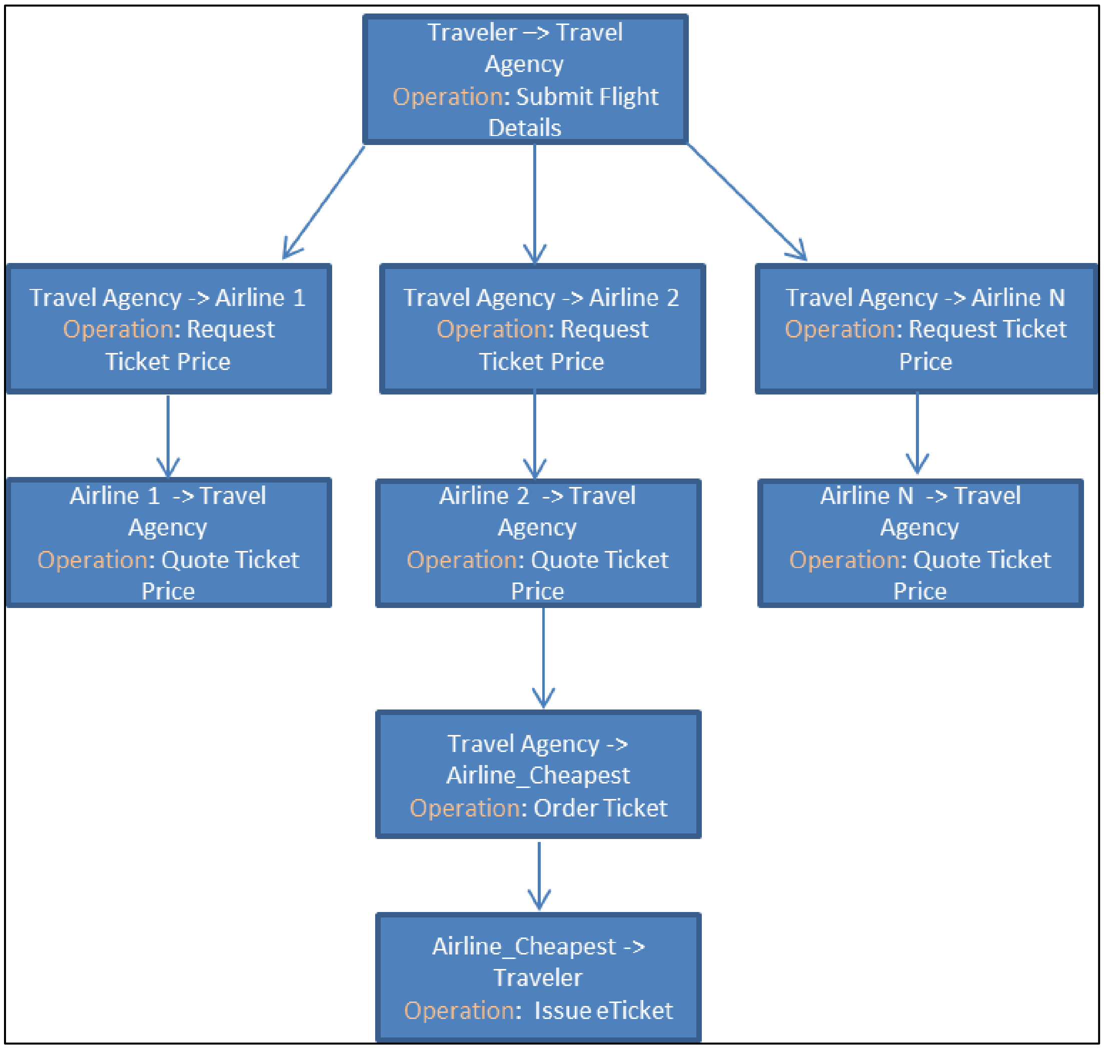

Traveler –> Travel Agency
Operation: Submit Flight Details
Travel Agency -> Airline 1
Operation: Request Ticket Price
Travel Agency -> Airline 2
Operation: Request Ticket Price
Travel Agency -> Airline N
Operation: Request Ticket Price
Airline 1 -> Travel Agency
Operation: Quote Ticket Price
Airline 2 -> Travel Agency
Operation: Quote Ticket Price
Airline N -> Travel Agency
Operation: Quote Ticket Price
Travel Agency -> Airline_Cheapest
Operation: Order Ticket
Airline_Cheapest -> Traveler
Operation: Issue eTicket

Figure 2.1 Interaction model choreography for FTBS scenario

Interconnection models have communication activities as basic building blocks and behavioral dependencies are defined between them on per-role bases [DEC09]. Each dependency should be assigned only to one role. BPMN and BPEL4Chor are the languages mainly used for designing interconnection choreographies. In example taken from [DKLW07] choreography is modeled using BPMN. Motivation scenario used throughout this thesis is similar but not same with this choreography example. Traveler, Traveler Agency and Airline are the involved participant types. Interconnection choreography example in Figure 2.2 Interconnection choreography example described in BPMN taken from [DKLW07] is same as interaction choreography example described above.

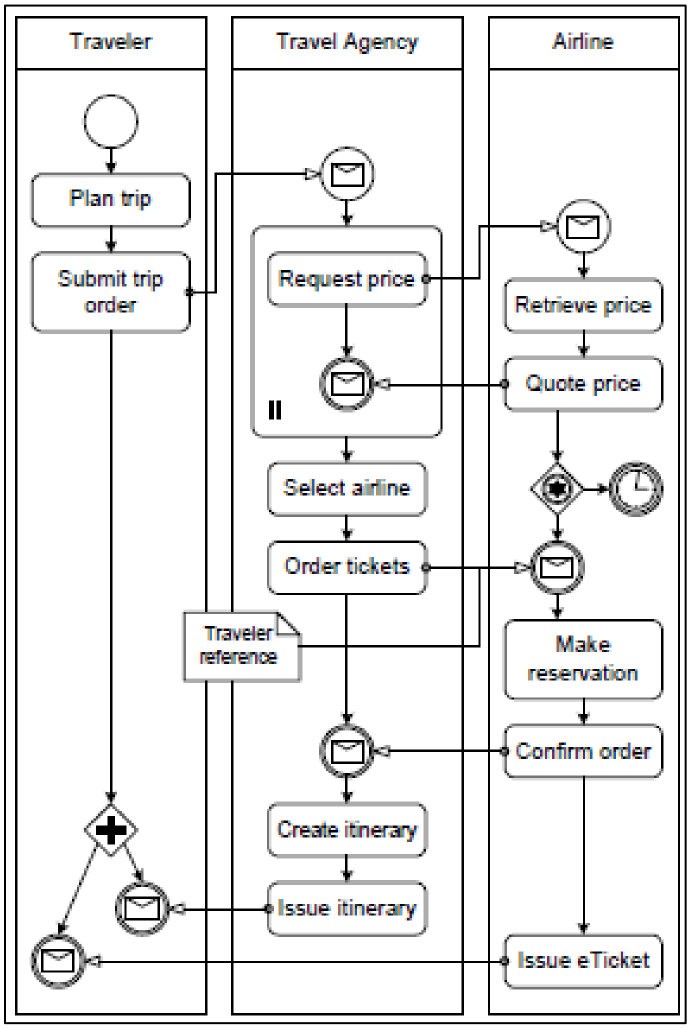


Figure 2.2 Interconnection choreography example described in BPMN taken from [DKLW07]

Interactions models can be mapped to interconnections models by using the approach described at Zaha et al. [ZDH06]. As not all the interaction models can be mapped to interconnection models, Kopp et al. [KLW10] has presented that safe and sound BPMN interconnection models and containing control flow without exception handling can be mapped to interaction BPMN models [KELLN11].

BPEL4Chor is another language for modeling interconnection models. Bpel4chor has three artifacts: participant topology, participant behavior description (PBD) and participant grounding. Different abstract BPEL process is used as basis for each PBD. PBDs are connected together using message links in participant topology. PBD and participant topology do not contain any technical configuration details, which provides high flexibility for reusing choreography for different technical setups, e.g., with different port types used [DKLW07]. Participant grounding is the only artifact which holds technical details such as links to WSDL definitions and XSD types. Let’s dance and BPEL4Chor are able to model all common interaction patterns described in [BDH05]. Figure 2.3 illustrates all three artifacts of BPEL4Chor:

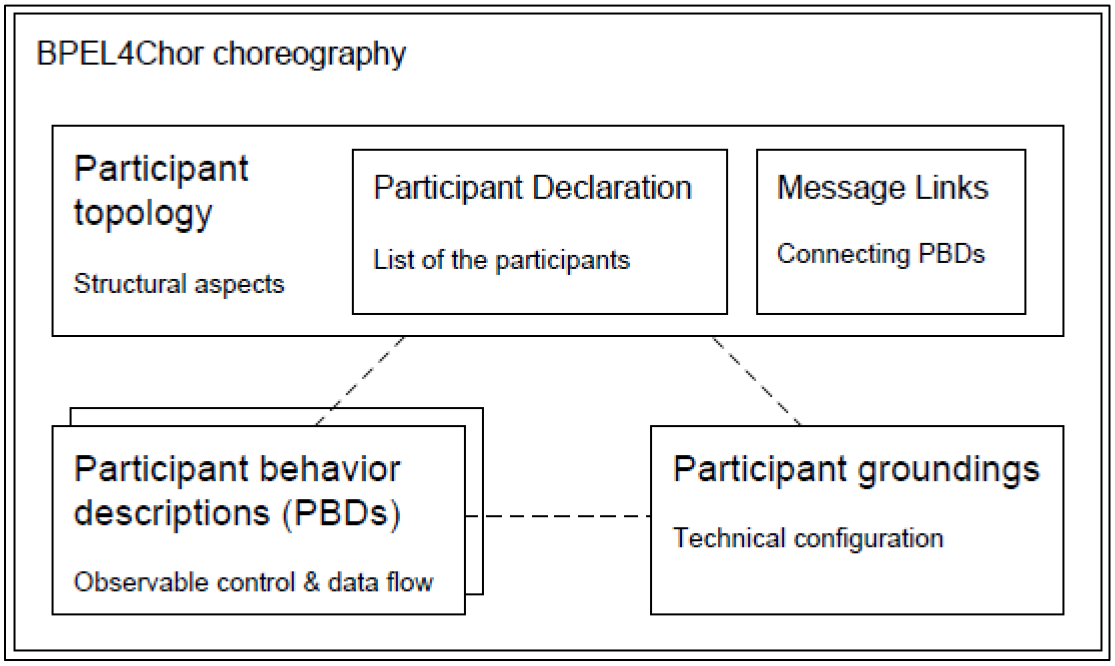


Figure 2.3 BPEL4Chor three artifacts, taken from [DKLW07]

2.2.1 Participant Behavior Descriptions (PBDs)

Communication activities and their control and data dependencies are major building blocks of choreographies. All activities of BPEL for control and data flow can be used unchanged in BPEL4Chor safely, which also enables using existing BPEL tools for choreography designing. But there exists few constraints while using BPEL activities for choreographies [DKLW07]:

- Each communicating activity inside choreographies must be identified uniquely. Due to fact that `<onMessage>` activity does not have name attribute, different `<onMessage>` activities cannot differentiated from each other. As a solution to this problem new *wsu:id* attribute having *xsd:id* type is introduced as new attribute for all communicating activities.

- *PartnerLink*, *portType* and *operation* attributes must not be specified for communication activities. By this constraint loose coupling of BPEL and WSDL interfaces is achieved.

- It is mandatory that *messageExchange* attribute exists in both receive and reply activities to relate each pair of receive/reply messages. But if a receive models an asynchronous operation, the attribute *messageExchange* must not be specified.

Abstract processes allow skipping some attributes of BPEL constructs, i.e. *partnerLink* and operation attribute of message can be omitted. Decker [DEC09] has introduced *Abstract Process Profile for Participant Behavior Descriptions* for describing the behavior of each participant. In other words one participant behavior description is enough for all participants of the same type, i.e. only one travel agency participant behavior description is created for all participants which has travel agency type. This profile satisfies all constraints of Abstract Process Profile for Observable Behavior specified by BPEL [DKLW07]. While using this profile, variables and variable types can be skipped in choreography design which gives very high flexibility in expressing branching conditions as plain text.

Even though *Abstract Process Profile for Participant Behavior Descriptions* are easier for designing choreographies, executable BPEL processes are used instead of abstract processes throughout this thesis.

```
<process name="Traveler"
         targetNamespace="http://xmlns.oracle.com/ACMETravel/Traveler/Traveler"
         xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
         xmlns:client="http://xmlns.oracle.com/ACMETravel/Traveler/Traveler"
         xmlns:ora="http://schemas.oracle.com/xpath/extension"
         xmlns:bpelx="http://schemas.oracle.com/bpel/extension"
         xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
         xmlns:ns1="http://xmlns.oracle.com/ACMETravel/TravelAgency/TravelAgency"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <import namespace="http://xmlns.oracle.com/ACMETravel/Traveler/Traveler"
        location="Traveler.wsdl" importType="http://schemas.xmlsoap.org/wsdl/"/>
    <partnerLinks>
        <partnerLink name="traveler_client" partnerLinkType="client:Traveler"
            myRole="TravelerProvider" partnerRole="TravelerRequester"/>
        <partnerLink name="TravelAgency" partnerLinkType="ns1:TravelAgency"
                    partnerRole="TravelAgencyProvider"
                    myRole="TravelAgencyRequester"/>
    </partnerLinks>
    <variables>
        <variable name="inputVariable" messageType="client:TravelerRequestMessage"/>
        <variable name="outputVariable" messageType="client:TravelerResponseMessage"/>
        <variable name="Invoke_TravelAgency_orderTrip_InputVariable"
            messageType="ns1:TravelAgencyTripOrderRequestMessage"/>
    </variables>
    <sequence name="main">
        <receive name="Receive_FlightDetails" partnerLink="traveler_client"
            portType="client:Traveler" operation="process" variable="inputVariable"
            createInstance="yes"/>
        <assign name="Assign_FlightDetails">
          ...
        </assign>
        <invoke name="Invoke_TravelAgency"
                wsu:id="InvokeOrderTrip"
                inputVariable="Invoke_TravelAgency_orderTrip_InputVariable"
                bpelx:invokeAsDetail="no"/>
        <invoke name="callbackClient" partnerLink="traveler_client"
            portType="client:TravelerCallback" operation="processResponse"
            inputVariable="outputVariable"/>
    </sequence>
</process>
```

Listing 2.14 Traveler participant behavior description

The xml snippet in Listing 2.14 describes traveler participant behavior description. Traveler PBD has declared `<partnerLinks>`, variables and set of activities coordinating the flow of messages across the services integrated within this PBD. This participant behavior description has only one communicating activity - `<invoke>` activity. As all communicating activities in general, this `<invoke>` activity also has *wsu:id* for differentiating it uniquely from other communicating activities of choreography. This

`<invoke>` activity communicates with `<receive>` activity (having wsu:id Receive_TravelDetails) of travel agency participant.

2.2.2 Participant Topology

As stated in previous section, for each participant one separate participant behavior description is created. Participant topology describes the structural aspects of choreography and helps to relate participant behavior descriptions to each other [DEC09]. Participant type, participant reference and message link are three new terms introduced in participant topology. Each participant behavior description represents one participant type [DKLW07].

```
<participantTypes>
    <participantType name="TravelerType"
        participantBehaviorDescription="cns:Traveler"/>
    <participantType name="AirlineType" participantBehaviorDescription="cns:Airline"/>
    <participantType name="TravelAgencyType" participantBehaviorDescription="cns:TravelAgency"/>
</participantTypes>
```

Listing 2.15 Participant type example from FTBS scenario

In the code snippet of Listing 2.15 three participant types are declared:

1) TravelerType participant type representing traveler participant behavior description

2) AirlineType participant type representing airline participant behavior description

3) TravelAgencyType participant type representing travel agency participant behavior description

Participant references point to participants.

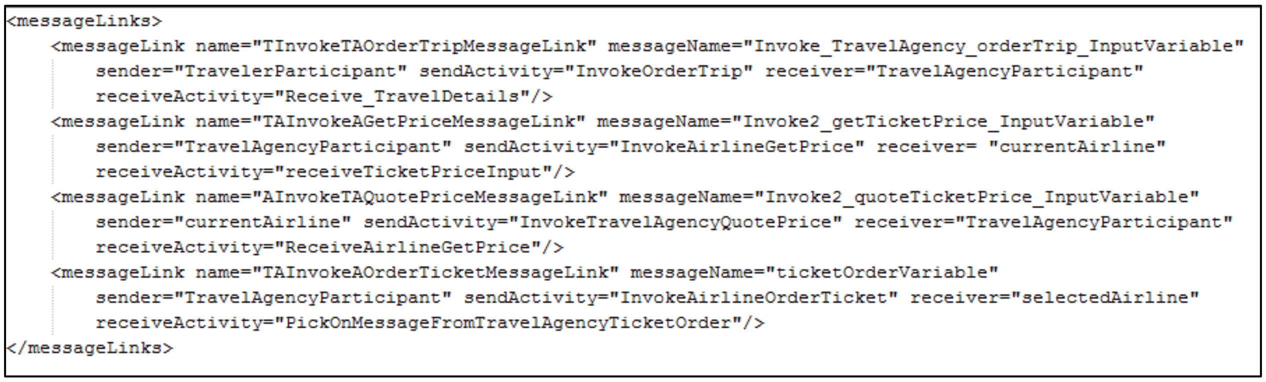
```
<participants>
    <participant name="TravelerParticipant" type="TravelerType" selects="TravelAgencyParticipant"/>
    <participant name="TravelAgencyParticipant" type="TravelAgencyType" selects="AirlinesParticipantSet"/>
    <participantSet name="AirlinesParticipantSet" type="AirlineType" forEach="cns:ForEach1">
        <participant name="currentAirline" type="AirlineType"/>
        <participant name="selectedAirline" type="AirlineType"/>
    </participantSet>
</participants>
```

Listing 2.16 Participant references example from FTBS scenario

In the Listing 2.16 two participants and one `<participantSet>` is declared. `<participantSet>s` are used for describing the case when several `<participant>`s of the same type participate in one choreography instance and the number of `<participant>`s can only be known at runtime. If number of `<participant>`s of same *participant type* can be known at design time, then same number of `<participant>`s is declared inside <participants> tags. Attribute *selects* indicates which `<participant>` selects which other `<participant>`. In the example above travel agency initiates conversation and selects airline of interest. The knowledge about `<participant>`s during a conversation is local to individual `<participant>`s [DEC09]. Traveler does not know about airlines involved in conversation with travel agency. In other words only direct partners of conversation know each other and no one else.

The term containment helps to explain the case when participant reference is contained inside participant set: that participant reference (having name *selectedAirline*) is selected from the set. The travel agency will order a ticket only from one selected airline which provides cheapest price. The other participant reference (having name *currentAirline*) which is also contained in `<participantSet>` represents one `<participant>` selected in each iteration of the parallel `<forEach>` branches.

Message links binds two `<participant>`s together which can communicate with each other. Alternatively it can be said that message links states interconnection of the participant behavior descriptions. Two attributes of message link, *receiveActivity* and *sendActivity* refers to the communicating activities of the PBDs. The ordering of these communicating activities can be understood from PBDs: sending activity initiates conversation and sends message over message link. Only one sender out of multiple senders with the same target type is allowed to send message. If a receiving activity is executed multiple times, several interactions can take place over one message link [DEC09]. Message links must be used under seven constraints which are described in [DKLW07].



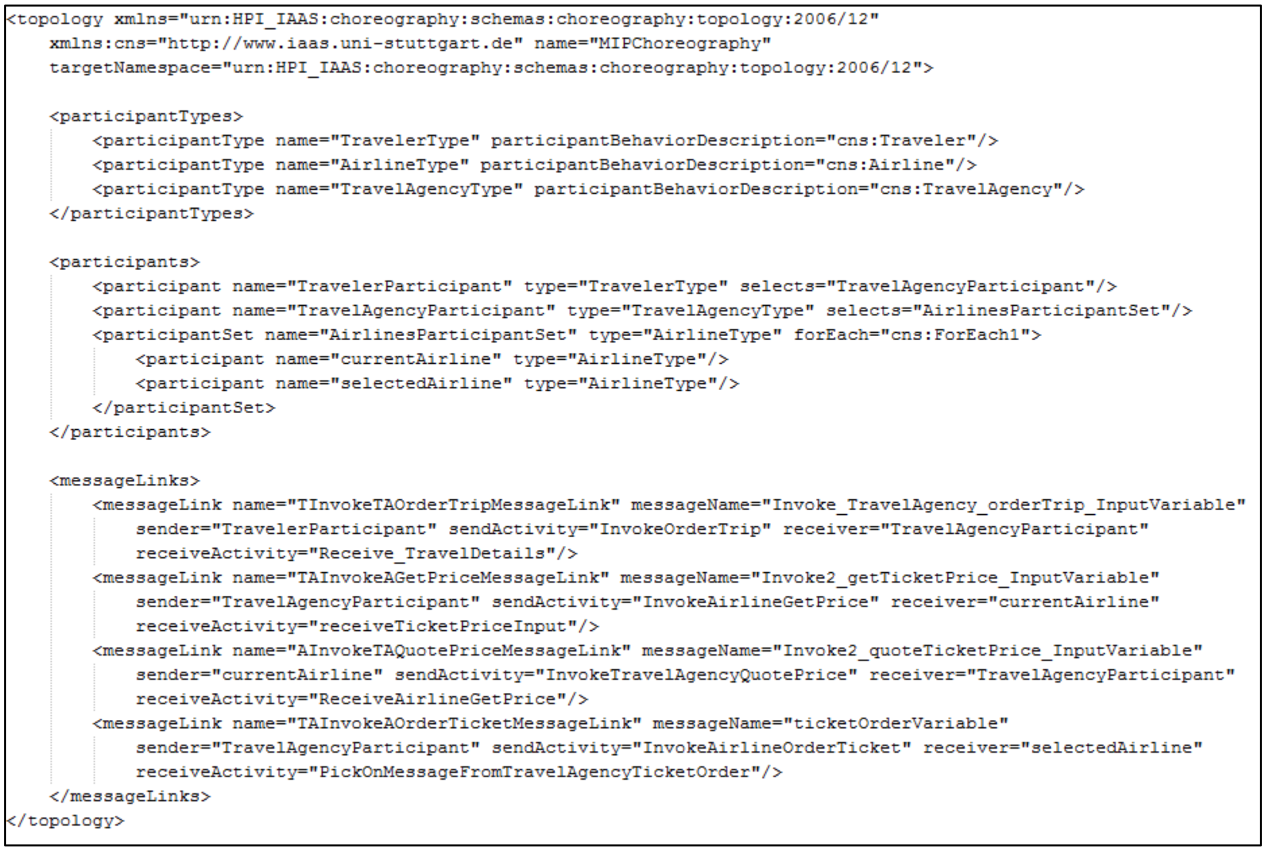
```
<messageLinks>
    <messageLink name="TInvokeTAOrderTripMessageLink" messageName="Invoke_TravelAgency_orderTrip_InputVariable"
        sender="TravelerParticipant" sendActivity="InvokeOrderTrip" receiver="TravelAgencyParticipant"
        receiveActivity="Receive_TravelDetails"/>
    <messageLink name="TAInvokeAGetPriceMessageLink" messageName="Invoke2_getTicketPrice_InputVariable"
        sender="TravelAgencyParticipant" sendActivity="InvokeAirlineGetPrice" receiver= "currentAirline"
        receiveActivity="receiveTicketPriceInput"/>
    <messageLink name="AInvokeTAQuotePriceMessageLink" messageName="Invoke2_quoteTicketPrice_InputVariable"
        sender="currentAirline" sendActivity="InvokeTravelAgencyQuotePrice" receiver="TravelAgencyParticipant"
        receiveActivity="ReceiveAirlineGetPrice"/>
    <messageLink name="TAInvokeAOrderTicketMessageLink" messageName="ticketOrderVariable"
        sender="TravelAgencyParticipant" sendActivity="InvokeAirlineOrderTicket" receiver="selectedAirline"
        receiveActivity="PickOnMessageFromTravelAgencyTicketOrder"/>
</messageLinks>
```

Listing 2.17 Message links from FTBS scenario

In the code snippet above three message links are declared. First message link named *TInvokeTAOrderTripMessageLink*, binds two participants - traveler and travel agency. This message link states that send activity (having wsu:id *Invoke_TravelAgency*) of traveler sends message (named *Invoke_TravelAgency_orderTrip_InputVariable*) to receive activity (having wsu:id *Receive_TravelDetails*) of travel agency participant.

Throughout this thesis only invoke activity is used as send activity, and receive activity is used as receive activity.

In Listing 2.18 you will find the topology artifact which describes motivating scenario FTBS:

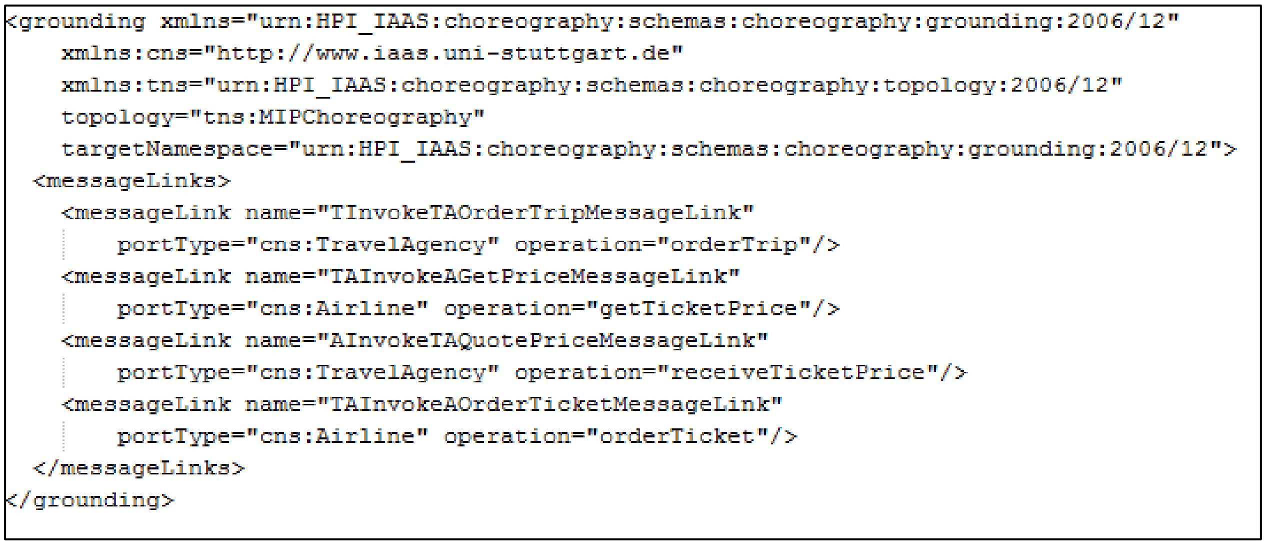


```xml
<topology xmlns="urn:HPI_IAAS:choreography:schemas:choreography:topology:2006/12"
    xmlns:cns="http://www.iaas.uni-stuttgart.de" name="MIPChoreography"
    targetNamespace="urn:HPI_IAAS:choreography:schemas:choreography:topology:2006/12">

    <participantTypes>
        <participantType name="TravelerType" participantBehaviorDescription="cns:Traveler"/>
        <participantType name="AirlineType" participantBehaviorDescription="cns:Airline"/>
        <participantType name="TravelAgencyType" participantBehaviorDescription="cns:TravelAgency"/>
    </participantTypes>

    <participants>
        <participant name="TravelerParticipant" type="TravelerType" selects="TravelAgencyParticipant"/>
        <participant name="TravelAgencyParticipant" type="TravelAgencyType" selects="AirlinesParticipantSet"/>
        <participantSet name="AirlinesParticipantSet" type="AirlineType" forEach="cns:ForEach1">
            <participant name="currentAirline" type="AirlineType"/>
            <participant name="selectedAirline" type="AirlineType"/>
        </participantSet>
    </participants>

    <messageLinks>
        <messageLink name="TInvokeTAOrderTripMessageLink" messageName="Invoke_TravelAgency_orderTrip_InputVariable"
            sender="TravelerParticipant" sendActivity="InvokeOrderTrip" receiver="TravelAgencyParticipant"
            receiveActivity="Receive_TravelDetails"/>
        <messageLink name="TAInvokeAGetPriceMessageLink" messageName="Invoke2_getTicketPrice_InputVariable"
            sender="TravelAgencyParticipant" sendActivity="InvokeAirlineGetPrice" receiver="currentAirline"
            receiveActivity="receiveTicketPriceInput"/>
        <messageLink name="AInvokeTAQuotePriceMessageLink" messageName="Invoke2_quoteTicketPrice_InputVariable"
            sender="currentAirline" sendActivity="InvokeTravelAgencyQuotePrice" receiver="TravelAgencyParticipant"
            receiveActivity="ReceiveAirlineGetPrice"/>
        <messageLink name="TAInvokeAOrderTicketMessageLink" messageName="ticketOrderVariable"
            sender="TravelAgencyParticipant" sendActivity="InvokeAirlineOrderTicket" receiver="selectedAirline"
            receiveActivity="PickOnMessageFromTravelAgencyTicketOrder"/>
    </messageLinks>
</topology>
```

Listing 2.18 Participant topology from FTBS scenario

2.2.3 Participant Grounding

Participant grounding is the only artifact of choreography which contains technical configuration details. Participant grounding specify the mapping to web-service specific configurations: links to WSDL definitions and XML schema types. In the code snippet below you can find grounding artifact of motivating scenario:



```
<grounding xmlns="urn:HPI_IAAS:choreography:schemas:choreography:grounding:2006/12"
    xmlns:cns="http://www.iaas.uni-stuttgart.de"
    xmlns:tns="urn:HPI_IAAS:choreography:schemas:choreography:topology:2006/12"
    topology="tns:MIPChoreography"
    targetNamespace="urn:HPI_IAAS:choreography:schemas:choreography:grounding:2006/12">
  <messageLinks>
    <messageLink name="TInvokeTAOrderTripMessageLink"
        portType="cns:TravelAgency" operation="orderTrip"/>
    <messageLink name="TAInvokeAGetPriceMessageLink"
        portType="cns:Airline" operation="getTicketPrice"/>
    <messageLink name="AInvokeTAQuotePriceMessageLink"
        portType="cns:TravelAgency" operation="receiveTicketPrice"/>
    <messageLink name="TAInvokeAOrderTicketMessageLink"
        portType="cns:Airline" operation="orderTicket"/>
  </messageLinks>
</grounding>
```

Listing 2.19 Participant topology from FTBS scenario

The above participant grounding grounds all three message links which were declared in participant topology. Grounding is only valid, if all message links are grounded [DEC09]. Message link named TAInvokeAGetPriceMessageLink is grounded by specifying port type and operation combination. This enables realization of one participant through different port types. If variables were declared at either receiving or sending activities, then the message type of the specified operation must match the given variable types.

The element *participantRefs* can be used inside participant grounding for passing participant references to third participant in BPEL4Chor choreographies, this concept is called link passing mobility [DKLW07]. But participant reference passing is not part of motivating scenario of this thesis, so will not be discussed further.

Weiß et al. [WASKV13] describe how to convert from a domain problem into executable business process. It is assumed that domain problem is explained in a plain text or in graphical process modeling language such as BPMN [WASKV13]. Then domain problem is modeled manually with choreography editor. Then this choreography can be transformed into BPEL4Chor choreography automatically [DKLW07], [DKLW09]. Afterwards three artifacts of BPEL4Chor choreography are used for generating abstract BPEL process. Basic Executable Completion takes an input abstract process and turns it into executable business process, but manual refinement is needed at the end.

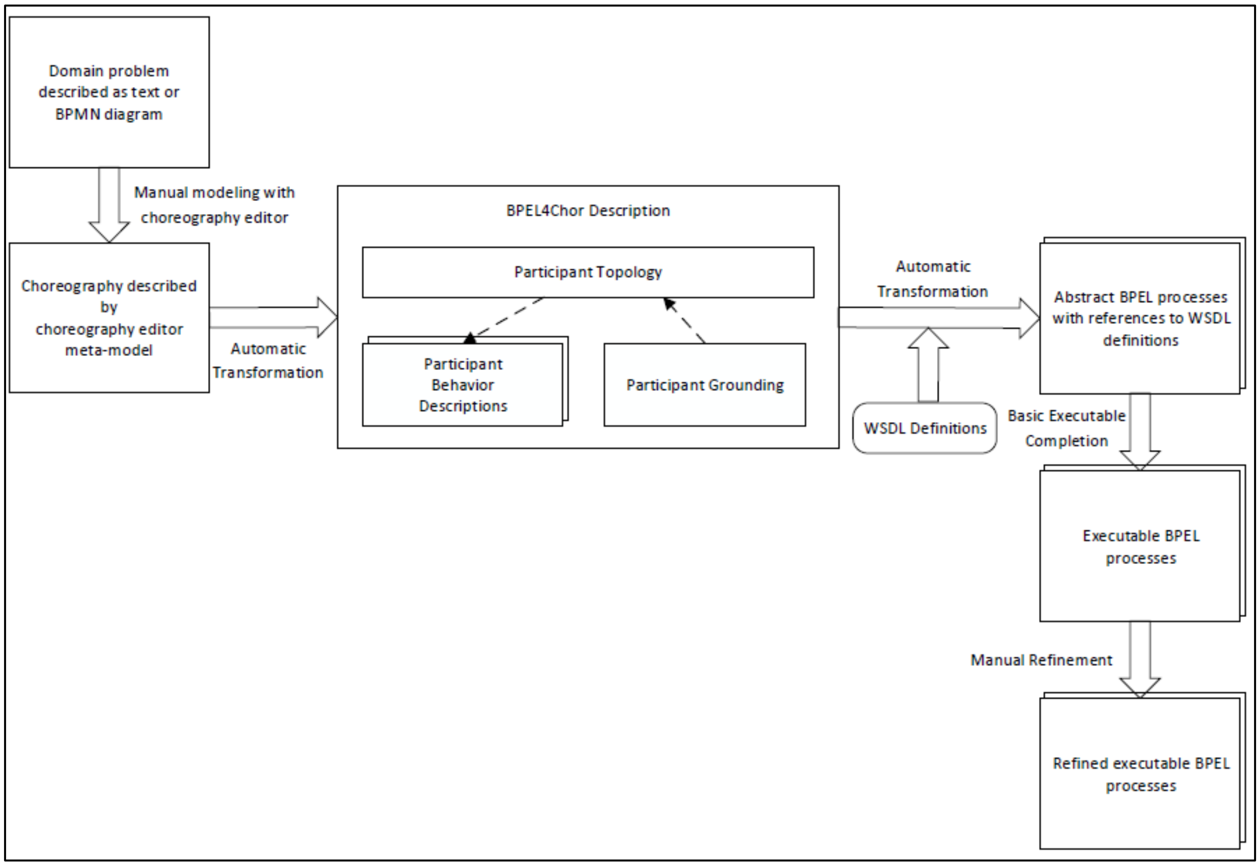

Domain problem described as text or BPMN diagram
Manual modeling with choreography editor
Choreography described by choreography editor meta-model
Automatic Transformation
BPEL4Chor Description
Participant Topology
Participant Behavior Descriptions
Participant Grounding
Automatic Transformation
WSDL Definitions
Abstract BPEL processes with references to WSDL definitions
Basic Executable Completion
Executable BPEL processes
Manual Refinement
Refined executable BPEL processes

Figure 2.4 How to convert a domain problem into an executable BPEL processes, taken from [WASKV13]

2.3 Allen's algebra

The Allen's algebra will be used for visual verification of correctness of merge patterns [WKL11]. Allen's algebra (which is also called interval algebra) covers thirteen distinct basic relations which can occur between two intervals *A* and *B*. Table 2.1 describes all these relations with pictorial and graphical examples. Each relation in Table 2.1 (except *A* equals *B* relation, which is singular relation without inverse relation) has inverse relation. Reverse relations are missing in this table as it's so obvious to derive from relation itself. For example, inverse relation of (*A* before *B*) is (*B* after *A*), with symbol representation of Allen's algebra $B > A$.

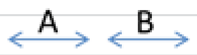
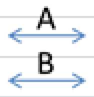
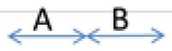
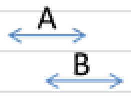
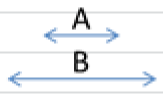
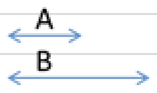
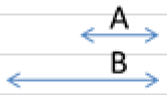
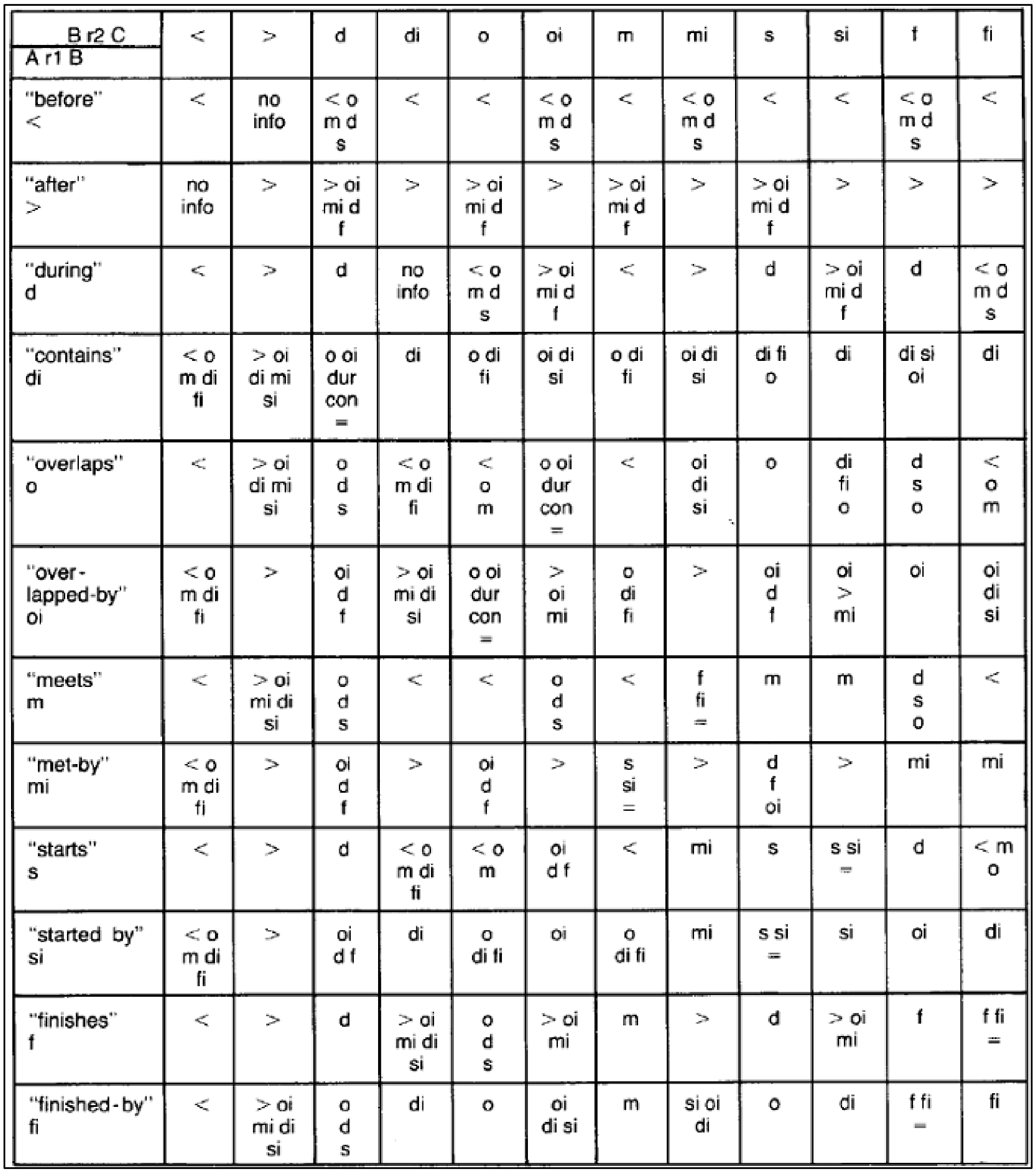
Relation	Symbol	Symbol for Inverse	Pictoral Example	Graphical Example
A before B	<	>	AAA BBB	
A equal B	=	=	AAA BBB	
A meets B	m	mi	AAABBB	
A overlaps B	o	oi	AAA BBB	
A during B	d	di	AAA BBBBBBB	
A starts B	s	si	AAA BBBBBBB	
A finishes B	f	fi	AAA BBBBBBB	

Table 2.1 Thirteen relations between A and B intervals of Allen's interval algebra adapted from [WKL12], [ALL83]

Allen [ALL83] has introduced how transitive relations can be derived for twelve relations (omitting equal relation). Table 2.2 describes the full table of transitivity relations which can be derived from thirteen distinct basic relations introduced before. Let's say there exist two relations *A* r1 *B*, and *B* r2 *C*. Let r1 represent before ("<") relation, in other words *A* before *B* relation; and let r2 represent before relation, in other words *B* before *C* relation. Then *A* before *C* transitive relation can be derived from these two basic distinct relations. All operations in on one cell of table can be combined with *OR* operation, which means any of these relations can be result of transitivity relation:



B r2 C / A r1 B	<	>	d	di	o	oi	m	mi	s	si	f	fi
"before" <	<	no info	< o m d s	<	<	< o m d s	<	< o m d s	<	<	< o m d s	<
"after" >	no info	>	> oi mi d f	>	> oi mi d f	>	> oi mi d f	>	> oi mi d f	>	>	>
"during" d	<	>	d	no info	< o m d s	> oi mi d f	<	>	d	> oi mi d f	d	< o m d s
"contains" di	< o m di fi	> oi di mi si	o oi dur con =	di	o di fi	oi di si	o di fi	oi di si	di fi o	di	di si oi	di
"overlaps" o	<	> oi di mi si	o d s	< o m di fi	< o m	o oi dur con =	<	oi di si	o	di fi o	d s o	< o m
"over-lapped-by" oi	< o m di fi	>	oi d f	> oi mi di si	o oi dur con =	> oi mi	o di fi	>	oi d f	oi > mi	oi	oi di si
"meets" m	<	> oi mi di si	o d s	<	<	o d s	<	f fi =	m	m	d s o	<
"met-by" mi	< o m di fi	>	oi d f	>	oi d f	>	s si =	>	d f oi	>	mi	mi
"starts" s	<	>	d	< o m di fi	< o m	oi d f	<	mi	s	s si =	d	< m o
"started by" si	< o m di fi	>	oi d f	di	o di fi	oi	o di fi	mi	s si =	si	oi	di
"finishes" f	<	>	d	> oi mi di si	o d s	> oi mi	m	>	d	> oi mi	f	f fi =
"finished-by" fi	<	> oi mi di si	o d s	di	o	oi di si	m	si oi di	o	di	f fi =	fi

Table 2.2 Transitivity Table for the Twelve Temporal Relations (omitting „=“) taken from [ALL83]

Allen’s interval algebra can be applied to determine relations between activities of same or different communicating processes. The advantage of using Allen’s algebra is that it is full algebra providing a set of operations for determining transitive relationships between activities [WKL12].

3 Consolidation of Multi Instance Partner Business Processes

The consolidation operation merges a set of n interacting processes which are part of same choreography into a single process named P_{Merged}. The consolidation process mostly keeps control flow dependencies between business activities of same process, as well as between business activities originating from different processes. The business activities are activities performing some business function. The peculiarity of business process consolidation operation is that it does not introduce any new BPEL language constructs or additions middleware for merging participating processes in choreography. The consolidated business process P_{Merged} will have better runtime and memory performance due to less communication (as communicating activities will be replaced by synchronization activities: `<assign>` and `<empty>` activities) between activities by exchanging SOAP messages.

In this chapter choreography based process consolidation approach is described for one-to-many interactions. Travel Ticket Management scenario is described as both an example of one-to-one and one-to-many interaction. Section 3.1 introduces how Allen's interval algebra can be used for visual verification of correctness of merge patterns. Section 3.2 introduces asynchronous and synchronous consolidation. In Section 3.3 traveler - travel agency interaction describes one-to-one interaction. Traveler states his requirements to travel agency. Assumption is made that Traveler contacts only one travel agency. Both traveler and travel agency plays role of "one" part of one-to-one interaction. In Section 3.4 travel agency – airline interactions describe one-to-many interactions. Travel agency contacts many/several airline for available tickets. In this scenario travel agency plays role of "one" part of interaction and airline plays "many" part of interaction. Section 3.5, Section 3.6, Section 3.7 describes how to determine MIP instantiation type which can be among static, dynamic and hybrid multi instance partner instantiations and which steps to follow for completing consolidation operation.

Two assumptions are made related to scenario used throughout this thesis:

1) It is assumed that there is only 1 traveler,1 travel agency, but N number of airline process instances

2) The best price means the cheapest price in all scenarios described in this thesis

Also several technical assumptions are made thorughout this thesis:

1) Nested loops are not supported in this thesis

2) It is assumed that `<forEach>` loops in different processes, each has unique name throughout the whole choreography

3) *startCounterValue* of all `<forEach>` loops starts from value 1.

4) Reference passing mechanism is not discussed for motivation scenario - FTBS.

5) It is assumed that there exists no parallel path to synchronization activities inside `<forEach>` loops, which makes fragmentation of `<forEach>` activity difficult.

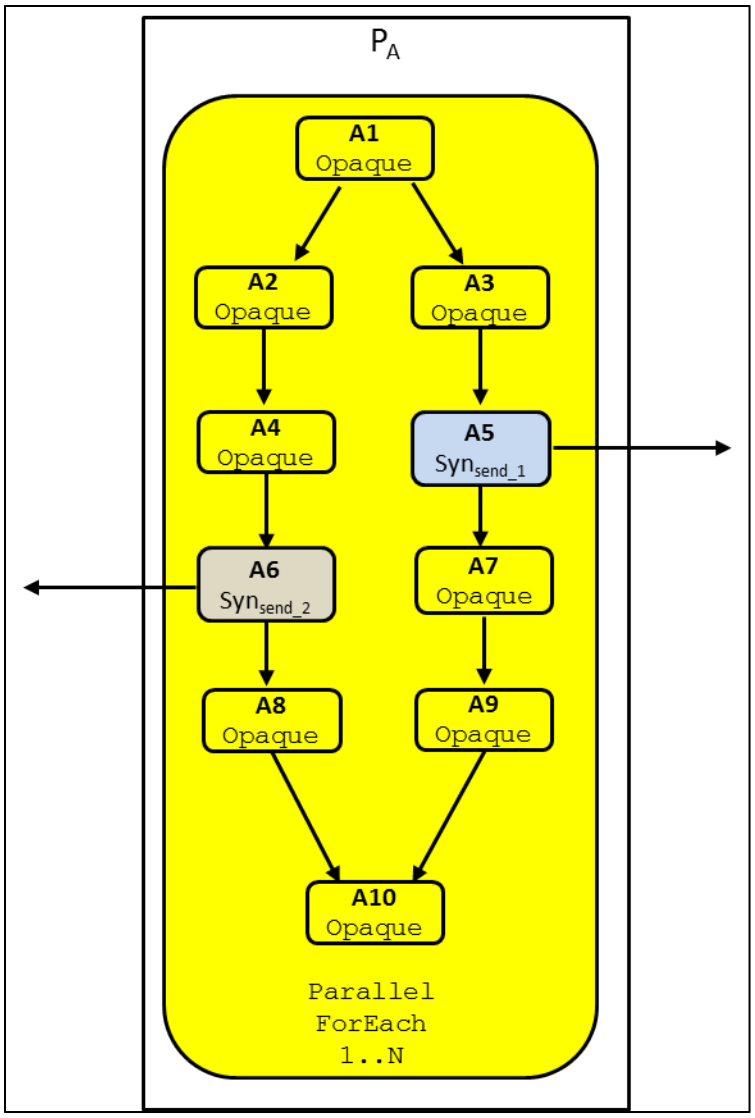
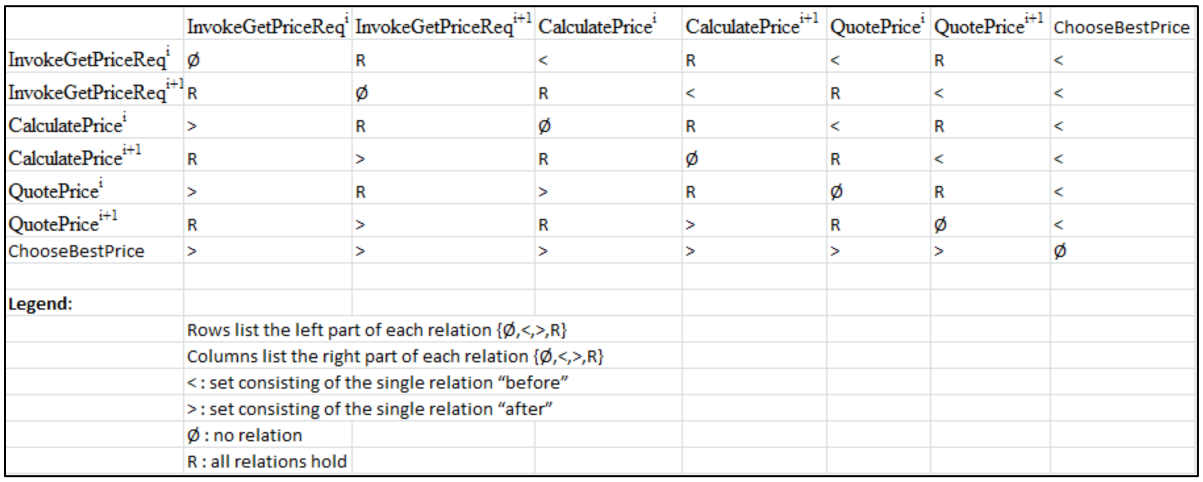


Figure 3.1 Parallel paths to synchronization activities inside `<forEach>` loop

Figure 3.1 illustrates process P_A, which has parallel `<forEach>` activity. There exist two synchronization activities *A5* and *A6*. This thesis does not cover this particular case during `<forEach>` loop fragmentation phase, described in Section 3.6.3. The problem with `<forEach>` loops containing activities parallel to communication activities is that it is impossible to propagate faults between parallel FE fragments. Let's say if n^{th} instance of FE_1 has failed, then all other related fragments (which are originating from the same `<forEach>` loop as FE_1) has terminate its n^{th} instance. But there is no way from outside making only n^{th} instance of another `<forEach>` loop to terminate.

3.1 Allen's Algebra applied to FTBS Scenario

After choreographies get merged, it still has to be proven that all relations which existed between activities in original choreographies are kept in merged choreography. Choreography based process consolidation is valid consolidation if all the control-flow dependencies between the activities in the merged choreography are reserved exactly same as in the originally choreography [WKL12]. Thus the Allen's algebra will be used for visual verification of correctness of merge patterns [WKL11].



	InvokeGetPriceReq^{i}	InvokeGetPriceReq^{i+1}	CalculatePrice^{i}	CalculatePrice^{i+1}	QuotePrice^{i}	QuotePrice^{i+1}	ChooseBestPrice
InvokeGetPriceReq^{i}	Ø	R	<	R	<	R	<
InvokeGetPriceReq^{i+1}	R	Ø	R	<	R	<	<
CalculatePrice^{i}	>	R	Ø	R	<	R	<
CalculatePrice^{i+1}	R	>	R	Ø	R	<	<
QuotePrice^{i}	>	R	>	R	Ø	R	<
QuotePrice^{i+1}	R	>	R	>	R	Ø	<
ChooseBestPrice	>	>	>	>	>	>	Ø

Legend:

Rows list the left part of each relation {Ø,<,>,R}

Columns list the right part of each relation {Ø,<,>,R}

< : set consisting of the single relation "before"

> : set consisting of the single relation "after"

Ø : no relation

R : all relations hold

Table 3.1 Allen's interval algebra applied to motivation scenario activities

Table 3.1 describes execution order relations between activities in travel agency and airline PBDs used throughout FTBS scenario. Only the relation of activities in the state "executing" is depicted and other states such as "faulted" are omitted here. Each activity has beginning state which is start-executing and end state which is end-executing. *InvokeGetPriceReq*^{i} activity means *InvokeGetPriceReq* activity which is called inside iteration *i* of `<forEach>` activity in travel agency choreography. As depicted in Table 3.1, all activities described in rows always enter and leave the state of "running" before an instance activity *ChooseBestPrice* (which is opaque activity in travel agency choreography). This is logical as all the prices from airline processes must be received in order to choose best price. The relation R indicates that there does not exist any control flow relations between activities, instances of airline process are independent of each other, and all those instances are created and executed concurrently, thus *QuotePrice*^{i} activity in airline process instance *i* and *QuotePrice*$^{i+1}$ activity in airline process instance *(i+1)* have no any control flow relation between each other. All these control flow constraints must hold after travel agency and airline process are consolidated in P_{Merged} process.

3.2 Asynchronous and Synchronous Consolidation

To capture the implicit control flow constraints between the execution orders of activities, the consolidation phase materializes those implicit control flow dependencies into explicit control flow relations. If one of the activities is long running activity, then implicit control flow constraints become hard to follow. In case of synchronous interaction, it is implied that the successor activities of a sending synchronous `<invoke>` activity are not started till it receives a response from the partner where it has sent a request message to before. Hence, the consolidation phase turns these implicit control flow constraints into the materialized control links.

The consolidation phase starts by putting all activities, control links and variables of two processes into a new single process P_{Merged} [WKL11]. The activities are put into a flow activity (child of P_{Merged} process) which enables a potential parallel execution of nested activities. During the consolidation phase new control links are added for keeping the original (before consolidation) execution order of the activities. Same variable names are renamed to achieve unique variable naming in the same scope. The purpose of consolidation phase is replacing communicating activities with synchronization activities. `<assign>` activities are used for synchronization.

During consolidation phase message links are replaced by control flow links [WKL13]. Message links need to be analyzed before determining consolidation type: synchronous or asynchronous consolidation. Let *ML* be a message link declared in topology artifact. Let *S* an asynchronous `<invoke>` activity and *RC* be `<receive>` activity declared in *ML*. In consolidation phase *S* is replaced by Syn_s `<assign>` activity and *R* is replaced by an `<empty>` activity Syn_{RC}. The visibility scope of v_{RC} variable is changed to scope of P_{Merged} process, so it can be accessed throughout the whole process. Syn_s activity copies the message from the input variable v_s of the invoke activity into the variable v_{rc} of receive activity. An `<assign>` activity inherits incoming and outgoing links of *S* and <empty> activity inherits incoming and outgoing links of *R* as its own incoming and outgoing links. The additional link from Syn_s to Syn_{RC} ensures that Syn_{RC} is started only after Syn_s activity. The graphical representation of asynchronous consolidation can be found in Figure 3.2 taken from [WKL13]:

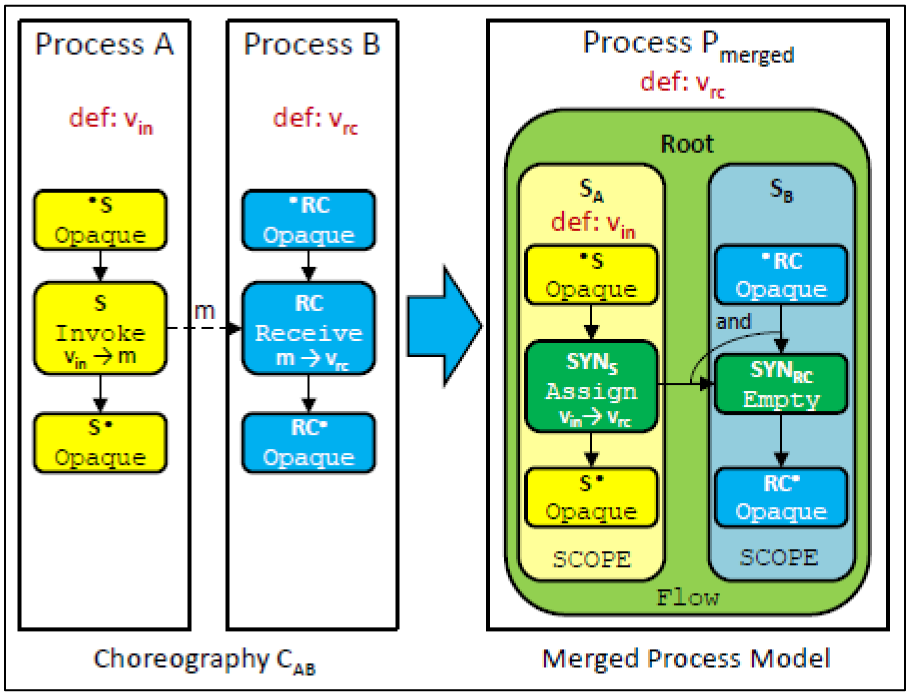


Figure 3.2 Asynchronous Merge Operation, taken from [WKL13]

In case of synchronous scenario, besides of `<invoke>` and `<receive>` activities there exists a `<reply>` activity *RP* to an `<invoke>` activity. In this case `<invoke>` activity plays role of sender in invoke-receive activities interaction and the role of receiver in reply-invoke activities interaction. The visibility scope of v_{OUT} variable is changed to the scope of P_{Merged} process, to make it accessible throughout the whole merged process. During consolidation phase the `<reply>` activity will be replaced by Syn_{RP} activity., which copies the value of v_{RP} variable into the variable v_{OUT}. *RC* `<receive>` activity is replaced by Syn_{RC}, and new Syn_{SR} activity is created to emulate receiving role of `<invoke>` activity *S*. The control links of *RC* are mapped into Syn_{RP} and the outbound links of *S* are mapped into Syn_{RC}. The newly created control link between Syn_{RP} and Syn_{SR} ensures that Syn_{SR} is started only after Syn_{RP}. Another newly created control link Syn_s and Syn_{RC} is added for the same purpose as in asynchronous consolidation.

The Figure 3.3 illustrates synchronous consolidation scenario graphically:

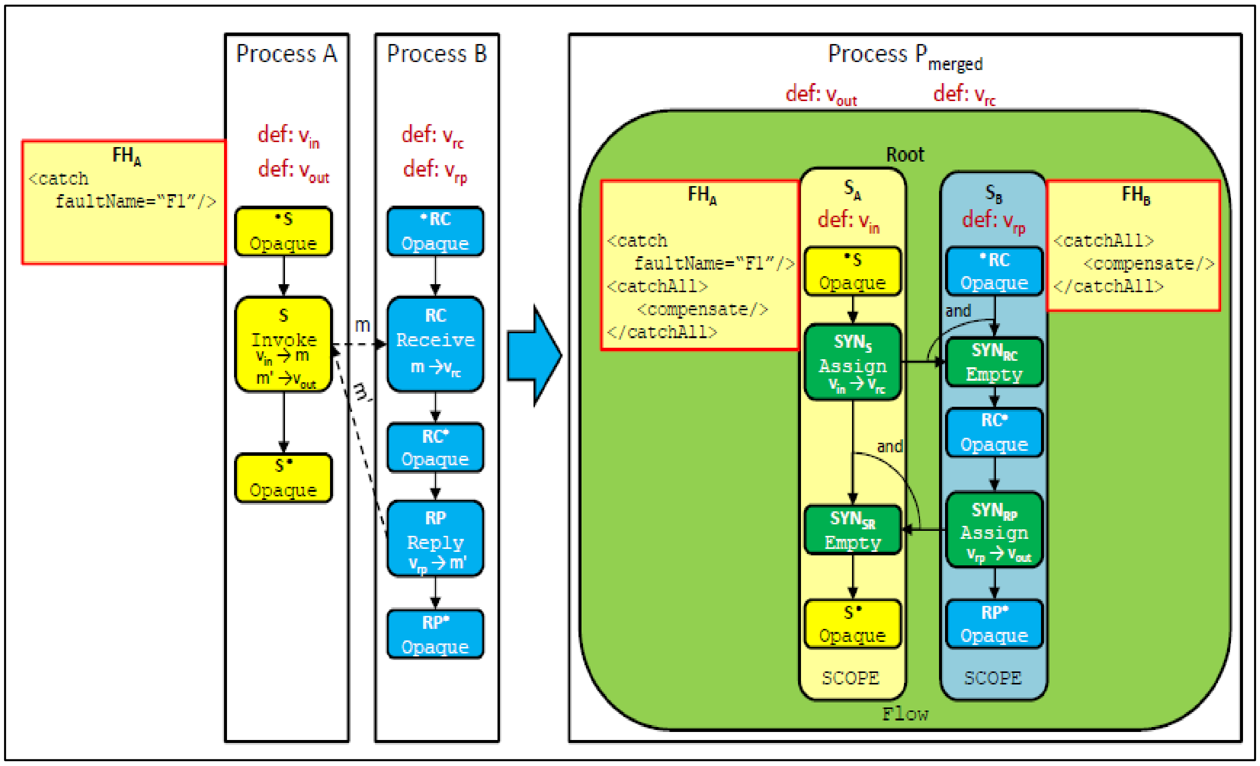


Figure 3.3 Synchronous Merge Operation, taken from [WKL13]

Throughout this thesis only asynchronous consolidation was used. Figure 3.4 illustrates consolidation model by applying asynchronous consolidation when merging travel agency and only one instance of airline processes (activities not related to travel agency and airline interaction are disregarded from this figure):

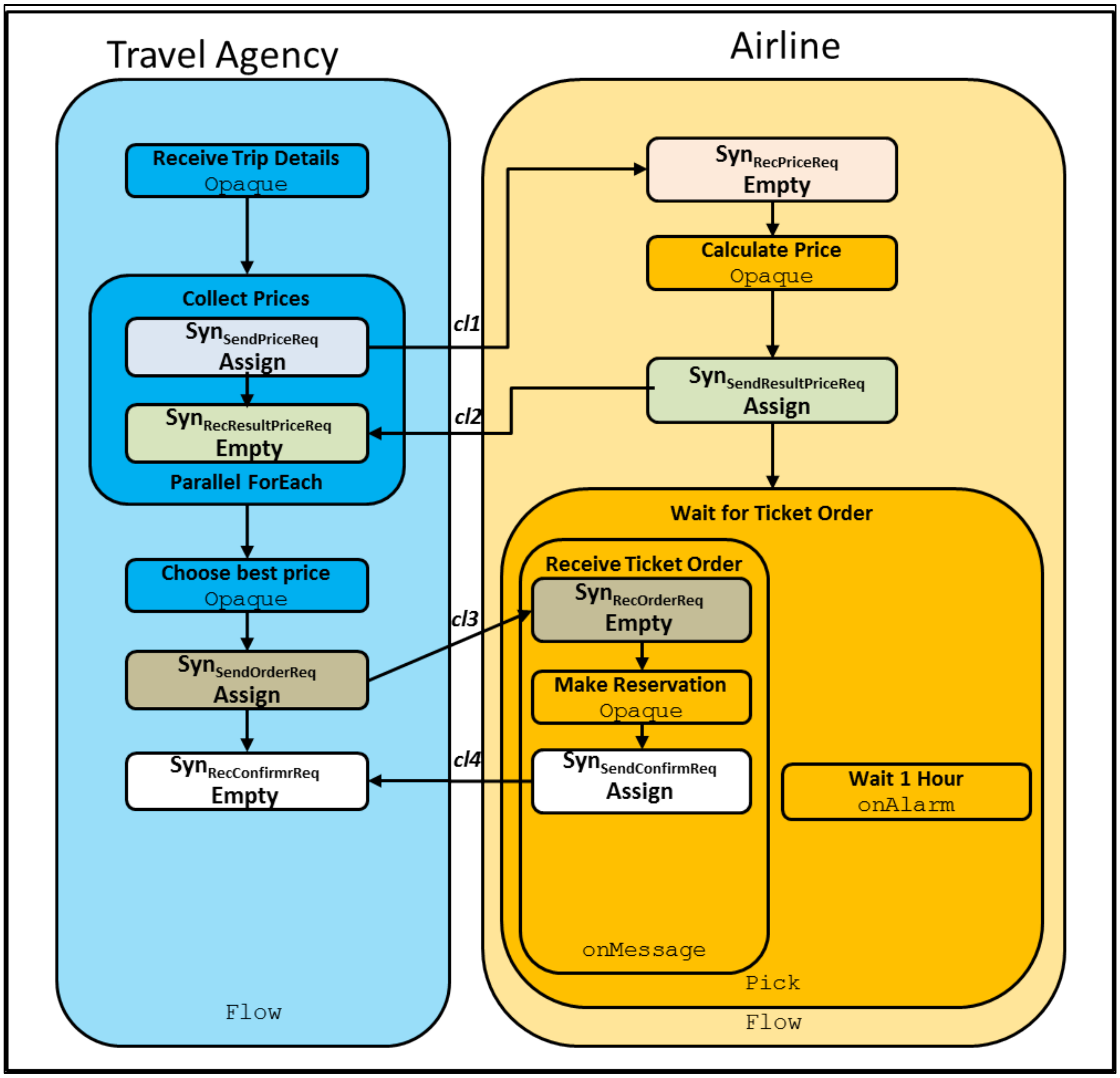


Figure 3.4 Travel Agency and Airline Consolidation Model

`<onMessage>` consolidation pattern is not supported yet, and that's why it will be ignored in consolidation phase.

Synchronization consolidation mechanism is applied four times in the scenario described above. Each synchronization activity has two parts: assign and empty part, which are painted in the same color to be differentiated easily.

3.3 FTBS Scenario: Traveler – Travel Agency as an Example of One-to-One Interaction

In this real world scenario – FTBS, there are 3 business processes interacting with each other:

- Traveler business process – is a customer who supplies travel details to travel agency.

- Travel agency business process – is a company receiving traveler flight details and contacting several airline companies for current ticket prices satisfying traveler flight requirements.

- Airline Company business process – receives flight details from travel agency, and provides up-to-date ticket prices to travel agency.

This section will focus on two participant types only: traveler and travel agency.

Existence of one-to-one interaction can be determined by analyzing topology artifact. In topology artifact each `<participant>`, as being child of `<participants>`, represents "one" part of one-to-one interaction. But participants inside `<participantSet>` must be ignored for both one-to-one and one-to-many interactions.

Traveler process is a calling process, and travel agency is a called process. In one-to-one interactions calling process will have exactly one `<invoke>` activity which calls `<receive>` activity with *createInstance="yes"* attribute in called process. `<receive>` activity with *createInstance="yes"* attribute is also exactly one in called process. `<invoke>` activity is the only activity used as instance of send activity throughout this thesis. It is worth to mention that the number of executed instances of invoke activities influences the number of created instances. In other words, if there are only mutual exclusive instance creating `<invoke>` activities then this is still a one-to-one scenario. Receiving activity can be instance of `<receive>` or `<pick>` activities. But in this thesis only `<receive>` activity is used as receiving activity in called process.

3.3.1 Determining Number of Containers for One-to-One Interaction

As a result of the process consolidation phase, in merged process P_{Merged} two containers will be generated. One container will be generated for calling process, which is traveler, and another container is created for called process, which is travel agency.

3.3.2 Container Generation Phase

Container is a `<scope>` activity for one-to-one interactions. New container will be generated for each `<invoke>` activity calling `<receive>` activity with *createInstance* attribute having „yes" value. Again only number of executed `<invoke>` activities influence the number of container generation, so it should be carefully revised when there exist mutually exclusive instance creating <invoke> activities – only one of them influences container generation, the one which is executed during run time. According to BPEL's execution semantics the container will only be executed if the invoke itself is had been executed, which will enable dead path elimination. Container generation for one-to-one interaction is 2 step process:

1) For traveler participant new container scope $CS_{traveler}$, and for travel agency participant new container scope $CS_{travelAgency}$ is created. This container scope is called static container. Static container is `<scope>` construct of BPEL language. Then these scopes are added as children of <flow name="MergedFlow"> in merged P_{Merged} process.

2) Traveler PBD is copied into $CS_{traveler}$, and TravelAgency PBD is copied into $CS_{travelAgency}$.

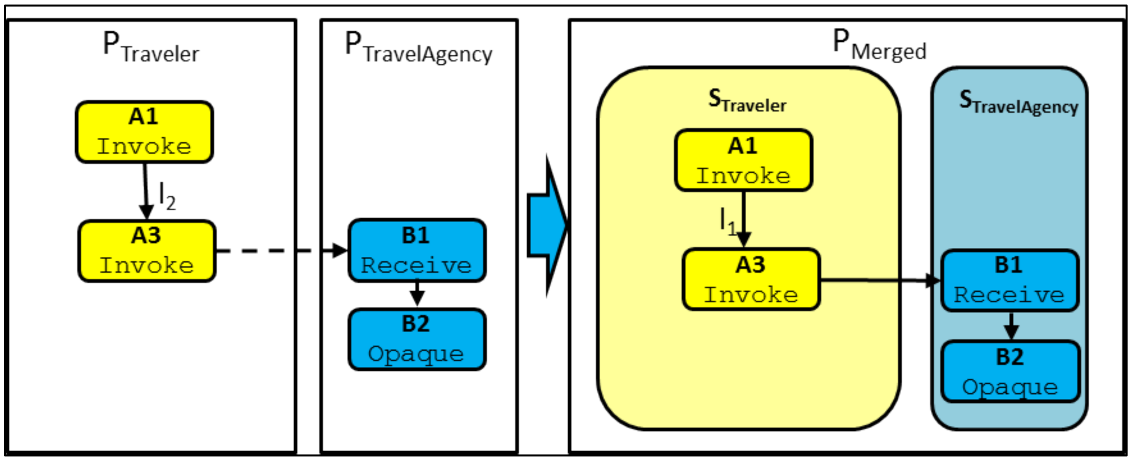


Figure 3.5 One-To-One Partner Instantiation and Container Generation

Figure 3.5 illustrates one-to-one interaction scenario where travel process instantiates only 1 instance of travel agency process. Hence only one container scope $S_{TravelAgency}$ needs to be created in P_{Merged} for partner process.

3.4 FTBS Scenario: Travel Agency – Airlines as an Example of One-to-Many Interactions

One assumption is made for one-to-many interaction part related to FTBS scenario:

- For one-to-many interaction the calling process must have `<forEach>` activity (or any other BPEL loop activities). `<forEach>` activity allows creating multi instances of same participant type.

There exist two different scenarios depending on type of `<forEach>` activity. In first scenario, `<forEach>` activity is parallel `<forEach>` activity, hence travel agency will invoke N number of airline participants at the same time. In this particular scenario `<forEach>` activity has *startCounterVariable* equals to one, and *finalCounterVariable* equals to three. Figure 3.6 and Figure 3.7 illustrate runtime of MIP instantiations, thus it differs graphically from design time representation of activities.

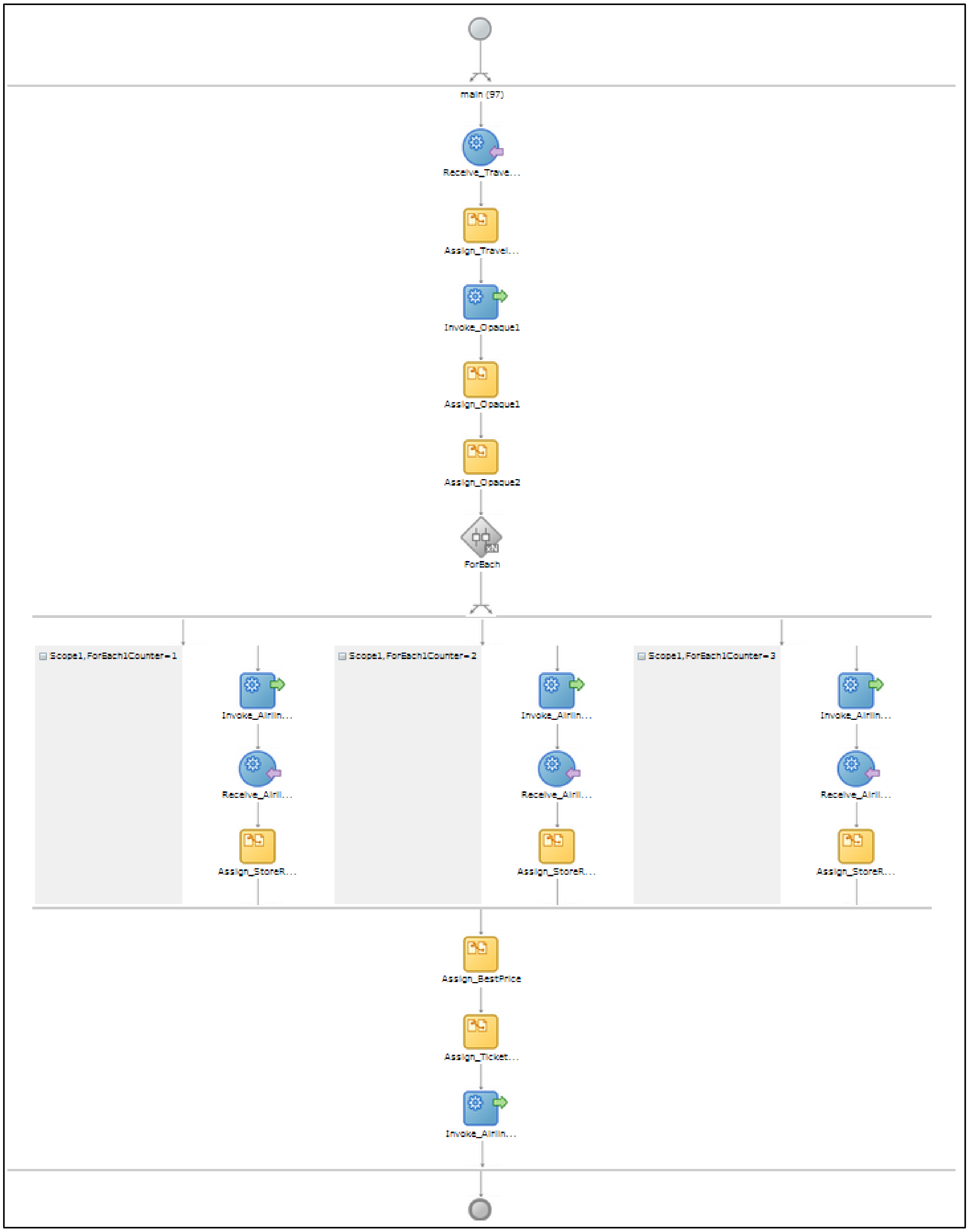


Figure 3.6 Parallel MIP instantiation

In second scenario, `<forEach>` activity is serial/sequential activity, then as shown in Figure 3.7, travel agency participant will invoke airline_1 participant, and only after receiving ticket price from airline_1 participant, then travel agency will invoke airline_2 participant and this sequence will be kept till the last – airline_N participant is invoked.

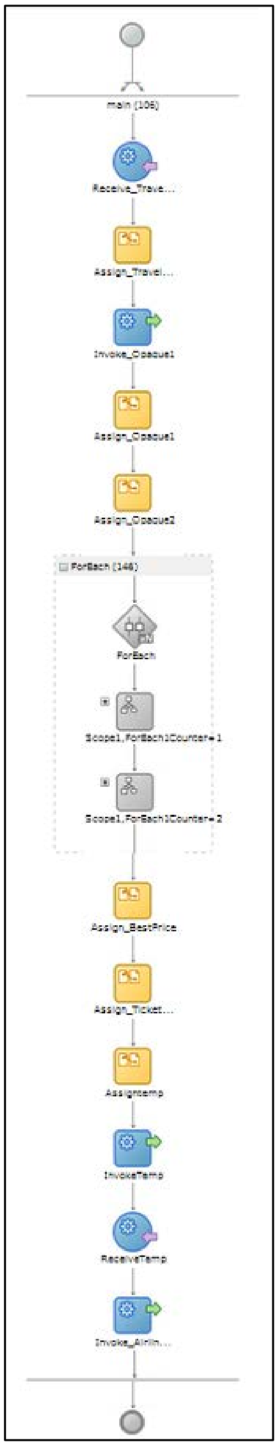


Figure 3.7 Sequential MIP instantiation

For better viewing activities inside `<forEach>` activity iterations are minimized.

Throughout this thesis only parallel multi-instance partner instantiation will be used.

3.4.1 Multi Instance Partner (MIP) Instantiation from BPEL4Chor Perspective

This section will describe how to determine the type of multi instance partner instantiation depending on analysis of topology and participant behavior description artifacts. After determining the type of MIP instantiation the corresponding container type will be created:

- If static MIP Instantiation, then static container will be created for corresponding PBD in merged process. Static container means corresponding PBD will be simple copied into new `<scope>` in merged process without any change.

- If dynamic MIP Instantiation, then dynamic container will be created for corresponding PBD in merged process. Dynamic container means corresponding PBD scope will be copied into corresponding `<forEach>` container which resides in new `<scope>` in merged process.

3.4.2 Determining Type of MIP Instantiation

Type of MIP Instantiation can be determined by analyzing topology and corresponding PBD artifacts together. The below two sections will describe how MIP instantiation types and counts can be determined by two approaches.

3.4.2.1 By Analysis of Participants and Participant Sets in Topology Artifact

The existence of participants and participant sets can be checked in topology artifact. The information about `<forEach>` activity's *finalCounterVariable*'s value can be determined from corresponding PBD artifact.
As described in Figure 3.8 , as a result of participant and participant set analysis in topology artifact, this is static MIP Instantiation in the cases:

1) If there does not exist a participant set in topology and there exists at least 1 participant of certain type

   OR

2) If there exists a participant set, which has at least 1 `<forEach>` activity, with *finalCounterVariable*'s value can be determined at design time – in other words *finalCounterVariable* has some constant integer value. Condition 2 in Figure 3.8 has to be checked for each `<forEach>` activity and corresponding path needs to be taken.

This is Dynamic MIP Instantiation in the case:

- If there exists a participant set with at least one `<forEach>` activity with *finalCounterVariable*'s value which can be known only at run time – in other words *finalCounterVariable* is variable, not constant.

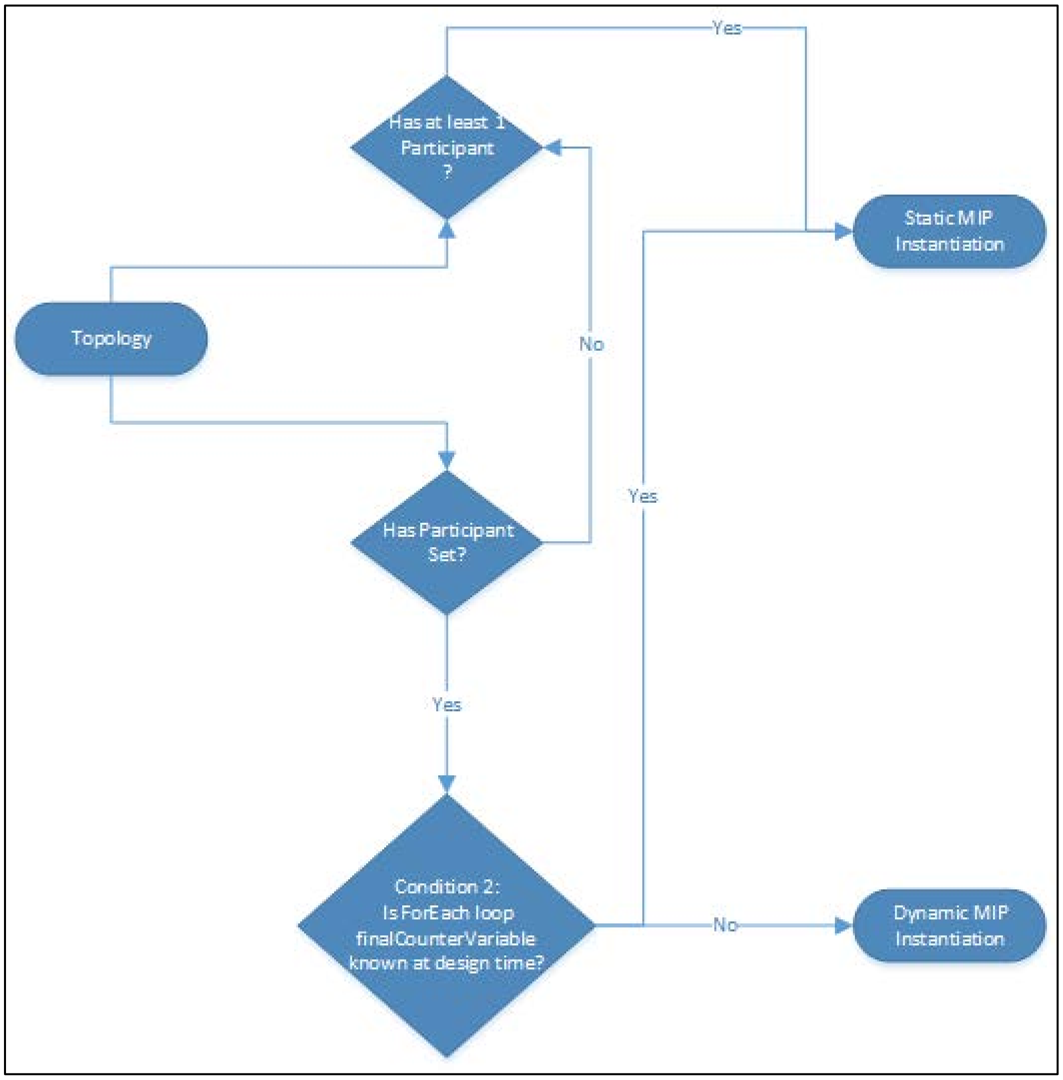

Yes
Has at least 1 Participant ?
Static MIP Instantiation
Topology
No
Yes
Has Participant Set?
Yes
Condition 2: Is ForEach loop finalCounterVariable known at design time?
No
Dynamic MIP Instantiation

Figure 3.8 Determining MIP instantiation type from topology artifact and PBDs

3.4.2.2 By Analysis of Message Links in in Topology Artifact

The existence of message links can be checked in topology artifact. Value of *createInstance* attribute of receiving activities can be checked from corresponding PBD artifact. Receiving activities with *createInstance* attribute can be `<receive>` activity. Also checking corresponding `<invoke>` activities (which communicates with `<receive>` activities with *createInstance*=”yes” attribute) reside in `<forEach>` activity or not, can be done by analysis of corresponding PBD. If Invoke activity resides inside `<forEach>` activity, then this is dynamic MIP Instantiation, else this is static MIP instantiation:

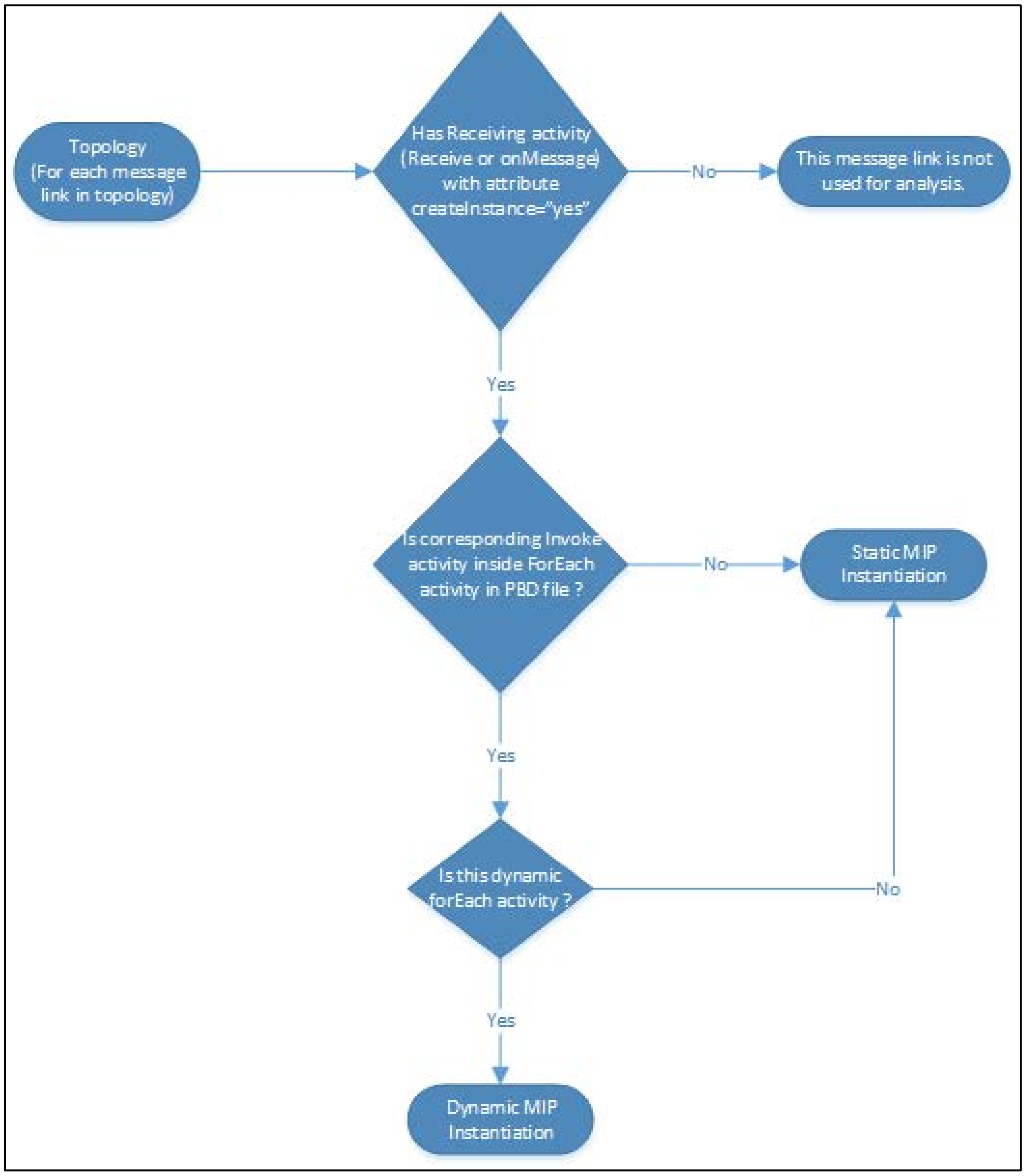

Topology (For each message link in topology)
Has Receiving activity (Receive or onMessage) with attribute createInstance="yes"
No
This message link is not used for analysis.
Yes
Is corresponding Invoke activity inside ForEach activity in PBD file ?
No
Static MIP Instantiation
Yes
Is this dynamic forEach activity ?
No
Yes
Dynamic MIP Instantiation

Figure 3.9 Determining MIP instantiation type from message links in topology artifact

3.5 Static MIP Instantiation in Merged Process

After type of MIP instantiation is determined static containers need to be created in merged process.

3.5.1 Determining Number of Static Containers to be created in Merged Process

At first, number of containers to be created in merged process has to be determined. In static MIP instantiation the number of containers is determined by analyzing message links in topology artifact. Containers must be created according to message links which has `<invoke>` activities as sender activity and `<receive>` activities as receiving activity with attribute *createInstance="yes"*. Only number of executed `<invoke>` activities influence the number of container generation, so it should be carefully revised when there exist mutually exclusive instance creating <invoke> activities – only one of them influences container generation, the one which is executed during run time. `<invoke>` activities must be differentiated in two ways:

1) `<invoke>` activity has `<forEach>` activity as parent. Then number of containers to be created is equal to *finalCounterVariable* of `<forEach>` activity.

2) `<invoke>` activity does not have `<forEach>` activity as parent. Then only one container scope needs to be created for each `<invoke>` activity in merged process.

3.5.2 Static Container Generation Phase

The reason of creating containers is same as for one-to-one interaction scenarios described in Section 3.3.2. Container or container scope is new `<scope>` activity, and in case of static MIP instantiation the container is called static container. After number of containers to be created in merged process is determined, then container generation phase starts. In static MIP instantiation container creation is two step processes:

1) New scope S_{static} is created as child of merged process P_{Merged}.

2) Corresponding PBD is copied into that scope S_{static} .

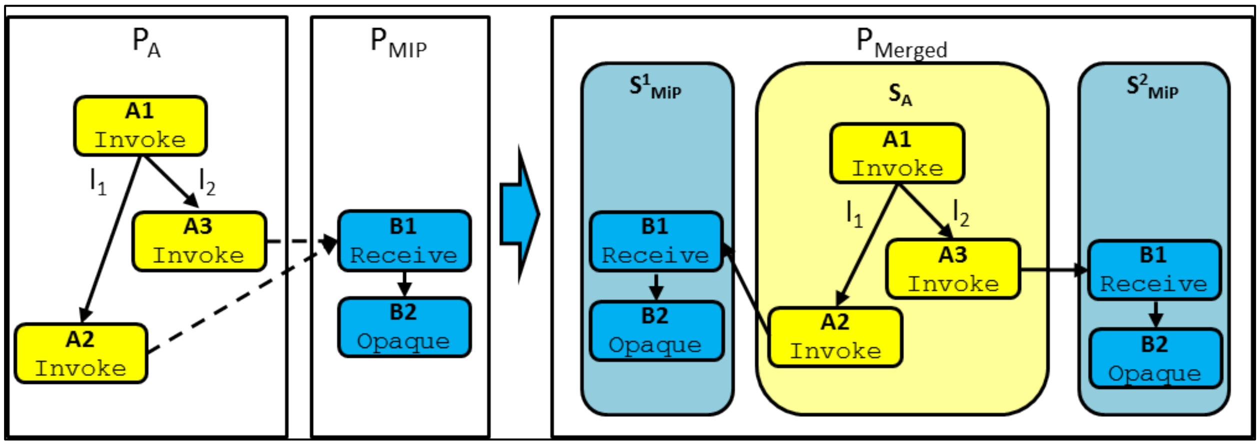


Figure 3.10 Static MIP instantiation and container generation: invoke case

Figure 3.10 illustrates static multi-instance-partner instantiation and container generation phases graphically. P_{MIP} is the instance of multi instance partner. P_{MIP} is called by process P_A, which contains two invoke activities: *A2* and *A*3, which are both creating new instance of P_{MIP} instances. *B2* `<receive>` activity of P_{MIP} process has *createInstance* attribute which is set to “yes” value. The right side of Figure 3.10 illustrates container generation phase. In this particular scenario, number of containers (which is two) is determined by number of `<invoke>` activities calling `<receive>` activity (with *createInstance=”yes”* attribute) in P_{MIP} process. S^1_{MiP} and S^2_{MiP} are two static container scopes generated during consolidation phase.

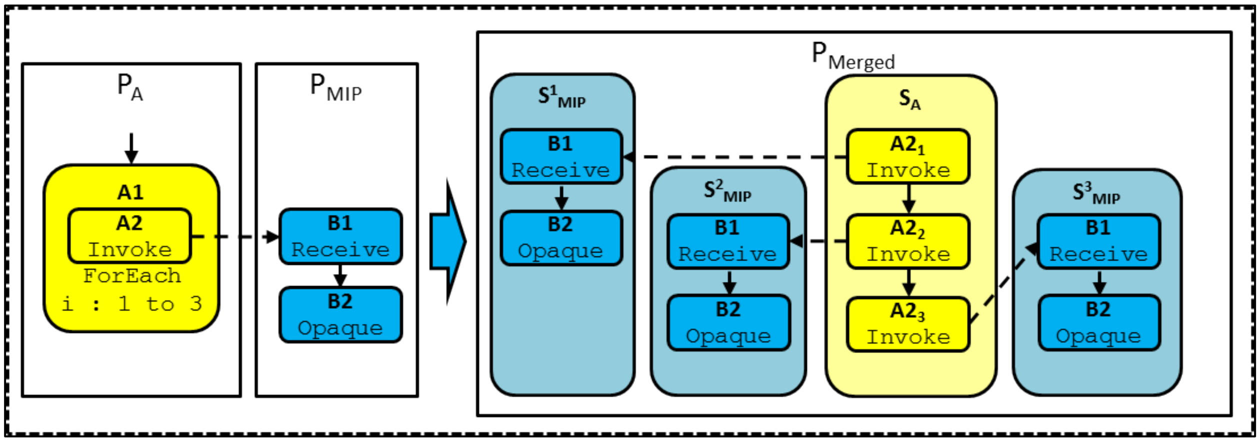


Figure 3.11 Static MIP instantiation and container generation: static sequential forEach case

Figure 3.11 illustrates static multi-instance-partner instantiation and container generation phase based on static sequential `<forEach>` activity. P_A process has `<forEach>` activity, which has *startCounterValue* equal to one, and *finalCounterValue* equal to three. `<invoke>` activity *A2* is child activity of `<forEach>` activity. Hence *A2* `<invoke>` activity will be called three times in total for three iterations of `<forEach>` activity. *B1* `<receive>` activity in P_{MIP} process, has attribute *createInstance* attribute set to "yes". Thus three static containers need to be generated. In case of parallel static `<forEach>` activity case, $A2_1$, $A2_2$, $A2_3$ `<invoke>` activities will be invoked concurrently and all three containers will be generated at the same time.

3.6 Dynamic MIP Instantiation

The number of instances to be created is not known at design time. In order to create a priory unknown number of instances parallel `<forEach>` activity will be used for container creation.

3.6.1 Determining Number of Dynamic Containers to be created in Merged Process

Number of containers to be created in dynamic MIP instantiation can be determined from message links in topology artifact. The interested message links are those which has `<receive>` activity with *createInstance="yes"* attribute. After those message links are determined, then send activities of those message links are analyzed. In dynamic MIP instantiation phase, we are only interested only in those invoke activities whose parents are dynamic `<forEach>` activities. If we have *m* `<invoke>` activities as children of `<forEach>` activity with *finalCoutnerVariable=n*, then number of containers to be created in merged process will be *(n*m)*. (Only executed `<invoke>` activities are assumed out of mutual exclusive instance creating `<invoke>`s) The steps below describe how to determine number of containers for dynamic MIP instantiation will be:

1) get all message links form topology artifact and store in $\text{Set}_{\text{allMsgLinks}}$

2) select message links from $\text{Set}_{\text{msgLinks}}$ where `<receive>` activity has attribute *createInstance="yes"* and store these message links in $\text{Set}_{\text{instanceCreatingMsgLinks}}$

3) get corresponding `<invoke>` activities from $\text{Set}_{\text{instanceCreatingMsgLinks}}$ and store in $\text{Set}_{\text{InstanceCreatingInvokes}}$

4) select `<invoke>` activities from $\text{Set}_{\text{InstanceCreatingInvokes}}$ which has as its parent dynamic `<forEach>` activity and store them in $\text{Set}_{\text{selectedInvokes}}$

5) Number of containers to be created in merged process will be size_of($\text{Set}_{\text{selectedInvokes}}$) x (finalCounterVariable_of_ ForEach)

3.6.2 Dynamic Container Generation Phase

The reason of creating containers is same as for one-to-one interaction scenarios described in Section 3.3.2. As the number of process instances involved in choreography is not known at design time, hence the multi-instance process cannot be unrolled into different containers as in static multi-instance creation scenarios. The only way for creating a priory unknown number of instances of certain process in BPEL is creating them using parallel dynamic `<forEach>` activity. Thus container in dynamic MIP instantiation is <scope> activity having dynamic parallel <forEach> activity as its child. This container is called dynamic container. Dynamic container generation phase is three step process:

1) New scope $\text{S}_{\text{dynamic}}$ is created as parent of merged process P_{Merged}.

2) New `<forEach>` activity $ForEach_{dynamicContainer}$ is created as the only child of $S_{dynamic}$. $ForEach_{dynamicContainer}$ will have same attributes and values as in calling process. The only attribute will matter is *suppressJoinFailure* attribute's value need to be taken from the called process.

3) Corresponding PBD is copied as child element of $ForEach_{dynamicContainer}$.

Figure 3.12 and Figure 3.13 illustrates dynamic parallel and serial `<forEach>` activity cases for multi-instance-partner instantiation and container generation.

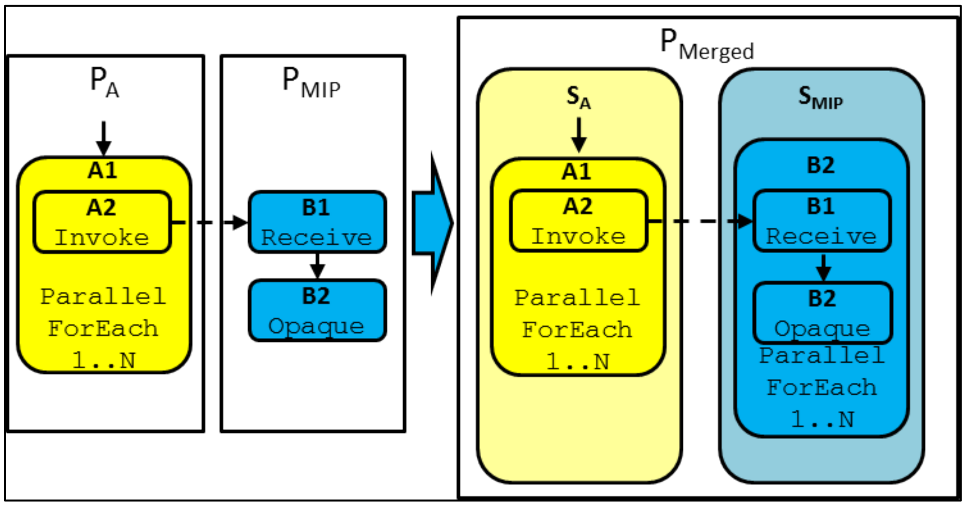


Figure 3.12 Dynamic MIP Instantiation and Dynamic Container Generation: Dynamic, Parallel `<forEach>` Case

In Figure 3.12, P_A process has one parallel `<forEach>` activity with *startCounterValue=1* and *finalCounterValue=N*. *A2* `<invoke>` activity is child activity of `<forEach>` activity. *B1* `<receive>` activity of P_{MIP} process has *createInstance* attribute which is set to "yes" value. As there is only one dynamic `<forEach>` activity (and also only one `<invoke>` activity calling *B1* receive activity in P_{MIP}), so only one dynamic container S_{MIP} need to be generated. S_{MIP} dynamic container is `<forEach>` activity which inherits all its attributes (except *suppressJoinFailure* attribute value must be inherited from P_{MIP} process) from `<forEach>` activity in P_A process.

Figure 3.13 describes dynamic serial `<forEach>` activity case. In this case the only difference from the scenario illustrated in Figure 3.12 is that, S_A container in P_{Merged} process contains dynamic serial `<forEach>` activity corresponding to dynamic serial `<forEach>` activity in P_A process. It's worth mentioning that in both dynamic parallel and serial `<forEach>` cases, the dynamic container S_{MIP} created in P_{Merged} process is dynamic parallel `<forEach>` activity.

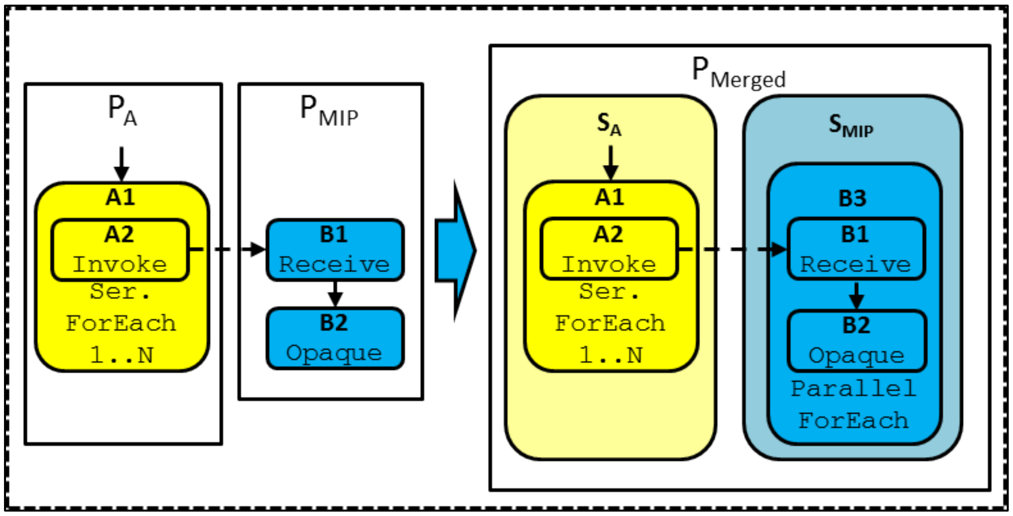


Figure 3.13 Dynamic MIP Instantiation and Dynamic Container Generation: Dynamic, Serial `<forEach>` Case

3.6.3 Loop Fragmentation for resolving Cross Boundary Link Violations

According to WS-BPEL 2.0 specification, a link must not cross the boundary of a repeatable construct. `<forEach>` activity is also repeatable activity. After choreography PBDs get consolidated into P_{Merged} process, new links are created for supporting communication between communicating activities residing in different `<forEach>` loops. Hence those links are violating cross-boundary constraint of WS-BPEL specification. Thus loop fragmentation technique is used as a solution for handling cross-boundary link violations.

This section introduces the algorithm for resolving cross boundary link violations by applying loop fragmentation approach. Loop fragmentation enables dividing `<forEach>` loop residing in dynamic container, created in P_{Merged}, into several `<forEach>` loop fragments. Later in this section link status propagation approach will be used for recovering broken control links between fragments.

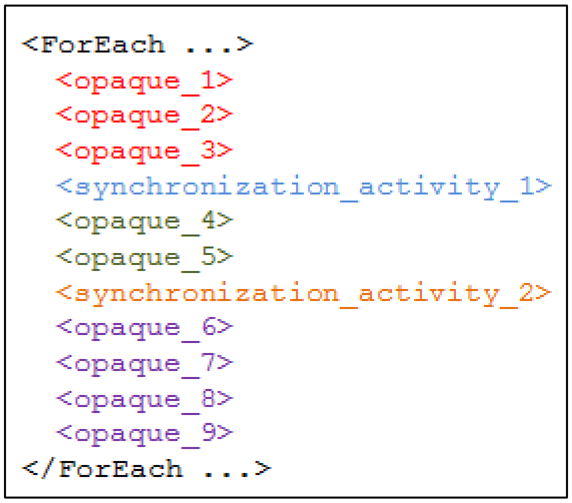
Loop fragmentation algorithms main purpose is dividing `<forEach>` loop, into several `<forEach>` fragments. Fragments are newly created `<forEach>` constructs of the BPEL language. Loop fragmentation will create new fragments for:

1) All activities until first synchronization activity will be grouped into a newly created fragment.

2) Synchronization activities: send activity - Syn_{send} , and receive activity Syn_{rec} will be moved into newly created fragment.

3) All activities coming after synchronization activity till next synchronization activity (if there is any) will be grouped into new different fragment.

In other words synchronization activities play the role of delimiter for creating new fragments.



```
<ForEach ...>
  <opaque_1>
  <opaque_2>
  <opaque_3>
  <synchronization_activity_1>
  <opaque_4>
  <opaque_5>
  <synchronization_activity_2>
  <opaque_6>
  <opaque_7>
  <opaque_8>
  <opaque_9>
</ForEach ...>
```

Listing 3.1 Abstract example for illustrating loop fragmentation technique

The above given example in Listing 3.1 describes how activities in `<forEach>` loop residing in dynamic container can be segmented into different fragments. Opaque activities can be any activity which is not synchronization activity. In other words opaque activities are business activities, but not communicating activities. Same color activities must be put into same fragment. As a result of loop fragmentation five new `<forEach>` loop fragments will be created.

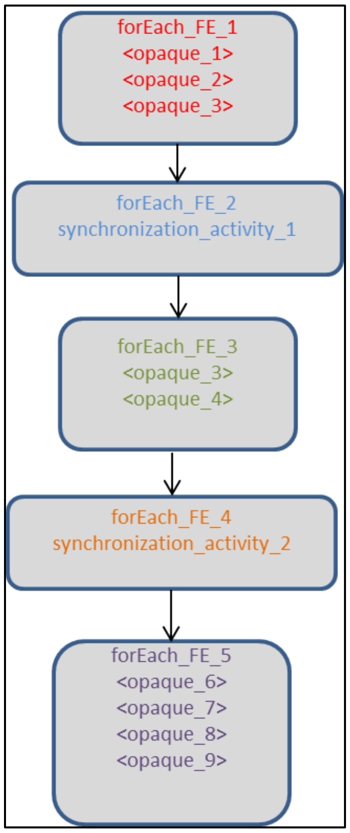


Figure 3.14 Loop fragmentation result, applied on example illustrated at Listing 3.1

Loop fragmentation algorithm is described by pseudo code in Listing 3.2:

```
(1)  performLoopFragmentation($act_{forEach\_send}$)
(2)  begin
(3)    … store all preceding activities of $act_{forEach\_send}$ in $ForEachPrecActs_{send}$ list
(4)    … store all succeeding activities of $act_{forEach\_send}$ in $ForEachSuccActs_{send}$ list
(5)    … store all preceding activities of $Syn_{rec}$ in $PrecActs_{rec}$ list
(6)    … store all succeeding activities of $Syn_{rec}$ in $SuccActs_{rec}$ list
(7)    … create new <forEach> $FE_{syn}$ in CS of $Syn_{send}$
(8)    … move each pair of $Syn_{send}$ and $Syn_{rec}$ into $FE_{syn}$
(9)    if ( $Syn_{send}$.precedingOpaqueActsInForEach() ≠ null )
(10)       … create new <forEach> $FE_{pred\_syn}$
(11)       … move all $Syn_{send}$.preceddingOpaqueActsInForEach() activities into $FE_{pred\_syn}$ fragment
(12)   fi
(13)   if ( $Syn_{rec}$.precedingOpaqueActsInForEach() ≠ null )
(14)       … create new <forEach> $FE_{pred\_rec}$
(15)       … move all $Syn_{send}$.precedingOpaqueActsInForEach() activities into $FE_{pred\_rec}$ fragment
(16)   fi
(17)   if ( $FE_{syn}$ != null && $FE_{pred\_rec}$ != null )
(18)       … create new control link $CL_{syn\_predRec}$($FE_{syn}$, $FE_{pred\_rec}$)
(19)   fi
(20)   if ( $FE_{syn}$ != null && $FE_{pred\_send}$ != null )
(21)       … create new control link $CL_{predSend\_syn}$($FE_{pred\_send}$ , $FE_{syn}$)
(22)   fi
(23)   if ( $Syn_{send}$.succeedingOpaqueActsInForEach() ≠ null )
(24)       … create new <forEach> $FE_{succ\_send}$
(25)       … move all $Syn_{send}$.succeedingOpaqueActsInForEach() activities into $FE_{succ\_send}$ fragment
(26)       … create new control link $CL_{syn\_succSend}$ ($FE_{syn}$, $FE_{succ\_send}$)
(26)   else
(27)       … connect $FE_{syn}$ to $ForEachSuccActs_{send}$ activities residing outside $act_{forEach\_send}$
(28)   fi
(29)   if ( $Syn_{rec}$.succeedingOpaqueActsInForEach() ≠ null )
(30)       … create new <forEach> $FE_{succ\_rec}$
(31)       … move all $Syn_{rec}$.succeedingOpaqueActsInForEach() activities into $FE_{succ\_rec}$ fragment
(32)       … create new control link $CL_{syn\_succRec}$ ($FE_{syn}$, $FE_{succ\_rec}$)
(33)   else
(34)       … connect $FE_{syn}$ to $SuccActs_{rec}$ activities residing after $Syn_{rec}$
(35)   fi
(36)   … all opaque activities that do not precede any synchronization activity are left in their
(37)   corresponding original <forEach> fragment and need to be connected to $FE_{syn}$ fragment
(38)   if ($ForEachPrecActs_{send}$ ≠ null)
(39)     if ($FE_{pred\_send}$ != null)
(40)                 … connect $ForEachPrecActs_{send}$ activity list to $FE_{pred\_send}$
(41)         else
(42)                 … connect $ForEachPrecActs_{send}$ activity list to FEsyn
(43)         fi
(44)     fi
…
```

```
(45)    if ($PrecActs_{rec} \neq null$)
(46)      if ($FE_{pred\_rec}$ != $null$)
(47)            … connect $PrecActs_{rec}$ activity list to $FE_{pred\_rec}$
(48)        else
(49)            … connect $PrecActs_{rec}$ activity list to $FEsyn$
(50)        fi
(51)      fi
(52)  end
```

Listing 3.2 Loop fragmentation algorithm

performLoopFragmentation($act_{forEach\_send}$) function takes one `<forEach>` activity as its argument. $act_{forEach\_send}$ represents `<forEach>` activity where send part of consolidation resides. $act_{forEach\_send}$ can be thought as `<forEach>` activity residing in travel agency process calling airline participant instances.

Lines [3,6] describes of getting all the activities preceding $act_{forEach\_send}$ `<forEach>` activity (Syn_{rec} activity) and storing them in $ForEachPrecActs_{send}$ ($PrecActs_{rec}$) lists.

Lines [7,8] creates new `<forEach>` loop container FE_{syn} in the container scope of Syn_{send}. The newly created fragment FE_{syn} inherits the attributes, handlers, start and end counter values from original `<forEach>` of Syn_{send}. Each pair of activities Syn_{send} and Syn_{rec} violating cross boundary link constraint must be moved into the fragment FE_{syn} .This steps guarantees that no data will be modified by Syn_{send} if another activity fails in container scope of Syn_{send}.

Lines [9,12] checks if there exist activities directly/indirectly preceding Syn_{send} , then create new fragment FE_{pred_send}. All opaque activities directly or indirectly preceding Syn_{send} (but not other synchronization activity) are removed from their original `<forEach>` loop and moved into fragment FE_{pred_send}.

Lines [13,16] checks if there exist activities directly/indirectly preceding Syn_{rec} activity (but not other synchronization activity) then create new fragment FE_{pred_rec} in the container scope of Syn_{rec} , as the opaque activities preceding Syn_{rec} resides in the scope of Syn_{rec}. All opaque activities directly/indirectly preceding Syn_{rec} (but not other synchronization activity) are removed from its original `<forEach>` loop and put into fragment FE_{pred_rec}.

Lines [17,19] creates new control link $CL_{syn\_predRec}(FE_{syn}, FE_{pred\_rec})$ and connects two <forEach> loop fragments - FE_{syn} and FE_{pred_rec}.

Lines [20,22] creates new control link $CL_{predSend\_syn}(FE_{pred\_send} , FE_{syn})$ and connects two `<forEach>` fragments - FE_{pred_send} and FE_{syn}.

Lines [23,28] connects FE_{syn} fragment either to the `<forEach>` fragment which contains the direct successor activities of Syn_{send}, or directly to successor activities of Syn_{send} that do not reside within `<forEach>` fragment.

Lines [29,35] connects FE_{syn} fragment either to the forEach fragment which contains the direct successor activities of Syn_{rec}, or directly to the successor activities of Syn_{rec} that do not reside within `<forEach>`.

All opaque activities that do not precede any synchronization activity (and residing inside forEach activity) are left in their corresponding original forEach loop fragments and need to be connected to FE_{syn} fragment which contains their preceding Syn_{send} and Syn_{rec} respectively.

Lines [38,51] are for handling activities not residing inside any of forEach activities. As they were grouped into two lists on lines [3,6], now those two lists of activities need to be connected respectively either to FE_{pred_rec} (or FE_{pred_send}), if exists, or to FE_{syn} directly.

Figure 3.15 illustrates abstract example of two processes - which need to be consolidated into one process, before loop fragmentation:

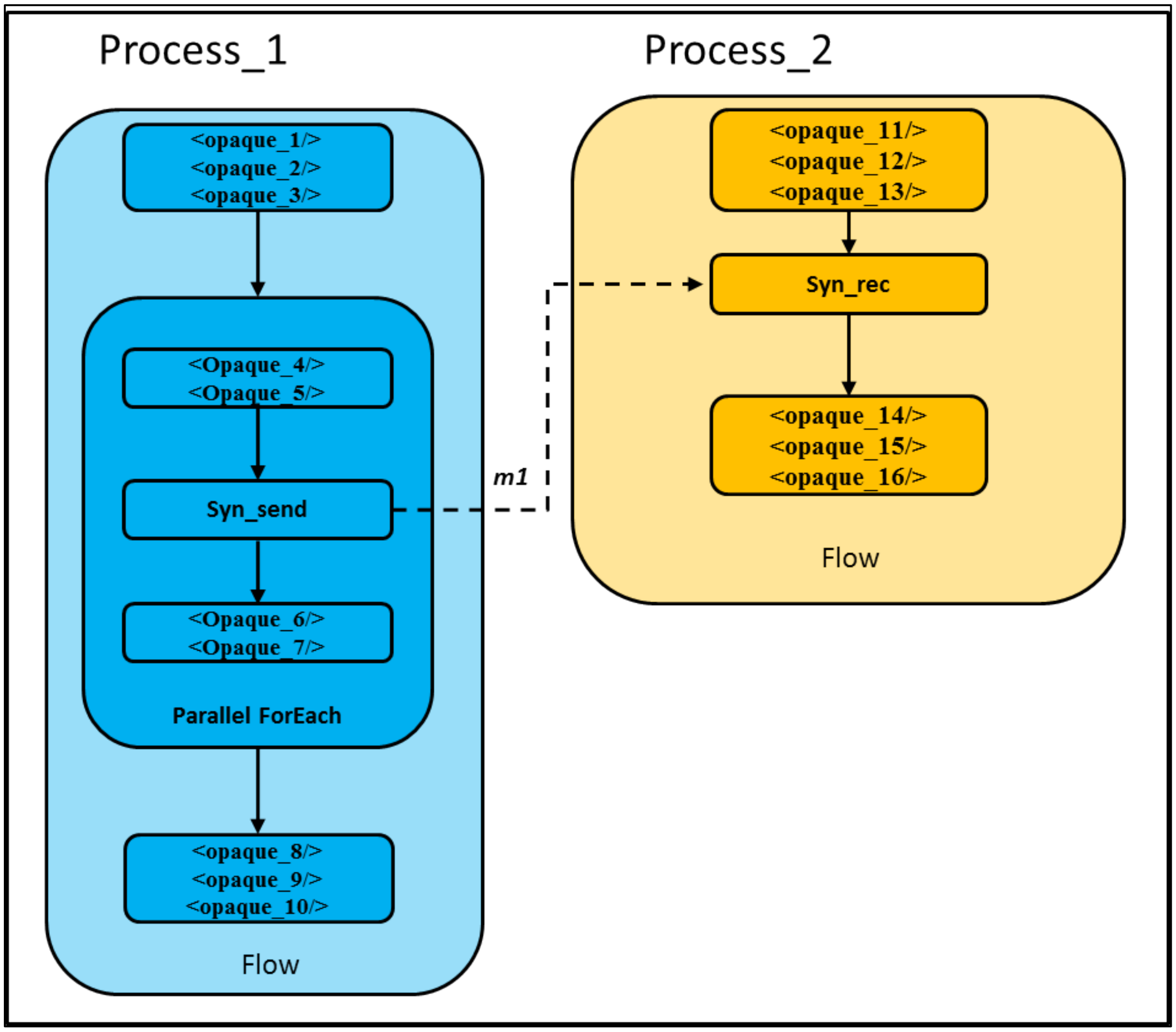


Figure 3.15 Two abstract business processes before applying loop fragmentation

Figure 3.16 demonstrates the result of loop fragmentation algorithm applied to the example in Figure 3.14:

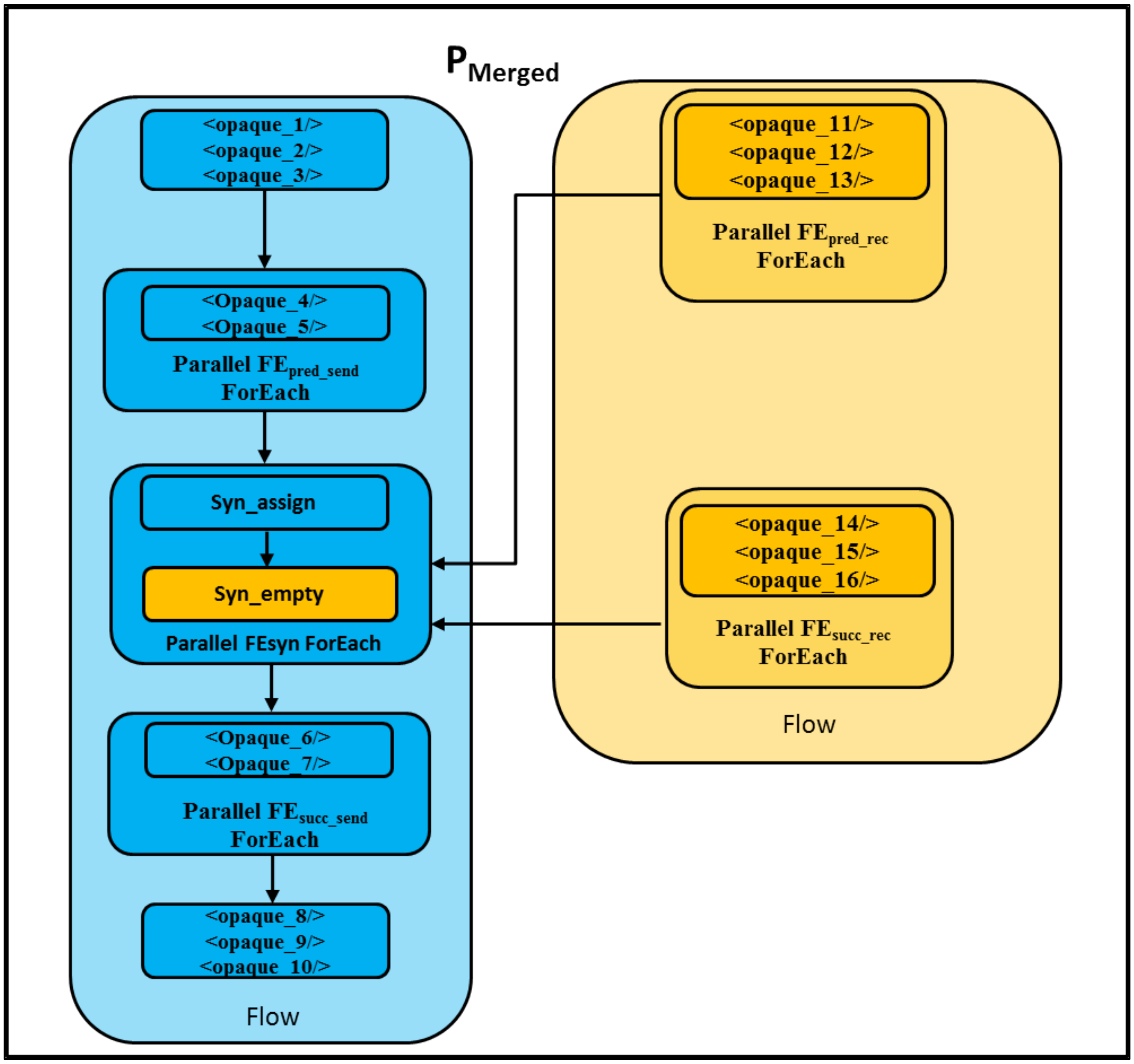


Figure 3.16 Loop fragmentation applied to example described in Figure 3.14

`<opaque>` activities from *1* to *3*, and from *8* to *10* reside outside parallel `<forEach>` activity, they remain unchanged after fragmentation algorithm is applied. `<opaque>` activities *4* and *5* – which reside inside `<forEach>` activity, also precede synchronization activity *Syn_send*, are moved from its original container into new `<forEach>` container named FE_{pred_send}. Pair of synchronization activities (Syn_{send} and Syn_{rec}) are replaced with `<assign>` and `<empty>` activities respectively and moved into new `<forEach>` container named FE_{syn}. `<opaque>` activities from *11* to *13* were preceding activities of Syn_{rec} activity, they are moved into new container called FE_{pred_rec} and connected to FE_{syn} fragment with new control link. `<opaque>` activities from 14 to 16 were the activities succeeding Syn_{rec}, are moved into new fragment named FE_{succ_rec} and connected to FE_{syn} fragment with new control link. `<opaque>` activities *6* and *7* – resided inside `<forEach>` activity and were succeeding activities of Syn_{send} activity, were moved into new container named FE_{succ_send}.

<opaque> activities from *1* to *3* are executed first. Secondly <opaque> activities residing inside *FE_{pred_send}* get executed. Execution of *FE_{syn}* fragment starts by execution of *FE_{pred_rec}* fragment, followed by execution of *Syn_assign* and *Syn_empty* activities and then proceeds with execution of *FE_{succ_rec}* fragment. *FE_{syn}* fragment finishes its execution only after *FE_{succ_rec}* fragment terminates. *FE_{succ_send}* fragment and opaque activities from 8 to 10 are executed in consecutive order.

Figure 3.17 illustrates the result of loop fragmentation algorithm applied to the executable business processes – travel agency and airline:

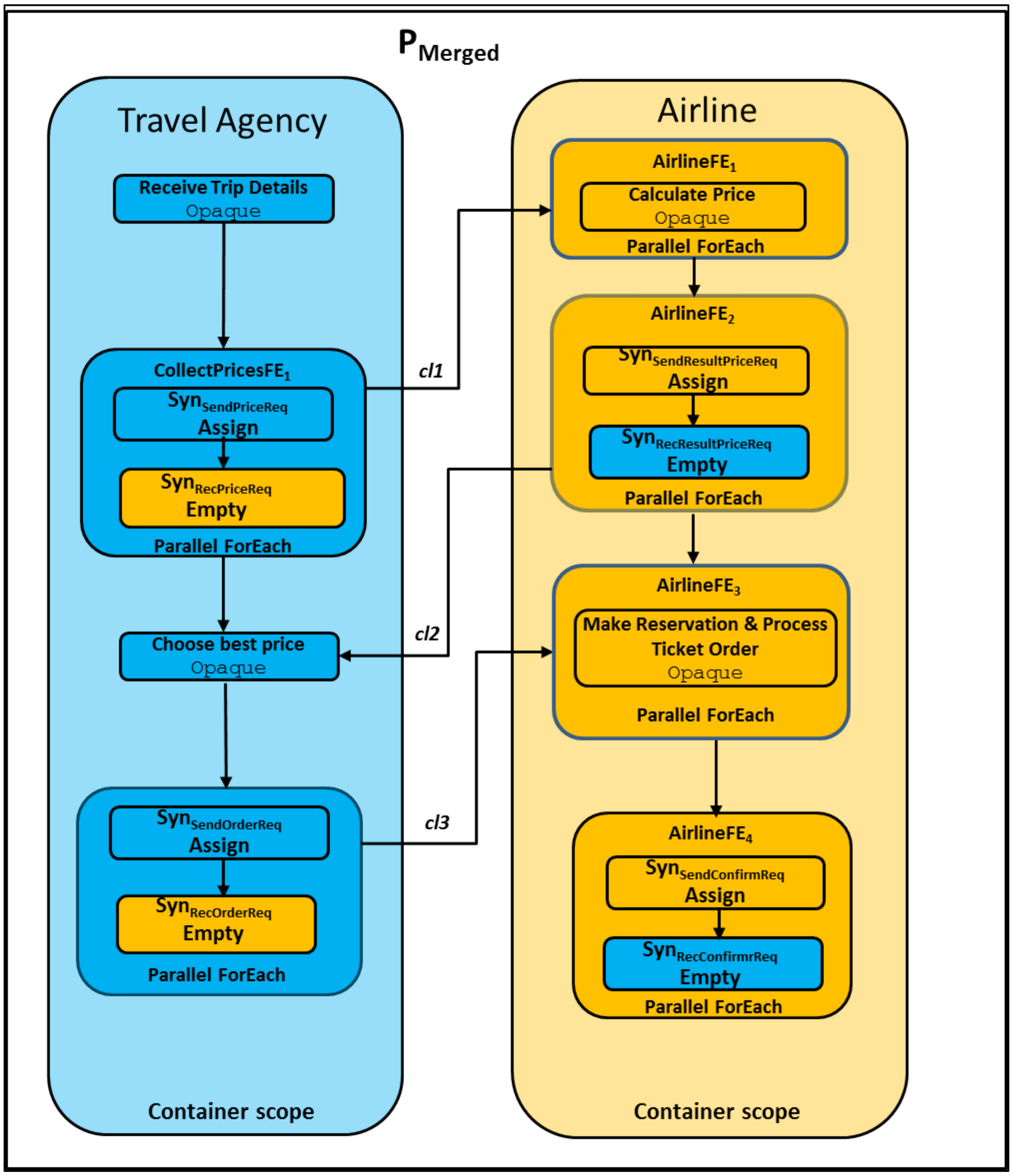


Figure 3.17 Travel agency and airline consolidation after fragmentation

Figure 3.17 illustrates result of applying loop fragmentation algorithm to travel agency and airline business process consolidation model. There is no preceding and succeeding activities of Syn_{send} in parallel ForEach loop in travel agency container scope, thus FE_{pred_send} is missing. AirlineFE$_1$ fragment represents FE_{pred_rec} fragment. It can also be seen that each `<empty>` activity part of synchronization consolidation is moved into same fragment of `<assign>` activity part of synchronization consolidation. Extra created control links *cl1*, *cl2* and *cl3* ensures that activity execution order is preserved as in original choreographies before consolidation.

3.6.4 Link Status Propagation Technique

After `<forEach>` activity in dynamic container scope is divided into several new `<forEach>` fragments, control links become broken. The links become broken due to the reason that source and target communicating activities may be residing in two different fragments after loop fragmentation is applied to the container scope. Thus, further steps need to be completed for handling broken control links between different `<forEach>` fragments:

1) The control links between activities which resides inside of same FE fragment must be kept in order to maintain the original execution (before consolidation) order of activities.

2) All incoming links of activities whose predecessors reside within another FE fragments are removed

3) When the successor activities of an activity are moved to another fragment, then all of its outgoing links are removed.

Khalaf and Leymann [KL06] describe an approach to split a single process into several individual process fragments, as a solution to broken control links, they described technique to propagate the link status from one process fragment to another fragment via message exchange. This thesis adapts this approach to propagate the link status from one FE fragment into another fragment by using variables, instead of message exchanges. Let activities *Src* and *Trg* be connected via control link *CL(Src,Trg, tc)* and transition condition *tc*. Before consolidation *Src* and *Trg* activities were in the same fragment - `<forEach>` container, but after consolidation they are put into different FE fragments FE_{src} and FE_{trg}. In order consolidation phase complete successfully FE_{src} and FE_{trg} fragments need to go through some changes. The steps described below need to be executed consecutively in FE_{src} fragment:

```
... create new LinkStatus boolean variable in the parent scope $FE_{src}$ and $FE_{trg}$
performLinkStatusPropagation($FE_{src}$, $FE_{trg}$)
begin
  … create new scope $Scope_{src}$ as child of $FE_{src}$
  $Scope_{src}$.supressJoinFailure=false
  … create new fault handler $FH_{src}$
  $Scope_{src}$.addFaultHandler($FH_{src}$)
  … create new assign activity $Assign_{true}$
  $Scope_{src}$.getChildren.add($Assign_{true}$)
  … create new assign activity $Assign_{false}$
  $FH_{src}$.getChildren().add($Assign_{false}$)
  … create new control link $CL_{src}$ (Src, $Assign_{true}$, tc)
...
```

Listing 3.3 Link status propagation - adapting changes to source fragment

At first new variable, named *LinkStatus*, is created in the parent scope of FE_{src} and FE_{trg} fragments. *LinkStatus* variable will hold the value of link status transition. Second new `<scope>` $Scope_{src}$ is created as child of FE_{src} fragment and its *suppressJoinFailure* attribute is set to "no". Third new fault handler FH_{src} is created and added into `<scope>` $Scope_{src}$. Fourth new `<assign>` activity $Assign_{true}$ is created and added as child of `<scope>` $Scope_{src}$. $Assign_{true}$ activity writes "true" to the value of *LinkStatus* variable. Fifth new `<assign>` activity $Assign_{false}$ is created and added as a child of fault handler FH_{src}. $Assign_{false}$ activity writes value "false" to variable *LinkStatus*. At last new control link $CL_{src}(Src, Assign_{true}, tc)$ is created.

These changes in FE_{src} fragment cover all potential paths of process exertion flow:

a) When transition condition *tc* in control link $CL_{src}(Src, Assign_{true}, tc)$ evaluates to "false", then *bpel-joinFailure* fault will be raised. This fault will be caught by FH_{src} and $Assign_{false}$ activity will be executed by setting *LinkStatus* variable to "false".

b) If *tc* n control link $CL_{src}(Src, Assign_{true}, tc)$ evaluates to "true", then $Assign_{true}$ activity will be executed by assigning "true" to *LinkStatus* variable.

In order link status propagation complete successfully, FE_{trg} fragment need to adapt these changes (in consecutive, order described in Listing 3.4):

```
  ... create new empty activity Emp
  $FE_{trg}$.getChildren().add(Emp)
  … create new control link $\mathbf{CL_{trg}}$(Emp, Trg, read(LinkStatus))
end
```

Listing 3.4 Link status propagation - adapting changes to target fragment

FE_{trg} fragment hosts Trg activity as its child. In order to adapt changes to target fragment for link status propagation, new `<empty>` activity *Emp* is created and added as child of FE_{trg} fragment. Then new control link CL_{trg}(*Emp*, *Trg*, *read*(*LinkStatus*)) is created for connecting *Emp* <empty> activity to *Trg* activity.

All changes done to FE_{src} fragment and then to FE_{trg} fragment ensure that:

1) The value of *LinkStatus* is always set before *Emp* <empty> activity reads it.

2) The instance of *Emp* `<empty>` activity is started as soon as the corresponding `<forEach>` branch becomes active.

3) The execution order of *Src* and *Trg* activities is preserved by the execution order of their hosting fragments FE_{src} and FE_{trg} respectively.

Figure 25 illustrates of link status propagation approach [20]. In this figure Airline scope and its two fragments are described. At first link status variable (named $var_{LS)}$ is created, and optionally initialized to value *false*. First fragment contains source activity – Calculate price, and newly created *CL (Src,* $Assign_{true}$*, tc)* control link and $Scope_{src}$ `<scope>` activities. $Assign_{true}$ `<assign>` activity is created and inserted into $Scope_{src}$ `<scope>` activity. Also fault handler FH_{src} is created and attached to $Scope_{src}$. $Assign_{false}$ `<assign>` activity is created and inserted into FH_{src}. Second fragment contains synchronization activity for quoting calculated ticket price. *Emp* `<empty>` activity and *CL'(Emp, Trg, read(*var_{LS}*))* control link are newly created activities inside fragment two of Airline scope.

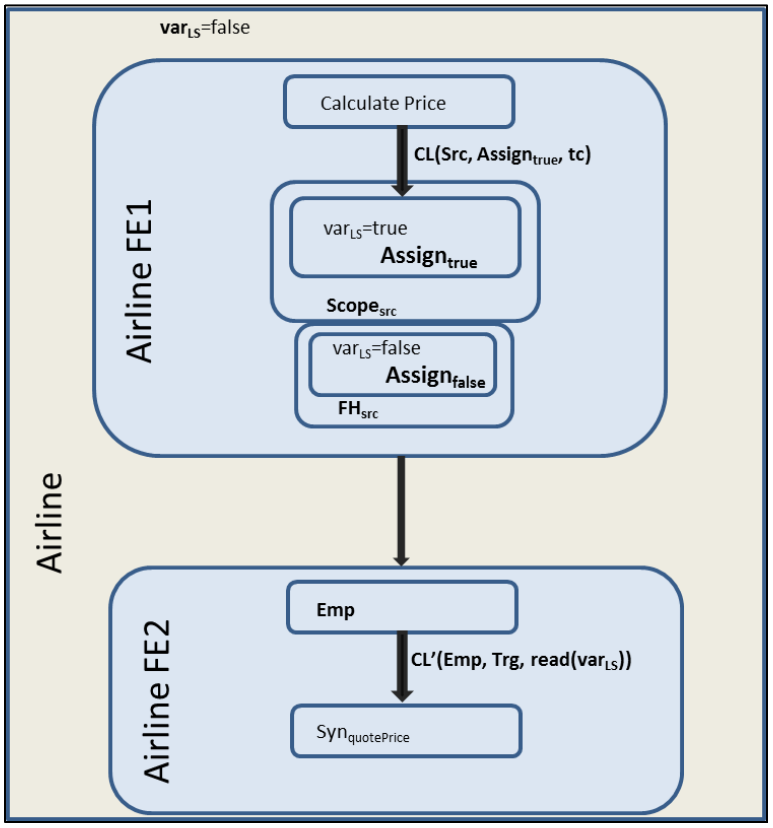


Figure 3.18 Link status propagation from one fragment into another adapted from

Another advantage of link propagation approach is that it also ensures dead-path elimination which can be performed easily. If an incoming link's transition condition evaluates to false, then synchronization activity it is connected to, will have all its outgoing links set to false.

3.6.5 Data Flow between Fragments in and across Container Scopes

After processes get consolidated in P_{Merged} certain variables and links need to be made global to enable data sharing between activities residing in different (container) scopes. This goal can be accomplished by applying variable-lifting approach. There are mainly two rules to follow in variable-lifting approach:

1) All variables that are used in different FE fragments but only in the same dynamic container scope CS, have to be made global only to that dynamic container scope CS.

2) All variables that are accessed in different container scopes have to be lifted up to the process scope P_{Merged}, in other words made global to the whole process P_{Merged}.

Figure 3.19 and Figure 3.20 illustrates abstract example of variable lifting approach. Variable *a* is declared in the scope of *Process_1* - inside forEach activity.. After *Process_1* and *Process_2* get consolidated and loop fragmentation is applied, variable a becomes unreachable for other fragments inside *Process_1* container scope. Figure 3.20 illustrates the solution for two cases. In first case *a)* it is assumed that variable *a* is only accessed throughout container scope of *Process_1* even after process consolidation. In second case *b)* it is assumed that after process consolidation phase variable *a* need also be accessed and updated in container scope of *Process_2*.

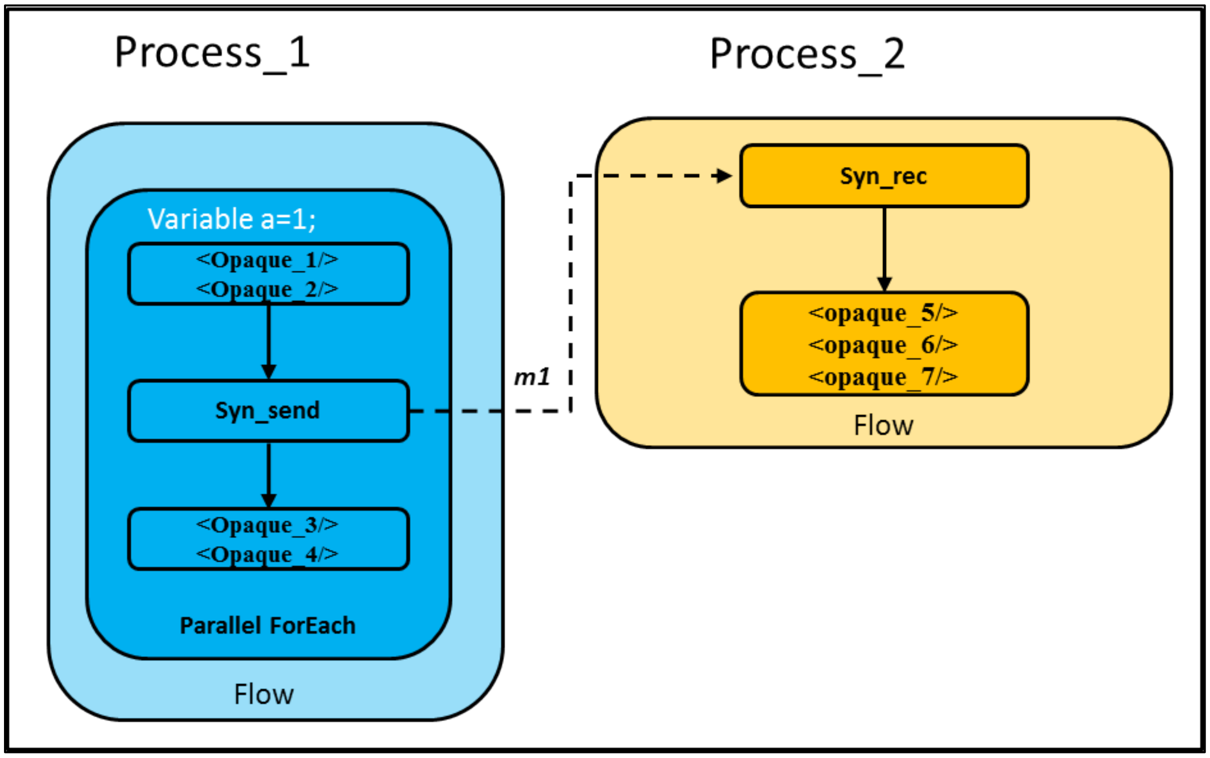


Figure 3.19 Before Applying Variable Lifting Technique

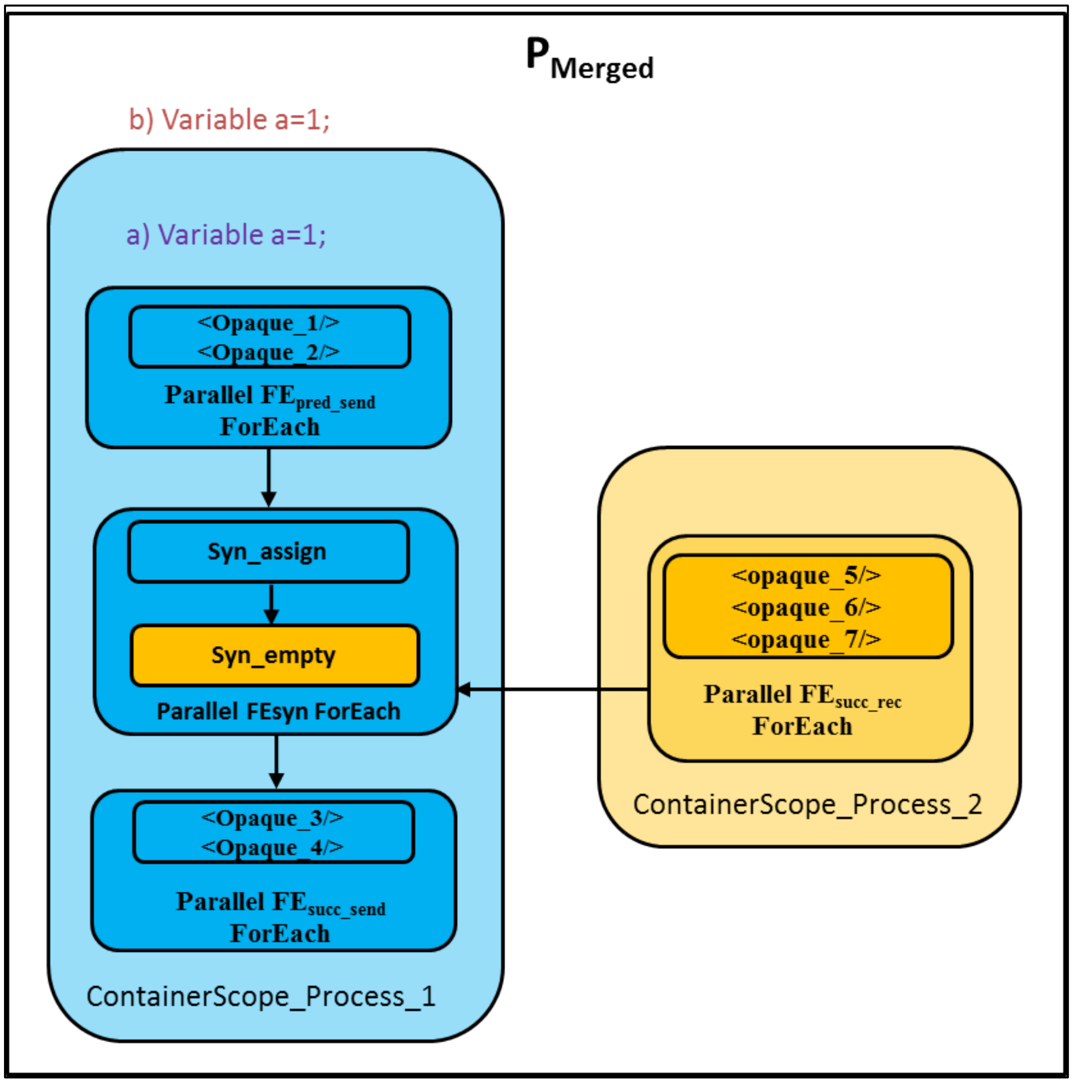


Figure 3.20 After Applying Variable Lifting Technique

In one-to-one interaction, only one variable of the same type is created, as only one instance of partner need to be consolidated in P_{Merged}. In case of one-to-many interaction scenario, let's say there are three instances of same partner choreography need to be consolidated in P_{Merged}. Then three variables of the same type need to be created for each instance. But in case of unknown number of instances to be consolidated, number of multi-instance variables (same variables serving for same purpose in different instances) is also unknown. To cover this case also, a map data structure is created for each variable type.

Let's say there is a variable named Var_{Price} declared in each instance of airline process. This variable holds the value of requested ticket price.

```
<variable name="price" type="TicketPrice" />
```

Listing 3.5 Price Variable

New variable of map data structure type is introduced. Let's call this map as $MIPM_{Price}$ – multi instance partner map for price variable. This map will have entries:

```
<xsd:ID, VarPrice>
```

Listing 3.6 *MIPM$_{Price}$* map entry

Xsd:ID is a key of each entry, and variable *Var$_{Price}$* will be the value of each entry. The key of the map *xsd:ID* is a unique key for determining instance of airline participant type. Var$_{Price}$ is the price variable used in instance of airline participant type determined by *xsd:ID* key. *MIPM$_{Price}$* map data structure can be expressed as message type in wsdl artifact of airline process (which will be automatically injected into it) as:

```
<complexType name="MIPM">
    <sequence>
        <element name="TicketPrice" type="xsd:double" />
    </sequence>
    <attribute name="id" type="xsd:ID" use="required" />
</complexType>

<variable name="priceMap" type="MIPM">
```

Listing 3.7 *MIPM* map complex type and variable *priceMap* of type *MIPM*

All references to *Var$_{Price}$* variable inside each instance of airline process (in assign activities, join or transition conditions) need to be modified: now they need to refer to the corresponding entry of *MIPM$_{Price}$* map. For instance the below code snippet shows how from part of `<assign>` activity need to be modified to adapt the changes:

```
<from variable="price" />

To

<from>priceMap[@id=i]</from>
```

Listing 3.8 Change of *from* part of `<assign>` activity

The *id* attribute helps to select the entry of *priceMap* with instance id equals to *i*. *Xsd:ID* field needs to uniquely identify the corresponding `<forEach>` activity and its corresponding iteration. This is the fact that once the instance started its execution, there is no technical way to insert instance id into instance of a `<forEach>` activity from outside. The *counterVariable* of `<forEach>` activity is the only information which is not changing during that iteration and can uniquely identify an instance within a `<forEach>` activity. But if there are several dynamic containers created in merged *P$_{Merged}$* process, then *counterVariable* value is not unique determining instance id throughout the whole process, but only within that particular dynamic container – `<forEach>` activity. In order to create global unique id, it is recommended to combine the static id *feid* (defined at design time) and the dynamic instance counter value *iid* (defined at runtime) are concatenated. Then *xsd:ID* will be equal to *xsd:ID=feid_iid*. Let's say n instances of airline process need to be consolidated. The first instance will get instance key *AirlinfeFE_1*, the second instance *AirlineFE_2* and nth instance will get instance key *AirlineFE_N*. In case of key

AirlineFE_1 (which is *feid_iid* key), *feid* is equal to "*Airline FE_*" part of the key (common name chosen for FE fragments being generated from same <forEach> loop), and "*1*" is *iid* part (dynamic instance counter value) of the key. If the container needs to be fragmented into several fragments during consolidation phase instance key does not need to be changed. Let's say *AirlineFE_1* fragment of instance one need to be divided into three fragments. Then the key for instance one and fragment one, for instance one and fragment two and for instance one and fragment three will be *AirlineFE_1*. In other words, only one entry in multi-instance variable map is created for the fragments created as division of one fragment. Thus it becomes obvious to determine all the fragments which are executed by the same instance of airline and travel agency choreography in one-to-many interaction scenario. *CollectPricesFE_1* and *AirlineFE_1* can be related to each other, and all related fragments can share data between each other through multi instance variable stored in multi instance map. By variable-lifting approach introduced above. All multi-instance variables which are updated inside related fragments need to be accessed through *xsd:ID* key in *priceMap*.

3.7 Hybrid MIP Instantiation in Merged Process

Hybrid MIP instantiation happens when there exist both dynamic and static `<forEach>` activities. There doesn't exist any specific rule for handling hybrid MIP instantiation, instead dynamic `<forEach>` activities in the PBD must be used for dynamic MIP instantiation, and static `<forEach>` activities must be used for static MIP instantiation.

Figure 3.21 illustrates hybrid multi-instance-partner instantiation and container generation scenario. P_A process has one dynamic parallel `<forEach>` activity (which contains *A4* `<invoke>` activity inside) and one *A2* `<invoke>` activity which are calling *B2* `<receive>` activity (with *createInstance* attribute set to "yes" value). In process P_A, *A2* `<invoke>` activity causes static container generation - covering static MIP case and *A4* `<invoke>` activity inside `<forEach>` activity causes dynamic container generation - covering dynamic MIP case.

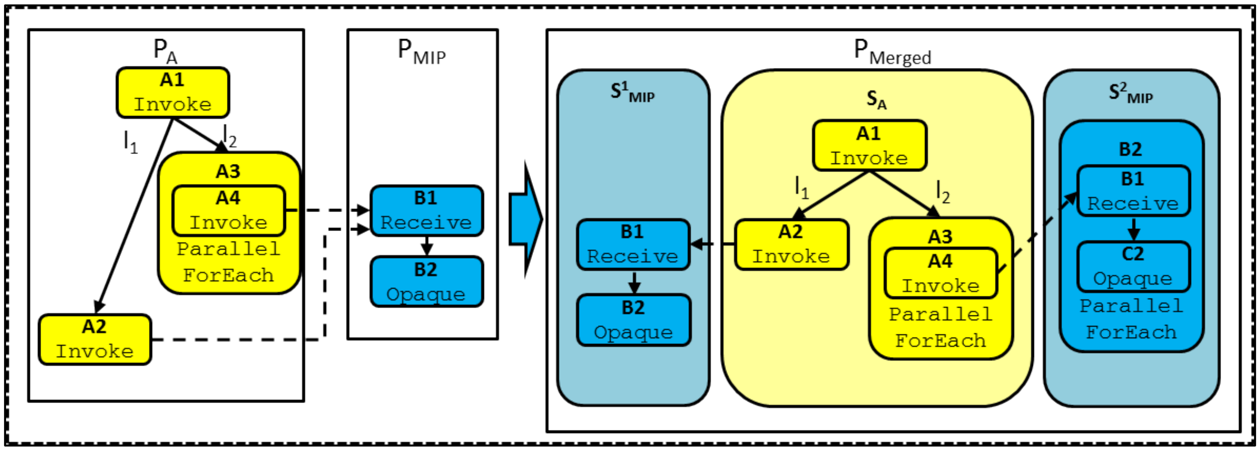


Figure 3.21 Hybrid MIP Instantiation and Container Generation

4 Implementation

In the following section implementation details for process consolidation in one-to-many interaction scenarios will be introduced. Eclipse was chosen as an integrated development environment. Debicki [DEB13] explains installing required packages and frameworks for setting up the environment.

4.1 Design

Figure 4.1, illustrates four consecutive steps for choreography based process consolidation in one-to-many interactions. First step is initialization phase – where input data is read from 3 (or more) files: participant behavior description(s), participant topology and participant grounding. Read input data are stored in data structures corresponding to WS-BPEL 2.0 specifications for further handling. Second step analyzes participant behavior descriptions and participant topology files for determining type and count of container scopes to be created.

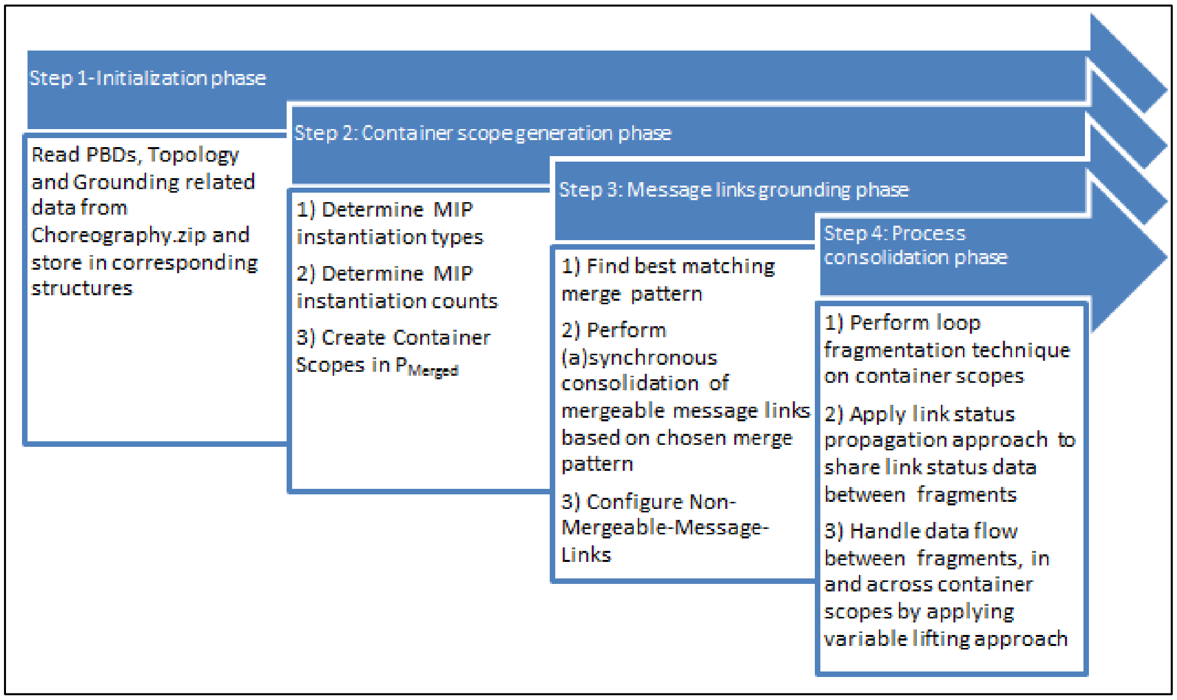

Step 1-Initialization phase
Read PBDs, Topology and Grounding related data from Choreography.zip and store in corresponding structures
Step 2: Container scope generation phase
1) Determine MIP instantiation types
2) Determine MIP instantiation counts
3) Create Container Scopes in P_Merged
Step 3: Message links grounding phase
1) Find best matching merge pattern
2) Perform (a)synchronous consolidation of mergeable message links based on chosen merge pattern
3) Configure Non-Mergeable-Message-Links
Step 4: Process consolidation phase
1) Perform loop fragmentation technique on container scopes
2) Apply link status propagation approach to share link status data between fragments
3) Handle data flow between fragments, in and across container scopes by applying variable lifting approach

Figure 4.1 Four steps for choreography based process consolidation in one-to-many interactions

Third step analyzes message links from participant topology and participant grounding artifacts. Message links can be merge able and non-merge able. If there exist asynchronous (synchronous) patterns for merging communicating activities, then best matching pattern is chosen and asynchronous (synchronous) consolidation takes place. For some communicating activities corresponding synchronization patterns are not introduced yet, thus not all message links can be grounded. The best matching pattern is pattern which covers the exact (or almost exact) scenario of interaction between communicating activities (i.e., two different scenarios - when there is activity succeeding the synchronization activity and when there is not, have different best matching patterns). If merge pattern is not yet implemented for communicating activities, then they cannot be replaced with synchronization activities, and the corresponding message link is moved into Non-Mergeable-Message-Links list.

Fourth step is mainly for handling merged process by applying loop fragmentation to `<forEach>` loop which is dynamic container in it. After container has been fragmented, control links need to be managed to keep most of control flow constraints between business activities. The business activities are the activities that implement a certain business function. Variable lifting approach changes locality of variables to scopes and in this way enables data flow between fragments, as well as across container scopes.

Figure 4.2 illustrates UML class diagram of *org.bpel4chor.mergeChoreography* package. *ChoreographyMerger* is responsible for determining type and count of MIP instantiations and generating corresponding container scopes, then applying consolidation techniques (loop fragmentation, link status propagation, variable lifting) to them. *ChoreographyPackage* represents the class for holding input read data about PBDs, grounding, topology and wsdl artifacts. *ChoreographyPackage* will also store new created merged process in its instance. *CommunicationMatcher* is mainly responsible for searching and finding best matching (a)sync pattern for merging communicating activities. Best matching pattern is selected among suitable matching patterns for merging. *MLEnvironment* class stores some required information such as preceding and succeeding activities of communicating activities, which is used for determining suitable pattern for merging communicating activities.

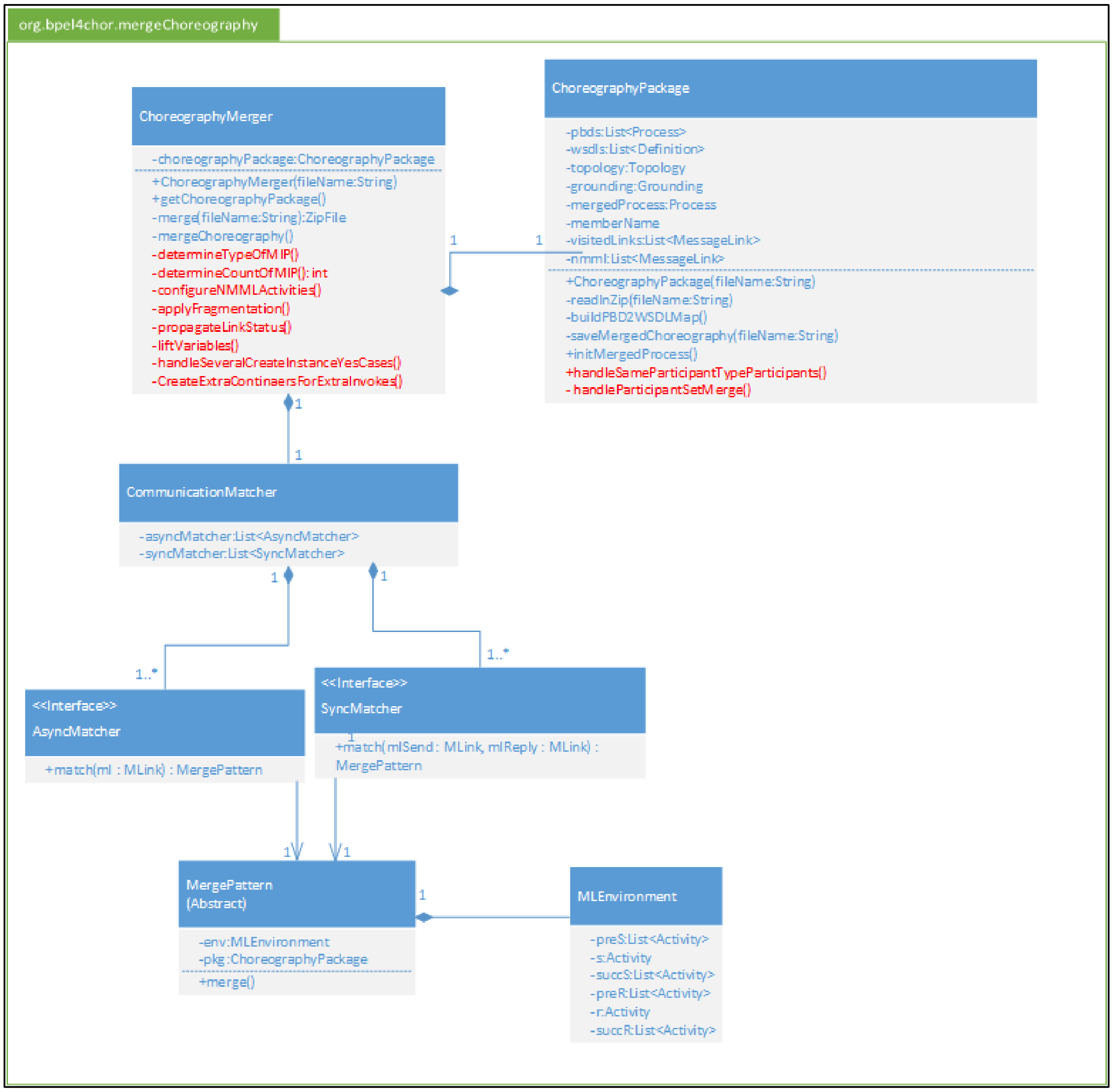

org.bpel4chor.mergeChoreography
ChoreographyMerger
-choreographyPackage:ChoreographyPackage
+ChoreographyMerger(fileName:String)
+getChoreographyPackage()
-merge(fileName:String):ZipFile
-mergeChoreography()
-determineTypeOfMIP()
-determineCountOfMIP(): int
-configureNMMLActivities()
-applyFragmentation()
-propagateLinkStatus()
-liftVariables()
-handleSeveralCreateInstanceYesCases()
-CreateExtraContinaersForExtraInvokes()
ChoreographyPackage
-pbds:List<Process>
-wsdls:List<Definition>
-topology:Topology
-grounding:Grounding
-mergedProcess:Process
-memberName
-visitedLinks:List<MessageLink>
-nmml:List<MessageLink>
+ChoreographyPackage(fileName:String)
-readInZip(fileName:String)
-buildPBD2WSDLMap()
-saveMergedChoreography(fileName:String)
+initMergedProcess()
+handleSameParticipantTypeParticipants()
- handleParticipantSetMerge()
1
1
1
1
CommunicationMatcher
-asyncMatcher:List<AsyncMatcher>
-syncMatcher:List<SyncMatcher>
1
1
1..*
1..*
<<Interface>>
AsyncMatcher
+match(ml : MLink) : MergePattern
<<Interface>>
SyncMatcher
+match(mlSend : MLink, mlReply : MLink) : MergePattern
1
1
MergePattern
(Abstract)
-env:MLEnvironment
-pkg:ChoreographyPackage
+merge()
1
MLEnvironment
-preS:List<Activity>
-s:Activity
-succS:List<Activity>
-preR:List<Activity>
-r:Activity
-succR:List<Activity>

Figure 4.2 UML Class Diagram of org.bpel4chor.mergeChoreographyPackage, adapted from [DEB13]

The method names written in red color are new methods implemented in *org.bpel4chor.mergeChoreography* for enabling choreography based process consolidation in one-to-many interactions.

4.2 Grounding Non-Mergeable-Message-Links (NMML)

This thesis does not cover all merge patterns for communication activities. Message links - whose communicating activities cannot be replaced by synchronization activities, are grouped as NMML list. Details about message links in NMML list can be found in participant grounding; technical details such as portType and operation can be gathered from participant behavior description artifact of choreography. Listing 4.1 (with Listing 4.2, Listing 4.3) illustrates asynchronous intra-process communication activities and the corresponding message link which is in NMML list. One extra partner link is added for each pair of asynchronous intra-process communicating activities which cannot be replaced by synchronization activities.

```
<topology ...>
    <messageLinks>
        <messageLink name="msgLinkSend" sender="PBD1" receiver="PBD2"
        sendActivity="send" receiveActivity="rec" messageName="msg"/>
        ...
    </messageLinks>
</topology>

<grounding ...>
  <messageLinks>
    <messageLink name="msgLinkSend" portType="ns1:send2recPType"
    operation="sendOperation"/>
    ...
  </messageLinks>
</grounding>

<definitions name="pbd2" >
    ...
    <plnk:partnerLinkType name="pbd12pbd2PLT">
        <plnk:role name="pbd12pbd2Role" portType="tns:send2recPType"/>
    </plnk:partnerLinkType>
    <portType name="send2recPType">
        <operation name="sendOperation">
            <input message="tns:pbd2invMessage"/>
        </operation>
    </portType>
    ...
</definitions>
```

Listing 4.1. Consolidation of asynchronous intra-process communicating activities-topology, grounding and wsdl artifact of PBD2, adapted from [DEB13]

```
<process name="processMerged"
targetNamespace="http://www.iaas.uni-stuttgart.de"
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:pbd1="http://BPEL4Chor/pbd1"
xmlns:pbd2="http://BPEL4Chor/pbd2"
xmlns:pbd3="http://BPEL4Chor/pbd3">
    ...
    <flow>
        <scope name="Scope_PBD1">
            <invoke name="send" inputVariable="var_send"/>
        </scope>
        <scope name="Scope_PBD2">
            <receive name="rec" variable="var_rec"/>
        </scope>
    </flow>
</process>
```

Listing 4.2 Consolidation of asynchronous intra-process communicating activities- merged process before consolidation, adapted from [DEB13]

```
<process name="processMerged"
targetNamespace="http://www.iaas.uni-stuttgart.de"
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:pbd1="http://BPEL4Chor/pbd1"
xmlns:pbd2="http://BPEL4Chor/pbd2"
xmlns:pbd3="http://BPEL4Chor/pbd3">
    ...
    <flow>
        <scope name="Scope_PBD1">
            <partnerLink name="pbd1pbd2PLS"
            partnerLinkType="pbd2:pbd12pbd2PLT"
            partnerRole="pbd12pbd2Role"/>
            <invoke name="send" partnerLink="pbd1pbd2PLS"
            portType="pbd2:s2rPType"
            operation="sendOperation"
            inputVariable="var_send"/>
        </scope>
        <scope name="Scope_PBD2">
            <partnerLink name="pbd1pbd2PLR"
            partnerLinkType="pbd2:pbd12pbd2PLT"
            myRole="pbd12pbd2Role"/>
            <receive name="rec" partnerLink="pbd1pbd2PLR"
            portType="pbd2:s2rPType"
            operation="sendOperation"
            variable="var_rec"/>
        </scope>
    </flow>
</process>
```

Listing 4.3 Consolidation of asynchronous intra-process communicating activities – merged process after consolidation, adapted from [DEB13]

As Listing 4.1 (with Listing 4.2, Listing 4.3) illustrates, *send* activity and *rec* activity remains as intra-process communicating activities in their corresponding scopes in MergedProcess. New partner link is created and inserted into scope of *send* and *rec* activities.

In case of synchronous intra-process communicating activities at least two message links (*msgLinkSend* and *msgLinkReply*) need to be analyzed and inserted into the corresponding scope. Listing 4.4(with Listing 4.5, Listing 4.6) describes synchronous intra-process communicating activities consolidation:

```
<topology ...>
    <messageLinks>
        <messageLink name="msgLinkSend" sender="PBD1" receiver="PBD2"
        sendActivity="send" receiveActivity="rec" messageName="msg1"/>
        <messageLink name="msgLinkReply" sender="PBD2" receiver="PBD1"
        sendActivity="reply" receiveActivity="s" messageName="msg2"/>
        ...
    </messageLinks>
</topology>

<grounding ...>
  <messageLinks>
    <messageLink name="msgLinkReply" portType="ns1:send2recPType"
    operation="sendOperation"/>
    <messageLink name="msgLinkSend" portType="ns1:send2recPType"
    operation="sendOperation"/>
    ...
  </messageLinks>
</grounding>

<definitions name="pbd2" >
    ...
    <plnk:partnerLinkType name="pbd12pbd2PLT">
        <plnk:role name="pbd12pbd2Role" portType="tns:send2recPType"/>
    </plnk:partnerLinkType>
    <portType name="send2recPType">
        <operation name="sendOperation">
            <input message="tns:pbd2invMessage"/>
            <output message="tns:pbd2replMessage"/>
        </operation>
    </portType>
    ...
</definitions>
```

Listing 4.4 Consolidation of synchronous intra-process communicating activities – topology, grounding and wsdl of PBD2, adapted from [DEB13]

```
<process name="MergedProcess"
xmlns:pbd1="http://BPEL4Chor/pbd1"
xmlns:pbd2="http://BPEL4Chor/pbd2" ...>
    ...
    <flow>
        <scope name="Scope_PBD1">
            <invoke name="send" inputVariable="var_send" outputVariable="var_reply"/>
        </scope>
        <scope name="Scope_PBD2">
            <receive name="rec" variable="var_rec"/>
            ...
            <reply name="reply" variable="var_reply">
        </scope>
    </flow>
</process>
```

Listing 4.5 Consolidation of synchronous intra-process communicating activities – merged process before consolidation, adapted from [DEB13]

```
<process name="MergedProcess"
xmlns:pbd1="http://BPEL4Chor/pbd1"
xmlns:pbd2="http://BPEL4Chor/pbd2"...>
    ...
    <flow>
        <scope name="Scope_PBD1">
            <partnerLink name="pbd1pbd2PLS"
            partnerLinkType="pbd2:pbd12pbd2PLT"
            partnerRole="pbd12pbd2Role"/>
            <invoke name="send" partnerLink="pbd1pbd2PLS"
            portType="pbd2:send2recPType"
            operation="sendOperation"
            inputVariable="var_send"
            outputVariable="var_reply"/>
        </scope>
        <scope name="Scope_PBD2">
            <partnerLink name="pbd1pbd2PLR"
            partnerLinkType="pbd2:pbd12pbd2PLT"
            myRole="pbd12pbd2Role"/>
            <receive name="rec" partnerLink="pbd1pbd2PLR"
            portType="pbd2:send2recPType"
            operation="sendOperation"
            variable="var_rec"/>
            <reply name="reply"  partnerLink="pbd1pbd2PLR"
            portType="pbd2:send2recPType" variable="var_reply">
        </scope>
    </flow>
</process>
```

Listing 4.6 Consolidation of synchronous intra-process communicating activities – merged process after consolidation, adapted from [DEB13]

Implementation hierarchy of synchronous and asynchronous merge patterns and matchers is illustrated as UML class diagram in Figure 4.3:

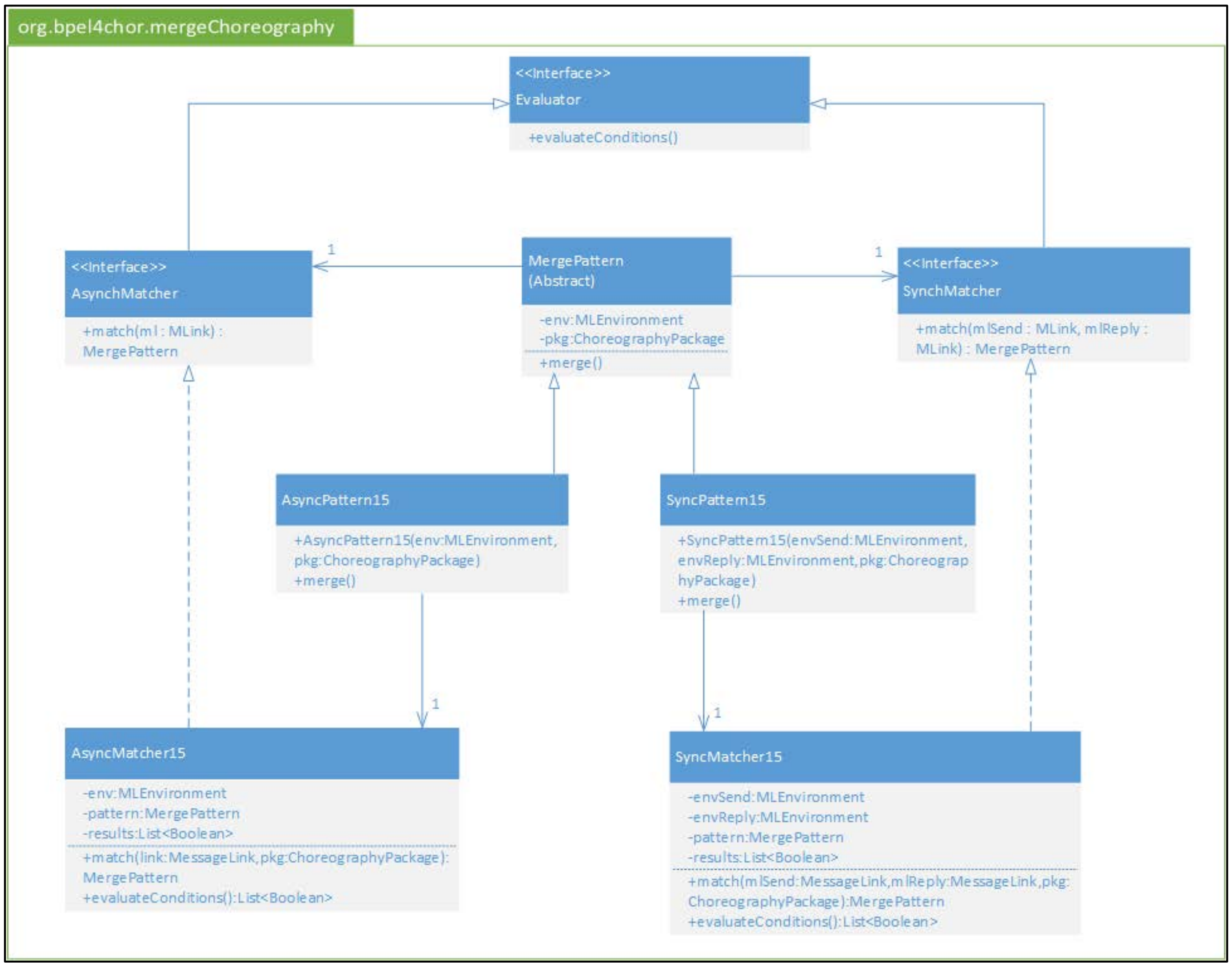

org.bpel4chor.mergeChoreography
<<Interface>>
Evaluator
+evaluateConditions()
<<Interface>>
AsynchMatcher
+match(ml : MLink) : MergePattern
MergePattern
(Abstract)
-env:MLEnvironment
-pkg:ChoreographyPackage
+merge()
<<Interface>>
SynchMatcher
+match(mlSend : MLink, mlReply : MLink) : MergePattern
1
1
AsyncPattern15
+AsyncPattern15(env:MLEnvironment, pkg:ChoreographyPackage)
+merge()
SyncPattern15
+SyncPattern15(envSend:MLEnvironment, envReply:MLEnvironment, pkg:ChoreographyPackage)
+merge()
1
1
AsyncMatcher15
-env:MLEnvironment
-pattern:MergePattern
-results:List<Boolean>
+match(link:MessageLink, pkg:ChoreographyPackage): MergePattern
+evaluateConditions():List<Boolean>
SyncMatcher15
-envSend:MLEnvironment
-envReply:MLEnvironment
-pattern:MergePattern
-results:List<Boolean>
+match(mlSend:MessageLink, mlReply:MessageLink, pkg: ChoreographyPackage):MergePattern
+evaluateConditions():List<Boolean>

Figure 4.3 UML Class Diagram describing relations between (a)sync matcher and (a)sync pattern, taken from [DEB13]

Figure 4.3, illustrates only AsyncPattern 15 and SyncPattern 15 pattern used as an example of merge patterns throughout this thesis. Full list of implemented merge patterns can be found in [DEB13].

4.3 Determining Type and Count of MIP Instantiation

Choreography is given as input to *org.bpel4chor.mergeChoreography* package. After choreography related data is read and stored in corresponding data structures, *merge()* method is called. Type of MIP instantiation is determined by analysis of topology artifact of choreography and PBDs. If `<participantSet>` was not defined in topology artifact, then this is one-to-one interaction. Then one container scope is created for each participant type, and all the activities are copied to the corresponding container scope. Number of container scopes to be generated will be equal to number of participants defined in topology artifact.

If `<participantSet>` was defined in topology artifact, then `<forEach>` activity names are extracted from *forEach* attribute value. Then the PBDs which need to be merged are searched for the given `<forEach>` name extracted from `<participantSet>`. It is assumed that all `<forEach>` activities in the same and different PBDs have unique names. Then found `<forEach>` activity is analyzed for determining if it is static (with *finalCounterValue* attribute has value known at design time) or dynamic (with *finalCounterValue* attribute having value N, which will be initialized only at run time).If `<forEach>` activity is dynamic, then dynamic MIP instantiation will be applied, and one dynamic container scope (which is `<forEach>` activity) will be created in consolidated process P_{Merged}. If `<forEach>` activity is static, then static MIP instantiation will be applied, and the number of static containers (static container is just `<scope>` activity containing all the activities of PBD whose instance need to be consolidated) to be created will be equal to *finalCounterVariable* of `<forEach>` activity.

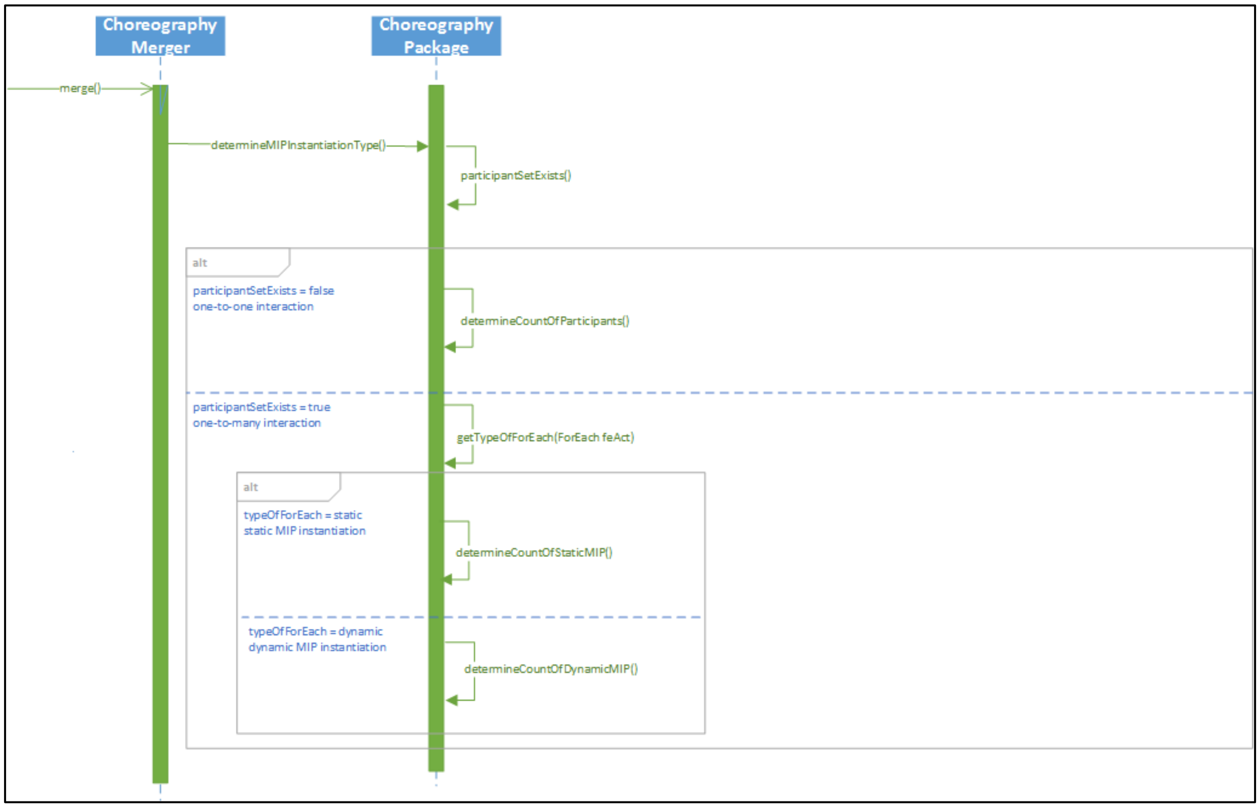

Choreography Merger
Choreography Package
merge()
determineMIPInstantiationType()
participantSetExists()
alt
participantSetExists = false
one-to-one interaction
determineCountOfParticipants()
participantSetExists = true
one-to-many interaction
getTypeOfForEach(ForEach feAct)
alt
typeOfForEach = static
static MIP instantiation
determineCountOfStaticMIP()
typeOfForEach = dynamic
dynamic MIP instantiation
determineCountOfDynamicMIP()

Figure 4.4 Determining type and count of MIP instantiations

determineMIPInstantiationType() and *participantSetExists()* methods try to determine MIP instantiation type by reading topology artifact. If there is no `<participantSet>` defined in topology artifact, then this is one-to-one interaction, and then the number of participants is calculated by counting the involved processes. If there is `<participantSet>` defined in topology artifact, then this is one-to-many interaction. *getTypeOfForEach(ForEach feAct)* determines type of `<forEach>` activity : if this is static `<forEach>` activity, then this will be static MIP instantiation; if this is dynamic `<forEach>` activity then this will be dynamic MIP instantiation. Then the number of multi-instance partners is determined by *finalCounterVariable* of `<forEach>` activity.

The next three sections demonstrate business process consolidation after choreography is being read and stored in corresponding data structures. The consolidation operation begins with calling of *merge()* method.

4.4 Consolidation of Business Processes in One-to-One Interactions

Choreography-based business process consolidation in one-to-one interactions starts by creating separate static container in merged process P_{Merged} for each PBD of involved choreography. Then communicating activities are replaced by synchronization activities. But as there does not exist matching patterns for all kinds of communicating activities, thus non-mergeable message links need to be handled in separate step. After involved business processes get consolidated, data flow in and across container scopes need to be analyze, and if required, variable lifting technique need to be applied. Finally, the container scopes are added choreography package, which results in executable merged business process. Figure 4.5 illustrates sequence diagram for finding message patterns for message links from topology artifact:

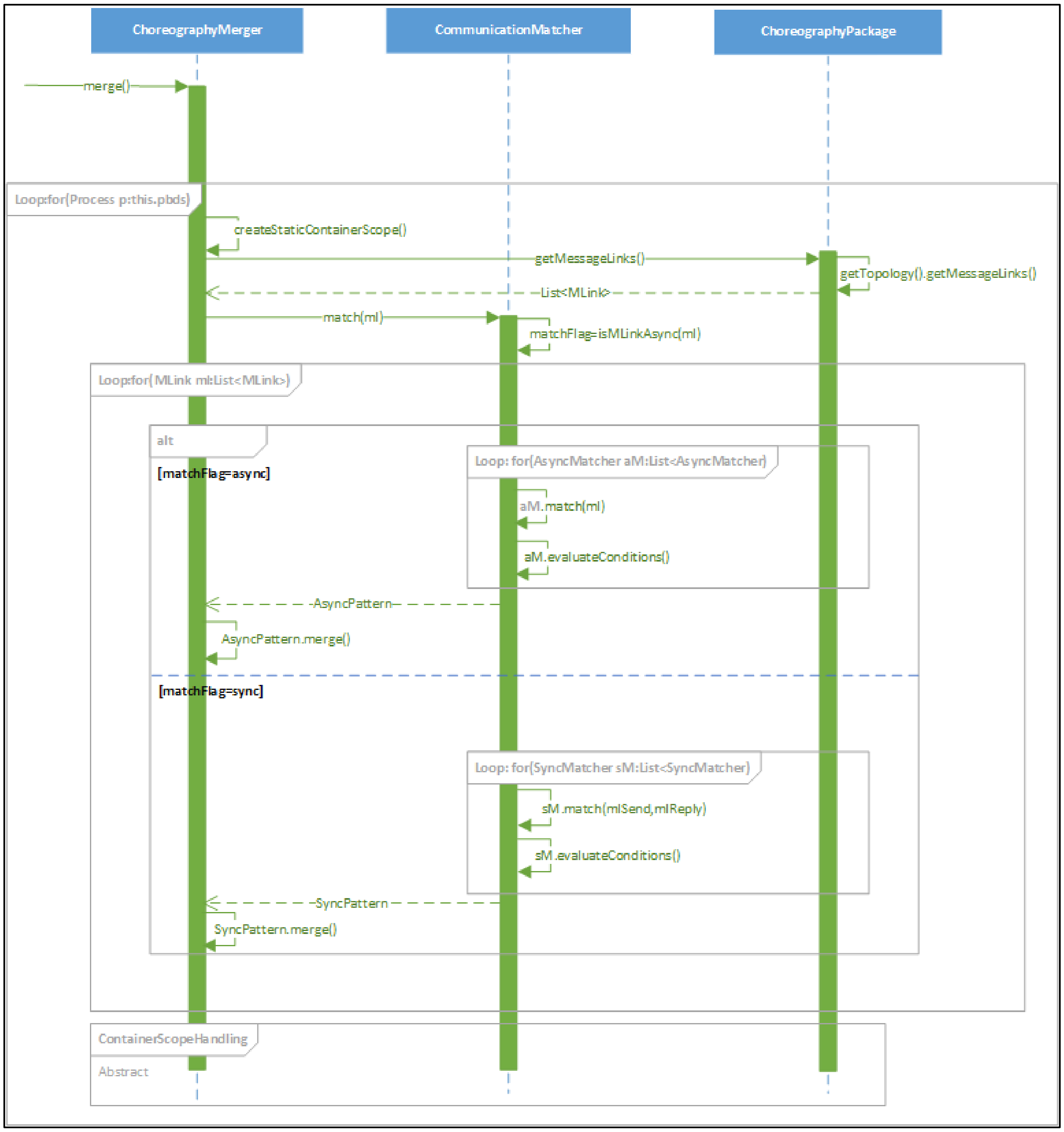


Figure 4.5 Sequence diagram for finding MergePatterns for message links in one-to-one interactions, adapted from [DEB13]

createSTaticContainerScope() creates new static container scope – which is `<scope>` activity. *match(ml)* method analyzes PBD files, and finds the matching merge patterns which suits for the interaction scenario of communicating activities. Interaction scenarios differ depending if there is any preceding (succeeding) activity after (before) communicating activities. Also the type of preceding (succeeding) activities influences on choice of merging patterns. *evaluateConditions()* method helps to choose the best matching merge pattern which resembles interaction scenario in closest way. *merge()* is applying the consolidation operation based on chosen best matching merge pattern.

4.5 Consolidation of Static MIP Instances

Static MIP instantiation and consolidation operation is logically same as business process consolidation in one-to-one interactions. The only difference is in the number of container scopes to be created. Thus additional step determines count of MIP instantiations. Count of container scopes of multi-instance process to be created in merged process P_{Merged} is equal to *finalCounterVariable*'s value of `<forEach>` activity in one part of one-to-many interaction. So sequence diagram will be same as in one-to-one interactions with the only change in outer loop, as the number of process instances will be more than PBDs in choreography. Then for each static MIP instantiation the following steps are executed in consecutive order:

1) Create new container scope corresponding to static MIP instantiation

2) For each message link in topology file, try to find corresponding merge pattern. If found apply merge pattern. Else add that message link to NMML list.

3) ConfigureNMMLActivities() method is handling merge of NMML.

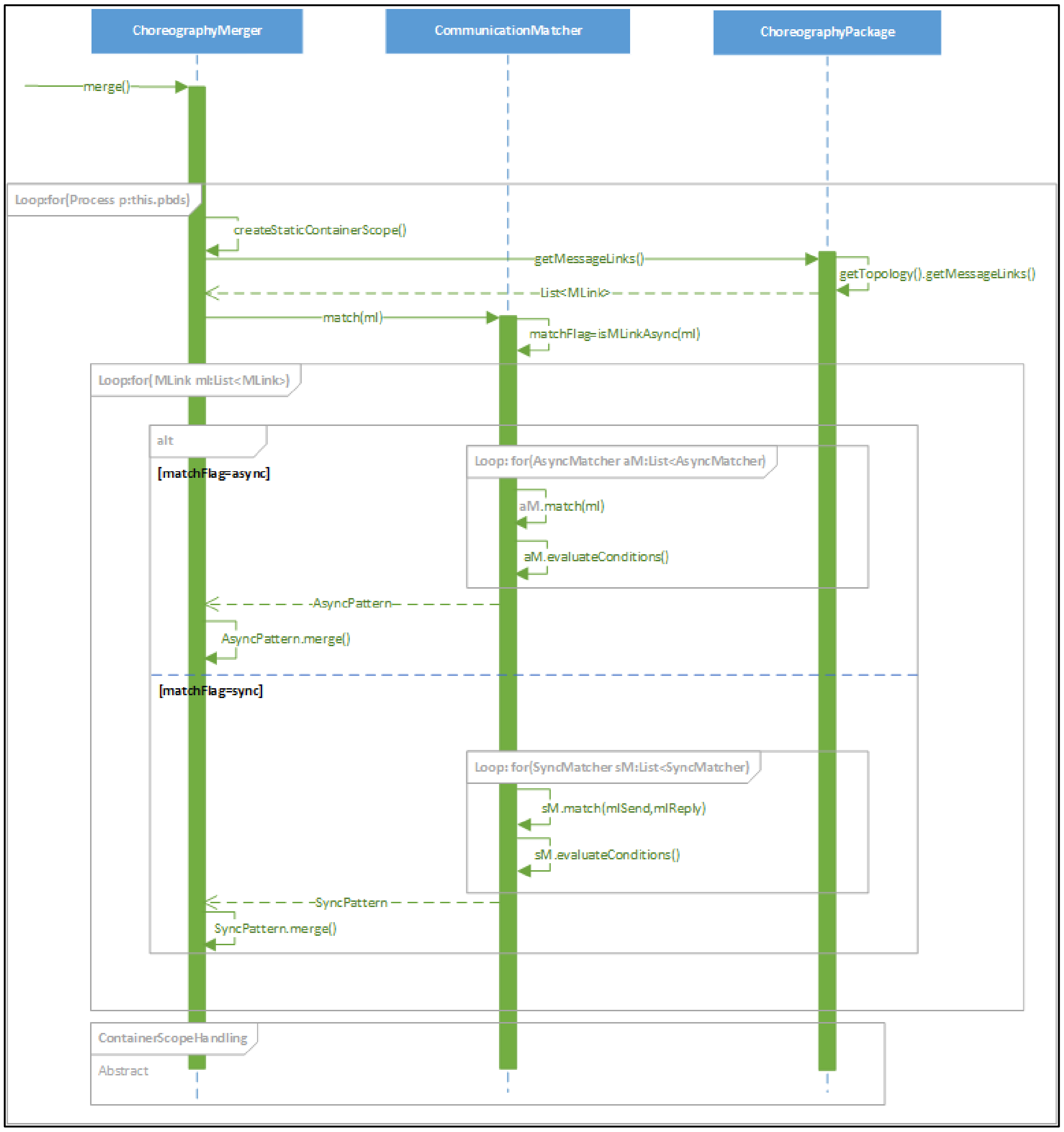


Figure 4.6 Sequence diagram for finding MergePatterns for message links in one-to-many interactions – static MIP consolidation, adapted from [DEB13][8]

4) applyVariableLiftingTechnique() method applies variable lifting technique for ensuring data flow across container scopes.

5) Finally addNewContinerScopeToChorPkg() method added container scope (which has gone the changes to enable data flow in and across container scopes) to merged process which resides in a choreography package.

[8] ContainerScopeHandling part is marked abstract and will be explained in next sequence diagrams

Step four and five are same for one-to-one interactions and static MIP consolidation in one-to-many interactions. Again the only difference will be on number of process instances the outer loop iterates through:

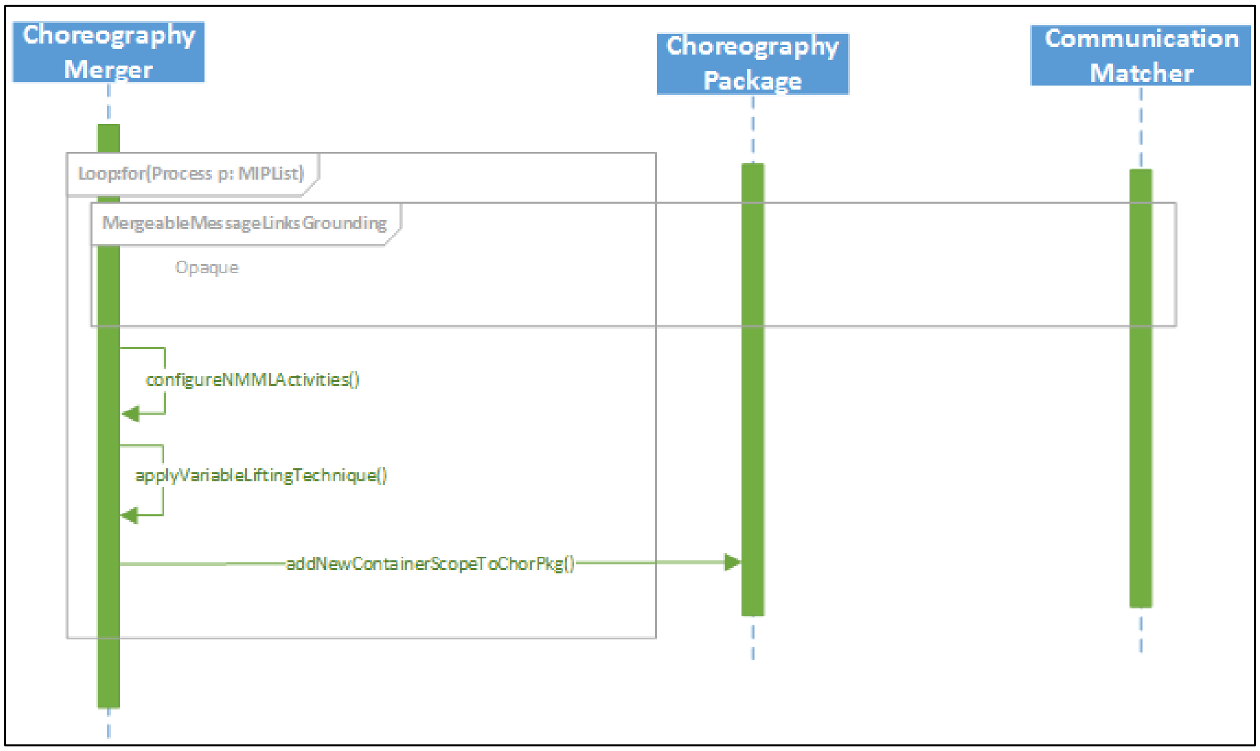

Choreography Merger
Choreography Package
Communication Matcher
Loop:for(Process p: MIPList)
MergeableMessageLinksGrounding
Opaque
configureNMMLActivities()
applyVariableLiftingTechnique()
addNewContainerScopeToChorPkg()

Figure 4.7 Static MIP consolidation in one-to-many interactions

4.6 Consolidation of Dynamic MIP Instances

In case of dynamic MIP consolidation, instead of *createStaticContainer()* method, *createDynamicContainer()* method is called in first part of consolidation (finding MergePatterns for message links).

Then for each instance the following steps are executed in consecutive order:

1) Create new container scope corresponding to dynamic MIP instantiation

2) For each message link in topology file, try to find corresponding merge pattern. If found apply merge pattern. Else add that message link to NMML list.

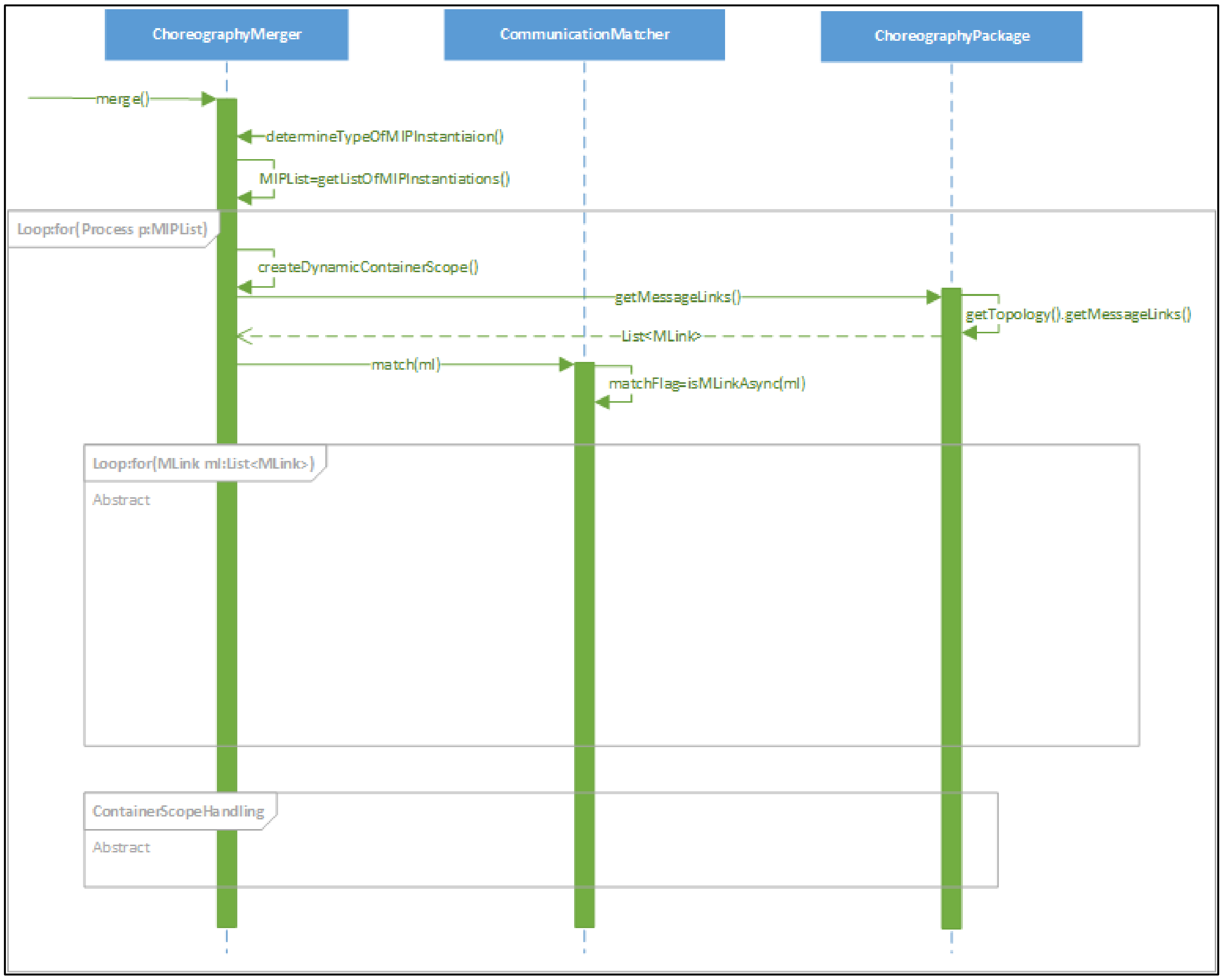

ChoreographyMerger
CommunicationMatcher
ChoreographyPackage
merge()
determineTypeOfMIPInstantiaion()
MIPList=getListOfMIPInstantiations()
Loop:for(Process p:MIPList)
createDynamicContainerScope()
getMessageLinks()
getTopology().getMessageLinks()
List<MLink>
match(ml)
matchFlag=isMLinkAsync(ml)
Loop:for(MLink ml:List<MLink>)
Abstract
ContainerScopeHandling
Abstract

Figure 4.8 Sequence diagram for finding MergePatterns for message links in one-to-many interactions – dynamic MIP consolidation, adapted from [DEB13]

3) *configureNMMLActivities()* method is handling merge of NMML.

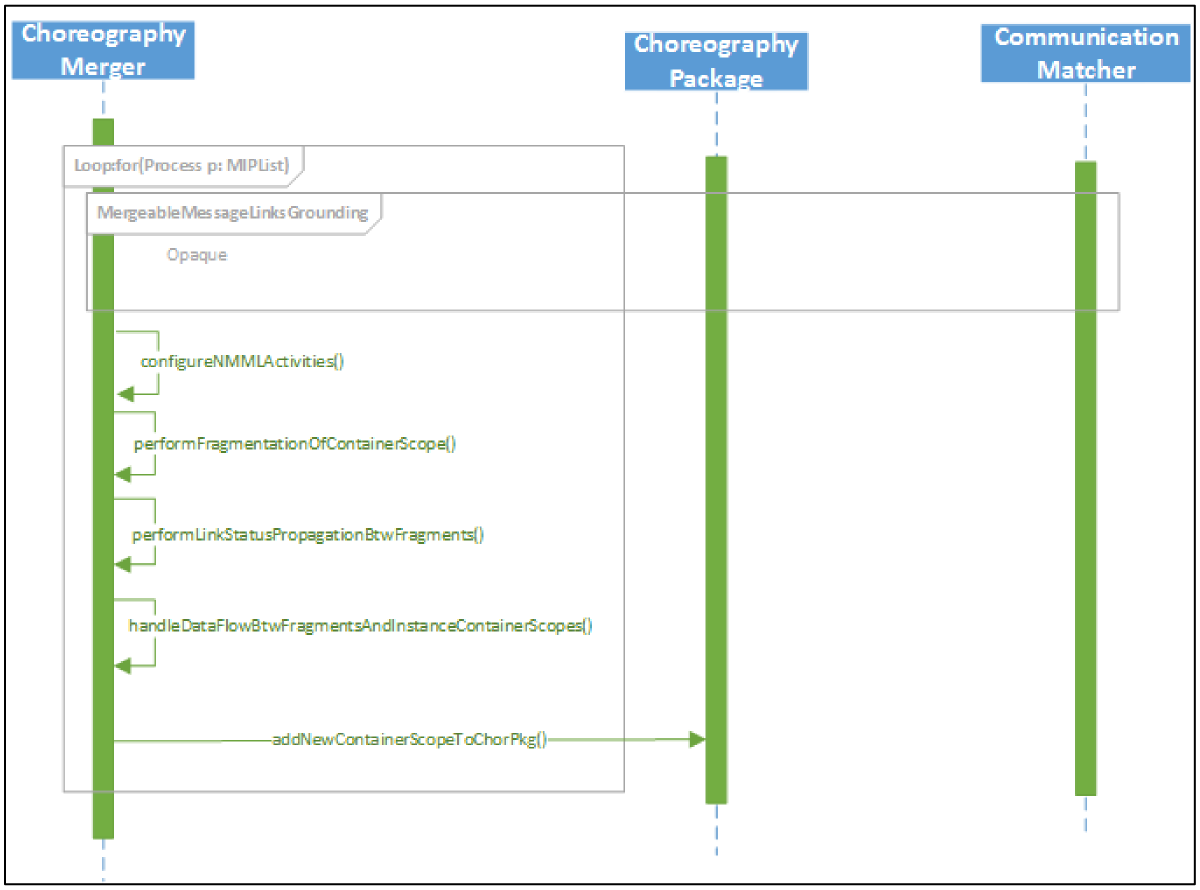
4) *peformFragmentationOfContainerScope()* method is dividing container scope into several fragments.

5) *performLinkStatusPropagationBtwFragments()* method handles cross-boundary link violations. This method adapts some changes to source and target fragments to enable correct execution order of fragments. Link status propagation, Section 3.6.4,explains the technique in detailed way.

6) *handleDataFlowBtwFragmentsAndInstanceContainerScopes()* method applies variable lifting technique for ensuring data flow between fragments, as well as in and across container scopes. This method adjusts visibility scope of variables accessed in and across container scopes.

7) Finally *addNewContinerScopeToChorPkg()* method added the container scope (which has undergone loop fragmentation, link status propagation and variable lifting techniques) to merged process which resides in choreography package.

Figure 4.9 demonstrates dynamic MIP consolidation operation. The major difference is creation of dynamic container in merged process P_{Merged}, which is `<forEach>` loop. Then that `<forEach>` loop is divided into several fragments for handling cross-boundary link violations entering and leaving `<forEach>` loop. Control links' statuses are propagated between connected fragments. Then variable lifting technique is applied for adjusting visibility scope of variables. The final step is adding container scopes to choreography package.



Choreography Merger
Choreography Package
Communication Matcher
Loop:for(Process p: MIPList)
MergeableMessageLinksGrounding
Opaque
configureNMMLActivities()
performFragmentationOfContainerScope()
performLinkStatusPropagationBtwFragments()
handleDataFlowBtwFragmentsAndInstanceContainerScopes()
addNewContainerScopeToChorPkg()

Figure 4.9 Choreography-based dynamic MIP consolidation in one-to-many interactions

4.7 Consolidation of Hybrid MIP Instances

There is no separate implementation for hybrid MIP consolidation. As hybrid MIP consolidation is mixture of static and dynamic MIP consolidations, the appropriate implementation is chosen for MIP instantiation type (static or dynamic).

5 Summary & Future work

The aim of this thesis was extension of choreography based process consolidation for one-to-many interactions. Introduced concepts were implemented as extension of BPLE4Chor choreography merge package. The input to this package are choreography artifacts: PBD that describes each participant process, control and data flow between activities in that process; topology artifact defines participant types, participant references and message links which binds communicating PBDs to each other; grounding artifact contains technical details about message links and the involved communicating activities from corresponding WSDL files. The output of the extended merge package is a consolidated executable BPEL process. The consolidated BPEL process retains most of the control flow dependencies between business activities of involved processes.

Business process consolidation in one-to-one interactions was introduced in Section 3.3. Section 3.4 demonstrates business process consolidation for one-to-many interaction scenarios. At first BPEL4Chor choreography artifacts are read and stored in corresponding data structures. One new process (consolidated process) is created to hold merged process activities. Determining type and count of MIP instantiations is next step to follow. There can be static, dynamic or hybrid MIP instantiation types. Thereafter, corresponding to MIP instantiation count and type, respective containers are created and filled with activities of corresponding PBDs in consolidated process. In case of one-to-one interaction scenarios and static MIP instantiation scenarios static container is created. Static container is new `<scope>` activity holding corresponding PBD's activities in it. In case of dynamic MIP instantiation dynamic container is created and then corresponding PBD's activities are added into it as its children. Dynamic container is new `<scope>` activity holding new `<forEach>` activity inside it. Afterwards, communicating activities are replaced by synchronization activities during (a)synchronous consolidation phase, which avoids the overhead of excess communication through sending and receiving SOAP messages. Depending on type of communication (asynchronous or synchronous) best matching pattern is searched for synchronization of communicating activities. Best matching pattern is the pattern the most resembling communicating scenario. For some communicating activities no merge patterns are defined yet, and those message links are added to NMML list. Then NMML list is grounded.

Consolidation phase generates control links – violating cross-boundary constraint, between synchronization activities that cross-boundaries of `<forEach>` loops. Only in the case of dynamic MIP instantiation, loop fragmentation technique is applied to divide original `<forEach>` activity into several `<forEach>` fragments as a solution to cross-boundary link constraint.

During the split of the original `<forEach>` activity into several fragments, it is possible that the source and the target activities that were connected by control link will be scattered into two different fragments by causing the break of that control link. Link status propagation technique was introduced for propagating link status from one `<forEach>` fragment into another by using variables.

Dividing original `<forEach>` activity into different fragment segments makes some variables unreachable to some fragments. Variable lifting technique is applied by changing the scope of unreachable variables – either global to container scope or global to the whole merged process scope, which enables data flow between different fragments of same container scope, also across different container scopes.

5.1 Future Work

Currently `<receive>` activity is the only supported receiving part of communication in the implementation. However, this implementation can be extended for different receive activities as receiving part of communicating activities. `<onMessage>` construct of `<pick>` activity can also be a receiving part of communicating activities.

This thesis has covered consolidation of multi-instance processes that are instantiated by an instance creating activity not residing inside any of loop activities or inside `<forEach>` loop only. Besides `<forEach>` activity, BPEL4Chor choreography supports other loop constructs provided by BPEL 2.0 specification, such as `<while>`, `<repeatUntil>`. Extending process consolidation by handling `<while>` and `<repeatUntil>` activities, as well as nested loops are also focus of future work.

Reference passing was not covered in this thesis, but it is one of the important aspects in multi-instance interaction scenarios. In case of FTBS scenario, travel agency could have passed endpoint references of several airlines - providing the same cheapest price flight tickets, to traveler process. This could have given a flexibility to the traveler in choice of airline.

Wagner et al. [WRKUL13] have stated that consolidation of several processes of choreography into one merged process can reduce execution time and performance of original choreography. The performance optimization is achieved by the reduction in number of message exchanges and message (de)serializations. In future work, performance of consolidated choreography (representing multi-instance processes scenarios) needs to be compared against non-consolidated choreography for deriving performance and runtime measures.

Furthermore this thesis could have been extended by analyzing how BPEL's compensation handling mechanism can be applied to different `<forEach>` fragments originated from single `<forEach>` loop.

As Business Process Model and Notation (BPMN) is a standard for business process modeling, it should be studied how consolidation approach, introduced in this thesis, can be applied to BPMN collaboration diagrams.

Bibliography

[ACKM04] G. Alonso, F. Casati, H. Kuno, V. Machiraju. *Web Services*. Book pp. 123-149, 2004. Online: http://www.inf.ethz.ch/personal/alonso/Web-book/Chapter-5.pdf

[AGA07] S. Agarwal. *Formal Description of Web Services for Expressive Matchmaking*. Dissertation, Universität Karlsruhe, Fakultät für Wirtschaftswissenschaften, Deutschland, 2007. Online: http://people.aifb.kit.edu/sag/papers/phdthesis.pdf

[ALL83] J.F. Allen. *Maintaining knowledge about temporal intervals*. Communications of the ACM, Volume 26 Issue 11, pp. 832-843, Nov. 1983

[BDH05] A. Barros, M. Dumas, A.H.M. ter Hofstede. *Service Interaction Patterns*. Lecture Notes in Computer Science, Volume 3649, 2005, pp. 302-318, 2005.

[JMS06] M.B. Jurich, B. Mathew, P. Sarang. *Business Process Execution Language for Web Services*. Book, 2nd edition, pp. 19-21, Jan. 2006

[DEB13] P. Debicki. *Choreographie-basierte Konsolidierung von BPEL Prozessmodellen*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Feb. 2013.

[DEC09] G. Decker. *Design and Analysis of Process Choreographies*. Dissertation, University of Potsdam, Potsdam, Germany, Jun. 2009.

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. *BPEL4Chor: Extending BPEL for Modeling Choreographies*. Web Services, pp. 296–303, 2007.

[DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. *Interacting services: from specification to execution*. Data & Knowledge Engineering, Volume 68 Issue 10, pp. 946–972, 2009.

[HL00] R. Hirscheim, M. Lacity. *The myths and realities of information technology insourcing*. Communications of the ACM, Volume 43 Issue 2 pp. 99-107, Feb. 2000.

[KELLN11] O. Kopp, L. Engler, T. van Lessen, F. Leymann, J. Nitzsche. *Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus*. Communications in Computer and Information Science, Volume 138, pp. 36-53, 2011.

[KL06] R. Khalaf, F. Leymann. *Role-based Decomposition of Business Processes using BPEL*. In International Conference on Web Services (ICWS 2006), pp. 770–780. IEEE Computer Society, 2006.

[KLW10] O. Kopp, F. Leymann, F. Wu. *Mapping Interconnection Choreography Models to Interaction Choreography Models*. Proceedings of the 2nd Central-European Workshop on Services and their Composition, Berlin, Germany, Feb. 2010.

[OAS07] OASIS. *Web Service Business Process Execution Language Version 2.0*, 11 April 2007. Online: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[PAS05] J. Pasley. *How BPEL and SOA Are Changing Web Services Development*. Internet Computing, IEEE, Volume 9 Issue 3. pp. 60-67, Jun. 2005.

[PEL03] C. Peltz. *Web services orchestration and choreography*. Computer, Volume 36 Issue 10. pp. 46-52, Oct. 2003.

[WASKV13] Weiß, V. Andrikopoulos, S.G. Saez, D. Karastoyanova, K. Vukojevi. *Modeling Choreographies using the BPEL4Chor Designer: an Evaluation Based on Case Studies*. Report 2013, University of Stuttgart, Institute of Architecture of Application Systems.

[WCLSF05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D.F. Ferguson. *Web Services Platform Architecture*. Book, 2005.

[WKL11] S. Wagner, O. Kopp, F. Leymann. *Towards Choreography-based Process Distribution In The Cloud*. Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems. pp. 490-494, 2011.

[WKL12] S. Wagner, O. Kopp, F. Leymann. *Towards Verification of Process Merge Patterns with Allen's Interval Algebra*. Proceedings of the 4th Central-European Workshop on Services and their Composition (ZEUS 2012). pp. 1-8, 2012.

[WKL13] S. Wagner, O. Kopp, F. Leymann. *Consolidation of Interacting BPEL Process Models with Fault Handlers*. Proceedings of the 5th Central-European Workshop on Services and their Composition, 2013

[WRKUL13] S. Wagner, D. Roller, O. Kopp, T. Unger, and F. Leymann. *Performance optimizations for interacting business processes*. IC2E, 2013

[ZDH06] J.M. Zaha, M. Dumas, A. ter Hofstede. *Service Interaction Modeling: Bridging Global and Local Views*. Enterprise Distributed Object Computing Conference, pp. 45-55, Oct. 2006.

All links were last followed on October 30, 2013.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Signature: