Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Master Thesis No. 0838-001

Consolidation of Process Models that Interact via Compensation Handlers

Prasad Phadnis

Course of Study:	Infotech
Examiner:	Prof. Dr. Frank Leymann
Supervisor:	Dipl.-Inf. Sebastian Wagner
Commenced:	02.02.2015
Completed:	04.08.2015
CR-Classification:	H.4.1

Abstract

Choreographies are generally used to model the interaction behaviour between the processes of different organizations and their suppliers. At times organizations make some business decisions like gaining more control over their suppliers and minimizing the transactional costs leading to the insourcing of these companies. This in turn causes the organization to merge the partner's processes with its own. In previous work an approach has been proposed to consolidate i.e. integrate interacting BPEL process models of different partners into a single BPEL process model by deriving control flow links between the process models from their interaction specification. The resulting merged BPEL process model may contain errors related to the derived control flow links in case of the BPEL constructs such as Fault Handler, Compensation Handler, Termination Handler and Event Handler (FCTE- Handler).

The focus of this thesis is to identify any control flow links violations in case of Compensation Handlers (CH) and resolve them. To achieve that, different scenarios have been identified wherein control flow link violations might occur for CH interacting with partner processes and a solution for each has been provided. The solution proposes an approach to emulate the behaviour of CH where any control link violation occurs thus resulting into a final merged BPEL process model containing no violations associated with CH but having the exact same behaviour.

Contents

1 Introduction ... 11

1.1 Problem statement ... 14

1.2 Scope of work ... 15

1.3 Outline ... 16

1.4 List of abbreviations ... 16

2 Fundamentals ... 18

2.1 Web Services Description Language (WSDL) ... 18

2.2 Business Process Execution Language (BPEL) ... 20

2.2.1 Invoke ... 20

2.2.2 Receive and Reply ... 21

2.2.3 Assign ... 21

2.2.4 Empty ... 21

2.2.5 Correlation Sets ... 22

2.2.6 Scopes and Isolated Scopes ... 22

2.2.7 Flow ... 24

2.2.8 Fault handler ... 25

2.2.9 Compensation handler ... 26

2.2.10 Termination Handler ... 28

2.2.11 Event Handler ... 29

2.3 BPEL4Chor ... 29

2.3.1 Participant Topology ... 30

2.3.2 Participant Behavior Descriptions (PBD’s) ... 31

2.3.3 Participant Grounding ... 31

2.4 Asynchronous and Synchronous Interaction ... 31

2.5 Existing System ... 32

3 Concept and Design ... 35

3.1 Consolidation of Process Models that interact via Fault handler ... 35

3.2 Problem appearing in Process Models interacting via Compensation handler 37
3.2.1 Compensation handler with no communication links 37
3.2.2 Compensation handler with synchronous communication link 38
3.2.3 Compensation handler with asynchronous communication link 39
3.3 Algorithm to determine Compensation Order Graph 40
3.3.1 Pseudocode for the algorithm 40
3.3.2 Application of the algorithm on an example 47
3.4 Consolidation of Process models that interact via Compensation handlers 48
3.4.1 Transformation of <compensateScope> activity in Fault handler 50
3.4.2 Transformation of <compensateScope> activity in Termination handler 51
3.4.3 Transformation of <compensateScope> activity in Compensation handler 52
3.4.4 Transformation of <compensate> activity in Fault handler 52
3.4.5 Transformation of <compensate> activity in Termination handler 54
3.4.6 Transformation of <compensate> activity in Compensation handler 55
3.5 Transformation in case of Nested Scopes 56
3.6 Transformation when Scopes do no complete successfully 58
3.7 Transformation when Scope is within a Repeatable Construct 61
3.8 Summary of Cases addressed 62

4 Implementation 65
4.1 Relationship between various Components 66
4.2 Consolidation Flow 67
4.3 Review of the Merged Process 69

5 Conclusion and Future Work 72

Bibliography 75

List of Figures

1.1 Choreography of two interacting BPEL processes between a manufacturer and supplier [4] 12

1.2 Creation of container process [4] 13

1.3 Control flow link materialization in the container process [4] 13

1.4 Consolidated BPEL process model with resolved control link violations [4] 14

2.1 Building blocks of WSDL 1.1 [8] 18

2.2 Ingredients of a WSDL [9] 19

2.3 Graphical representation of control flow restrictions [7] 25

2.4 Graphical representation of definitions 28

2.5 BPEL4Chor artifacts [17] 30

2.6 Asynchronous and Synchronous interactions transformations 32

2.7 Consolidation – Existing system 33

3.1 Message link pointing into Fault handler [5] 36

3.2 Merged process model with out-factored FH logic [5] 36

3.3 CH with no communication link 37

3.4 Synchronous communication initiating from a CH 38

3.5 Synchronous communication with CH from outside CH 39

3.6 CH with asynchronous communication link 39

3.7 Example business process model [7] 43

3.8 The tree structure containing the COG 48

3.9 Transformation of <compensateScope> activity in FH 50

3.10 Transformation of <compensateScope> activity in TH 51

3.11 Transformation of <compensateScope> activity in CH 52

3.12 Transformation of <compensate> activity in FH 53

3.13 Transformation of <compensate> activity in TH 54

3.14 Transformation of <compensate> activity in CH 55

3.15 Example process containing nested scopes 56

3.16 Process after applying first transformation 57

3.17 Area to be considered while applying second transformation 57

3.18 The final transformed process 58

3.19 Example process when scopes do not complete successfully 59

3.20 Transformed process when scopes do not complete successfully 60

3.21 Transformed process when scopes do not complete successfully 62

4.1 Consolidation process for choreography 65

4.2 Component diagram for choreography consolidation [7] 66

4.3 Component diagram for testing the consolidated process [7] 67

4.4 Sequence diagram for choreography consolidation [7] 68

4.5 Sequence diagram for MergePostProcessor dealing with FCT handlers 70

List of Listings

2.1 BPEL-Element Scope syntax [12] 22

2.2 BPEL-Element Flow syntax [12] 24

2.3 BPEL-Element faultHandlers syntax [12]... 26

2.4 BPEL-Element compensationHandler syntax [12] 26

2.5 BPEL-Element terminationHandler syntax [12]... 29

List of Algorithms

3.1 Pseudocode for the starting function 43

3.2 Pseudocode for the function to create tree.. 44

3.3 Pseudocode for the function to determine the compensation order graph 46

1 Introduction

At the heart of any business lies its business process. It is the most vital part for any business since it drives the business towards achieving its vision and generating profit at the same time. In the past, the businesses were having their own workflows which have evolved over time to the more structured and transparent existing business processes but the motive behind them remains the same. A business process is, "a set of one or more linked procedures or activities which collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional goals and relationships" [1]. Business process execution language (BPEL) is "an XML-based language that enables task sharing in a distributed computing or grid computing environment." It is used to define executable business processes [2]. All the business processes referred in this document are BPEL processes.

It is a common practice among organizations to outsource some task or use an existing service from a third party to reach the final product or service. There are various benefits of outsourcing such as cost advantages, increased efficiency, concentration on core processes rather than supporting ones, etc. and on the other hand there could be other factors such as risk of exposing confidential data, hidden costs, synchronizing the deliverables, etc. which might be disadvantageous [3]. Hence the organization has to make a decision on which parts of the process can be outsourced.

The interaction specifications for the processes of the collaborating organizations are modeled by choreographies via message links interconnecting their communication activities. Considering some of the disadvantages of outsourcing, an organization may decide to insource the partner's business process to save transactional costs or to gain more control over the complete process. This decision results into the consolidation or integration of two communicating BPEL processes into one merged BPEL process model. This conversion process is explained below in brief.

The process consolidation methodology for one-to-one interactions (one instance of a process is communicating with one instance of another process) is divided into four major steps. The utmost important thing to be ensured is; the original control flow relations between all atomic activities that were specified in the choreography must be preserved in the consolidated process. This in turn also ensures that the data flow implied by the control flow is kept as in the BPEL processes. The four steps are listed below [4]:

- Analyzing the control flow relations
- Creation of the container process
- Control flow materialization
- Resolving control link violations

Each of the above mentioned steps is elaborated with the help of an example given below.

Consider an example where an organization manufactures a product but outsources the ordering of parts to a third party supplier organization. The interaction behaviour between the business processes

of the manufacturer and supplier is modeled by choreography as shown in Fig 1.1. The consolidation of such one-to-one interaction is carried out in four major steps as shown below [4]:

1. Analyzing the control flow relations: As one of the most important aspects of the consolidation process is to maintain the original control flow between the activities and thus the first step is to analyze the control flows in the choreography. In this step, the control flow relations between the non-communicating (opaque) activities are identified. The example choreography is depicted in Fig 1.1.

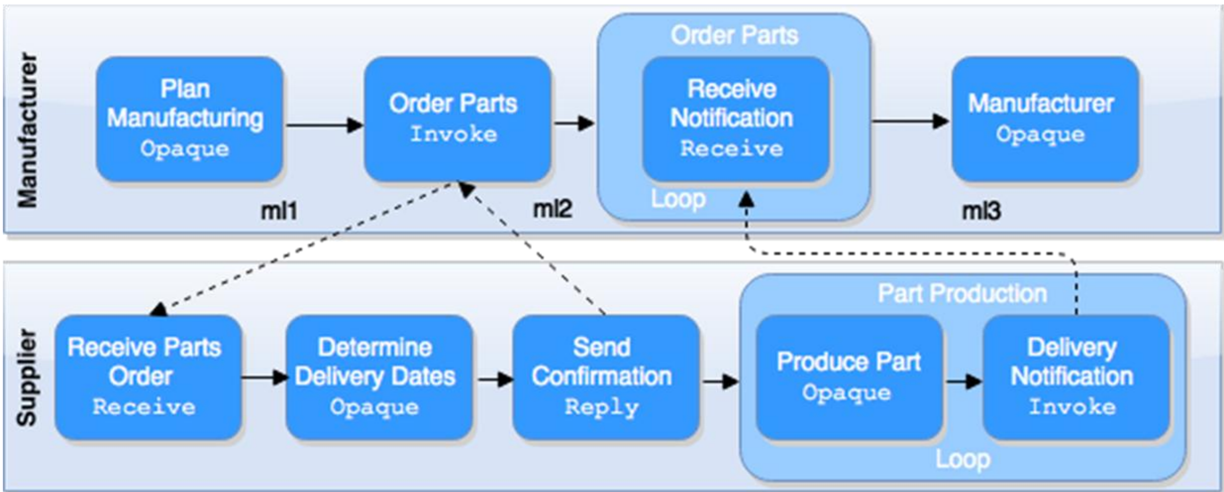

Fig 1.1: Choreography of two interacting BPEL processes between a manufacturer and supplier [4]

2. Creation of container process: Once the control flow relations have been analyzed, to create a consolidated process a container process is created which holds both the processes in an isolated manner. To achieve that a new process is created containing the activities of both the processes in different scopes thus isolating the activities of different processes within the container. For the example considered in Fig 1.1, a new container process named 'Manufacturer' is created and the processes of the manufacturer and the supplier are isolated in two different scopes '$S_{Manufacturer}$' and '$S_{Supplier}$' respectively marked with red boundaries as shown in the Fig 1.2.

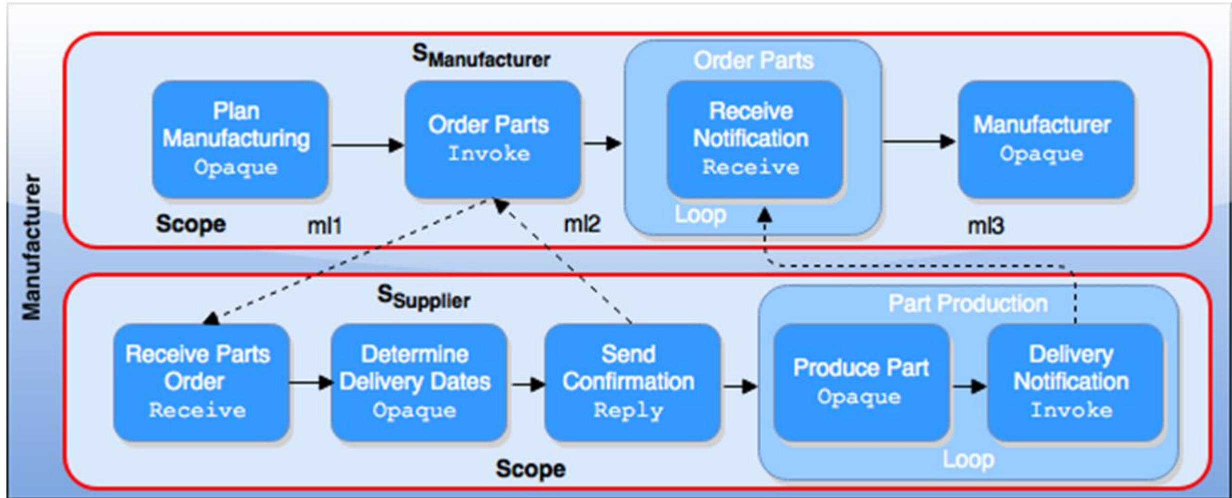

Fig 1.2: Creation of container process [4]

3. Control flow materialization: After analyzing the control flow relations and creating the container process, the analyzed control flow relations are materialized by replacing the message flows between the processes by control links since message flows become obsolete in the single merged business process. The control flow is materialized from the message flow based on the interaction style. The interaction style could be asynchronous or synchronous (refer to section 2.4) but for the example in consideration it is synchronous .The outcome of this materialization is shown in the Fig 1.3.

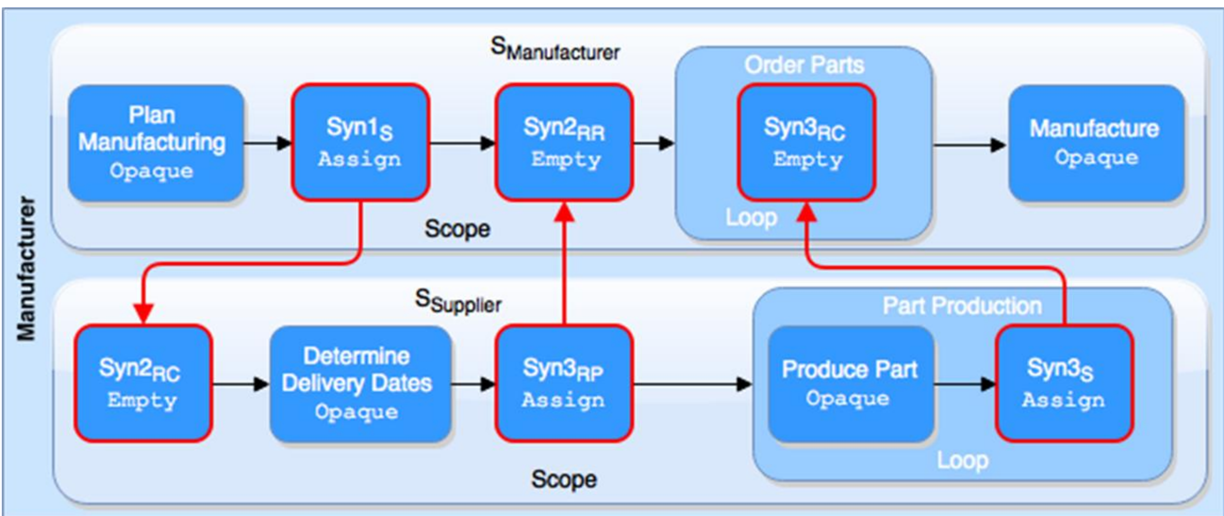

Fig 1.3: Control flow link materialization in the container process [4]

4. Resolving control link violations: To complete the consolidation process this step is very important since the outcome of step three might create a process containing cross boundary link violations which must be resolved. All the cross boundary link violations are identified and a solution must be provided for each to fix them. For the considered example, in Fig 1.3 the cross boundary constraint for loop is violated and hence it must be resolved. Since in this context it does not affect the control flow between non communicating (opaque) activities, the loop Order Parts containing the $Syn3_{RC}$ activity is simply omitted as shown in the Fig 1.4. The consolidated process does not contain any control link violation as all the control links comply with the BPEL specifications and thus is the final merged process for this particular example.

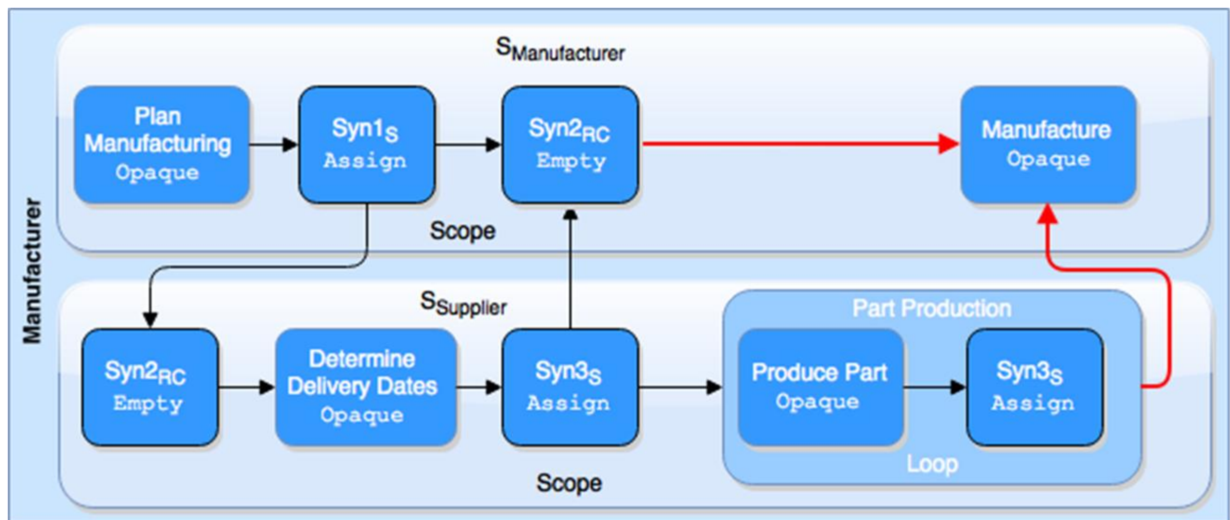

Fig 1.4: Consolidated BPEL process model with resolved control link violations [4]

The above described process consolidation methodology serves as the base for the problem statement at hand explained in the following section 1.1.

1.1 Problem statement

It is very common in the business world to coordinate with each other to achieve a certain goal and in terms of business process it means that the business processes of two different organizations communicating with each other. Choreographies are used to realize such interaction behaviour between the business processes of different organizations and their suppliers. Such kind of interaction can be seen between a manufacturer and supplier business process as shown in Fig 1.1. But there could be various reasons when one organization decides to in source the partner's business process such as to gain more control, reduce the overall transactional costs and sometimes maybe even for data privacy reasons. This kind of scenario occurs during a merger or an acquisition of one company by another. This

leads to the integration of not only the organizational structure but also the processes within the companies. The focus of this work is on the integration at process level specifically on consolidating complementing BPEL process models whose interaction behaviour is described by choreography [5]. Also, the overall performance of the consolidated BPEL process is better as compared to that of choreography [6].

The consolidated BPEL process model must abide to the BPEL specifications. BPEL specifications impose constraints on control flow links such as no inbound or outbound control flow links are allowed for compensation handlers whereas for fault handlers outbound links are permitted but no inbound links. This study focuses on identifying and removing any kind of control link violations for compensation handlers in the consolidated BPEL process model. The scope of this study thesis is described in the following section 1.2.

1.2 Scope of work

In previous work, a technique has been designed and implemented to consolidate two interacting BPEL process models. The resulting BPEL process model contains a merged process wherein all the message flow links have been replaced by control flow links as per the interaction behaviour specification to maintain the control flow. This resulting merged process model might contain some control link violations such as cross boundary link violations and these violations must be identified and fixed. As shown in the Fig 1.3, a scenario has been identified wherein control links are not allowed to cross the loop boundaries as per the BPEL specifications. The solution for that is quite trivial since the example process model is not that complicated and is shown in Fig 1.4. Similarly, there could be different scenarios where control link violations might occur in case of other BPEL constructs such as fault handlers, compensation handlers, termination handlers and event handlers (FCTE handlers).

A solution has been devised to resolve the control link violations occurring in case of fault handlers and termination handlers in [7]. The above approach does not work in case of the compensation handlers due to the additional constraints imposed by the BPEL specifications on the control links. Neither inbound nor outbound control links are allowed in case of compensation handlers. Thus an extension to the solution proposed in [7] is needed to fix the problem associated with compensation handler (CH). The scope of this thesis is to identify all the scenarios associated with compensation handlers where any control link violation occurs after the consolidation of the process models and then emulate the behaviour of compensation handlers where any such violation occurs. The final outcome would be the merged BPEL process model with no control link violations related to compensation handlers.

1.3 Outline

The outline of the thesis document is described in the following manner:

Chapter 2: Fundamentals - In this chapter, the basics of WSDL, BPEL and BPEL4Chor are discussed. All the relevant BPEL activities along with the concepts of fault handler, compensation handler and termination handler are also described.

Chapter 3: Concept and Design – During consolidation the problems associated with compensation handler have been identified and various base case scenarios are described with a solution for each case. An algorithmic solution has been proposed with a working example.

Chapter 4: Implementation – This chapter provides the implementation details of the conceptual solutions proposed in the previous chapter with the help of component and sequence diagrams.

Chapter 5: Conclusion and Future work - A brief summary of the objective of the thesis and how it was achieved is described in this chapter along with the related future work.

1.4 List of abbreviations

BPEL – Business Process Execution Language

BPEL4Chor – BPEL for Choreography

CH – Compensation Handler

COG – Compensation Order Graph

COL – Compensation Order Link

EH – Event Handler

FCT – Fault, Compensation and Termination

FCTE – Fault, Compensation, Termination and Event

FH – Fault Handler

SOA – Service Oriented Architecture

TH – Termination Handler

WSDL – Web Service Definition Language

2 Fundamentals

In this chapter, some basic concepts that are needed for understanding the topic in a better manner have been discussed. The following sections include introduction to WSDL, BPEL, BPEL4Chor, asynchronous and synchronous interaction and the state of the existing system.

2.1 Web Services Description Language (WSDL)

Service oriented architecture (SOA) is an architectural style of developing software that is based on services and there are three basic principles of SOA namely, Publish, Find and Bind [8]. First a service to be provided is defined in an abstract manner and published to a central repository. Next, when someone needs to use a particular service, they query this central repository as per their search criteria and obtain services that closely match these criteria. And finally, the returned information also contains the information about how to access a particular service i.e. how to bind to a service.

Web services are a technical implementation of SOA style. Web Services Description Language (WSDL) is an XML format for describing web services and it also enables to separate the description of the abstract functionality of a service from its concrete detailed description [9].

Fig 2.1: Building blocks of WSDL 1.1 [8]

The Fig 2.1 shows a rough structure of WSDL 1.1 which is used in the BPEL 2.0 specifications and its components are described below:

- **Types:** Contains the definitions of all the data types needed.
- **Message:** It is an abstract definition of the data exchanged.
- **PortType:** All the abstract actions supported by the service are called as operations and a set of operations supported by the service is defined by the port type. The below mentioned four communication patterns are available [10]:
    - **One-way:** Here, simply the web-service receives a message.
    - **Request-Response:** Web-service receives a message and sends a response.
    - **Solicit-Response:** Web-service sends a message and receives a response.
    - **Notification:** Here, simply the web-service sends a message.
- **Binding:** It defines a concrete protocol and data format used to implement a port type i.e. it answers the question of 'how' to invoke a service.
- **Service:** A port is an individual 'end point' identified by a network address supporting a particular binding and service is a collection of such related 'end points'. Thus, it answers the question of 'where' to find a service.

The Fig 2.2 summarizes the ingredients of a WSDL and how they interact with each other.

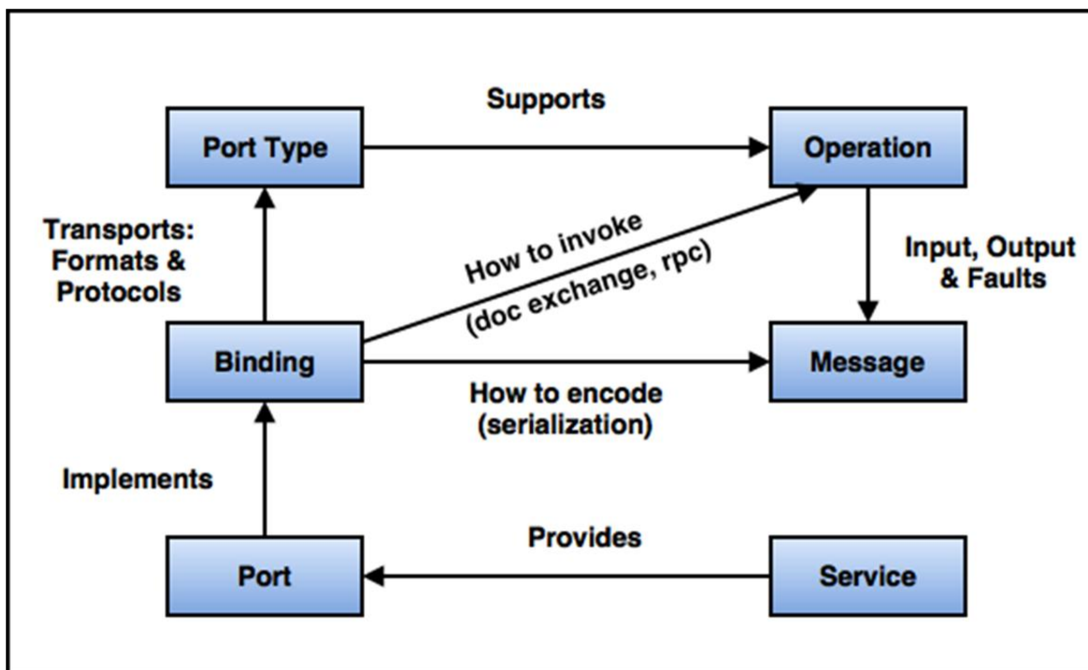

Fig 2.2: Ingredients of a WSDL [9]

2.2 Business Process Execution Language (BPEL)

The Business Process Execution Language (BPEL) is an XML-based language which specifies the business process and its behavior based on web services (refer to section 2.1). Thus BPEL and web services are closely related. To define it abstractly, BPEL is simply a recursive aggregation model for web services. This definition has two important keywords namely, aggregation and recursive. A business process model can tie together a set of web services into one or more new web services i.e. aggregation. These newly created web services can again be tied together into other new web services i.e. recursive [11].

For describing a BPEL process the below listed concepts should be considered:

- Data dependent behaviour is included in the business processes (for example number of items in an order). Such behavior is dealt with by using conditional and time out constructs.
- There has to be a way to specify exceptional conditions and their consequences along with the recovery sequences.
- Multiple nested units of work with its own data requirements are included in long running interactions and business processes frequently require cross partner coordination at various level of granularity.

The above concepts can be applied in one of two ways, abstract process or executable process. An abstract process is not intended to be executed and it is a partially specified process. A process must be explicitly declared as 'abstract' to make it abstract. On the other hand, an executable process is fully specified and thus can be executed. Abstract process serves more of a descriptive role (for example can be used to define a process template). There are different tags and constructs of BPEL which are relevant to this study thesis and are briefly described in the below sections. The whole business process is defined within a <process> tag containing other multiple tags which specify the business logic.

In BPEL, activities are responsible for performing the process logic and are divided into two categories; basic and structured. The elemental steps of the process behavior are described by basic activities. The control-flow logic is encoded by the structured activities and thus they can recursively contain other basic and/or structured activities [12]. The following sub-sections describe few of the basic and structured activities relevant to this study thesis.

2.2.1 Invoke

The main purpose of <invoke> activity is to call web services offered by service providers i.e. to invoke an operation on a service and it is considered as a basic activity. The invocation could either be one-way or request-response (refer to section 2.1). To correlate the business process instance with a stateful

service at the partner's side, zero or more correlationSets (refer to section 2.2.5) can be specified. Although being a basic activity it can enclose other activities inlined in the CH or FH associated with it.

During the invocation of a web service call, a fault might occur and thus the <invoke> activity can have <catch> or <catchAll> blocks to deal with such situations. It can also be associated with another activity that acts as its compensation handler (refer to section 2.2.9). Because of such possibility, it is semantically equivalent to the presence of an implicit <scope> (refer to section 2.2.6) activity immediately enclosing the <invoke> activity providing these handlers.

2.2.2 Receive and Reply

The <receive> activity is used to receive the service requests from the partners of a business process. The <receive> activity plays a vital role in the lifecycle of a business process. It is used to instantiate a business process. A <receive> is a blocking activity in the sense that it is not completed until a matching message is received by the process instance.

The <reply> activity is used to send a response to a previously accepted request such as through a <receive> activity. Only for request-response interactions (i.e. synchronous interactions) these responses are meaningful whereas for one-way interactions a one-way 'response' can be sent by invoking a corresponding one-way operation on the partnerLink.

2.2.3 Assign

The <assign> activity is used to copy the data from one variable to another. It can also be used to construct and insert new data using expressions. Expressions are used to produce a new value by operating on variables, properties and literal constants. One more use of <assign> activity is to copy endpoint references to and from partnerLinks.

2.2.4 Empty

The <empty >activity is used when no action has to be taken i.e. doing nothing, for example in case of a fault that needs to be caught and suppressed. One more use of this activity is to provide a synchronization point in a <flow>.

2.2.5 Correlation Sets

As the name suggests, correlation sets are used to correlate a set of message exchanges by defining correlation identifiers. These are the fields in messages with a business meaning. For example, multiple instances of a process could be running in the BPEL engine communicating with its partners. Thus to be able to identify which response corresponds to which request these correlation identifiers are used in the correlation sets. The properties used in a <correlationSet> must be defined using XML schema simple types.

2.2.6 Scopes and Isolated Scopes

A context is provided by the <scope> activity for the execution of its enclosed activities which impacts their execution behavior. Variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler and a termination handler are included in this behavioral context as shown in the syntax for scope in the listing 2.1:

Listing 2.1: BPEL-Element Scope syntax [12]

```
<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
  standard-attributes>
  standard-elements
  <variables>?
    ...
  </variables>
  <partnerLinks>?
    ...
  </partnerLinks>
  <messageExchanges>?
    ...
  </messageExchanges>
  <correlationSets>?
    ...
  </correlationSets>
  <eventHandlers>?
    ...
  </eventHandlers>
  <faultHandlers>?
    ...
  </faultHandlers>
  <compensationHandler>?
    ...
  </compensationHandler>
  <terminationHandler>?
    ...
  </terminationHandler>
  activity
</scope>
```

The context provided by <scope> activity can be nested hierarchically by having a complex structured activity with many nested activities to arbitrary depth as its primary activity. The above structure is very much similar to a <process> construct except the differences being, <process> construct is not an activity, it cannot have a compensation handler or a termination handler and the isolated attribute cannot be attached to a <process> construct.

Scope States

A scope can be in one of the below listed states [13], [14]:

- **Active:** A scope reaches an active state as soon as the scope's inner activity is activated and it remains in that state as long as positive control flows in a scope.
- **Completed:** A scope reaches a completed state after the successful faultless execution of its inner activity and all of its event handlers have also finished.
- **Compensated:** When the compensation handler associated with a scope is activated for the first time then a scope changes its state to compensated.
- **Faulted:** The scope attains a faulted state when the scope execution is stopped by the occurrence of a fault which is handled by its corresponding fault handler.
- **Terminated:** A scope reaches a terminated state when an error occurs outside of its context i.e. as soon as the first termination activity is reached.

Error handling in scopes

During the execution of a process errors can occur and to deal with such situations it is very important to have some error handling mechanism. BPEL provides some constructs such as fault handler (refer to section 2.2.8), compensation handler (refer to section 2.2.9) and termination handler (refer to section 2.2.10) for the same. All of these BPEL constructs are associated with the <scope> activity either explicitly or implicitly.

Isolated scopes

A scope providing control of concurrent access to shared resources such as variables, partner links and control dependency links is called as an isolated scope. A scope can be made an isolated scope by setting the attribute isolated="yes". By default a scope is not isolated. For example when two concurrent isolated scopes S1 and S2, access these common set of variables or partners links and are carrying out some read write operations, then these read/write operations are conceptually reordered in such a way as if all such activities of one scope were executed before the other. It is very much analogous to the standard isolation level "serializable" used in database transactions.

Isolated scopes must not contain any other isolated scopes whereas it may contain scopes that are not marked as isolated. Also, for an isolated scope the compensation handler does not share the isolation domain of the associated scope whereas the fault handler and the termination handler share the isolation domain of the associated scope.

2.2.7 Flow

Concurrency and synchronization are the two important aspects provided by the <flow> activity. The general syntax is shown in the listing 2.2:

Listing 2.2: BPEL-Element Flow syntax [12]

```
<flow standard-attributes>
  standard-elements
  <links>?
    <link name="NCName">+
  </links>
  activity+
</flow>
```

The fundamental semantic effect of a <flow> activity is to enable concurrency among all of its enclosed activities. A <flow> completes only after all of its enclosed activities are completed. An activity is also considered completed when its enabling condition evaluates to false. The <link> construct is used to define the synchronization dependencies for the enclosed activities. A <link> is associated with the <source> and <target> element which are nested within <sources> and <targets> construct. These are used to establish synchronization relationships through a <link>. The <source> element can specify an optional <transitionCondition> which acts as a guard for following the specified link and when it is not specified then the default value is true. A link can have true, false and undefined status. The <targets> as a whole can specify an optional <joinCondition> which is responsible for the evaluation of all the incoming links and when it is not specified then it is considered as disjunction i.e. logical OR operation by default. Consider an activity 'x' is the target of a link whose source is activity 'y', then the activity 'x' is said to have synchronization dependency on activity 'y'. A link is said to cross the boundary of a construct when it either enters or leaves a construct. The control flow restrictions on links are listed below (see Fig 2.3):

- A link must not cross the boundary of a repeatable construct (<while>, <repeatUntil>, <forEach>, <eventHandlers>) or the <compensationHandler> element.
- For elements like <catch>, <catchAll> or <terminationHandler>, if a link crosses the boundary then it must be an outbound link.
- A <link> declared within a <flow> must not create a control cycle.

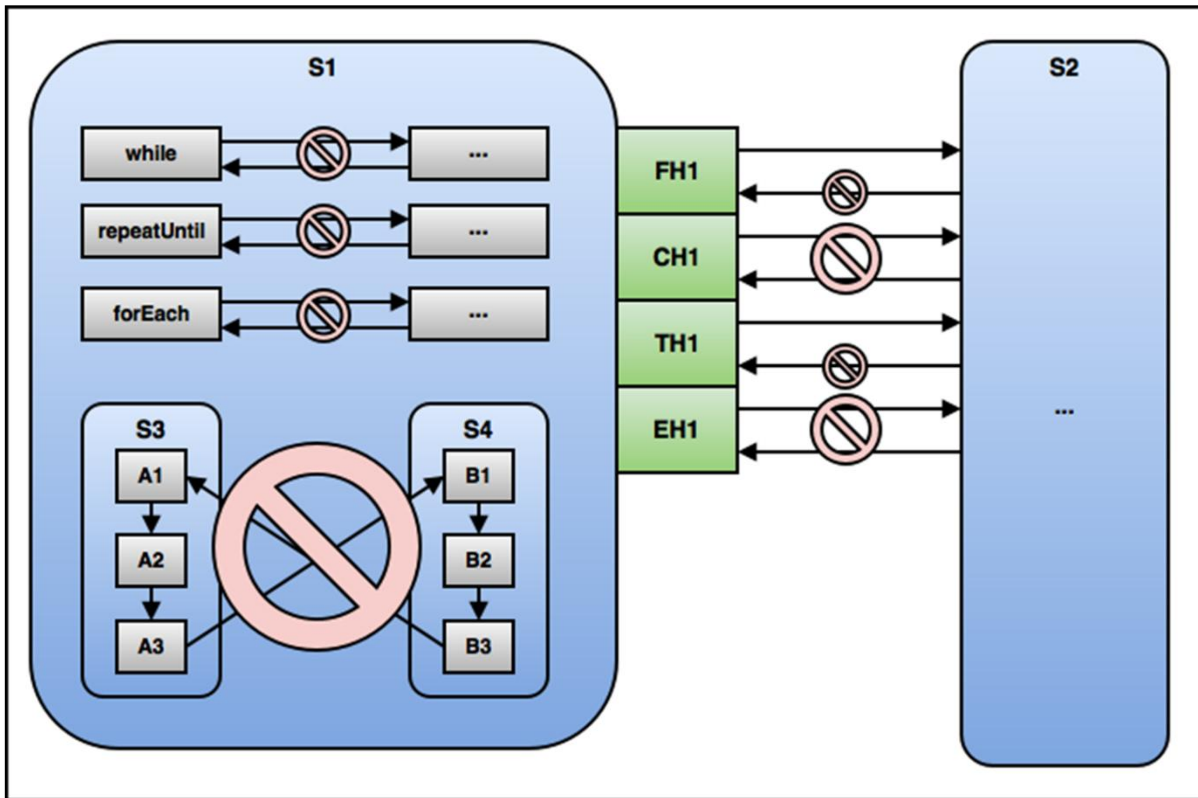

Fig 2.3: Graphical representation of control flow restrictions [7]

Dead-Path Elimination

Dead-Path elimination is used to get rid of all the dead paths [12], [15]. In cases where the control flow is defined by the links and the attribute suppressJoinFailure is set to 'yes', then it means when the join condition for an activity is evaluated to false then it must not be executed i.e. the fault bpel:joinFailure must not be generated. When an activity is not executed because of the <joinCondition> being evaluated to false, then all of outgoing links from such an activity must be assigned a false status. And this in effect propagates the false link status in a transitive manner to all the successive links until some join condition is reached that evaluates to true. This approach is known as Dead-Path Elimination (DPE). By default, for a <process> element the value of the suppressJoinFailure is set to 'no' which avoids suppressing a well-defined fault. This attribute value is inherited by all the nested activities else except when this attribute value is overridden by some other nested activity [12].

2.2.8 Fault handler

When an error occurs during the execution of a scope or process then the execution goes to the fault handler associated with that particular scope or process. In case of scopes, the scope state is changed to faulted (refer to section 2.2.6). The aim of a fault handler is to undo the partial and unsuccessful work of

a scope where a fault occurs. The compensation handler associated with a scope which has faulted is not enabled since it did not complete its execution successfully. A general syntax of a fault handler is shown in the listing 2.3:

Listing 2.3: BPEL-Element faultHandlers syntax [12]

```
<faultHandlers>
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    ( faultMessageType="QName" | faultElement="QName" )? >*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

When an explicit fault handler is defined for a scope, it provides a way to use custom fault handling activities defined by <catch> and <catchAll> constructs. Such a fault handler must have at least one <catch> or <catchAll> element and this requirement must be statically enforced. When faults occur in a scope and the execution goes to its corresponding fault handler, then usually preference is given to a matching <catch> corresponding to that fault to deal with it. If there is no matching <catch> element then the fault will be dealt by <catchAll> element if present. Otherwise, the fault will be handled by the default fault handler. The default fault handler i.e. the implicit fault handler contains a <catchAll> element which has a <sequence> element containing a <compensate> (refer to section 2.2.9)followed by a <rethrow> construct thus making sure, that either a fault will be dealt with properly or it will be re-thrown to its immediately enclosing parent. A <rethrow> activity can only be used within a fault handler. Also, only one explicit or default FCT handler can run for the same scope under any circumstances.

2.2.9 Compensation handler

The ability to undo the work done by successfully executed business logic is one of the key aspects of WS-BPEL. It is achieved by providing a compensation handler for scopes containing the work which can be reversed. The <compensationHandler> construct acts as a wrapper for the activity containing the compensation logic as shown in the listing 2.4:

Listing 2.4: BPEL-Element compensationHandler syntax [12]

```
<compensationHandler>
  activity
</compensationHandler>
```

A compensation handler can either be associated with a scope or an invoke activity (refer to section 2.2.1). For scopes the associated compensation handler is only installed after the scope is successfully executed i.e. the scope state has changed to completed (refer to section 2.2.6). Once a scope completes

successfully, the data context of the scope at the time of its completion is preserved by creating a scope snapshot. This scope snapshot is used by the compensation handler to undo the work done by the scope.

Invoking a Compensation handler

Compensation activities (<compensateScope> and <compensate>) are used to invoke a compensation handler. These compensation activities can only be used within <catch>, <catchAll>, <compensationHandler> and <terminationHandler>.Thus, the compensation activities can only be used within FCT handlers associated with a particular scope. The <compensateScope> activity is used to compensate a specific successfully completed nested scope which can be specified by setting the target attribute (containing the name of the scope to be compensated). The <compensate> activity is used to compensate all the successfully completed nested scopes in default order.

Default Compensation Order

Scenarios wherein multiple compensation handlers must be executed (for example a parent scope fails which contains multiple nested successfully completed child scopes), the BPEL process engine determines a default compensation order in which the compensation handlers must be called. Thus the understanding of how this order is determined is very important. According to BPEL specifications, there are two rules that address the different aspects of order relation. The two rules are mentioned below.

Definition (Control Dependency): When one activity must complete before the execution of another activity starts then those activities are said to be control dependent. For example, when activity A must complete before activity B begins then, activity B has a control dependency on activity A. Control dependencies might occur due to constructs like <sequence> and control links in <flow>. In case an explicit <throw> is used then it is not considered as a control dependency (see Fig 2.4).

Rule 1: Informally, Rule 1 states that the forward order of execution for the scopes being compensated must be respected by the default compensation and thus the compensation order would be the reverse of order of completion. As per BPEL specs to state it exactly, "*Consider scopes A and B such that B has a control dependency on A. Assuming both A and B completed successfully and both must be compensated as part of a single default compensation behavior, the compensation handler of B MUST run to completion before the compensation handler of A is started*" [12]. This rule allows for scopes that were executed concurrently on the forward path to be compensated concurrently on the reverse path.

Definition (Peer-Scopes): When two scopes are enclosed within the same parent scope (including process scope) then such scopes are called as peer scopes (see Fig 2.4).

Definition (Scope-Controlled Set): An activity A is considered to be in the scope-controlled set of a scope S if either it is the scope S itself or is nested within the scope S at any depth (see Fig 2.4).

Definition (Peer-Scope Dependency): When two peer scopes have nested activities which have control dependency among each other then such scopes are said to have a direct peer-scope dependency. For example, scope S1 and S2 are peer-scopes containing activities A and B respectively in their scope-

controlled sets such that B has a control dependency on A. then scope S2 has a direct peer-scope dependency on scope S1. The transitive closure of the direct peer-scope dependency relation is defined as peer-scope dependency relation (see Fig 2.4).

Rule 2: Informally, Rule 2 states that peer scopes must not have cyclic dependencies to be able to be compensated successfully when needed. As per BPEL specs to state it exactly, *"The peer-scope dependency relation MUST NOT include cycles. In other words, WS-BPEL forbids a process in which there are peer scopes S1 and S2 such that S1 has a peer-scope dependency on S2 and S2 has a peer-scope dependency on S1. A process definition containing a cyclic peer-scope dependency relation MUST be rejected."* This rule is enforced by static analysis [12].

The default compensation order derived by following the above mentioned two rules is consistent with the strict reverse order of completion.

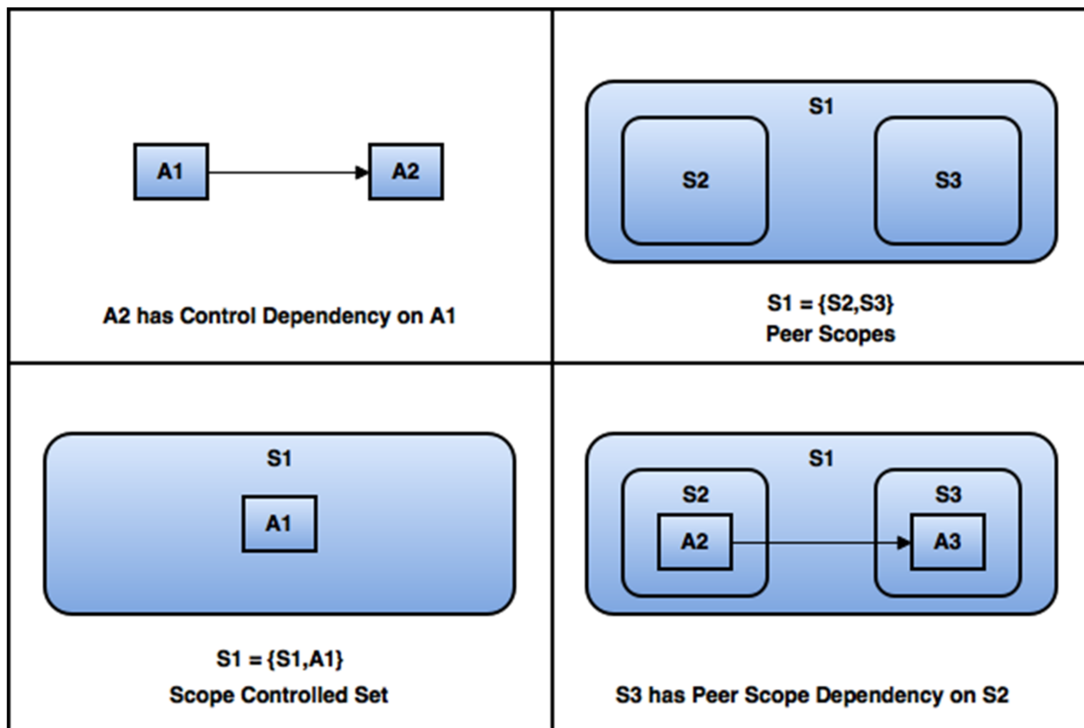

Fig 2.4: Graphical representation of definitions

2.2.10 Termination Handler

When a termination is forced, it disables the event handlers associated with that scope and terminates its primary activity and all the running event handler instances. After this, the custom <terminationHandler> for the scope is run if it is present or the default termination handler is run. The syntax of a termination handler is shown in the listing 2.5:

Listing 2.5: BPEL-Element terminationHandler syntax [12]

```
<terminationHandler>
  activity
</terminationHandler>
```

The termination handler of a scope applies only when the scope is in a normal processing mode. Once a scope has faulted, then the termination handler is uninstalled and the forced termination has no effect. When applying the forced termination, different activities in the scope are treated differently as mentioned below,

- The <assign> activities may be allowed to complete since they are sufficiently short-lived.
- Activities like <wait>, <receive>, <reply> and <invoke> must be interrupted and terminated prematurely.
- The <empty>, <throw> and <rethrow> activities may be allowed to complete but once an <exit> activity starts, it must not be terminated.
- All the structured activity behavior is interrupted.

A termination handler in itself can use the same range of activities as a fault handler including <compensate> and <compensateScope> activity. When a fault occurs in the termination handler, the execution of all the running contained activities must be terminated as a termination handler cannot throw any fault.

2.2.11 Event Handler

Event handlers can be associated with each scope or even with the process scope. As the name suggests, event handlers are invoked when an event occurs and they have the capability to run in a concurrent manner. There are two types of events, first it could be inbound messages corresponding to a WSDL operation and second, alarms that are triggered after user-set times. The child activity within an event handler must be a <scope> activity. Unlike FCT handlers, event handlers are considered a part of normal behavior of the scope.

2.3 BPEL4Chor

An architectural style for building the software system based on services is called Service oriented architecture (SOA). In such an environment, Business Process Execution Language (BPEL) is an established standard for describing long-running business processes. The orchestration of web services into a single business process is done using BPEL [16].

BPEL being an orchestration language specifies the order of execution of activities within an individual process. It also specifies the sequence and conditions for message exchanges in a process. Thus it provides the overview of an individual process and how it communicates with its partners. Choreography on the other hand provides the complete overview of all the processes and how they communicate with each other. It also provides the order and conditions for message exchanges between the processes. BPEL4Chor is a choreography language created to get the global view of all the process by extending the orchestration language BPEL.

BPEL4Chor has three main artifacts [16], [17] as shown in the Fig 2.5:

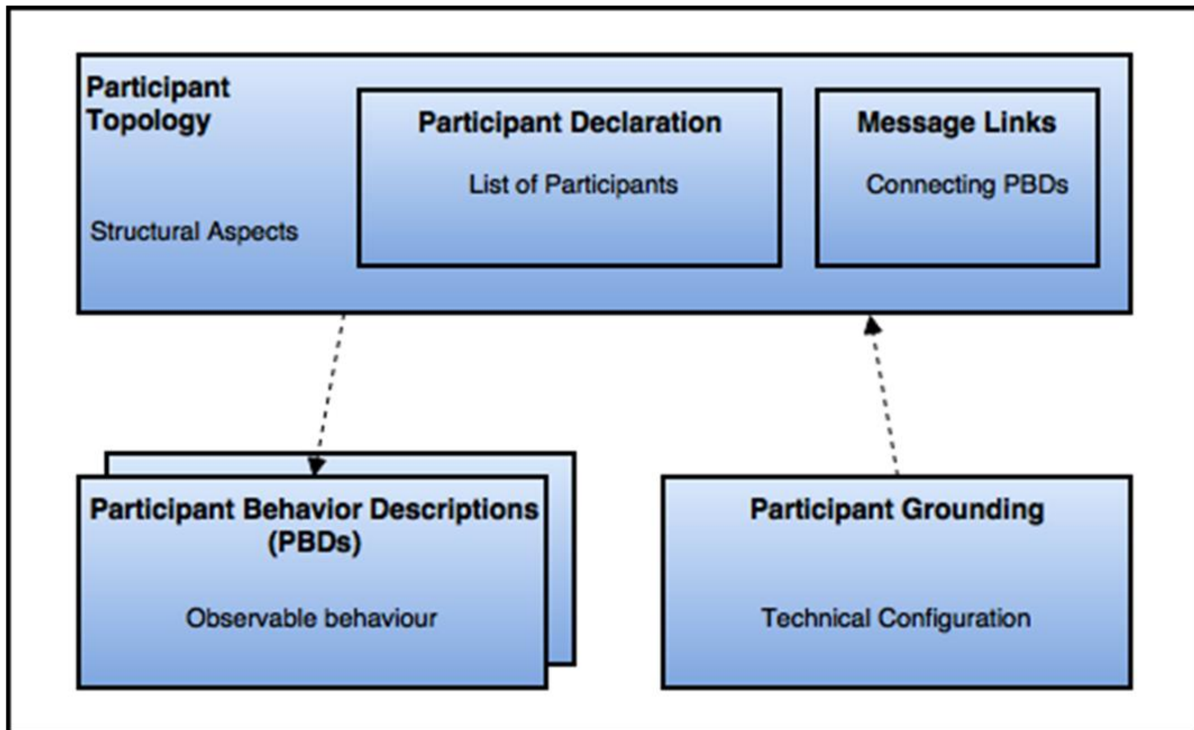

Fig 2.5: BPEL4Chor artifacts [17]

2.3.1 Participant Topology

The structural aspects of choreography are defined by a participant topology. It acts as 'glue' between the participant behavior descriptions and it has introduced the following three notions:

participant type: Each participant behavior description corresponds to one participant type implying that the same participant behavior description is applied to all the participants of the same type.

participant reference: As the name suggests it is simply a pointer to a participant i.e. participant reference points to a participant.

message link: Which participant can potentially communicate with which other participants is defined by message links.

2.3.2 Participant Behavior Descriptions (PBD's)

At the heart of choreographies lies their communication activities (message send and receive) along with their control and data flow dependencies. BPEL provides a rich set of constructs for the same and they are used unchanged in BPEL4Chor. Based on the abstract process profile for observable behavior specified by BPEL, the abstract process profile for participant behavior descriptions is derived stating the requirements for defining the behavior of each participant. To be able to uniquely reference activities from abstract process models an identifier is needed for each activity in a process. Not all the activities in BPEL have a name attribute (for example onMessage branches) and thus a new attribute wsu:id of type xsd:id is introduced as a new attribute for communication activities and onMessage branches. This new attribute is used to reference any activities in the participant topology by message links for any message exchange.

2.3.3 Participant Grounding

Participant topologies and PBD's contain no technical configuration stuff and thus the mapping to the web service specific configurations is introduced in participant groundings. A grounding is only considered valid when all the message links are grounded. Each and every PBD can be transformed by following the abstract process profile for observable behavior to an executable BPEL process only after choreography is completely grounded.

2.4 Asynchronous and Synchronous Interaction

As discussed in Chapter 1, the consolidation process consists of four steps. In the third step, 'Control flow materializations' where control links are derived from the message links and also the communicating activities are transformed depending on the type of interaction. There are two kinds of interactions described below.

First, asynchronous interaction wherein the activity that sends the message (sender) does not wait for a reply and it continues with its execution. The receiving activity (receiver) waits for a message to be received before continuing its execution thus imposing control flow relations between the activities of the sender and the receiver (for example, activity B2 must be executed after activity A1 completes). For such interaction, the <invoke> activity is replaced by an <assign> activity and the <receive> activity is

replaced by an <empty> activity which acts as a synchronization point for the control flow at the receiver side. Then the original link is materialized as a control flow link as shown in the left side of the Fig 2.6 thus maintaining the control flow relations imposed by asynchronous interactions.

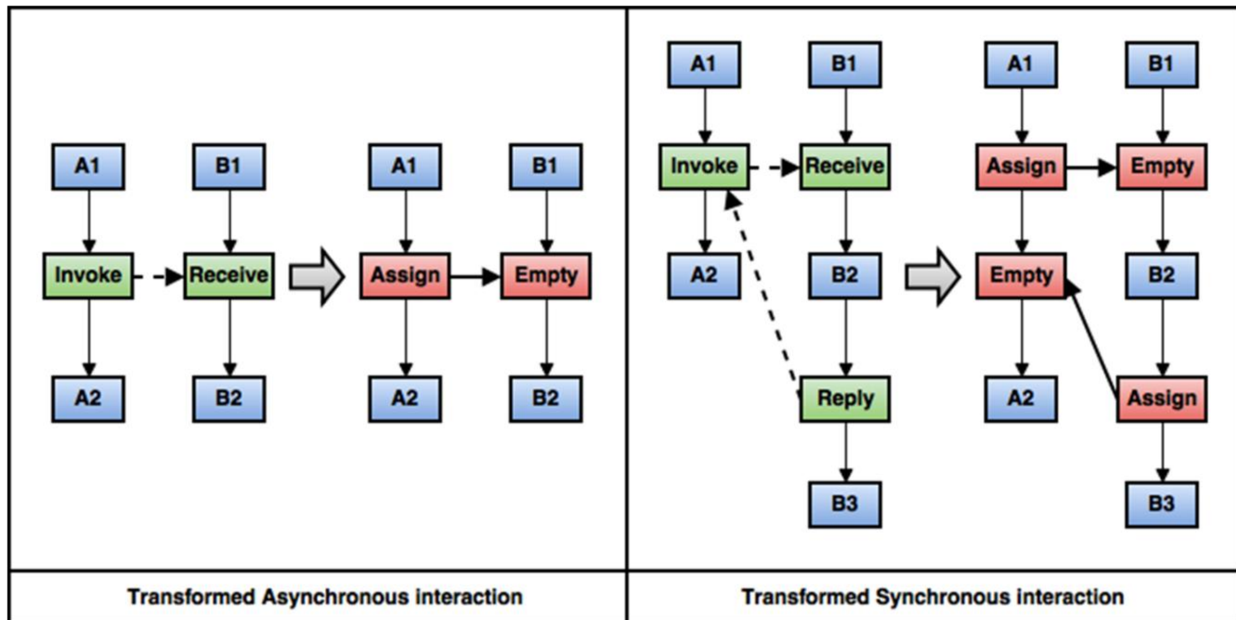

Fig 2.6: Asynchronous and Synchronous interactions transformations

Second type, synchronous interaction wherein the activity that sends the message (sender) waits for a reply and it halts its execution. Once a reply is received then the sender resumes with its execution. This kind of interaction imposes control flow relations between the activities of the sender and the receiver (for example, activity A2 must be executed after activity B2 completes which in turn can only be executed after activity A1 completes). For such interaction, the <invoke> and <receive> activities are replaced by <assign> and <empty> activities respectively. In addition to that, an <empty> activity is added to the sender's side as a synchronization point. As in the asynchronous interaction case, the two message flows are materialized as control flows. In spite of this, a new control link is created between the <assign> activity and newly added <empty> activity at the sender's side to ensure the original control flow relation. This transformation is shown on the right hand side of the Fig 2.6.

2.5 Existing System

The current state of the system is based on the previous works of [18], [7] and [19]. The existing system takes a zip file as input containing the choreography in BPEL4Chor and its associated WSDL's as shown in the Fig 2.7. The consolidation process takes place in the merge module.

It contains three parts, first a Pre-Merge Processor which converts all the <invoke> activities which are not in a scope into a scope. It is just an alternative representation of the <invoke> activity. Second, the actual merge module where the choreography is converted to an abstract BPEL process and the consolidation takes place. Here, the process containing a <flow> activity is created. This <flow> activity in turn contains the scopes enclosing the logic of the individual PBD's. All the message links (refer to section 2.3.1) are examined to match with a consolidation pattern and the transformation takes place on that basis. In the end, all the technical information from Participant Grounding (refer to section 2.3.3) is added to the process. Third, a Post-Merge Processor in which all the control link violations associated with fault handler are identified and fixed. The final output of the merge module is a zip file containing the consolidated process and its associated WSDL's as shown in the Fig 2.7.

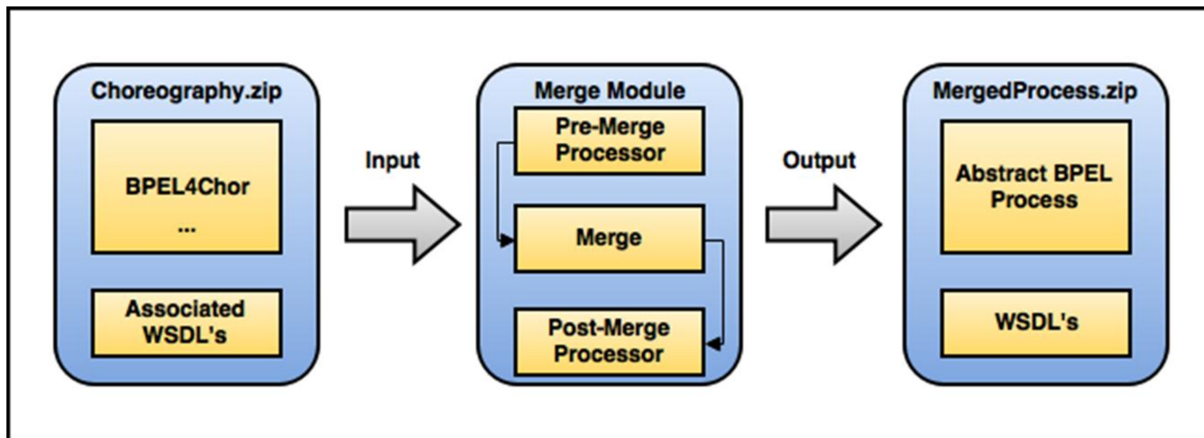

Fig 2.7: Consolidation – Existing system

3 Concept and Design

This chapter elaborates the problem at hand and answers questions like how the problem arises along with identifying the various scenarios in which the problem might arise and provides a solution for the same. As described in Chapter 1, the third step of the consolidation process is 'Control flow materialization' after which the consolidated BPEL process model may contain control link violations such as cross boundary link violations. The final step aims at resolving all such control link violations. Briefly, this study thesis aims at providing a solution for cross boundary link violations occurring in consolidation of process models that interact via compensation handlers.

All the analysis in this study thesis is done based on the following pre-conditions:

- The choreography is based on one-to-one (i.e. one instance of a process communicating with one instance of another process) interactions [20].
- All the given processes are correct and without any deadlock.
- Repeatable constructs do not contain any communication links with other activities through message links (a possible solution is mentioned briefly in the section 3.7).

3.1 Consolidation of Process Models that interact via Fault handler

In this section, a brief description of the solution where consolidation of the process models interacting via FH is provided. As per the BPEL specifications, no inbound control links are allowed for FH's whereas outbound links are permitted.

Consider a scenario as shown in the Fig 3.1, where a message link is pointing into a FH due to the synchronous interaction behaviour between two processes; process A and process B and the consolidated process P_{merged} containing materialized control flows.

As seen in the Fig 3.1 the control flow link l_2 crosses the boundary of FH_A in order to realize that activity A5 is executed after activities B1 and B2 respectively. This violates the cross-boundary link constraint on the FH. The developed solution for such violations proposes that all the activities containing any inbound control flow links be taken out of the fault handler. They are replaced by an empty activity (which does nothing) within a fault handler and this empty activity has a control link to the out-factored activities of the FH containing the FH logic (see Fig 3.2).

As shown in the Fig 3.2, an empty activity A7 is created which has an outbound control flow link to the activity A3 containing the FH logic, thus making sure that whenever this FH is called its FH logic is executed and all of this is enclosed in the new scope S_{FH} ensuring access to all the required variables.

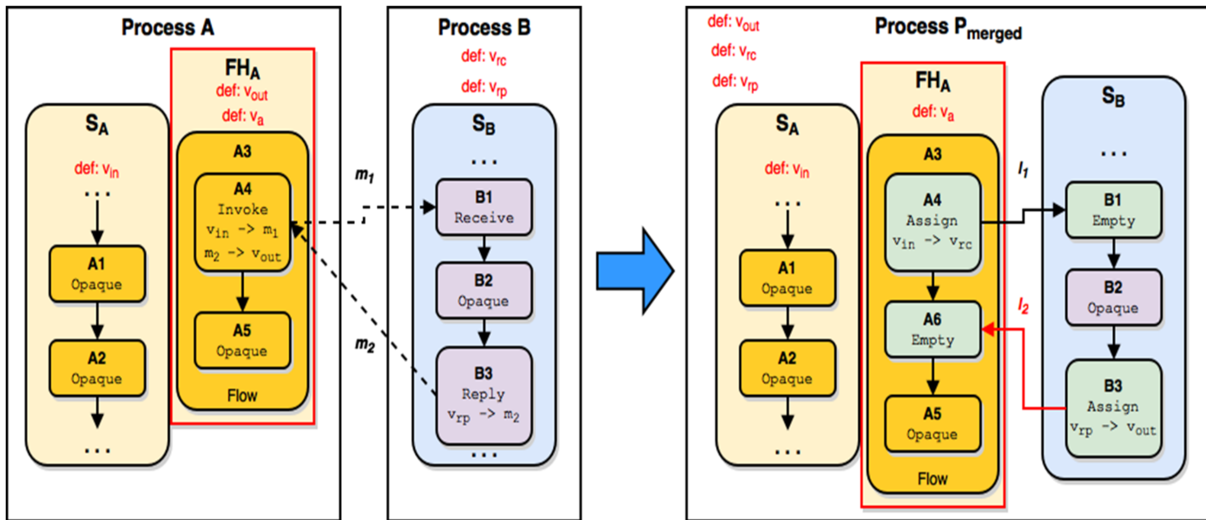

Fig 3.1: Message link pointing into Fault handler [5]

The control flow relation between the non-communicating activities (for example activity A5 must be executed after activity B1 and B2 respectively) as defined in the choreography is maintained in the proposed solution [5].

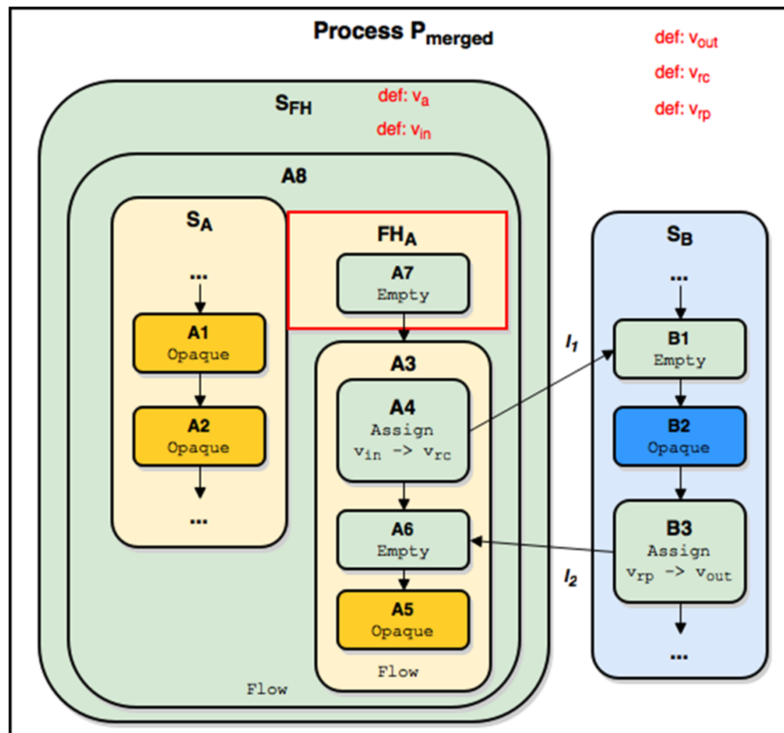

Fig 3.2: Merged process model with out-factored FH logic [5]

3.2 Problem appearing in Process Models interacting via Compensation handler

In this section, various scenarios have been identified where control link violations might or might not occur in case of consolidation of process models interacting via CH. One of the most important steps in the consolidation of process models is 'Control flow materializations' wherein all the message links in the choreography are replaced by control links. This serves as the basis to identify scenarios where this step might result in the control link violations for CH's. Thus it is necessary to segregate scenarios based on the basis of message links i.e. if compensation handlers contain any message links and if so what kind of message links. Each identified scenario is listed below and elaborated in the following subsections,

- Compensation handler with no communication links
- Compensation handler with synchronous communication link(s).
- Compensation handler with asynchronous communication link(s).

3.2.1 Compensation handler with no communication links

This subsection describes the trivial scenario wherein the CH contains no communication links at all. The Fig 3.3 illustrates this case.

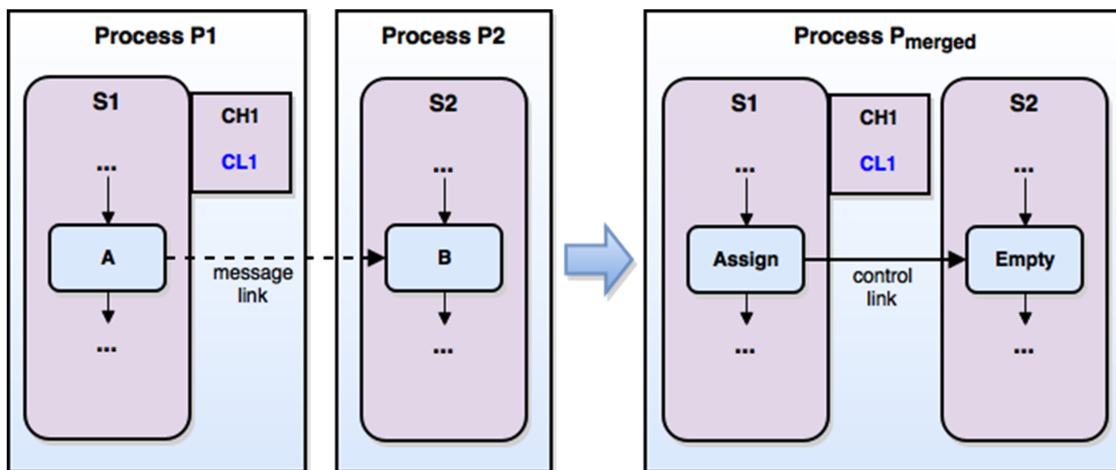

Fig 3.3: CH with no communication link

The diagram on the left side in the Fig 3.3 shows a simple choreography between two processes P1 and P2 where they interact via a message link originating from scope S1 of process P1 pointing into scope S2 of process P2. The scope S1 in process P1 contains a compensation handler CH1 associated with it. There are no communication links going in or out of the compensation handler CH1. The right side of the Fig 3.3 shows the consolidated process P_{merged} wherein the message link is replaced by a control link. The enclosing scopes for scope S1 and S2 have been omitted for the sake of simplicity. There is no change in

the compensation handler CH1 associated with scope S1. Thus, after transformation there is no control link violation associated with the CH.

3.2.2 Compensation handler with synchronous communication link

This subsection elaborates a scenario where the CH initiates a synchronous communication call as shown in the Fig 3.4. The left side depicts choreography between two processes P1 and P2. The process P1 has a scope S1 and its associated compensation handler CH1. The invoke activity within this CH1 sends a synchronous message m to the receive activity within the scope S2 of process P2. Since the message m is a synchronous message the control flow of the compensation handler CH1 is blocked until it receives the message m' from the reply activity within the scope S2 of process P2. Once this reply message is received the compensation handler CH1 can resume the execution with the further activities and also the process P2 executes normally.

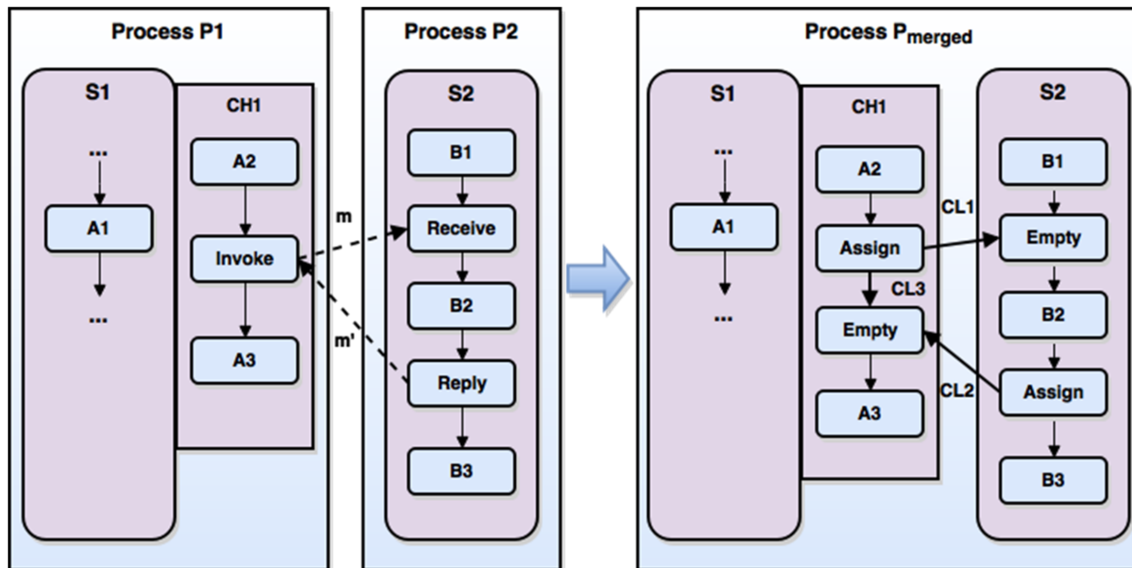

Fig 3.4: Synchronous communication initiating from a CH

On the right side of Fig 3.4 is the consolidated process P_{merged} wherein the invoke activity in the compensation handler CH1 of scope S1 is replaced by an assign activity and an empty activity and the receive activity in scope S2 is replaced by an rmpty activity and the reply activity is replaced by an assign activity in scope S2. Also, the message flow links have been replaced by the control flow links CL1 and CL2.

The Fig 3.4 illustrates a scenario wherein the synchronous call is initiated from within the compensation handler CH1 associated with scope S1 of process P1. A similar situation might arise when the call is initiated from outside the compensation handler but still resulting in synchronous inbound as well as outbound message links as shown in the below Fig 3.5,

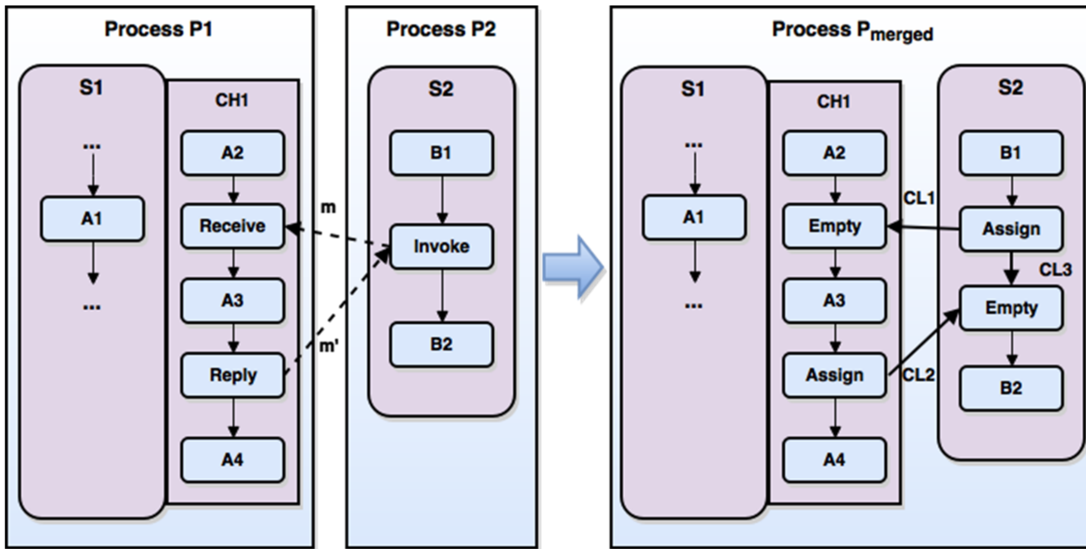

Fig 3.5: Synchronous communication with CH from outside CH

As seen clearly on the right in both the Fig 3.4 and Fig 3.5, the control flow links CL1 and CL2 are violating the cross boundary link constraints of the CH as per the BPEL specifications.

3.2.3 Compensation handler with asynchronous communication link

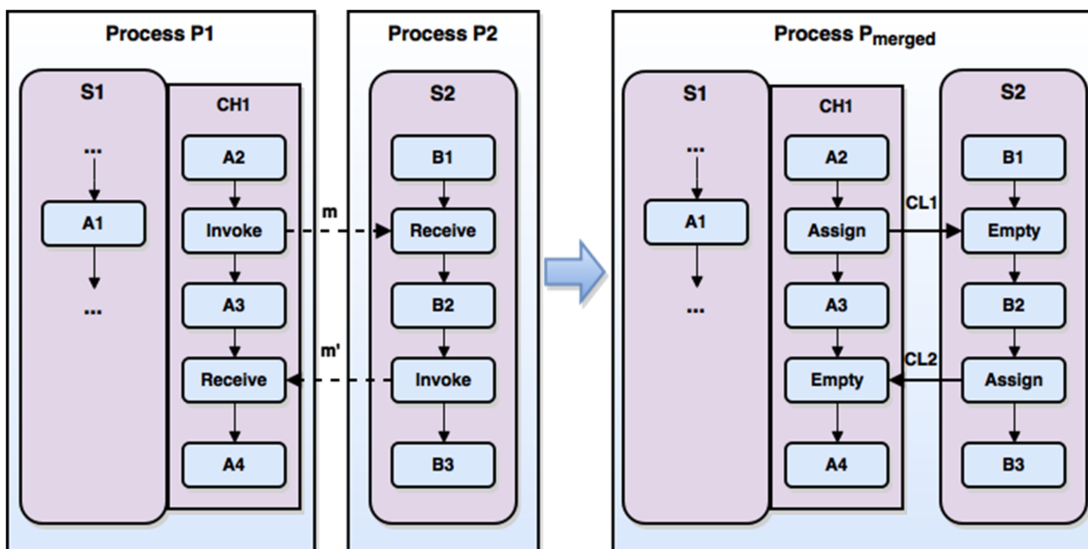

Fig 3.6: CH with asynchronous communication link

This subsection elaborates a scenario where the CH contains both asynchronous inbound and outbound communication links as shown in the Fig 3.6. The left side in figure depicts choreography between two processes P1 and P2. The process P1 has a scope S1 and its associated compensation handler CH1. This compensation handler CH1 has an outbound asynchronous message link from an invoke activity to the receive activity within the scope S2 of process P2 and similarly it has an inbound asynchronous communication message link as well. Point to note here is none of the execution waits to receive any kind of reply here since the communication is asynchronous.

On the right side of the Fig 3.6 is the process P_{merged} wherein all the invoke activities have been replaced by assign activities and the receive activities have been replaced by empty activities. Also, message links have been replaced by the control links CL1 and CL2. As seen clearly these control flow links are violating the cross boundary link constraints on the CH as per the BPEL specifications.

3.3 Algorithm to determine Compensation Order Graph

When a business process is being executed and for some unforeseen reason a fault occurs, compensation handlers for all the successfully completed scopes are invoked by the execution engine and they are executed in a particular order known as default compensation order as described in Chapter 2. This section elaborates an algorithm that has been devised to determine this default compensation order and construct Compensation Order Graph (COG) for any given business process containing multiple compensation handlers. The following section 3.3.1 provides the pseudocode for the devised algorithm along with the detailed explanation.

3.3.1 Pseudocode for the algorithm

The general structure of any business process consists of a main process containing a set of other activities such as flow, sequence, invoke, etc. nested within it. The activities such as flow, scope, sequence can further have more nested activities and so on. The closest data structure that can represent such a structure efficiently is tree data structure since it also has a root node that may contain other nodes as its nested children and so on. Thus the main idea behind this algorithm is to first convert the input BPEL process into a tree structure and then process this tree to determine the correct COG. The most important activity into consideration is a scope since a CH can only be associated with a scope and it can only be called from immediately enclosing parents FH or CH or TH which are also associated with scopes. Also a scope can be nested within a flow activity or a sequence activity and thus the nodes in the tree structure created are differentiated or categorized into 3 types; sequence, flow and scope. Based on the type of the node the corresponding processing is carried out.

For sequence and flow nodes the processing is trivial because Compensation Order Link (COL) is added to its parent and the algorithm continues processing the next element. But the processing gets complex if the node type is scope. In this case also a COL is added to its parent along with a check to see if this node has child nodes and if so COL's are added from each of them to itself. The processing logic also checks if it has any control dependent nodes. In that case COL's are added from the control dependent nodes to each of its children if it has any else to itself.

Before diving deep into the pseudocode of the algorithm, a Data Dictionary (DD) has been defined explaining all the terms and functions that are used in the pseudocode.

Data Dictionary

Term	Description
TreeNode(TN)	A node structure for a tree containing links to its children, parent and other control dependent TN's. Also it contains the information about the node type i.e. if it is a scope, flow or sequence.
rootNode	It is the root of the tree structure for the given process.
createTree(P) [Algorithm 1.2]	It is a function that takes any process P as the input and generates the complete tree structure for that process.
createTreeIterator(TN)	A function that returns an iterator for the input tree node TN containing a list of all of its immediate children along with immediately control dependent nodes(if any).
createImmediateChildIterator(TN)	This function returns an iterator for the input tree node TN containing a list of all of its immediate children.
createImmediateControlDepIterator(TN)	This function returns an iterator for the input tree node TN containing a list of all of its immediate control dependent tree nodes.
hasMoreElements(Ti)	A function to check if the input tree iterator Ti has more elements in the list.
getNextIteratorElement(Ti)	This function returns the next element to be processed from the input tree iterator Ti.
getType(TN)	Returns the type of the input tree node TN i.e. if it is a sequence or flow or scope.
Compensation Order Link(COL)	It is a link from one TN to another imposing a compensation order between them.
Control Link (CL)	It is a link indicating the control dependency from one node to another.
Nested Link (NL)	It is link indicating parent-child relation between two nodes.
addCOL(fromTN, toTN)	This function adds a COL from source tree node fromTN to target tree node toTN.

isProcessed(TN)	Checks if the input tree node TN has already been processed i.e. visited.
setProcessed(TN)	Sets the processed flag of the input tree node TN to true.
createCompensationOrderForProcess(P) [Algorithm 1.1]	This function creates the tree structure for input process P and calls another function to create the compensation order.
createCompensationOrder(rootNode) [Algorithm 1.3]	The actual logic to determine the compensation order for the given input tree resides in this function and the output of this method is the compensation order graph.
addToQueue(TreeNode tn)	Adds a new tree node tn to the queue.
getNextQueueElement(Queue q)	Returns the next element in the input queue q.
getNestedElementIterator(TreeNode tn)	This function returns an iterator for the nested children list of the input tree node tn.
getDependentElementIterator(TreeNode tn)	This function returns an iterator for the dependent tree nodes list of the input tree node tn.
addNL(fromTN, toTN)	This function adds a nested link from source tree node fromTN to target tree node toTN.
addCL(fromTN, toTN)	This function adds a control link from source tree node fromTN to target tree node toTN.
inQueue(TreeNode tn)	This function returns a true or false depending on whether the tree node tn is present in the queue.

The approach that this algorithm follows is a '**Depth First Search**' wherein the tree is traversed one branch at a time till the leaf nodes are reached while processing each node on the way, thus it takes care of the nested scopes as well as control dependent nodes while determining the correct compensation order for the input process. To better understand the pseudocode a sample business process is taken from [7] as shown in the Fig 3.7.

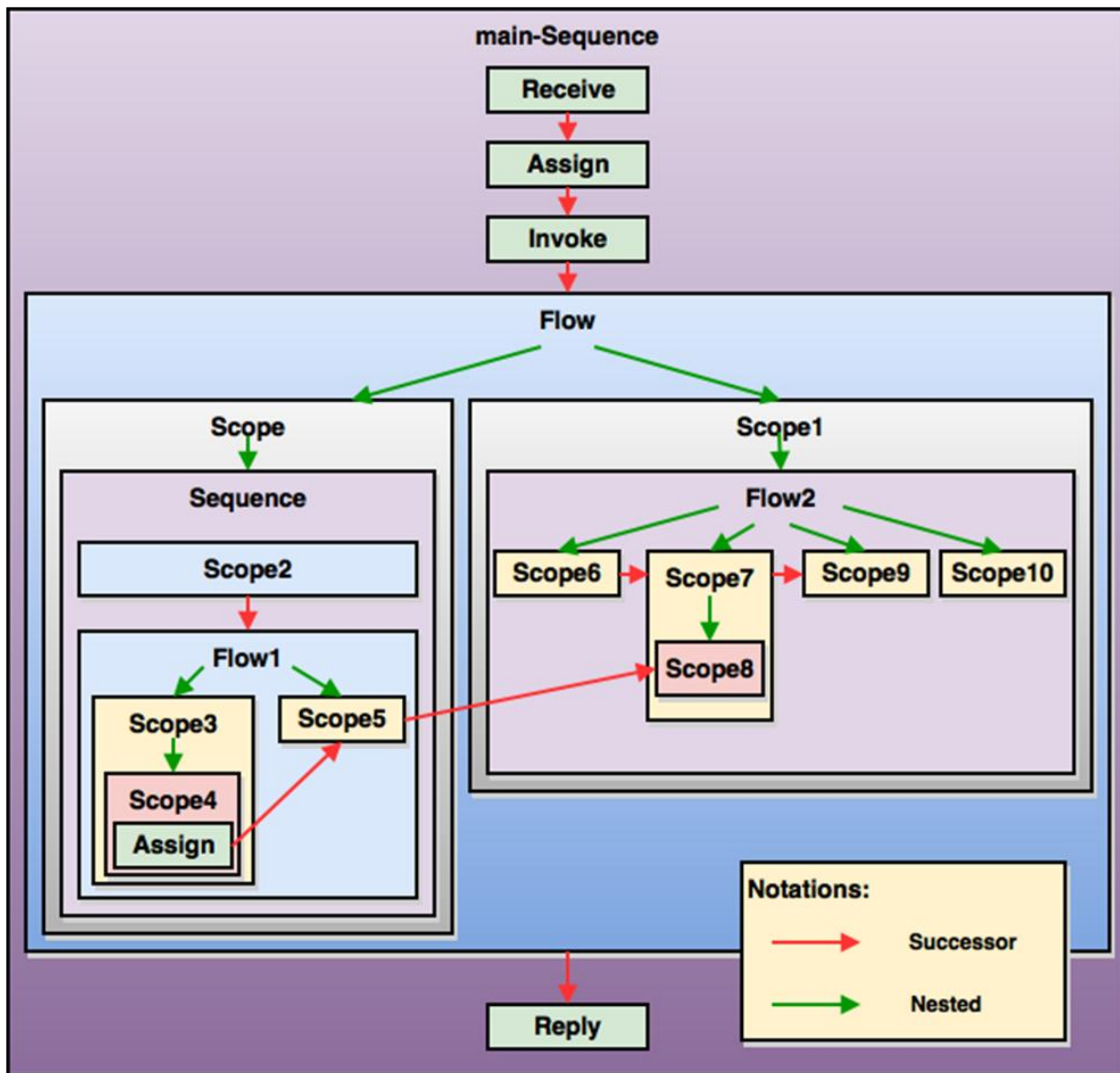

Fig 3.7: Example business process model [7]

The complete algorithm is divided into three parts listed below,

Algorithm 3.1 pseudocode for the starting function

1. ***procedure*** *createCompensationOrderForProcess(Process P)*
2. *rootNode = NULL*
3. *rootNode = createTree(P)*
4. *createCompensationOrder(rootNode)*
5. ***end procedure***

The above pseudocode acts as the entry level function from where the call to create the COG starts. The pseudocode for the delegated function calls are given below.

Algorithm 3.2: pseudocode for the function to create tree

```
1.  procedure createTree(Process P)
2.      processTN = new TN
3.      elementQueue = NULL
4.      addToQueue(processTN)
5.      while hasMoreElements(elementQueue) do
6.        nextElement = getNextQueueElement(elementQueue)
7.        setProcessed(nextElement)
8.        nestedElementIterator = getNestedElementIterator(nextElement)
9.        while hasMoreElements(nestedElementIterator) do
10.         nestedElement = getNextIteratorElement(nestedElementIterator)
11.         addNL(nextElement,nestedElement)
12.         if isProcessed(nestedElement) OR inQueue(nestedElement) then
13.           Continue
14.         else
15.           addToQueue(nestedElement)
16.         end if
17.       end while
18.       dependentElementIterator = getDependentElementIterator(nextElement)
19.       while hasMoreElements(dependentElementIterator) do
20.         dependentElement = getNextIteratorElement(dependentElementIterator)
21.         addCL(nextElement,dependentElement)
22.         if isProcessed(dependentElement) OR inQueue(dependentElement) then
23.           Continue
24.         else
25.           addToQueue(dependentElement)
26.         end if
27.       end while
28.   end while
29. end procedure
```

The algorithm 3.2 uses a '**Breadth First Search**' approach while constructing the tree from the given input process thus maintaining the parent-child relations in case of the nested scopes. Also the control dependent nodes are taken into consideration while constructing the tree.

The input to the function is the complete process. To realize the breadth first search approach, queue is used as a data structure. This queue holds all the tree nodes to be processed. The processing continues

till the queue is empty i.e. there are no more nodes to be processed. Each tree node has a corresponding processed flag indicating if that node has already been processed and if so that node must not be added to the queue again. The flow of the pseudocode is briefly elaborated below.

Lines 4-7: The very first step is to get all the immediate nested children and add them to the queue. For the sample business process shown in Fig 3.7, if the activity in consideration is ‘Flow2’ then all of its immediate nested children i.e. ‘Scope6’, ‘Scope7’, ‘Scope9’ and ‘Scope10’ are added to the queue.

Lines 8-17: Process one element at a time from the queue, extract one element from the queue and get all of its immediate children. While iterating through the list of children add a nested link from the child to the parent and check if the child node is already processed or is present in the queue then continue with the children list otherwise add it to the queue. In the sample business process shown in Fig 3.7, if ‘Scope7’ was extracted from the queue, it is marked processed, a nested link is added from ‘Scope7’ to ‘Scope8’ and ‘Scope8’ is added to the queue if it is not processed or it is not in the queue.

Lines 18-27: Now for the extracted element from the queue, get all the immediate control dependent nodes. While iterating through that list add a control link from parent to that node and again add it to the queue if that not has not been processed yet or is not in the queue. For the example in Fig 3.7, consider ‘Scope7’ was the extracted element, a control link is added from ‘Scope7’ to ‘Scope9’ and ‘Scope9’ is added to the queue if it is not in the queue or has not been processed yet.

The algorithm 3.3 generates a COG as an output and using that graph the compensation order is determined for the whole input process. In case only a specific part needs to be compensated then simply pass that as the input root node instead of the whole process thus making it work for specific cases as well as the whole process.

The input to the function is the root node of the tree structure created by algorithm 3.2 for the input business process. To realize the depth first approach, this algorithm is designed in a recursive manner thus implementing the stack data structure on internal memory. The flow of the pseudocode is briefly elaborated below.

Lines 2-4: Checks if the input node is already processed then the procedure ends. It is very important part of this pseudocode since it indicates the exit condition of the recursive call.

Lines 5-6: Creates an iterator for the input tree node that iterates over the immediate children of that node.

Lines 7-11: Gets the first iterator element and checks for the type of the node. If it is a SEQUENCE or FLOW then it sets the parent node to processed, adds a COL from this node to its parent node and calls this function recursively by passing this node as an argument. For the sample process shown in Fig 3.7, if the extracted node is ‘Flow2’ then it adds a COL from ‘Flow2’ to ‘Scope1’ (its parent) and sets ‘Scope1’ to processed. Recursively call the same function to passing ‘Flow2’ node as the argument.

Algorithm 3.3 pseudocode for the function to determine the COG

```
1.  procedure createCompensationOrder(rootNode)
2.      if isProcessed(rootNode) then
3.          return
4.      end if
5.      treeIterator = createTreeIterator(rootNode)
6.      while hasMoreElements(treeIterator) do
7.          nextTN = getNextIteratorElement(treeIterator)
8.          if getType(nextTN) is SEQ|FLOW then
9.              addCOL(nextTN, rootNode)
10.             setProcessed(rootNode)
11.             createCompensationOrder(nextTN)    // Recursive call
12.         elseif getType(nextTN) is SCOPE then
13.             addCOL(nextTN, rootNode)
14.             setProcessed(rootNode)
15.             childIterator = createImmediateChildIterator(nextTN)
16.             while hasMoreElements(childIterator) do
17.                 childTN = getNextIteratorElement(childIterator)
18.                 addCOL(childTN, nextTN)
19.             end while
20.             ctrlDepIterator = createImmediateControlDepIterator(nextTN)
21.             while hasMoreElements(ctrlDepIterator) do
22.                 ctrlDepTN = getNextIteratorElement(ctrlDepIterator)
23.                 if childIterator != NULL then
24.                     while hasMoreElements(childIterator) do
25.                         childTN = getNextIteratorElement(childIterator)
26.                         addCOL(ctrlDepTN, childTN)
27.                     end while
28.                 else
29.                     addCOL(ctrlDepTN, nextTN)
30.                 end if
31.             end while
32.             createCompensationOrder(nextTN)    // Recursive call
33.         end if
34.     end while
35. end procedure
```

Lines 12-19: If the type of the node is SCOPE then mark its parent node as processed and for all of its immediate children add a COL from each of them to itself. So for the sample shown in Fig 3.7, if the

node is ‘Scope7’, then it marks node ‘Flow2’ (its parent) as processed and creates a COL from ‘Scope8’ (its child) to itself i.e. ‘Scope7’.

Lines 20-32: Now get all the immediate control dependent nodes for that node and add a COL from each of the control dependent nodes to itself only if it does not have any immediate children. If it has children then add COL from each control dependent nodes to each of its children. Recursively call the same function with next node to be processed as the argument. So for the example in Fig 3.7, if the node is ‘Scope7’, then ‘Scope9’ is its immediate control dependent node and since ‘Scope7’ has immediate child ‘Scope8’ a COL is added from ‘Scope9’ to ‘Scope8’ and ‘Scope8’ is passed as the argument to recursively call the same function.

3.3.2 Application of the algorithm on an example

In this section, the algorithms mentioned in section 3.3.1 are applied on an example shown in the Fig 3.7 to determine the COG for that business process. The combined output containing the tree structure along with the COG is shown in the Fig 3.8.

When the algorithm 3.2 mentioned in the section 3.3.1 is applied, a tree structure corresponding to the input business process is created. The algorithm distinguishes between child nodes and control dependent nodes and thus the tree structure contains two different link types, a nested link and a control link. The Fig 3.8 shows the complete tree structure (i.e. the area without the green links and the numbers) for the example process shown in Fig 3.7.

This created tree structure is passed to the algorithm 3.3 listed in the previous section and a corresponding compensation order graph is created. The added compensation order links are shown in green color and the numbers in the Fig 3.8 indicate the sequence in which the COL’s are added by the recursive algorithm. Using this COG, the compensation order for the whole business process can be determined. For example, ‘Scope7’ can only be compensated after ‘Scope8’, which in turn can only be compensated after ‘Scope9’.

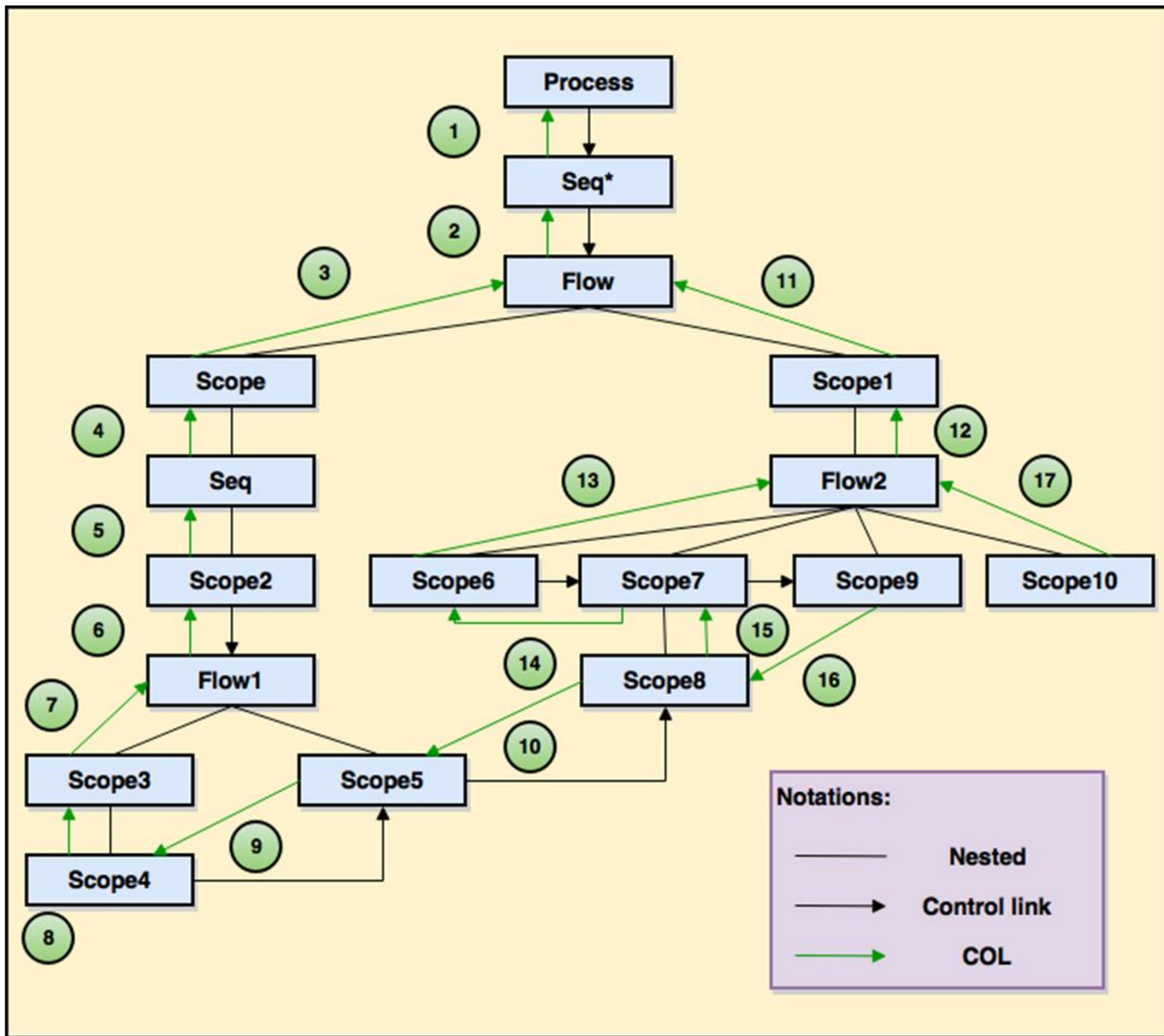

Fig 3.8: The tree structure containing the COG

3.4 Consolidation of Process models that interact via Compensation handlers

In section 3.2 different scenarios were identified where any violation of control flow link might occur when consolidating two processes interacting via CH. And in section 3.3, an algorithm to determine the correct COG has been proposed. In this section, different elemental base case scenarios have been identified that might occur in the merged process model to transform the behaviour of CH in case any cross boundary link violation occurs.

The proposed approach is inspired by the solution for consolidation of process models that interact via fault handlers [7] which has been described briefly in the section 3.1. But the same solution would not fit in case of the compensation handlers since even an outbound control link violates the cross boundary link constraint. Thus, the important points that describe the devised solution have been mentioned below in a brief manner:

- Identify all the compensation handlers that violate the cross boundary link constraint. For each of such CH, the compensation logic associated with it is propagated to the immediately enclosing callers FCT handlers (since a CH can be invoked only from the FCT handlers of its immediately enclosing parent).
- The compensation logic in each of the above identified CH's is replaced by an empty activity since it does nothing and a compensation handler must have an enclosed activity (refer to section 2.2.9).
- The compensation order is of utmost importance and it is derived by using the algorithm described in the section 3.3. This order must be known when multiple compensation logics need to be integrated in immediately enclosing parents FCT handlers. These compensation logics are arranged by using the proper combination of <flow> and <sequence> activities to maintain the derived compensation order.
- The FCT handlers associated with a particular scope have access to the data context of that scope along with the data context of its ancestors. The data context consists of the following four things:
    - Variables
    - Partner links
    - Message exchanges and
    - Correlation sets
- The compensation logic in the CH has access to the data context of that scope which might be used for compensation and thus, the complete data context associated with that scope must also be propagated to the immediately enclosing parent scope i.e. integrated with the parent scopes data context.
- Compensation handlers that do not contain any cross boundary link violations are kept as they are without any changes.

The general solution has been briefly described above and the further sub-sections provide a detailed solution for all the base cases that have been identified based on the way a compensation handler is invoked. These base cases are not mutually exclusive i.e. the proposed solutions could be applied in combination depending on the way a CH has been invoked (from FH or CH or TH or a combination of these). For the sake of simplicity, the data context is not shown in any of the diagrams but as discussed above it is also propagated to the immediately enclosing parent scope. Also, all the derived base cases assume that the scope completes its execution successfully so that the CH associated with that scope is installed. The scenario when scopes do not complete successfully or are unreachable has been handled separately in the section 3.6.

3.4.1 Transformation of <compensateScope> activity in Fault handler

To elaborate the usage of <compensateScope> activity from within a fault handler a simple scenario has been shown wherein, a compensation handler for a scope is called from its immediately enclosing fault handler using a <compensateScope target="S2"> as shown in the Fig 3.9.

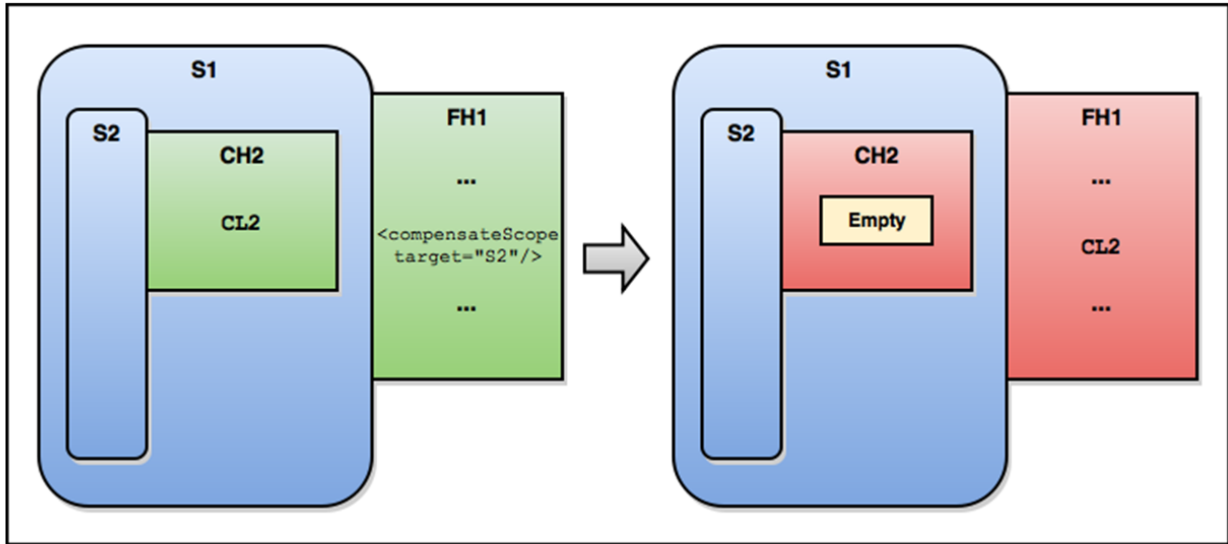

Fig 3.9: Transformation of <compensateScope> activity in FH

Considering that the child scope S2 completes successfully, its corresponding compensation handler is installed, now if a fault occurs in its parent scope S1 then its associated fault handler FH1 is called which in turn calls the compensation handler CH2 of scope S2. Regular flow of activities in the fault handler is mentioned below:

1. All the statements/activities before the <compensateScope> activity are executed.
2. The statement <compensateScope target="S2"> makes the BPEL engine to invoke the compensation handler CH2 associated with scope S2.
3. The compensation logic CL2 within the compensation handler is executed and the flow returns back to the fault handler.
4. The activities/statements after the <compensateScope> activity are executed.

To emulate this behavior the process is transformed to the process as shown in the right side of Fig 3.9 and the following changes are made:

1. Replace the <compensateScope target="S2"> in the fault handler with the actual compensation logic CL2 from the compensation handler CH2 of scope S2.
2. Replace the compensation logic CL2 in the compensation handler CH2 of scope S2 with an <empty> activity.

After the above mentioned transformation the flow of the activities in case the fault handler is called takes place in the following order:

1. All the activities/statements before the compensation logic CL2 are executed.
2. The compensation logic CL2 is executed.
3. All the activities/statements after the compensation logic CL2 are executed.

As seen from the flows before and after transformation, the major difference is that the invocation of the compensation handler is removed while maintaining the control flow between the basic activities in the process. Thus the behaviour of the compensation handler is emulated for this scenario with the prerequisite that the scope S1 completes successfully.

3.4.2 Transformation of <compensateScope> activity in Termination handler

The scenario mentioned in section 3.4.1 is slightly varied i.e. in this case the compensation handler is being invoked from a termination handler instead of a fault handler by using the same activity <compensateScope> as shown in the Fig 3.10.

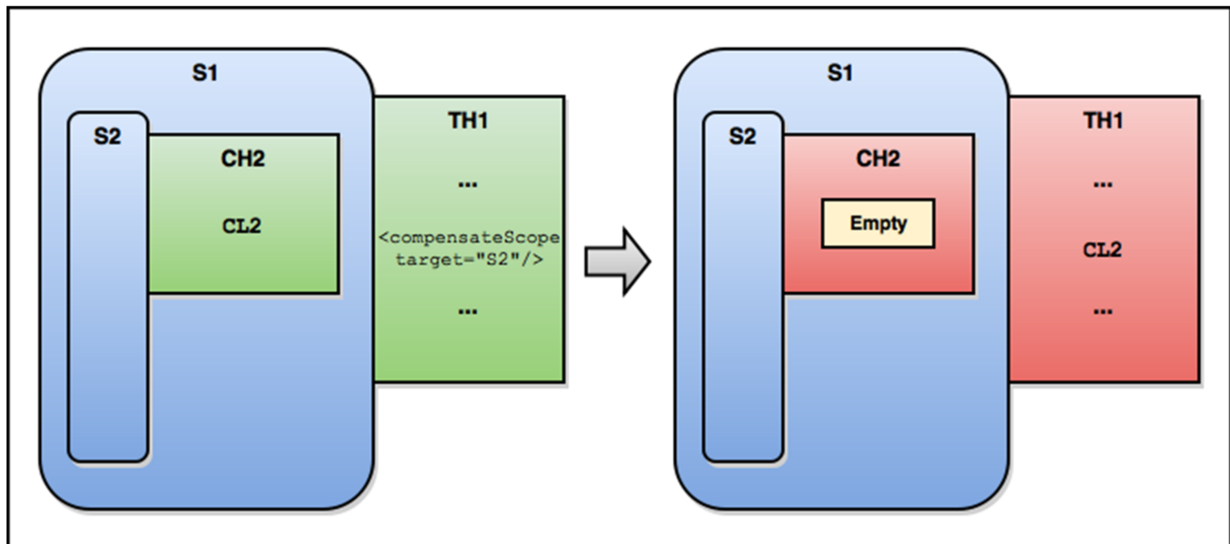

Fig 3.10: Transformation of <compensateScope> activity in TH

The flow in case of termination handler while calling the nested compensation handler and even after transformation is similar to the flow for fault handler described in section 3.4.1, so only the transformation steps are briefly described below:

1. Replace the <compensateScope target=”S2”> in the termination handler with the actual compensation logic CL2 from the compensation handler CH2 of scope S2.

2. Replace the compensation logic CL2 in the compensation handler CH2 of scope S2 with an <empty> activity.

In this way the behaviour of the compensation handler being called from a termination handler is emulated.

3.4.3 Transformation of <compensateScope> activity in Compensation handler

Similar to the scenario in section 3.4.1 and 3.4.2, <compensateScope> activity can also be called from within a compensation handler and the transformation process is also the same. So the behaviour of the nested compensation handler can be emulated as shown in the Fig 3.11.

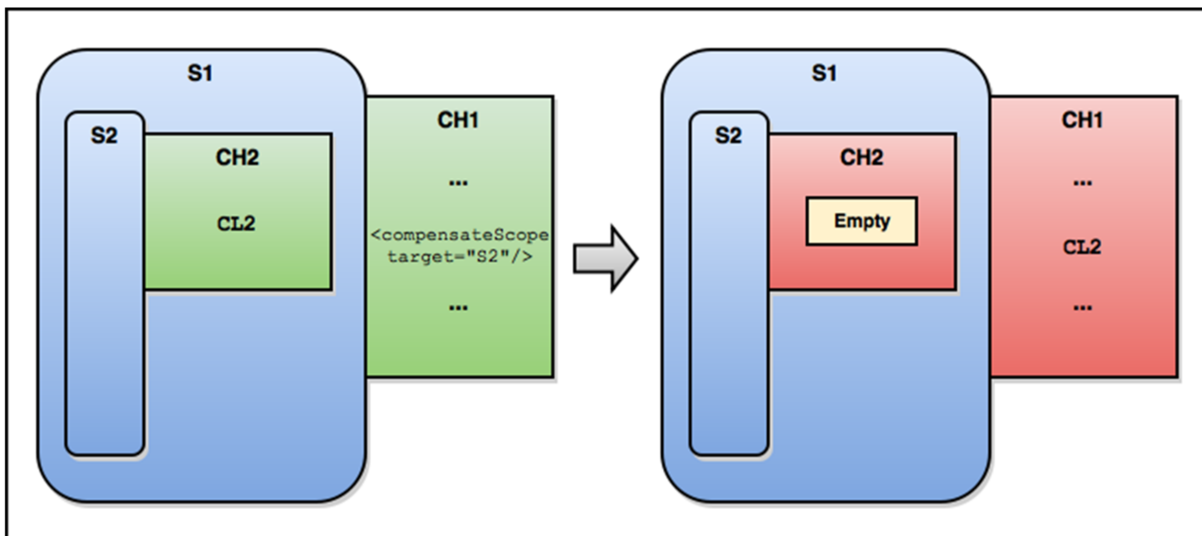

Fig 3.11: Transformation of <compensateScope> activity in CH

3.4.4 Transformation of <compensate> activity in Fault handler

The only other option that can be used to call a compensation handler is a <compensate> activity and it is used mostly in case of default FCT handlers but it can also be used explicitly by the user. A scenario is described when <compensate> activity is used from within a fault handler to call the compensation handlers associated with all the directly nested scopes.

As described in section 3.4.1, the regular flow of activities in case of fault handlers is elaborated and then the flow in case of the transformed scenario is explained with the help of the Fig 3.12.

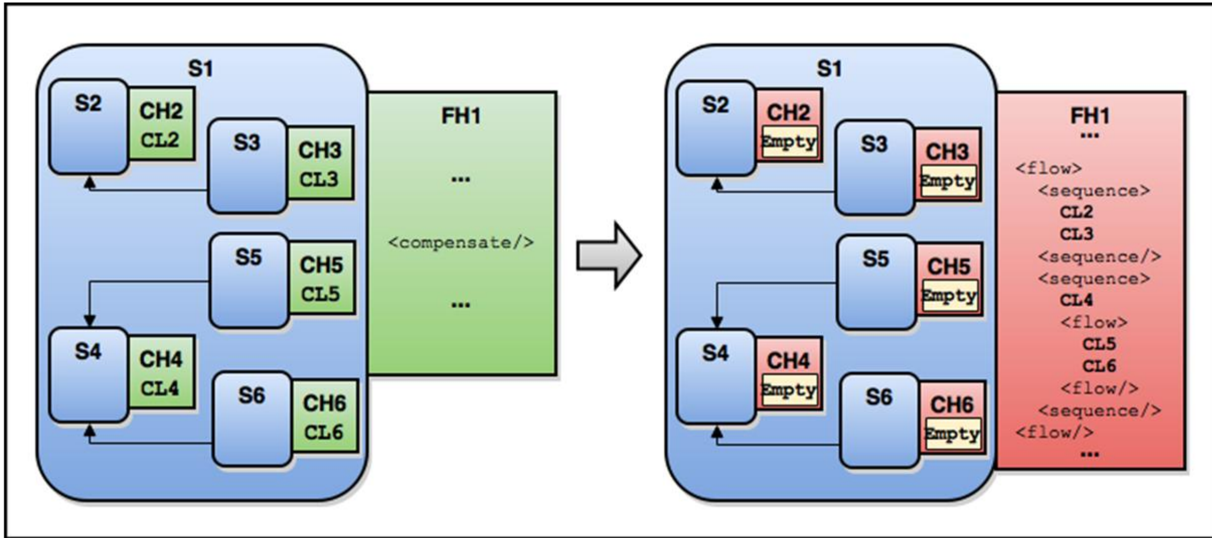

Fig 3.12: Transformation of <compensate> activity in FH

As seen in the Fig 3.12, scope S1 has a fault handler FH1 associated with it containing the activity <compensate> and also scope S1 has five directly nested scopes which have control dependencies among each other. Mentioned below is the regular flow without transformation assuming all the nested scopes completed successfully and thus their corresponding compensation handlers are installed and a fault occurs in scope S1 and thus the fault handler FH1 is invoked:

1. All the activities/statements before the <compensate> activity are executed.
2. The <compensate> activity is invoked and thus the BPEL engine must first determine the compensation order since there are many nested scopes. It does so by using the two compensation rules already discussed in the section 2.2.9 of Chapter 2.
3. The compensation handlers associated with the nested scopes are called as per the compensation order determined in step 2 and the compensation logic within each of them is executed.
4. The execution returns back to the fault handler.
5. All the activities/statements after the <compensate> activity are executed.

Now to get rid of the compensation handlers of all the nested scopes we transform the process as mentioned below:

1. Pass the scope S1 to the algorithm described in the section 3.3.1 to determine the compensation order graph for scope S1 and all of its nested scopes.
2. Using the COG obtained from step 1, the exact order in which the compensation handlers must be invoked is known. The compensation logic of CH's that can be executed in parallel are added within <flow> activities and when an order has to be imposed then within <sequence> activities. The arranged compensation logic associated with each nested compensation handler of scope S1 is shown in the right side of Fig 3.12.
3. Replace the <compensate> activity in the fault handler with the flow generated in step 2.

4. Replace the compensation logic of all the nested compensation handlers with <empty> activities.

After following the above transformation, the transformed process is obtained as shown in the right side of the Fig 3.12. The flow of the transformed process is briefly described below:

1. All the activities/statements before the <flow> are executed.
2. The <flow> contains the correct determined order of execution of the compensation logic of the nested scopes and is thus executed.
3. All the activities/statements after the <flow/> are executed.

Thus as seen from the flow, it can be concluded that the behaviour of the nested compensation handlers is emulated correctly by getting rid of them and transforming the compensation logic to the fault handler.

3.4.5 Transformation of <compensate> activity in Termination handler

The scenario mentioned in section 3.4.4 is tweaked with a minor change i.e. in this case the compensation handler is being invoked from a termination handler instead of a fault handler by using the same activity <compensate> as shown in the Fig 3.13.

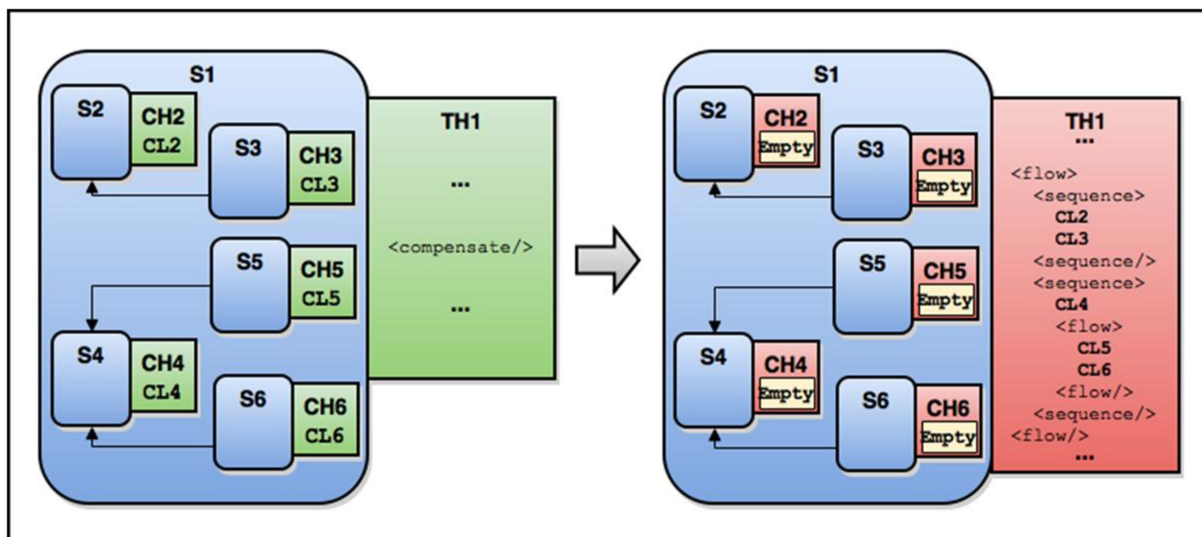

Fig 3.13: Transformation of <compensate> activity in TH

The regular flow in case of termination handler while calling the nested compensation handlers and even after transformation is similar to the flow for fault handler described in section 3.4.4, so only the transformation steps are briefly described below,

1. Pass the scope S to the algorithm described in the section 3.3.2 to determine the Compensation order graph for scope S and all of its nested scopes.
2. Using the COG obtained from step 1, the exact order in which the compensation handlers must be invoked is known. The compensation logic of CH's that can be executed in parallel are added within <flow> activities and when an order has to be imposed then within <sequence> activities. The arranged compensation logic associated with each nested compensation handler of scope S1 is shown in the right side of Fig 3.13.
3. Replace the <compensate> activity in the termination handler with the flow generated in step 2.
4. Replace the compensation logic of all the nested compensation handlers with <empty> activities.

In this way, the behaviour of the nested compensation handlers is emulated by transforming the compensation logic to the termination handler.

3.4.6 Transformation of <compensate> activity in Compensation handler

Similar to the scenario in section 3.4.4 and 3.4.5, <compensate> activity can also be called from within a compensation handler and the transformation process is also the same. So the behaviour of the nested compensation handlers can be emulated as shown in the Fig 3.14.

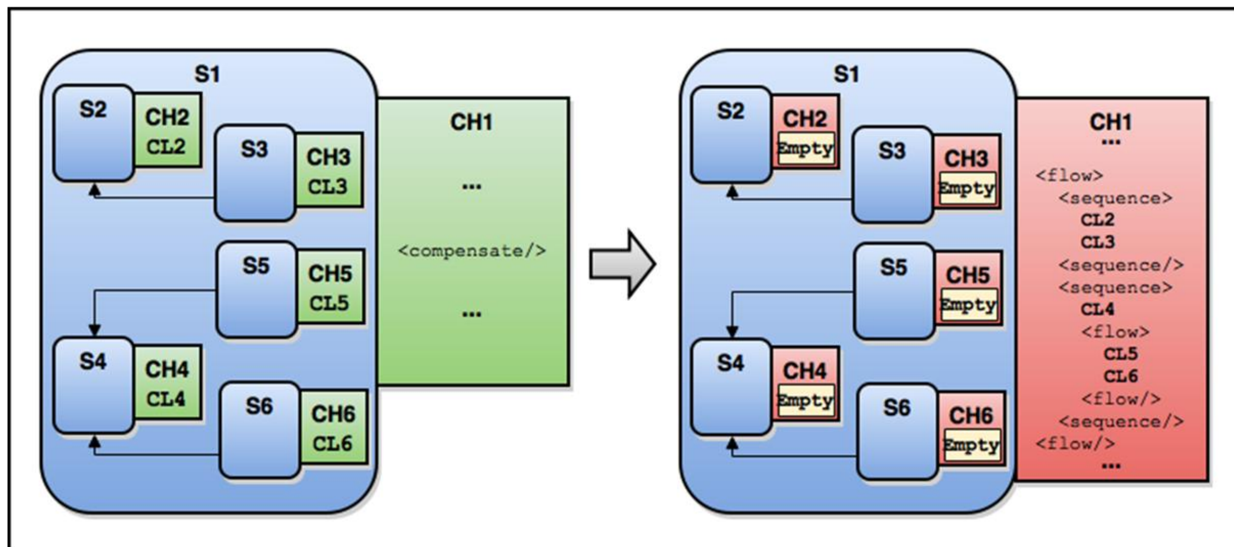

Fig 3.14: Transformation of <compensate> activity in CH.

3.5 Transformation in case of Nested Scopes

In the previous section, different base cases related to the way CH's can be called were discussed along with a solution for each case. But all those cases were elemental in the sense considering the complexity of the scenarios. The transformations were trivial in nature since they only consider the relationship between parent and directly nested child scopes. In practice a process consists of scopes that are nested arbitrarily deep. Thus in most of cases, the six elemental cases described in section 3.4 are not encountered but they definitely act as the fundamental elements in the transformation of the whole process.

To transform a real world process, think of it as tree, the root of the tree being the process itself and its immediately nested scopes as its immediate child nodes. Each of these scopes can have further nested scopes which in the tree are represented by further child nodes at the next level. Thus the deepest nested scopes in the process are represented by leaf nodes in such a tree structure. The elemental transformation is applied to these leaf nodes i.e. to the deepest nested scopes in a recursive manner climbing up the tree thus finishing the transformation of the whole tree i.e. the process itself. An example scenario for such a transformation is shown in the Fig 3.15.

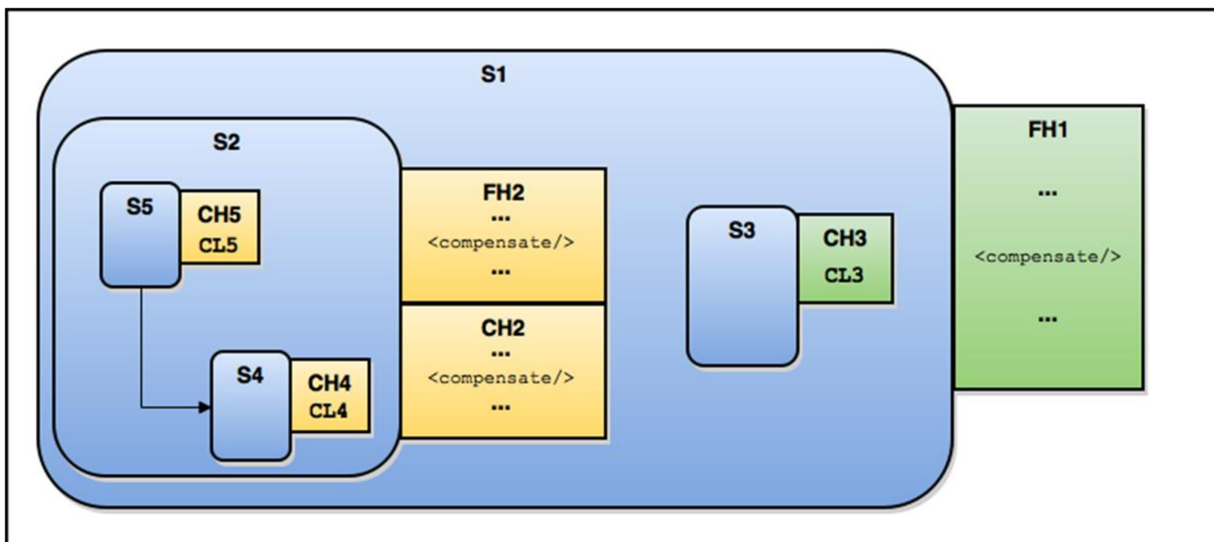

Fig 3.15: Example process containing nested scopes

As shown in the Fig 3.15, the scope S1 can be considered as the root here and it has two immediately nested scopes S2 and S3. The scope S2 has further nested scopes S4 and S5. This is the deepest level for this example. These scopes S4 and S5 have associated CH's containing compensation logic CL4 and CL5 respectively. Since the CH associated with any scope can only be invoked by its immediately enclosing parents FCT handlers the area impacted during the transformation is colored in yellow as shown in the Fig 3.15. The elemental transformations that apply in this scenario fall under the base cases mentioned in section 3.4.4 and 3.4.6. After transformation the impacted areas are shown in red in the Fig 3.16.

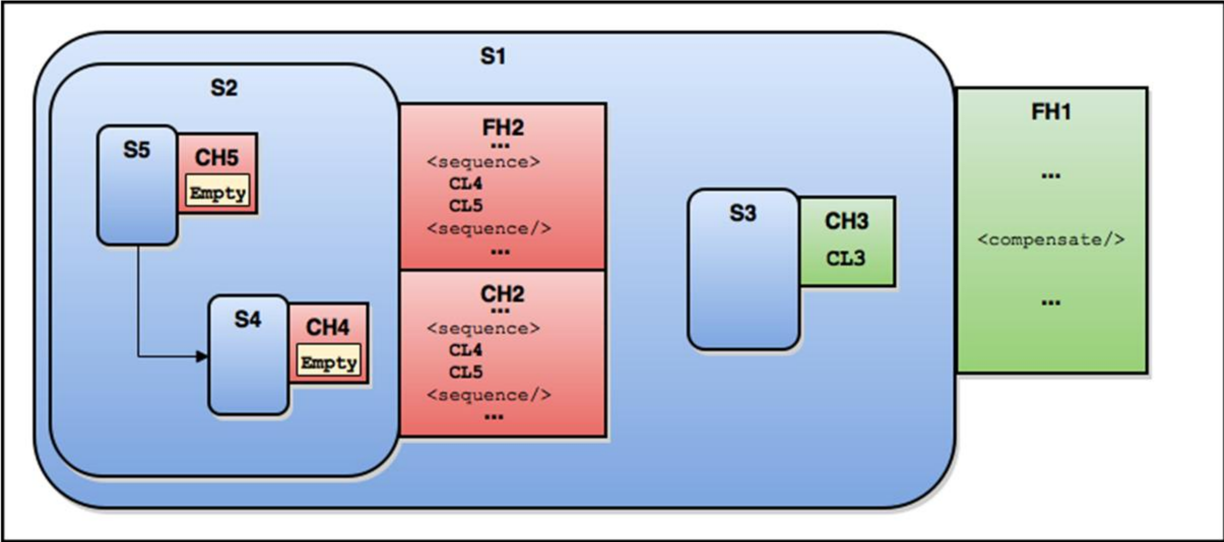

Fig 3.16: Process after applying first transformation

Now the elemental transformations are applied recursively one level up till we reach the parent process. Now the scopes under consideration are S2 and S3 and the areas into consideration are colored in yellow in the Fig 3.17.

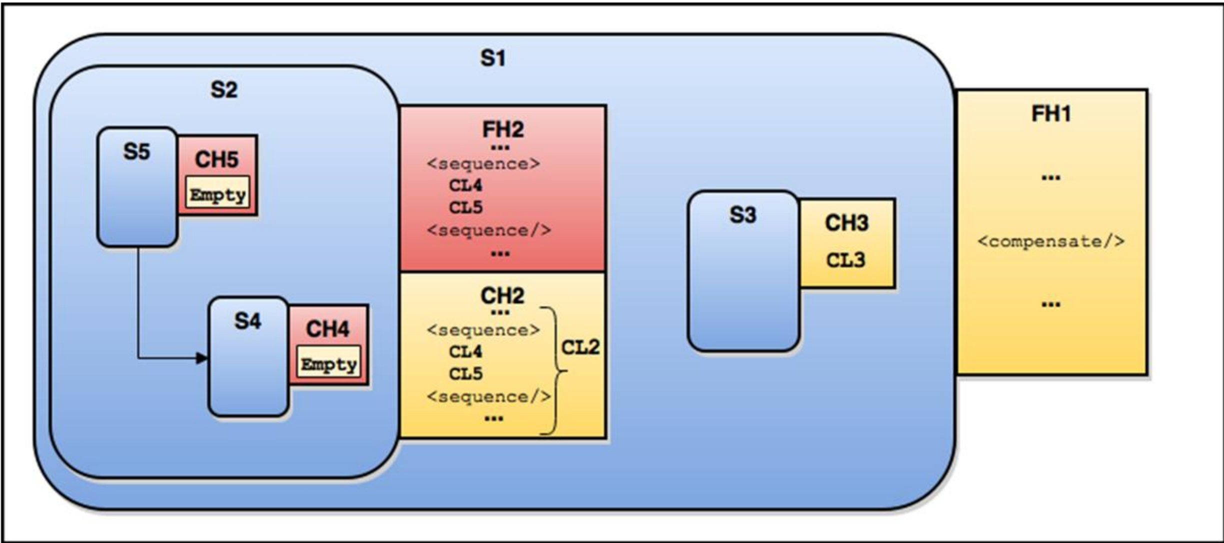

Fig 3.17: Area to be considered while applying second transformation

For the scenario in the Fig 3.17, the elemental transformations that can be applied fall under the base case described in the section 3.4.6. After the transformation is applied the changes are shown in red in the Fig 3.18.

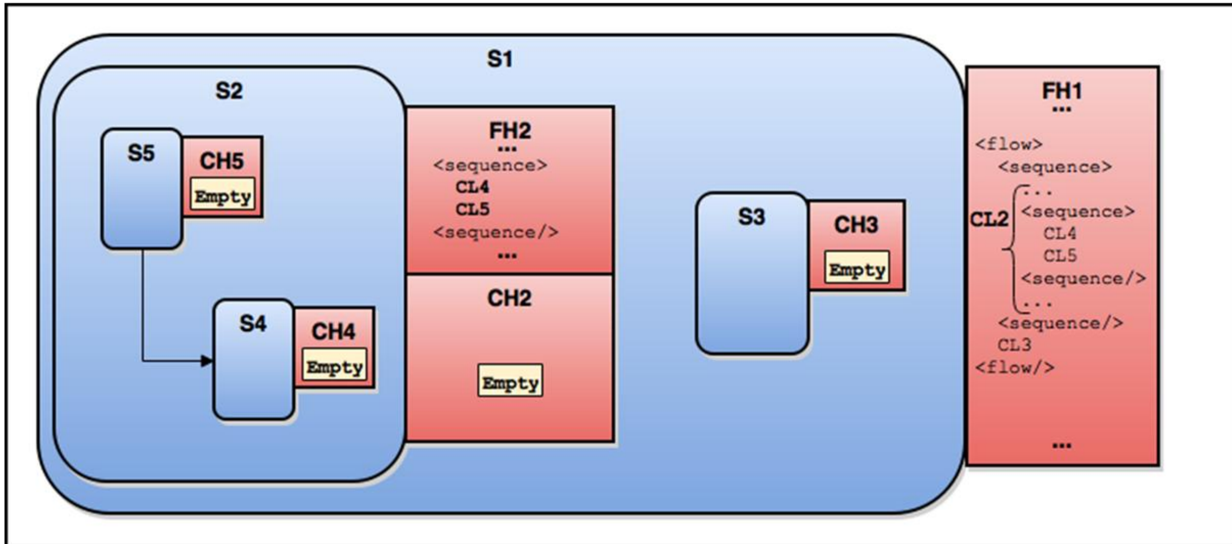

Fig 3.18: The final transformed process

Thus the behaviour of the CH's is emulated in case of nested scopes by applying the elemental transformation recursively starting at the deepest level up to the root level i.e. the whole process.

3.6 Transformation when Scopes do no complete successfully

During regular execution, when a scope does not complete successfully, then the execution flow goes to the FH or TH associated with that scope, and its CH is never installed. It indicates that the CH of a scope which does not complete successfully is not reachable and hence it is never invoked. To emulate this behaviour, once the compensation logic within the CH associated with a scope is propagated to the immediately enclosing parents FCT handlers, a check must be added to decide if that compensation logic should be executed and that check indicates if the scope with which the compensation logic is associated, completed successfully.

For this purpose, a new variable '*hasScopeCompletedSuccessfully*' is created and is kept in the process scope itself thus making it accessible anywhere within the process. If by default this value is set to 'false' and is changed to 'true' only at the end of the scope, then there might arise a situation when the execution of the scope logic completes but before assigning 'true' to this variable some fault occurs in the parent scope and thus this assignment operation is not executed. This leads to a false scenario wherein the variable associated with that scope remains 'false' whereas the scope completed successfully and must be compensated. Thus by default, it is assumed that each and every scope will complete its execution successfully and hence this variable is initialized with a value 'true'. There is one variable for each scope thus segregating all the scopes within the process. As discussed in the above section, when a fault occurs or if for some reason a scope terminates without completing, then the

execution is transferred to the FH/TH associated with that scope. This is the place where the variable associated with that scope will be set to 'false'.

Now that a variable has been added indicating whether a scope completed successfully or not, the compensation logic associated with any scope is only executed if the value of the variable associated with that scope is 'true'. This way, the scenario when a scope does not complete successfully is emulated by adding a check before executing any compensation logic.

BPEL also has a special feature of 'Dead Path Elimination' (DPE) (refer to section 2.2.7) which is activated when the property 'suppressJoinFault=yes' is set. The behavior when this property is set to true and a scope fails, then the Boolean value associated with all the outgoing links from that scope is set to 'false'. Whether a scope will be activated is determined by the join condition of all the incoming links and hence it is statically not possible to determine if a scope will be activated in the lifetime of the process. By default the join condition for all the incoming links is 'OR' [12].

To cope up with such situation, a new variable is introduced indicating if a scope was ever reached, '*isScopeReached*' and the assumption here is none of the scopes in the process are ever reached and thus this variable (one variable per scope) associated with each scope is initialized to 'false'. As soon as a scope is activated or reached, before starting the activity associated with that scope this variable associated with that scope is set to 'true' thus indicating that the particular scope was reached. The final condition which is added before executing any compensation logic is that both these variables '*hasScopeCompletedSuccessfully*' and '*isScopeReached*' must be set to 'true'. These two variables make it possible to emulate the behavior when a scope does not complete successfully or is never activated.

To understand it better consider a sample process shown in the Fig 3.19.

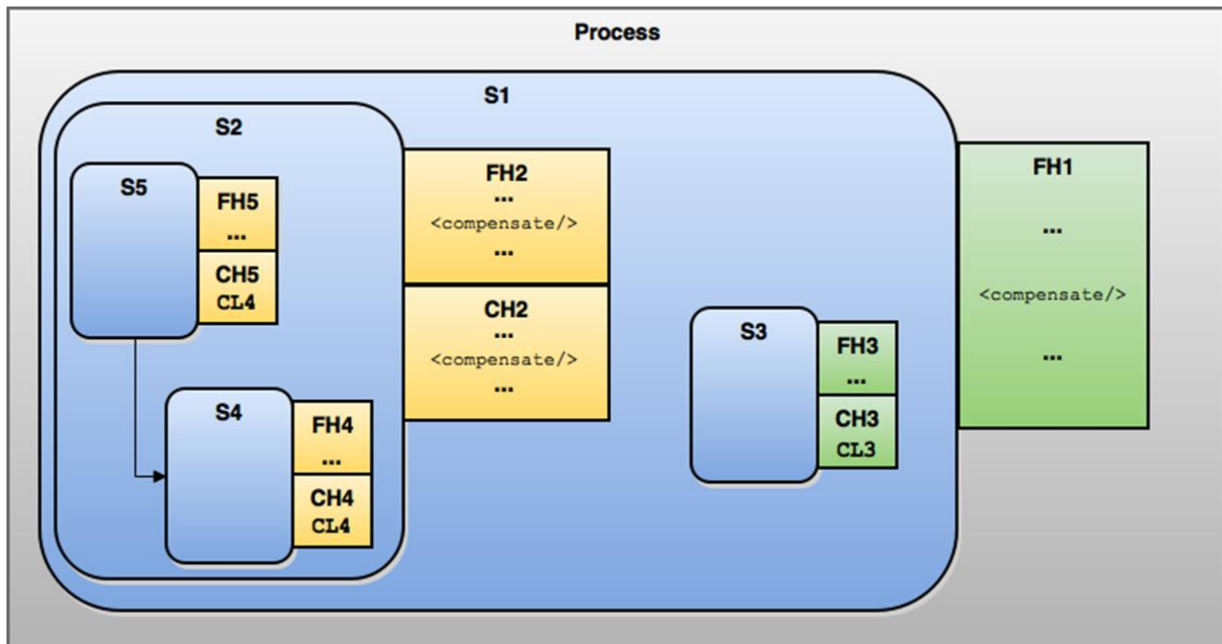

Fig 3.19: Example process when scopes do not complete successfully

For the sake of simplicity, the TH's associated with the scopes are not shown in the figure. This example shows scopes nested within each other as well as having control dependencies. The transformations that are applied have already been discussed in the section 3.5 and thus the transformed process along with the variables described above is shown in the Fig 3.20.

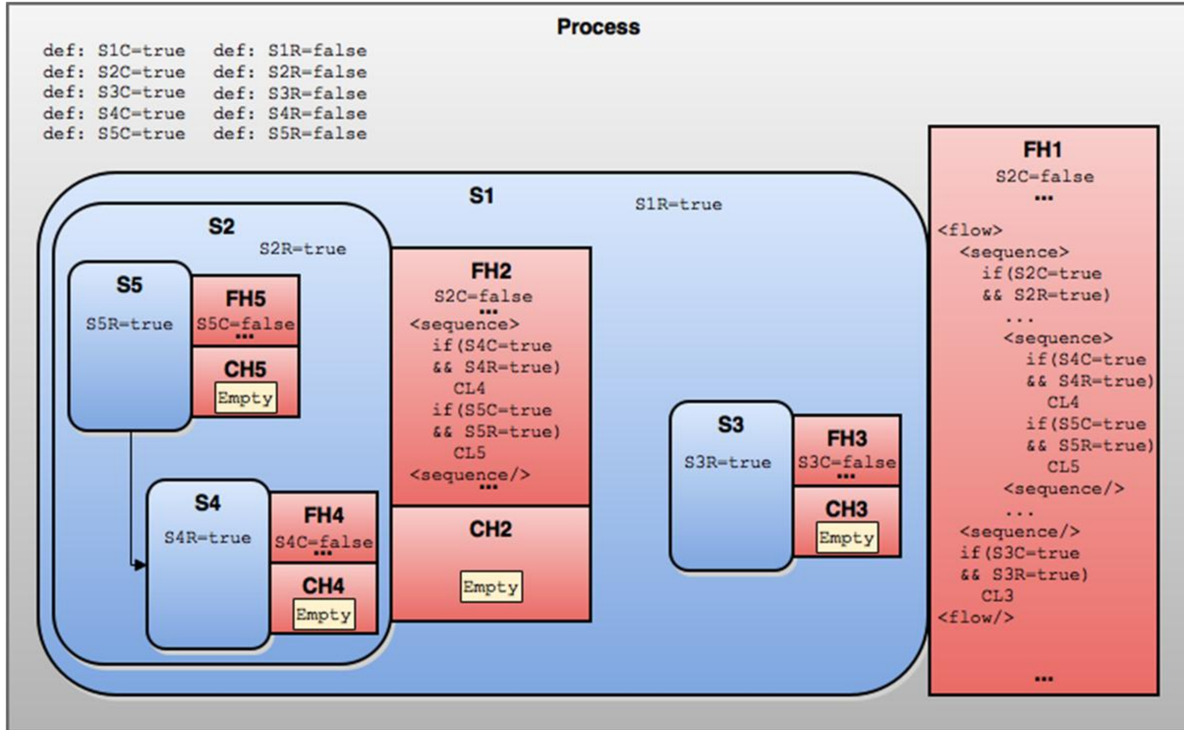

Fig 3.20: Transformed process when scopes do not complete successfully

In the Fig 3.20, the declarations of the variables associated with the scopes have been shown. The discussed variable '*hasScopeCompletedSuccessfully*' is represented by the scope name followed by 'C' and has been initialized with value 'true' (For example, S2C implies hasScope2CompletedSuccessfully). Similarly the variable '*isScopeReached*' is represented by the scope name followed by 'R' and has been initialized with value 'false' (For example, S4R implies isScope3Reached).

As discussed, all the variables associated with the completion of a scope are set to 'false' in the FH's associated with that scope. Also, all the variables that check whether a scope was reached have been set to 'true' as soon as the flow enters the scope. Before executing compensation logic associated with any scope, a condition has been added to verify if both these variables are set to 'true' (indicating the scope was reachable and it completed successfully) and only then the compensation logic is executed.

3.7 Transformation when Scope is within a Repeatable Construct

For all the cases considered in the above section, it was assumed that the scopes are not within a repeatable construct. In this section, a possible conceptual solution is proposed for the case when a CH associated with a scope within a repeatable construct violates the cross boundary link constraint.

In regular flow, when a scope is within a repeatable construct and it needs to be compensated, for each successful iteration of the loop the BPEL engine creates a scope instance (also known as scope snapshot)and along with that it also creates CH instance for that particular scope instance. This CH instances are known as compensation handler instance group and the BPEL engine uses this CH instance group to compensate the scope in the default compensation order. Repeatable construct could be loop or an event handler and are further categorized as non-parallel loops (containing <while>, <repeatUntil> and non-parallel <forEach>) and parallel loops (containing parallel <forEach> and event handlers). In case of non-parallel loops the invocation of the installed CH instances in successive iterations must be in reverse order whereas for parallel loops no ordering is imposed for the compensation of associated scope [12].

To emulate such behavior and to impose the reverse order of execution for such CH instances the best suitable data structure is a stack. The entries in this stack could be the scope instances containing only the data context. The proposed solution is briefly described below,

- Create one stack associated with one scope within a repeatable construct in its immediately enclosing parent scope. This stack stores the scope instances.
- For each successful iteration of the scope, take the snapshot of its data context (this can be emulated by copying the data context of this scope in a new scope with its primary activity as an empty activity) and push it (i.e. the newly created scope) on the stack associated with that scope.
- The compensation logic associated with this scope will be propagated to the FCT handlers of the immediately enclosing parent scope and thus it will have access to this stack.
- Execute the compensation logic in a loop on each element (i.e. the scope containing the data context) of the stack, till the stack is empty thus emulating the behaviour similar to the regular execution.

To get a better idea of the proposed approach, a sample transformed process is shown in the Fig 3.21. As seen the scope S2 is within a repeatable construct i.e. loop and assuming the compensation handler CH2 associated with it contains a control link violation the transformation is carried out. A stack representing scope S2 is declared in its parent scope i.e. in scope S1. After each successful iteration of scope S2, a snapshot of the data context is taken by creating a new scope containing this data context of scope S2 and an empty activity as its primary activity. This newly created scope is pushed on the declared stack S2_stack.

The propagated compensation logic CL2 in the fault handler FH1 has access to this declared stack variable S2_stack. This compensation logic is executed in a loop while popping out the elements in the stack one at a time. Thus, the compensation logic is executed for all the scope instances pushed on the S2_stack in the reverse order of execution. This loop runs till the stack is empty thus emulating the behaviour of a compensation handler instance group.

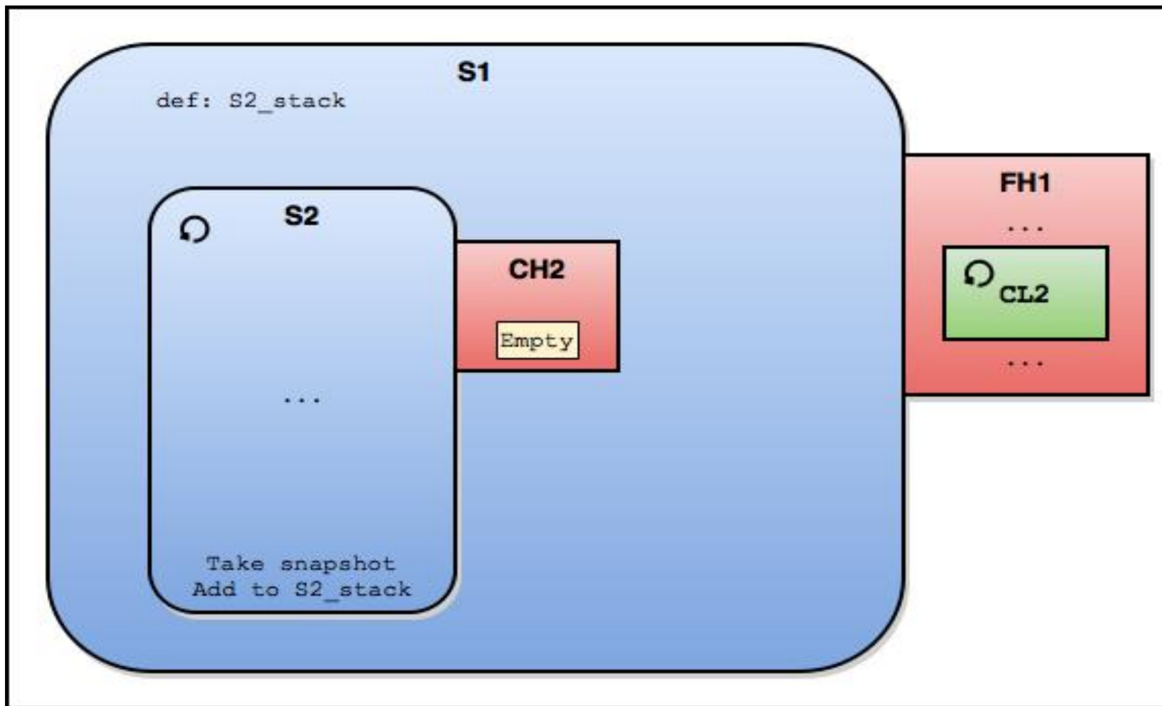

Fig 3.21: Transformed process when scopes do not complete successfully

3.8 Summary of Cases addressed

The main focus of this study is to identify and correct any control link violations that occur during the consolidation of the process models that interact via compensation handlers. To describe the solution briefly, the compensation logic of the CH's is propagated to the FCT handlers of the immediately enclosing parent scope.

There are two types of compensation handlers, explicit and implicit and the way both these scenarios are covered in this study are described below,

1. Explicit compensation handler: The way to invoke an explicitly defined CH is by using the <compensateScope> and <compensate> activities and these in turn can be called from the <catch>, <catchAll>, <compensationHandler> and <terminationHandler> constructs. Of these, the <catch> and <catchAll> can only be used in the FH. Considering these ways the following permutation and combinations occur,

a. <compensateScope> in FH – solution covered in section 3.4.1
b. <compensateScope> in TH – solution covered in section 3.4.2
c. <compensateScope> in CH – solution covered in section 3.4.3
d. <compensate> in FH – solution covered in section 3.4.4
e. <compensate> in TH – solution covered in section 3.4.5
f. <compensate> in CH – solution covered in section 3.4.6

Also, the nested scope scenario is covered in the section 3.5, wherein the transformation is carried out from the deepest level, one level at a time reaching to the parent using the above mentioned elemental transformations.

2. Implicit compensation handler: Implicit CH is nothing but a default CH. According to the BPEL specifications, the way it determines the compensation order is “reverse order of execution”. When no CH is provided for a scope, the BPEL engine associates it with a default CH. The assumption is that it does not violate any control link violations imposed by the BPEL specifications since it is defined by the BPEL engine itself.

Thus, each scenario has been taken into consideration and solution is provided for all the cases where control link violations occur. Hence, we can safely argue that this approach is able to get rid of CH’s for all type of scenarios.

4 Implementation

This chapter provides the details of the implementation of the proposed solution discussed in the previous chapter. The implementation is based on the existing state of the system described in section 2.5. As a development environment, the IDE Eclipse Kepler Service Release One (SR1) along with Java 7 was used. Initially, the BPEL engine Apache ODE v.1.3.6 (Orchestration Director Engine) was used to develop sample business processes.

The implementation details and the flow of the existing system are briefly described as shown in the Fig 4.1.

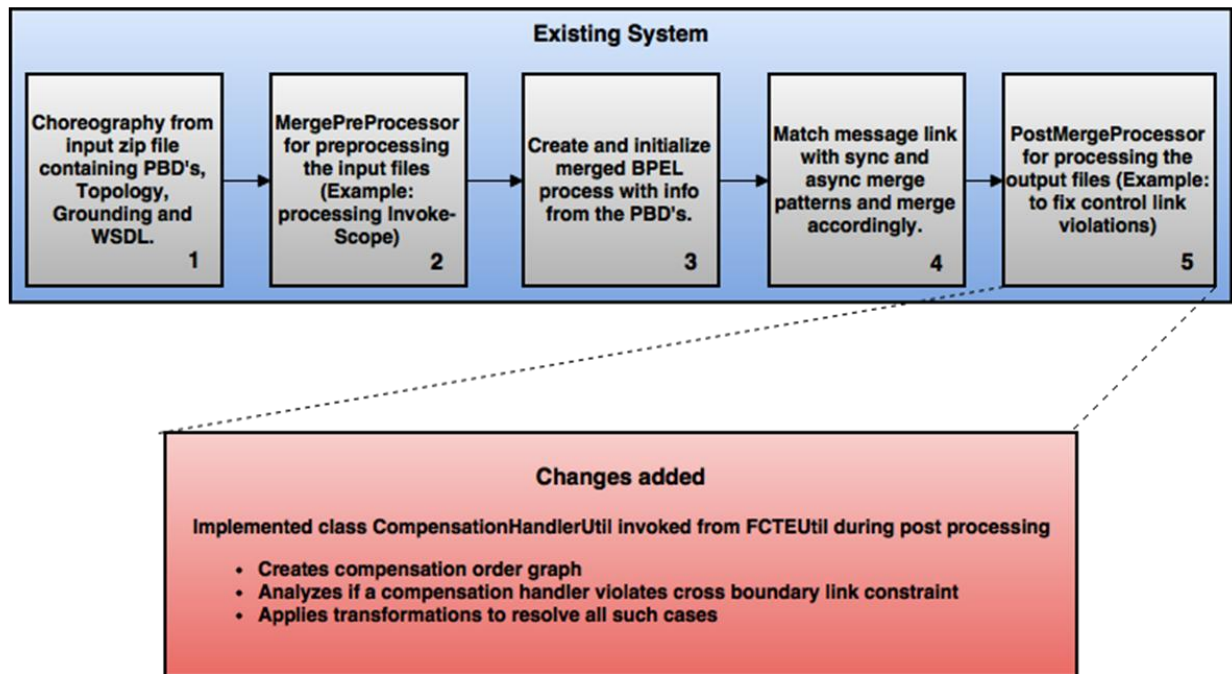

Fig 4.1: Consolidation process for choreography

As seen in Fig 4.1, the implementation of the consolidation process (refer chapter 1) consists of five steps. First the choreography files i.e. the PBD's, WSDL, topology and grounding are read from the provided input zip file. In the second step, the class MergePreProcessor is invoked wherein any preprocessing related to the input files can be done. In the current implementation [7], the invoke activities are transformed to have a surrounding enclosing scope activity. Then, a merged process P_{merged} is created containing the activities from the processes in the input PBD's each enclosed within a separate scope. In the fourth step, the consolidation is done using the predefined consolidation patterns. After consolidation the last step deals with all the post processing that needs to be done and for this purpose the class MergePostProcessor is used. The current implementation [7] deals with fixing all the control link violations related to the fault handler, termination handler and event handler.

4.1 Relationship between various Components

The current implementation has a lot of projects which are used in conjuncture to obtain the final consolidated process. The relationship between these various projects is shown in the component diagram in the Fig 4.2.

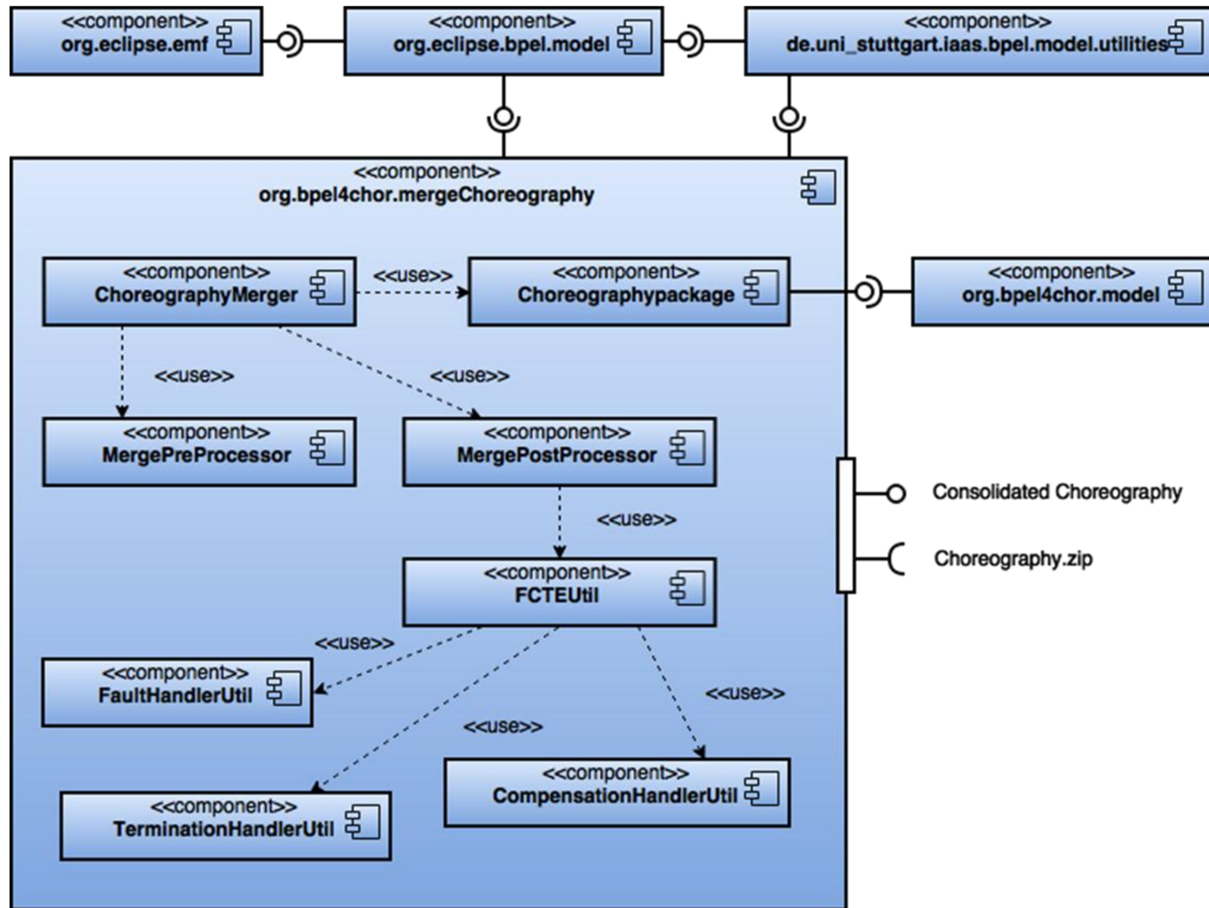

Fig 4.2: Component diagram for choreography consolidation [7]

The project org.bpel4chor.mergeChoreography is the most important project which contains the main class ChoreographyMerger which drives the complete consolidation process. It uses the classes MergePreProcessor, MergerPostProcessor and ChoreographyPackage to obtain the final merged process. The class ChoreographyPackage is responsible for reading the input BPEL4Chor files, initializing the merged process P_{merged} and storing it. With the help of the project org.bpel4chor.model which was developed by [19], the BPEL4Chor files are stored in BPEL4Chor objects which are used to read the input files. There are a lot of utility classes containing useful reusable functionalities (for example to zip and unzip files) which are bundled together in the project de.uni_stuttgart.iaas.bpel.model.utilities. These classes are used by the ChoreographyPackage to traverse through the BPEL and BPEL4Chor objects

during the consolidation process. In addition to that, the eclipse EMF projects org.eclipse.bpel.model and org.eclipse.bpel.common.model are used for the representation and the processing of the BPEL objects. Another component diagram related to the testing of the merged process (refer to section 4.3) is shown in the Fig 4.3.

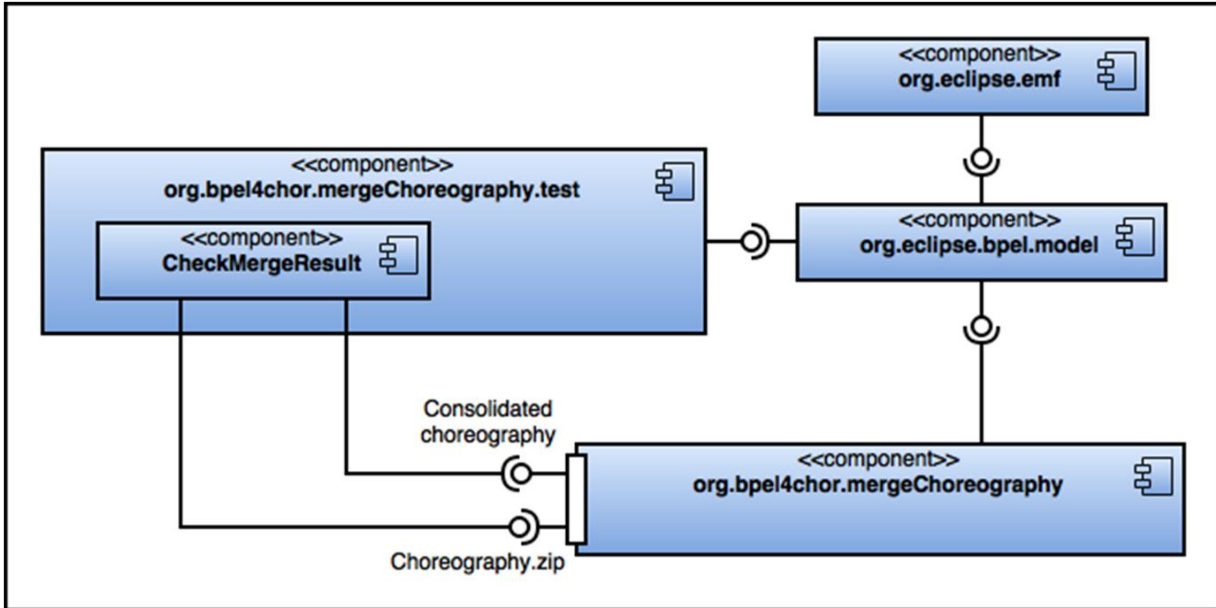

Fig 4.3: Component diagram for testing the consolidated process [7]

As seen, the project org.bpel4chor.mergeChoreography.test contains the component CheckMergeResult which is used to validate the merged process against the expected output process. To obtain the merged process it uses the interfaces provided by the org.bpel4chor.mergeChreography project.

4.2 Consolidation Flow

The sequence diagram showing the consolidation process in the current implementation is shown in the Fig 4.4. The method merge(fileName) from the class ChoreographyMerger is invoked which acts as the main method to drive the overall consolidation flow. First the input choreography files are read using the readInZip(fileName) method of the class ChoreographyPackage. This is the part of the step one shown in the Fig 4.1. Next, the method startPreProcessing(choreographyPackage) from the class MergePreProcessor is invoked where in all the preprocessing needed prior to the actual consolidation logic is carried out. This is the second step shown in Fig 4.1. The third and the fourth step shown in the Fig 4.1 are executed by calling the method mergeChoreography() of the class ChoreographyMerger. This method in turn invokes the initMergedProcess() method of the class ChoreographyPackage which is responsible for creating and initializing the merged process P_{merged}. Then an instance of the class CommunicationMatcher is created which is used to loop over all the message links to find a matching merge pattern for that message link and then the corresponding merge takes place. Once the loop

execution is complete the fifth step shown in the Fig 4.1 starts. The method startPostProcessing(choreographyPackage) of the class MergePostProcessor is invoked which is responsible for dealing with the control link violations in the merged process P_{merged} associated with the FCTE handlers. This is shown in more details with the help of Fig 4.5. In the end, before the merge(fileName) method finishes its execution, the method saveMergedChoreography(fileName) from the class ChoreographyPackage is invoked which contains the logic to save the consolidated process P_{merged} along with its associated WSDL files to a configurable output location.

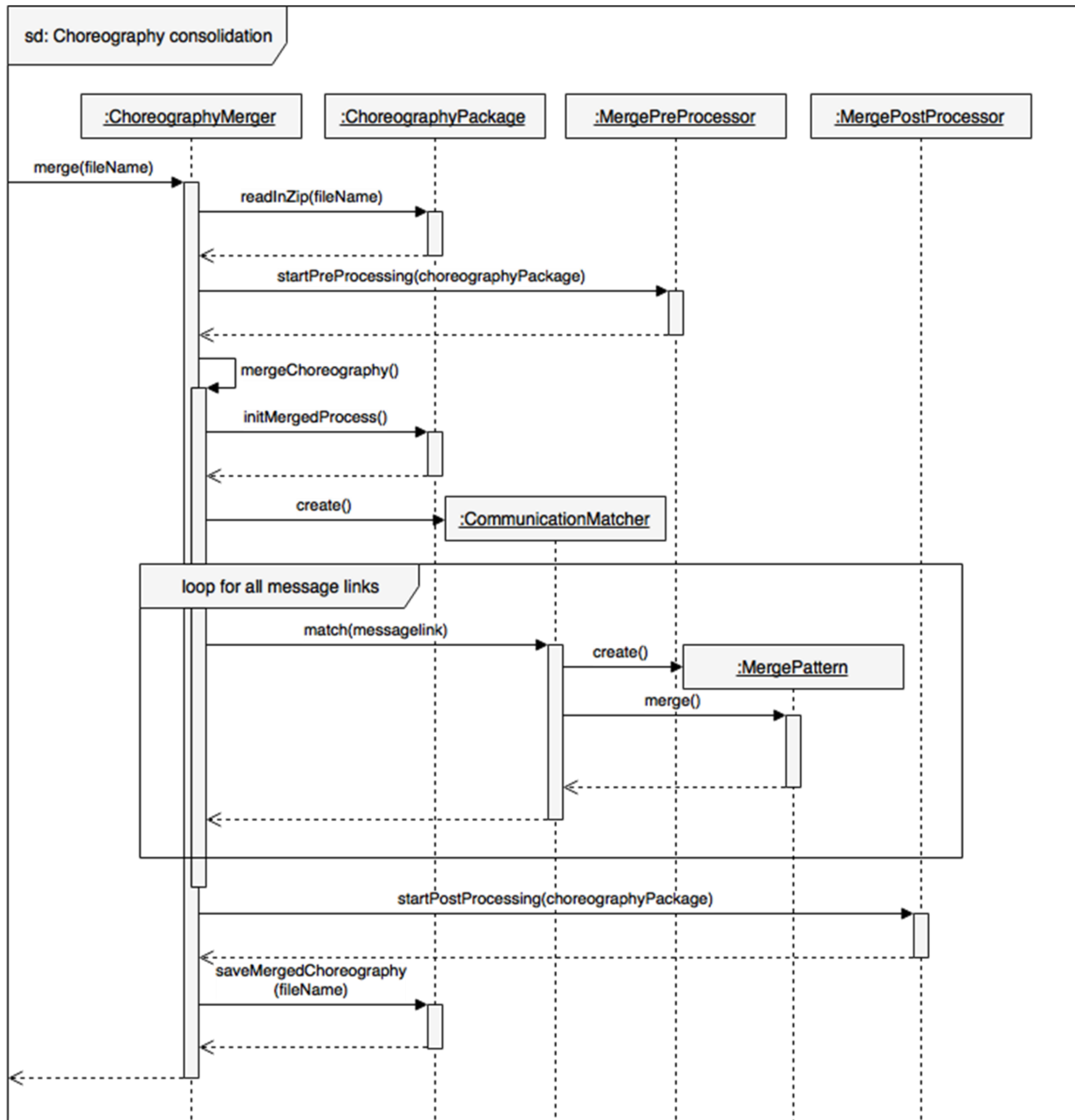

Fig 4.4: Sequence diagram for choreography consolidation [7]

The sequence diagram in the Fig 4.5 shows the post processing flow after the actual consolidation process finishes. The class FCTEUtil contains utility methods required for the processing of the FCTE handlers. Some of the existing utility methods were modified and new methods were added to accommodate the changes corresponding to the compensation handler (for example to check if an activity has any incoming link). The class CompensationHandlerUtil was modified to contain the actual logic to deal with the compensation handlers containing any control link violations. The algorithms described in section 3.3.1 to determine the compensation order graph have been implemented in this class. New class to define the node structure of the tree has also been created for the same purpose. Also, the transformations needed to fix the control link violations associated with compensation handlers (refer to sections 3.4, 3.5 and 3.6) are carried out in this class. The classes FaultHandlerUtil and TerminationHandlerUtil are responsible for fixing the control link violations associated with fault and termination handlers respectively [7]. When the execution invokes the method startPostProcessing(choreographyPackage) of the class MergePostProcessor, then it in turn invokes the methods processCompensationHandler(mergedProcess) and processScopesFT(mergedProcess) of the class FCTEUtil. The method processCompensationHandler(mergedProcess) creates an instance of the class CompensationHandlerUtil and then invokes the method processCompensation(activity) for all the scopes in the merged process P_{merged} (this method is modified to implement the algorithms described in the section 3.3.1). Similarly, the method processScopesFT(mergedProcess) creates instances of the classes FaultHandlerUtil and TerminationHandlerUtil and invokes the corresponding methods processFaultHanlder(activity) and processTerminationHandler(activity) respectively for all the scopes in the merged process.

4.3 Review of the Merged Process

An implementation to verify the correctness of the merged process was developed in [7] which is reused for verifying the correctness of the merged process after applying the transformations to fix the control link violations associated with compensation handlers during the consolidation process. The class CheckMergeResult requires the file paths for two files, first the file path to the merged process P_{merged} and second the file path to the expected process $P_{expected}$. The process $P_{expected}$ (developed for the scenarios described in the sections 3.4, 3.5 and 3.6) contains the structural design which corresponds to the expected merged process. Thus the obtained process P_{merged} after consolidation is checked against the process $P_{expected}$ to verify if the process P_{merged} conforms to the structural design of the process $P_{expected}$. Also, regular expressions are used to check if the activity names match. It is very important to note that the process $P_{expected}$ cannot contain any structure that violates any BPEL specification and thus it must be developed carefully. For example, a scope can have only one activity directly enclosed within itself (refer section 2.2.6). Therefore scopes containing more than one activity must not be defined in the process $P_{expected}$.

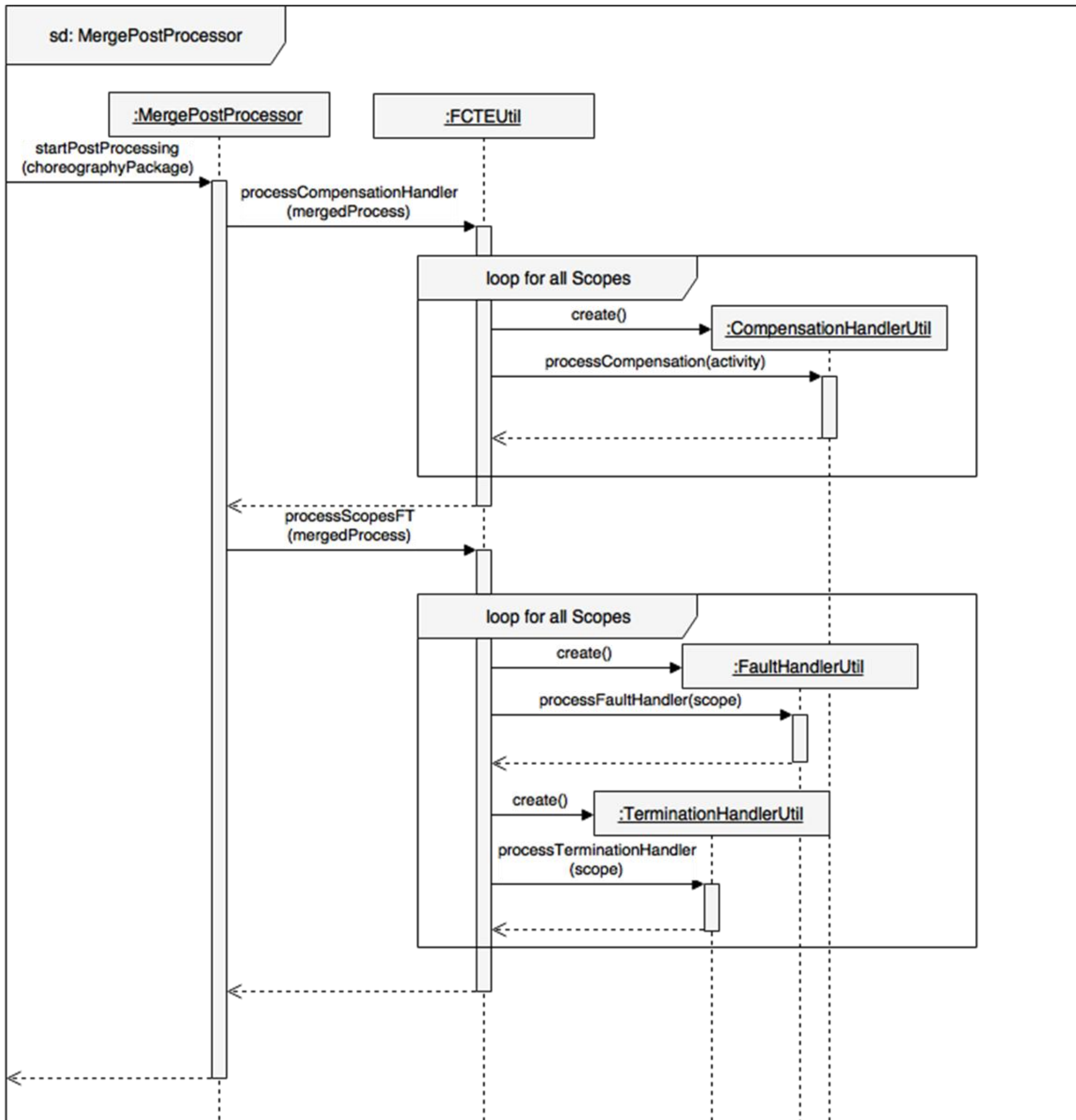

Fig 4.5: Sequence diagram for MergePostProcessor dealing with FCT handlers

5 Conclusion and Future Work

The main objective behind this study thesis was to identify and resolve scenarios when consolidation of process models that interact via compensation handlers results in merged processes containing control link violations related to compensation handlers. This objective was achieved by identifying and grouping the scenarios (refer to section 3.2) into logical units and a solution for each unit was proposed. An implementation of the proposed solution was integrated with the existing system (refer to section 2.5) by creating choreographies for each scenario, applying the consolidation process and checking the merged process with the expected correct consolidated process.

Chapter 1 discussed the various reasons that can lead to the decision of insourcing a partner's business process into its own and the advantages behind this consolidation. The consolidation process was described briefly with an example. In chapter 2, all the fundamental concepts that are needed to understand the basics of BPEL and BPEL4Chor were described.

In chapter 3, the solution for control link violations in case of fault handlers [7] was described briefly and it was used as the basis of the approach for compensation handlers. Based on the ways a compensation handler can be invoked, base scenarios were identified, analyzed and a solution was proposed for all the identified cases. An algorithm (refer to section 3.3) was proposed which is used to derive the compensation order graph complying with the default compensation order rules (refer to section 2.9).The compensation logic from the compensation handlers violating the control link constraints was propagated to the immediately enclosing parents FCT handlers and it was arranged in a way that follows the default compensation order derived by applying the proposed algorithm. Boolean variables indicating if a scope was reached and completed successfully were created which in turn were used as a check before executing the compensation logic for any scope. A conceptual solution was also proposed for a scenario wherein a scope is nested within a repeatable construct. The control flow relations among the basic activities from the choreography were maintained after applying the transformations to obtain the final merged process.

Chapter 4 elaborated the implementation details of the algorithms (refer to section 3.3) and the transformations (refer to sections 3.4, 3.5 and 3.6) needed to fix the control link violations associated with the compensation handler. Also, an overview of the integration of this implementation with the existing components and their correlation was described with the help of component and sequence diagrams.

Future Work

The concept of the isolated scopes has been briefly described in the section 2.2.6. The proposed solution assumes that the compensation handler associated with any scope is not an isolated scope. Thus the current solution needs an extension to provide an approach for compensation handler associated with an isolated scope.

In section 3.7, a conceptual approach has been proposed for the case when a scope is enclosed within a repeatable construct. The current implementation does not cover the implementation of this solution and thus it has to be extended accordingly.

The most important aspect of the consolidation process is to retain the control flow relations between the basic activities in the choreography and these must not be altered by the transformations described in the sections 3.4, 3.5 and 3.6. Thus, there is a need to devise a formal verification method which can verify that the control flow relations between the basic activities are maintained.

Bibliography

[1] "WFMC Terminology and Glossary English," Workflow Management Coalition, [Online]. Available: http://www.wfmc.org/.

[2] "BPEL (Business Process Execution Language)," TechTarget, [Online]. Available: http://searchsoa.techtarget.com/definition/BPEL.

[3] "The Advantages and Disadvantages of Outsourcing," flatworld solutions, [Online]. Available: https://www.flatworldsolutions.com/articles/advantages-disadvantages-outsourcing.php.

[4] S. Wagner, "Poster - Choreography-based BPEL Process Consolidation," Institute of Architecture of Application Systems, Universität Stuttgart, Stuttgart, Germany, 2013.

[5] S. Wagner, O. Kopp and F. Leymann, "Consolidation of Interacting BPEL Process Models with Fault Handlers," in *Proceedings of the 5th Central-European Workshop on Services and their Composition (ZEUS 2013)*, CEUR Workshop Proceedings, Rostock, Germany, 2013.

[6] S. Wagner, D. Roller, O. Kopp, T. Unger and F. Leymann, "Performance Optimizations for Interacting Business Processes," in *2013 IEEE International Conference on Cloud Engineering (IC2E)*, 2013.

[7] P. Berger, "Konsolidierung von BPEL Prozessmodellen im Kontext von Interaktionen über Fehlerbehandlungskonstrukte," Diplomarbeit, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, Stuttgart, Germany, 2013.

[8] S. Weerawarana, F. Curbera, F. Leymann, T. Storey and D. Ferguson, Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More, Prentice Hall PTR, 2005.

[9] F. Leymann, "Lecture notes on WSDL.," Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany, 2013.

[10] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," 15 March 2001. [Online]. Available: http://www.w3.org/TR/2001/NOTE-wsdl-20010315.html.

[11] F. Leymann, "OASIS BPEL Webinar," 12 March 2007. [Online]. Available: https://www.oasis-open.org/committees/download.php/23070/.

[12] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. Rijn, P. Yendluri and A. Yiu, "Web Services Business Process Execution Language Version 2.0," 2007. [Online]. Available:

http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[13] C. Stahl, "A Petri Net Semantics for BPEL," 2005. [Online]. Available: http://www2.informatik.hu-berlin.de/Institut/struktur/systemanalyse/preprint/stahl188.pdf.

[14] O. Kopp, S. Henke, D. Karastoyanova, R. Khalaf, F. Leymann, M. Sonntag, T. Steinmetz, T. Unger and B. Wetzstein, "An Event Model for WS-BPEL 2.0," Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany, 2007.

[15] C. Stahl, "Einführung in BPEL Teil 2," 2005. [Online]. Available: http://www2.informatik.hu-berlin.de/top/lehre/WS05-06/taskforce/bpel-einfuehrung_2.pdf.

[16] G. Decker, O. Kopp, F. Leymann and M. Weske, "BPEL4Chor: Extending BPEL for Modeling Choreographies," in *ICWS 2007, IEEE Computer Society*, 2007.

[17] G. Decker, "Design and Analysis of Process Choreographies," Dissertation, Business Process Technology Group, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, 2009.

[18] P. Debicki, "Choreographie-basierte Konsolidierung von BPEL Prozessmodellen," Diplomarbeit, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, Stuttgart, Germany, 2013.

[19] D. Cui, "Splitting BPEL Processes," Diplomarbeit, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, Stuttgart, Germany, 2012.

[20] A. Barros, M. Dumas and A. Hofstede, "Service Interaction Patterns," Lecture Notes in Computer Science, Volume 3649, 2005, pp. 302-318, 2005.

All the links were last followed on August 3, 2014.

Acknowledgement

I am sincerely thankful to my mentor and supervisor Sebastian Wagner from the University of Stuttgart for his help, guidance, motivation and support during all the phases of my master thesis. I would also like to thank Prof. Frank Leymann for giving me this wonderful opportunity to do my master thesis at the Institute of Architecture of Application Systems. I am also thankful to my family and friends for their help and moral support during the tenure of my thesis.

Prasad Phadnis

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. 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.

Stuttgart, 04.08.2015 (Prasad Phadnis)