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

Diplomarbeit Nr. 3037

Transaction Flow Generation for SWOM

Li Li

Course of Study: Informatik

Examiner: Prof. Dr. F. Leymann

Supervisor: Dipl.-Phys. Dieter H. Roller

Commenced: June 01, 2010

Completed: December 01, 2010

CR-Classification: D.2.11, D.1.3, H.2.8

Acknowledgment

I am heartily thankful to my supervisor, Mr. Roller, whose help, stimulating suggestions and encouragement helped me in all the time of development and writing of this thesis.

Thanks to Prof. Dr. Leymann, he made my thesis at the Institute of Architecture of Application Systems possible.

Deepest gratitude is also due to my family, for their understanding and endless love through the duration of my studies.

Contents

List of Figures

List of Listings

List of Algorithms

1 Introduction

Web Services based on the service-oriented architecture framework serve as the foundation for modern distributed, heterogeneous applications. They are perfectly suited as the function layer of the two-level programming model that is characteristic for workflow-based applications.

Workflow-based applications [LD00] are composed of two distinct pieces: a process model that describes the sequence in which the different activities making up the process model are being carried (programming in the large) and the individual components that implement the various activities (programming in the small). In the Web services environment, Web Services Business Process Execution Language (WS-BPEL) is used to describe process models.

The purpose of a workflow management system (WFMS) implementing the WS-BPEL specification is the management of the life cycle of business processes, the navigation through the associated process models, and the invocation of the appropriate Web services. The Stuttgarter Workflow machine (SWoM) partially implements the appropriate WS-BPEL standard. It is implemented on top of IBM WebSphere® and IBM DB2®.

The performance of workflow-based application depends to a great extent on the performance of the workflow engine. The performance of a workflow engine can be significantly improved through appropriate optimization techniques. One approach is to determine the transaction flow that is executed by the navigator, which is then used to carry the different optimization tasks.

The purpose of this work is to implement the transaction flow generator that generates transaction flows from the WS-BPEL process model and stores the transaction flow as an XML file in the flow execution plan.

This thesis is structured into the following chapters:

- “Background” covers basic knowledge about WS-BPEL, process model, activities in transaction, variables in transaction, transaction boundary, the notion of transaction flow, and a brief introduction to the results of manual optimization test
- “Flow Execution Plan” introduces in general the components of the flow execution plan
- “The Algorithm” presents the actual algorithm and explains all the details necessary to grasp the analysis, a motivating example is presented
- “Implementation” provides an overview over the test environment, the implementation is described steps by steps
- “Summary and Outlook” concludes the work
- “Appendix” presents the generated flow execution plan of a given BPEL process model

The figure 1.1 shows an overview of the whole work. We retrieve the model loader, get all the information about BPEL file and process deployment descriptor in the memory. We traverse a given BPEL process model, create transactions according to certain criteria. Then we get a transaction set at the end of the traverse, and through comparing the source activities and target activities of each transaction we can assert the source transactions and target transactions of each transaction, it means, the transaction flow will be generated. Then we

generate the flow execution plan through XML Beans, and save the xml file in the build time database.

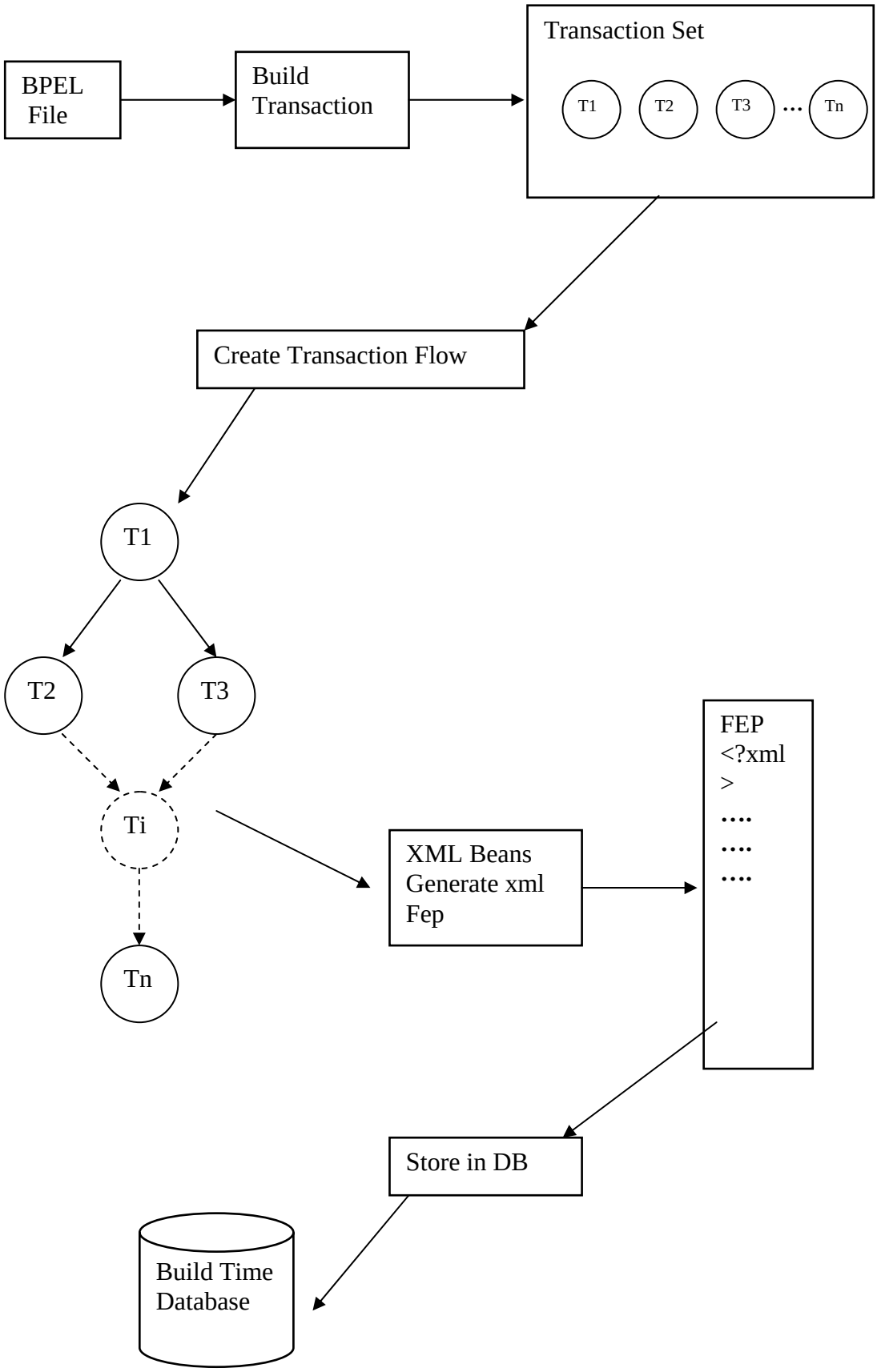


Figure1.1 overview of this work

2 Background

This chapter introduces the necessary related basics and technologies that are needed to understand the content of this thesis.

The following sections cover these topics:

- 2.1 WS BPEL
- 2.2 Process Model
- 2.3 Activities
- 2.4 Variables
- 2.5 Introduction of transaction Flow
- 2.6 Determination of transaction boundary
- 2.7 Manual optimization

2.1 WS BPEL

The Web Services Business Process Execution Language 2.0 (BPEL for short) is a “workflow based language that aggregates services by choreographing service interaction” [WCL+05] in XML.

According to [Wes07], “a business process consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations.” This connotes that BPEL is a programming language used to describe activities to produce goods, or in this case services, for a customer.

In BPEL, a business process executes steps to achieve a business goal. That goal can be the completion of a business transaction, or the fulfilling a job of service. The steps in the BPEL process execute activities to finish work. Those activities are focused on invoking partner services to perform tasks and return results back to the process. The aggregate work, the collaboration of all the services, is a service orchestration.

BPEL supports two ways to describe business processes: Executable and abstract business processes. An abstract business process leaves out details of the process, focusing on the broader interaction. “Interaction” in this case is the communication between a BPEL process and a Web service. Executable business processes are actual code, defining interactions detailed enough for execution by a BPEL engine. This work is about control flow in executable business processes and does not discuss any details of abstract business processes.

BPEL builds on the foundation of XML and Web services. The commands of the BPEL language are called “activities”. The activities will be introduced in section 2.3.

2.2 Process Model

Also according to [Wes07], “a process model represents a blue print for a set of process instances with a similar structure. Process models have a two level hierarchy, so that each process model consists of a set of activity models. Each process model consists of nodes and directed edges.”

There are different methods to describe a process model. The most common one is a process graph. The structure of a process model is made up of activities, control connectors, transition conditions, input containers, output containers, and data connectors [LR00].

The process logic will be defined in a process model. It specifies two major facets: the control flow and the data flow. The control flow defines the execution sequence of various activities which are defined as the tasks that need to be carried out as part of a process. According to [WCL+05], control flow in BPEL "is defined using a combination of structured activities and control links." The control links can only be defined in a <flow> activity, at arbitrary levels of nesting (an example of <flow> activity can be found in next section). In this work, we focus only on the control flow, the data flow is not discussed.

Figure 2.1 shows the control flow in a sample process graph, the nodes represent the activities. An activity is an abstraction of a piece of work that has to be performed within a process, in other words, it describe the tasks to be performed. Note that this is not a definitive graphical notation for WS-BPEL processes. It is used here informally as an aid to understanding.

The arrows (so-called incoming links and outgoing links) of each activity that connect different activities are control connectors that define the flow of control from one activity to the next activity. In other words, they describe the potential sequence in which the activities are to be carried out. We should notice that control connector must not form loops, it means, the control connector must not point backward. In the figure 2.1 we can also see, an activity can be the source or target of multiple control connectors, these activities will be introduced in next section.

Processing of a process graph is generally called navigation. It starts with the processing of the start activities. Before the process started, all activities are in a default state. After an activity has been processed, the outgoing control connectors will be followed, the navigation continues. In figure 2.1, activity A is the start activity, in a BPEL process it must be a receive activity. Then the navigation continues by following the link AB, and so on. The details will be introduced in chapter 4 and 5.

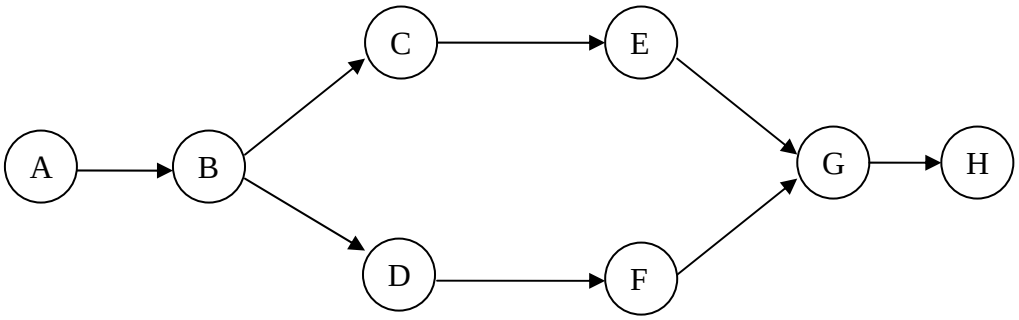


Figure 2.1 sample process

As we in figure 2.1 see, activity A is the initial receive activity, it starts the process. All the activities and their types will be shown in figure 2.2. Each activity will be explained in next section.

Figure 2.2 shows the activity type of these activities in the sample process shown in figure 2.1.

Activity Type	Activity name
Receive	A, E, F
Invoke	C, D, H
Assign	B, G
Fork	B
Join	G

Figure 2.2 activity types

2.3 Activities

WS-BPEL activities perform the process logic [OAS07]. Activities are divided into two classes: basic and structured. Basic activities are those which describe elemental steps of the process behaviour. Structured activities encode control-flow logic, and therefore other basic and/or structured activities will be recursively contained. In this section some BPEL activities, which play important role in transaction flow, are listed.

Flow Activity

A flow activity is a structured activity, in a flow container activity, contained activities are run concurrently. They can be synchronized and ordered through BPEL links. It should be noticed that BPEL links can not form control cycles. That is, the target of a link may not be a preceding activity of the link source in any way. As a result, one link may be at most used to connect two activities.

BPEL links have a transition condition in a specified expression Language that will yield a true or false result. If no transition condition is specified, it is true by default. The transition condition can refer to any data available to the link, such as BPEL variables. The link status is the status of the evaluation of the transition condition. This can be true, false or unset.

The standard-attributes and standard-elements (for example, in listing 2.1, the <links>-</links> element) for activities nested within a <flow> are significant because the standard attributes and elements exist to provide link semantics to the activities. Each WS-BPEL activity has the optional containers <sources> and <targets> which contain collections of <source> and <target> elements respectively. These elements are used to set up synchronization relationships through a <link>. The value of the “linkName” attribute of the <source> or <target> must be the name of a <link> declared in an enclosing <flow> activity. As Listing 2.1 shows, the linkName attribute within <sources> element “linkAB” has been declared in <links> element. The linkName attribute within <target> element “linkAB” has also been declared in <links> element.

Listing 2.1 presents a sample flow activity in a BPEL process model. As mentioned in section 2.2, all the control links are defined at the beginning of the <flow> activity. As we see, this flow container activity contains a receive activity A, an assign activity B and an invoke activity C, the other activities in this flow activity have been ignored. The <link> named “linkBC” starts at assign activity B and ends at invoke activity C. The requirement of the <link> that declared in this sample activity is satisfied.

```
<flow>

    <links>
       <link name="LinkAB"/>
       <link name="LinkBC"/>
       <link name="LinkBD"/>
       …
    </links>

    <receive createInstance="yes"
             name="A"
             …
             variable="InRequest">
       <sources>
          <source linkName="LinkAB"/>
       </sources>
    </receive>

    <assign name="B">
       <targets>
          <target linkName="LinkAB"/>
       </targets>
       <sources>
          <source linkName="LinkBC"/>
          <source linkName="LinkBD"/>
       </sources>
       …
    </assign>

    <invoke name="C">
      …
    </invoke>

    <invoke name="D">
    …
    </invoke>
    …
</flow>
```

Listing 2.1 flow activity

Sequence Activity

Sequence activity that executes the contained activities in the order they appear in the BPEL process. Sequence activity provides sequential control between activities.

A <sequence> activity contains one or more activities that are performed sequentially, in the lexical order in which they appear within the <sequence> element. The <sequence> activity completes when the last activity in the <sequence> has completed. The sequence activity can be used to organize the activities so that they are executed in a pre-defined, sequential order. In fact, a <sequence> contains implicit links. A <sequence> activity can be represented by a <flow> with properly links. Listing 2.2 is an example of a sequence activity. This <sequence> has two activities begins by activating its first activity— receive activity A. After that A runs to completion, the <sequence> activates its second activity— invoke activity B. After B completes, the <sequence> is finished.

```
<sequence>
     <receive name=A>… </receive>
     <invoke name=B> … </invoke>
</sequence>
```

Listing 2.2 sequence activity

Assign Activity

Variable update occurs through the <assign> activity, the <assign> activity copies data between variables and other constructs [OAS07].

The <assign> activity contains one or more elementary operations. The <assign> activity copies data from one location to another atomically, using a list of copy statements that copy one value each [WCL+05]. The <copy> element is one of the standard element of <assign> activity, as following shows.

```
<assign>
     <copy>
        from-spec to-spec
     </copy>
</assign>
```

Listing 2.3 the copy element in an assign activity

The <assign> activity copies a type-compatible value from the source ("from-spec") to the destination ("to-spec"), using the <copy> element.

There exist following variants of the “from-spec” types in executable process:

- Variable
- Variable part
- Partner link
- literal

Listing 2.4 illustrates copying a variable part to another compatible element type. It just shows the assign activity B and the two child activities of B in the sample process.

There exist two <copy> elements in this activity. And the from-spec type is variable part. The “from- variable” has two parts, one part’s name is “content1”, the other part’s name is ”content2”. The first <copy> copies the first part “content1” of the variable “InRequest” into the variable “OutRequest1”, which will be used as input variable by the invoke activity C. Analogous to the first copy element the second <copy> copies the second part “content2” of the variable “InRequest” into “OutRequest2” which will be used as input variable by the following invoke activity D. The invoke activity will be introduce later.

```
<assign name="B">
   <targets>
      <target linkName="LinkAB"/>
   </targets>
   <sources>
      <source linkName="LinkBC"/>
      <source linkName="LinkBD"/>
   </sources>
   <copy>
      <from variable="InRequest" part="content1" />
      <to variable="OutRequest1" />
   </copy>
    <copy>
      <from variable="InRequest" part="content2" />
      <to variable="OutRequest2" />
   </copy>
</assign>

<invoke name="C"
        ...
        inputVariable="OutRequest1">
   ...
</invoke>

<invoke name="D"
        ...
        inputVariable="OutRequest2">
   ...
</invoke>
```

Listing 2.4 example of an assign activity

Receive Activity

The <receive> activity plays an important role in the lifecycle of a business process. This activity has an important attribute called “createInstance”. A business process can be instantiated only when the “createInstance” attribute of the receive activity set to "yes” (see listing 2.5). The default value of this attribute is "no". In listing 2.5 the receive activity A with the “createInstance =yes” attribute initiates the process. This receive activity will also be called as “root activity” in this thesis. In other words, a root activity must be a receive activity. And this receive activity receives a message from the given partnerLink, portType and operation. In section 5.3.2 we can see the process of finding out this root activity in a business process.

Receive activity has also special impact on transaction flows, which can be seen in section 2.5.

```
<receive createInstance="yes"
         name="A"
         operation="start"
         partnerLink="TTProcAPL"
         portType="tns:TTProcAPT"
         variable="InRequest">
   <sources>
      <source linkName="LinkAB"/>
   </sources>
</receive>
```

Listing 2.5 receive activity

Invoke Activity

Invoke a service operation, specified by partnerLink, portType and operation. A partnerLink encapsulates the structure of the relationship between BPEL and a Web service through a Web service interface. A portType represents a set of operations available from the partnerLink and other endpoints. An operation is an action supported by the Web service. More information on Web service terminology can be found in [OAS07] and [Wor01].

Variables are read from the inputVariable and fromVariable and written to the outputVariable and toVariable.

The operation invocation can be a one-way invocation or a request-response invocation. If the operation is one-way and no response is expected, the inputVariable or fromVariable is needed only. The invocations in the two invoke activities C and D in listing 2.4 are one- way invocation. For request-response operations, invoke waits for a reply, an additional variable: outputVariable or toVariable is used.

Here is an example (see Listing 2.6) of a invoke activity. The operation invocation in this invoke activity is a request-response operation. Variable are read from the inputVariable-- InInvoke and written to the outputVariable-- OutInvoke.

```
<invoke name="H"
        operation="startTTProcD"
        partnerLink="TTProcDPL"
        portType="tns:TTProcDPT"
        inputVariable="InInvoke"
        outputVariable="OutInvoke">
   <targets>
      <target linkName="LinkGH"/>
   </targets>
   <sources>
      <source linkName="LinkHI"/>
   </sources>
</invoke>
```

Listing 2.6 invoke activity

Wait Activity

The <wait> activity specifies a delay for a certain period of time or until a certain deadline is reached [OAS07]. If the specified duration value in <for> is zero or negative, or a specified deadline in <until> has already been reached or passed, then the <wait> activity completes immediately. A typical use of this activity is to invoke an operation at a certain time.

Listing 2.7 presents an example of the wait activity. In this example, a deadline in <until> has been specified, it means, when this deadline has been reached, it's time to hand in the diplomarbeit, the operation "diplomarbeit abgeben" should be carried out.

```
<wait>
     <until>'2010 -12- 01 12:00' </until>
</wait>
<invoke paternerLink="Prüfungsamt", operation="diplomarbeit abgeben"
```

Listing 2.7 wait activity

Pick Activity

The <pick> activity waits for the occurrence of exactly one event from a set of events, then executes the activity associated with that event. Once an event has already been selected, the other events are no longer accepted by that <pick> activity.

The <pick> activity is made up of a set of branches, each containing an event-activity pair. The <pick> activity completes when the selected activity completes. The <pick> activity's events occur in two forms:

- The <onMessage> is similar to a <receive> activity, in that it waits for the receipt of an inbound message.
- The <onAlarm> corresponds to a timer-based alarm. If the specified duration value in <for> is zero or negative, or a specified deadline in <until> has already been reached or passed, then the <onAlarm> event is executed immediately. Again, the handling of race conditions depends on implementation.

There must be at least one <onMessage> in each <pick> activity. The following listing 2.8 [OAS07] shows, the activity's event comes in the form of <onMessage>.

A special form of <pick> is used when a new instance of a business process is to be created upon the receipt of an <onMessage> event. This form of <pick> has a createInstance attribute with a value of yes (the default value of the attribute is no). In such a case, the events in the <pick> must all be <onMessage> events. This requirement must be statically enforced.

```
<pick>
   <onMessage paternerLink="buyer"
      portType="orderEntry"
      operation="inputLineItem"
      variable="lineItem">
   </onMessage>
…
…
</pick>
```

Listing 2.8 example of pick activity

Join Activity

An activity that is the target of multiple control connectors is called join activity [LR10]. A join activity has more than one incoming control connector, and it can be processed only when all the incoming control connector entered. In figure 2.1 the sample process activity G is a join activity, it has two incoming links, link EG and link FG. The navigation stops at activity G until the link EG and FG have been evaluated. It means, G can be processed, only when the link EG and link FG have entered. Because this special property join activity has also special effect on the transaction flow, it will be discussed later.

Fork Activity

An activity from which multiple connectors leave is called fork activity [LR10]. A fork activity has more than one outgoing links. It means the activities at the end of the control connector are performed in parallel. In figure 2.1 activity B is a fork activity, it has two outgoing links, link BC and link BD, and the targets activities of these two links are performed in parallel.

The above listed activities play important role in determination of transaction boundary. In section 2.6 will be explained, how these activities influence the transaction boundary and transaction flow.

2.4 Variables

BPEL variables can be used for holding messages that constitute a part of the state of a business process. BPEL variables can be WSDL message types, XML Schema types or XML Schema elements. Variables are defined in the variables element of a BPEL process or scope activity. Variables can be accessed in two ways in BPEL. Some activities require a variable to be selected in one of their attributes. In some places, XPath queries can be used to access variables. The details on variable access can be found in [OAS07].

Complex variables contain a hierarchy of variable elements. When an activity writes to a variable element, it might write to any number of sub-elements. Again, this cannot be statically determined, so any writer to a variable element also has to be considered to be a writer to any sub-element.

Variable declarations can appear directly under the process element, which means that they are visible to all BPEL constructs within the BPEL process, or under a scope element, which means it is only visible to the children of that scope. So at the beginning of a BPEL process, all global variables in uninitialized state should be defined.

The following listing (Listing 2.9) shows an example of a <variable> declaration. In this example, the variable uses WSDL message types declared in a WSDL document with the target namespace: Message1. The variable name is “InRequest”.
The variable access process will be introduced in chapter 4.

```
<variables>
  <variable messageType="tns:Message1" name="InRequest"/>
</variables>
```

Listing 2.9 example of variable declaration

2.5 Introduction to Transaction Flow

This section provides an introduction and overview over transaction flow. As presented in [Rol10] a transaction flow is the set of transactions that the navigator carries out. The transaction flow is generated by contracting activities of the process model into transactions. There exist five basic transaction types: very short, short, medium, long and ultimate. It is necessary to set a transaction boundary for a wait, receive, and pick activity. We call these activities as boundary activities. Because for these activities, the further navigation of the process instance must be suspended, until activity can be processed, it means, the required message for these activities arrives or the occurrence of a waited event. Except the fixed transaction boundaries an additional criteria for each transaction types are also provided in [Rol10]. It will be shown in the figure 2.3.

Figure 2.3 [Rol10] shows these five types and their transaction boundaries criteria. As it shows, in the very short transaction type, each activity and each link would be processed in a separate transaction, and in the short transaction type, each activity and the associated links comprise a separate transaction. In the medium transaction type, the fork and join activity terminate the current transaction. In the long transaction type the fork activity has no impact on the transaction boundary, the transaction will terminated only by the receive, wait, pick and join activity. From the figure we can also see that the only difference between the long transaction type and ultimate transaction type: A join activity is transaction boundary activity for long transaction type, but a non- finished join activity is transaction boundary activity for ultimate transaction type. This difference will be explained with an example in following.

transaction types	**transaction boundaries criteria**
very short	activity, link
short	activity
medium	fork activity, join activity, receive, wait, pick activity
long	join activity, receive, wait, pick activity
ultimate	non-finished join activity, receive, wait, pick activity

Figure 2.3 Transaction Types

In the next section the concrete determination of transaction boundaries of each transaction type will be introduced.

2.6 Determination of Transaction Boundary

For a better understanding we take the sample process as an example. The determination of the transaction boundary will be explained with this example. We analyse this process based on different transaction types.

If the transaction type is very short, then an activity or a link itself can build a transaction. So activity A is a separate transaction. If the transaction type is short, then activity A and its outgoing links linkAB construct a separate transaction.

If the transaction type is medium, the activity A and activity B builds a separate transaction, because activity B is a fork activity, it has two outgoing links. According to the transaction boundaries criteria of the medium transaction type the current transaction terminates after processing the fork activity B. Figure 2.4 illustrate the transaction flow of the medium

transaction type. According to the above described criteria in figure 2.3 the transaction flow is constructed by six transactions. As we see, the first transaction terminates after processing activity B, the initial receive activity A and the assign activity B comprise transaction T1. The processing of the invoke activity C and D are called parallel, the receive activity E and F complete separately transaction T2 and T3. T4 and T5 process the messages that are received by the receive activities E and F, these two transactions terminate after processing the join activity G. From the figure 2.4 we see, the receive activity E (F) are part of two transactions, they are contained by transaction T2 (T3) and T4 (T5). Activity G itself construct a transaction T6.

If the transaction type is long, then the first transaction terminates until a receive activity or a join activity is read, in the sample process, activity E and F are receive activities, so the first transaction terminates after processing the activity E and the activity F, it contains now the invocation activities C and D and the associated receive activities E and F, as the following figure 2.5 shown. The receive activity E and join activity G comprise the transaction T2, and receive activity F and join activity G comprise the transaction T3. The performation of transaction T3 is the same as the perfomation of transaction T2. Transaction T4 performs at last. As we see, when the process is carried out using the long transaction type, the transaction flow is made up of only four transactions.

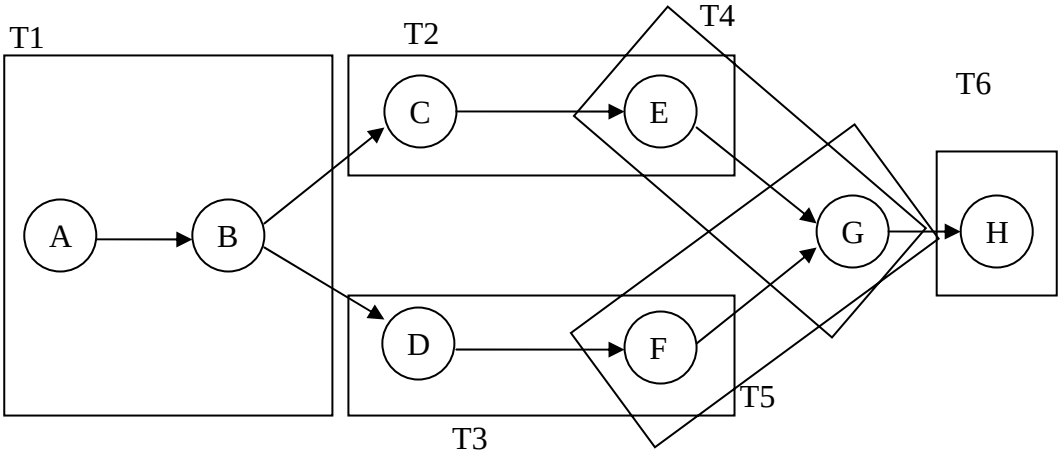


Figure 2.4 transaction type – medium

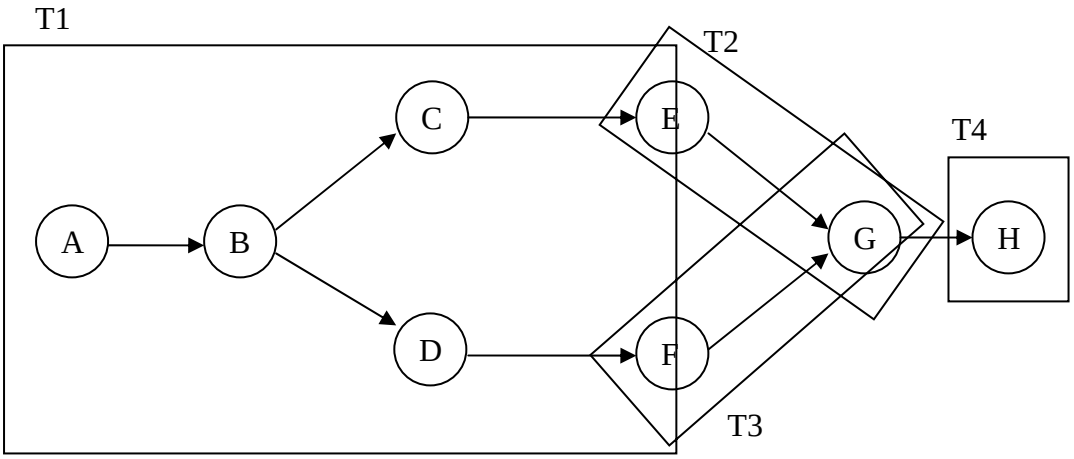


Figure 2.5 transaction type – long

If the transaction type is ultimate, from figure 2.3 we have already known that the only difference between ultimate and long is the handling of the join activity. In long transaction type, the transaction terminates after processing the join activity, but in ultimate transaction type the transaction terminates not certainly at join activity. We should check if for the join activity all necessary incoming links have entered. So for a join activity a state variable to indicate the join condition is needed, the default value of this state variable is “false”, only when all the incoming links of this join activity have arrived, the state variable can be set as “true”.

It is obviously to see the difference between figure 2.6 and 2.7. When this process is carried out using the ultimate transaction type, the transaction flow is made up of only three transactions. When transaction T3 has processed activity G, it must suspend until Transaction T2 has finished, that means, all the incoming links of G have entered. So we set a state variable: joinable, to indicate this join condition in activity H. The “joinable” will be initialized as “false”, only when linkEG and linkFG have entered in activity G, the “joinable” can be set as “true”.

There exist an alternative for ultimate transaction type, as the figure 2.7 shows, because we don’t know which transaction will be processed firstly. In the fact there exists no difference between these two alternatives. Whether T2 arrives before or T3 arrives before have no effect on the processing. The transaction flow T1→ T2 → T3 and T1→ T3 → T2 are actually the same transaction flow. T2 and T3 will be processed parallel.

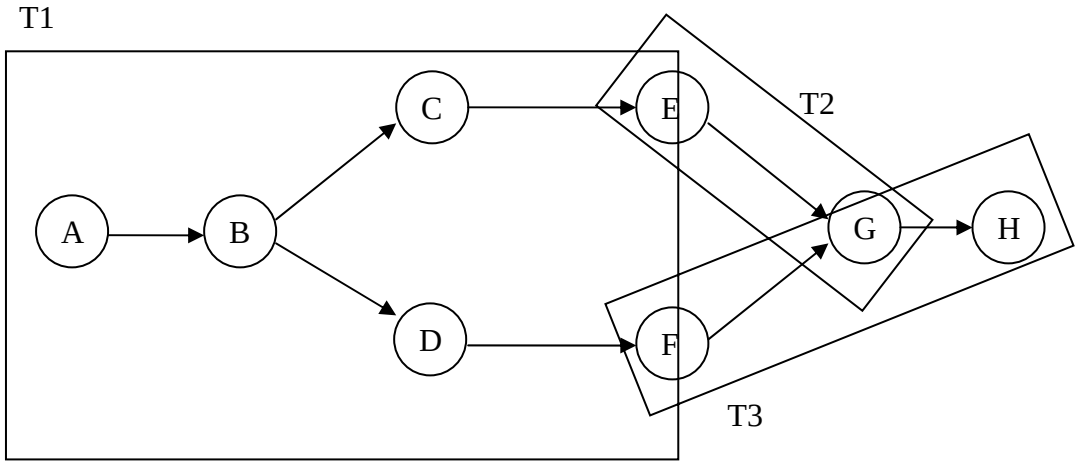


Figure 2.6 ultimate transaction type- alternative 1

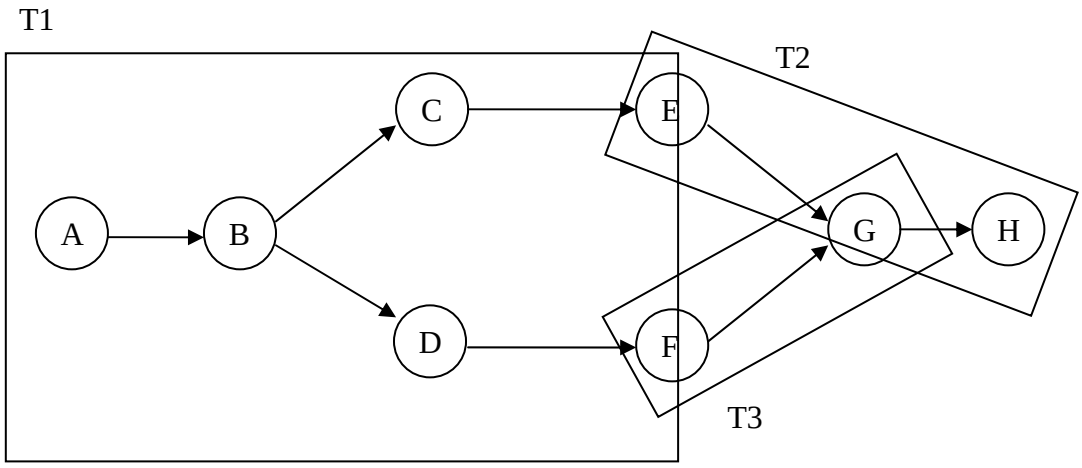


Figure 2.7 ultimate transaction type- alternative 2

The number of transactions for each of the different transaction types can be as following (figure 2.8) [Rol10] summarized. As can be seen, the ultimate transaction type is carrying out the minimum number of transactions.

Transaction type	Number of navigator transactions	Number of service invoker transactions
very short	22	4
short	13	4
medium	7	4
long	5	4
ultimate	4	4

Figure 2.8 Number of Transactions

2.7 Manual Optimization[1]

In [Rol10] a manual optimization and the test results are presented. Via this manual optimization we can see significant performance improvement.

The following figure shows three test scenarios, which simulate the different conditions under which the processes are run. Test case 1 focus on the determination of the scalability related to the CPU load. Test case 2 is used to determine of the impact of the higher parallelism in the incoming requests with various numbers of the parallel requests. Test case 3 is used to determine the impact of the different transaction types.

TC ID	Parallel requests	CPU Load	Message size
1	1	20/30/50/70	200
2	10/30	50	200
3	30	var	200

Figure 2.9 three test cases of manual optimization

The following figure shows the performance test result from the test case 1. In this case the client was generating one request at a time. As the figure shows, the workflow engine scales pretty linear.

CPU load (%)	28	32	40	52	70
Main Process (1/min)	289	323	419	637	778

Figure 2.10 result of test case 1

The following figure shows the performance test result from the test case 2. As can be seen, the workflow engine scales relative well.

[1] All test results in this section come from [Rol10]

Number parallel Requests/ CPU load	1	10	30
50	555	612	564

Figure 2.11 result of test case 2

The results of the test case 1 and 2 present that the workflow engine scales almost linearly with various CPU load and request parallelism.

The following figure shows the performance test result from the test case 3. As can be seen, the amount of process instances that are carried out, is improving from the normal approach of running each activity within a transaction to the ultimate transaction type, where several activities are carried in a transaction, improves by 37 %. The increase in performance can be seen that the throughput of the workflow engine is quite high.

Transaction type	CPU load	Main process (1/min)	Total process (1/min)	Improvement
short	53	902	4510	-
medium	42	1137	5685	26
long	40	1195	5975	32
ultimate	36	1238	6190	37

Figure 2.12 result of test case 3

As can be seen, the proper setting in manual optimization provides significant performance improvements. It can be expected that even greater performance can be achieved through a workflow optimizer, which automatically determines the correct settings. The design of such an optimizer is just the purpose of this work.

3 Flow Execution Plan

In this chapter the flow execution plan is to be introduced.
The flow execution plan (FEP) controls the execution of the SWoM engine that means the navigator, the invocation handler, and some of the shared components. It is associated with a particular process model. Each process instance is associated with a particular version of the Flow Execution Plan [FEP].

A FEP consists of two major parts:

- The transaction flow that identifies the different transactions and the way of processing them
- The execution options which are derived from information collected by the statistics manager.

In this work we generate only the first part— the transaction flow.

Generally speaking, as the figure 3.1 shows, we should write an optimizer to generate a FEP from a BPEL process, and FEP is saved in the form of an XML-File.

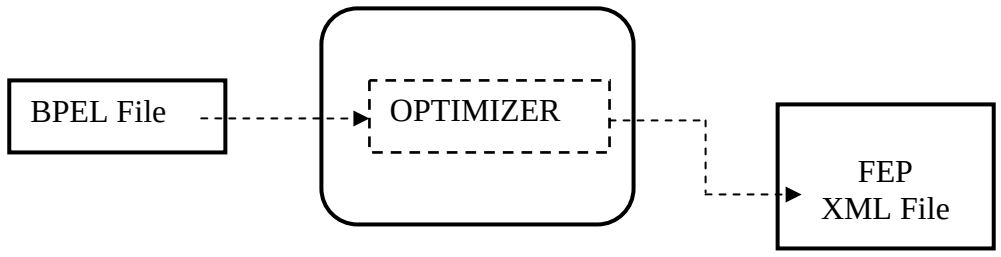


Figure 3.1 location of optimizer

Transaction flow defines the transaction flow that is used to optimally carry out process instances [FEP]. Transaction flow is made up of set of transactions. Transaction is made up of set of activities and set of variables. We traverse a BPEL process through analysis of each activity.
The following list shows what information we should specify in a transaction.

- Transaction ID: identifier the transaction
- Sources: identifies the transaction, which this transaction is the source
- Targets: identifies the transaction, which this transaction is the target
- Activities in Transaction: identifies the activities that make up this transaction. It should be note that an activity may be part of two transactions, for example, a receive activity or a join activity. For this part we should still define some properties:
    - Name: identifier the activity
    - Position: identifies the position of the activity within the transaction. There exist four position types: start, middle, end, and singleton
    - Joinable: indicates whether the activity is anchoring the following transaction, this property is especially for the join activity, for another activity this property is not considered.
- Variables in Transaction: identifies the variables that are consumed or created within the transaction. We should also define some properties for this part.
    - Name: identifier the variable

- Start state: defines the start state of the variable within the transaction, there exist three states: new, old, unknown. If this variable is newly created by an activity, then the start state of this variable should be set to “NEW”. Otherwise if the variable is already created by another activity, we just use it to save something, then the start state of this variable should be set to “OLD”. For some activities, for example, join activity, because we can not determine, which all the incoming links of this join activity have already entered. So for such an activity we can not make sure of the start state of this activity, so we can only set the start state of those activities as “UNKNOWN”.
- End state: defines the end state of the variable within the transaction, there exist also three states: keep, delete, unknown. If this variable is no longer used, so we can set the end state of the variable as ”DELETE”. On the converse if we need it later, the value of this variable should be kept, so the end state of this variable should be set as “KEEP”. Obviously if we are not sure, if the variable should be used later or not, then the end state of the variable can only be set as “UNKNOWN”.
- Creation activity: identifies the activity, which creates the variable, this property will be set only when the start state of the variable is “START”.

The following list presents the general structure of the flow execution plan.

```
<transactionFlow>
            <transactions>
               <transaction ID="">
                       <sources>
                                <sourceTransaction ="--">
                                <sourceTransaction ="--">
                       </sources>
                       <targets>
                                <targetTransaction ="--">
                                <targetTransaction ="--">
                       </targets>
                       <activitiesInTransaction>
                               <activityInTransaction name="--">
                                    <position>--</position>
                                    <joinable>--</joinable>
                               </activityInTransaction>
                               <activityInTransaction name="--">
                                    <position>--</position>
                                    <joinable>--</joinable>
                               </activityInTransaction>
                       </activitiesInTransaction>
                       <variablesInTransaction >
                             <variableInTransaction name="--">
                                  <startState>--</startState>
                                  <creationActivity>--</creationActivity>
                                  <endState>--<endState>
                             </variableInTransaction>
                             <variableInTransaction name="--">
                                  <startState>--</startState>
                                  <creationActivity>--</creationActivity>
                                  <endState>--<endState>
                             </variableInTransaction>

                          </variablesInTransaction name=”--“>
               </transaction>
               <transaction>
```

```
            ……..
            ……..
          </transaction>
        </transactions>
</transactionflow>
```

The following figure illustrates the tree structure of the transaction flow. The root node of this tree is the Transaction flow, it conforms to the root element of the above listed flow execution plan.

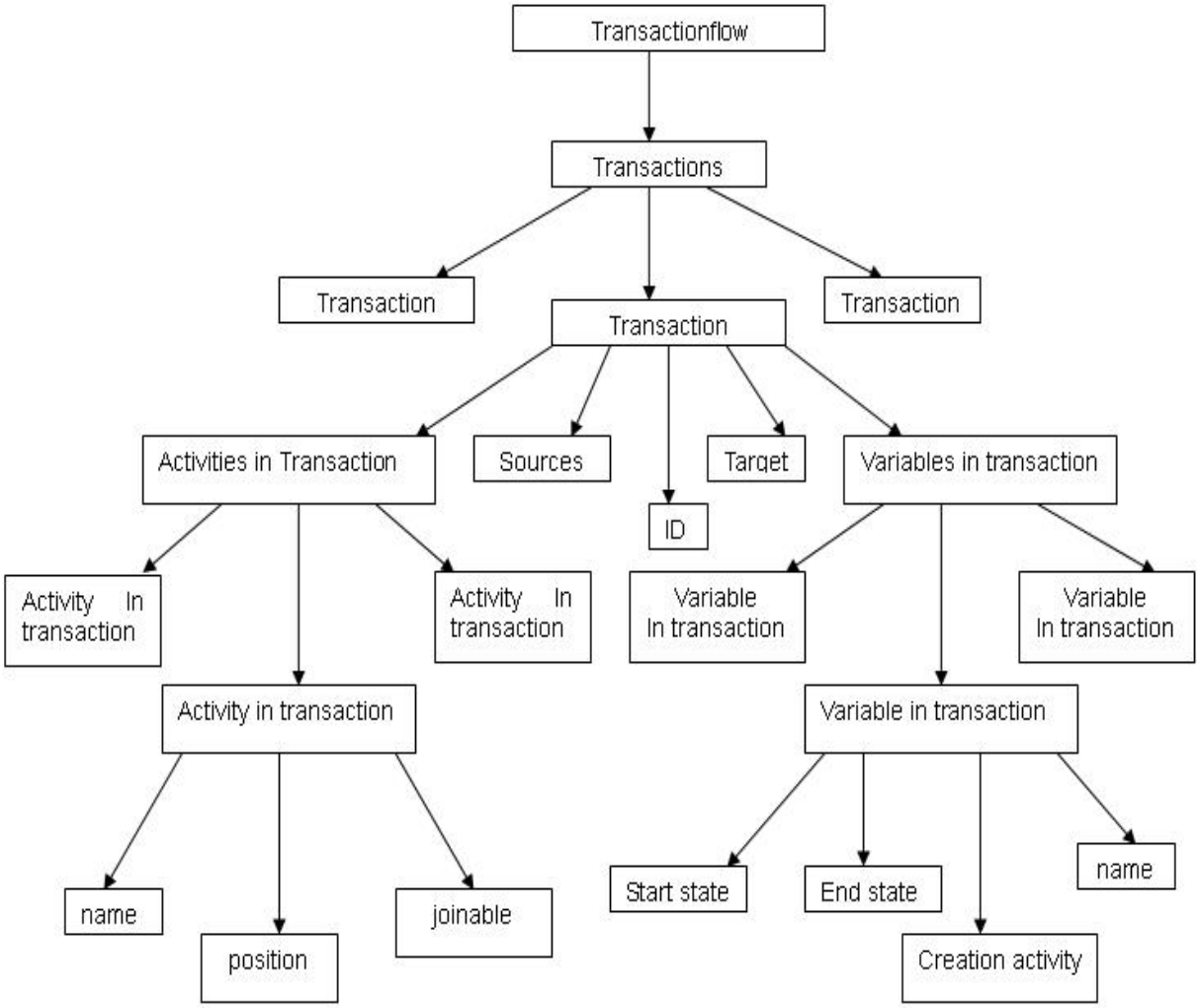


Figure 3.2 tree structure of transaction flow

4 Algorithm

The following chapter will provide details of the different parts of the algorithm. In the process of doing this, I am also going to repeat some of the explanations of the algorithm which have already been made in the referenced sources. It should thus be possible to understand the presented algorithm without reading any references.

The following sections cover these topics:

- Section 4.1 “basic idea” provides an introduction of the algorithm
- Section 4.2 “general activity analysis” specifies how to analyse a activity
- Section 4.3 “general variable analysis” explains how to get the variable within an activity
- Section 4.4 “create transaction flow” explains how to generate transaction flow

4.1 Basic Idea

In this section we introduce the basic ideas.

It is possible that a BPEL process with multiple start activities, but in this thesis we focus on the BPEL process with only one start activity. For a given BPEL process model, we get the first activity at first. All the outgoing links of the first activity will be followed, analyzed. The first activity is either a flow activity or a sequence activity. If the first activity is a flow activity, then we get the start activity within the flow activity. If the first activity is a sequence activity, the successor will be analyzed. By this means all child nodes will be obtained and added into a FIFO (i.e., First In, First Out) queue. According to the principle of the FIFO queue all the child nodes in the queue are to be handled in the order of their arrival. The core of traverse procedure is the "breadth first traverse".

During the traverse each activity will be analyzed, the general analysis will be realized in the following steps:

- Get the child activity
- Check the type of each child activity
- Run special procedure according to the activity type

These steps will be explained in next section.

4.2 General Activity Analysis

In this section we introduced the analyse process of an activity.

If an activity has outgoing links, the links are followed, analyzed and analysis of the target activities starts. If the activity is part of a sequence, the successor is analyzed. We define two FIFO queues for storing different activities for further use. One is called QA, for storing the child activity of an activity; the other is called QNT, for storing the activity, which runs a new round of traverse when the current QA is emtpy. The concrete analysis of an activity is performed by the following steps:

1. Get Child Activity**:** get the outgoing links of the current activity at first, then get the target activity of each outgoing link, all the target activity are just the child activity of the current activity. These target activities should to be put into the FIFO queue QA.
2. Determine the type of Child activity: get the top element of the QA, and then check

the type of the child activity.

- Case2.1: activity is of the type “receive”, “wait”, or “pick”
- Case2.2: activity has the logic relation: “fork” or “join”
- Case2.3: other activities: activities, which have no effect on transaction boundary

For the above two cases 2.1 and 2.2, the current transaction should finish, and a new transaction should be created. The current activity should carry a new round of traverse. It means the navigation begins from this activity next times. For the case 2.3, because these activities have no effect on transaction boundary, we just need to continue to get the child activity and put the child activity into the queue QA.

3. Handling Activity: After determining the type of activity, run special handler procedures based on the activity type. The algorithm for handling activity will be introduced in next page.

4. Create Transaction: Root transaction begins from the root activity. When the first round of the traverse finishes, the current transaction finishes too, and all the activities in this transaction as well as their positions are stored for further use. The first transaction has been created. All the activities in this transaction are divided into three classes according to their positions: start activity, middle activity, and end activity. These three classes provide the possibility to generate the transaction flow. It will be explained in next section.

These steps are realized in the procedure ActivityAnalyse and CreateTransaction. The procedure ActivityAnalyse is described in Algorithm 1.

Generally, this procedure ActivityAnalyse(a) will take an activity (a) as argument. We define a set called “ActivityInTransaction”, each element in this set is a two-tuples: activity and its position, namely (Activity, Position). The activity’s position identifies the position of the activity within the transaction [Rol10].

The procedure begins from the root activity, we save the first two-tuples: root activity and its position, the position of the root activity is: start, so the first two-tuples in the “ActivityInTransaction” is: (rootActivity, start), and then call the function “GetChildActivity(a)” to get the child activity of the root activity, this is done in lines 4 to 6. After getting the child activity of the root activity, we save the child activity in QA, now we should check, whether the queue is empty or not, if yes, we do not need to go on handling the activity. If not, we get the top element of the QA, and call the function “HandlingActivity(a)”. This is done in lines 7 to 11. When the current traverse finishes, the function CreateTransaction () (line 13) will be called to generate the first transaction. After the first transaction being generated, we should check the queue: QNT, whether it is empty or not. If not, we get the top element of QNT, and the procedure ActivityAnalyse will be called again, it means, this procedure will be recursive called while the QNT is not empty. If the QNT is empty, then the traverse of this BPEL process finishes. Our task is done. This is done in lines 13 to 19.

Now we take a BPEL process graph as an example, as the figure 4.1 shows, we put the root activity A and its position (a, start) in “ActivityInTransaction” at first, and call the function “GetChildActivity”, get the child activity of A: B and C, put B and C into the queue QA, and after checking we know the QA is not empty, then we get the top element of the queue, here is B, then put B and its position into the set “ActivityInTransaction”, and at the same time the function handlingActivity(a) should be called to handle the activity B. As the figure 4.2 shows,

the type of activity B was checked, it is not the transaction boundary activity, so we continue to get its child activity: D and put D into QA, and now the analysis of activity B finishes. The analysis procedure goes back to get the top element of QA, while the QA is not empty. The top element of QA: C will be poped. The analysis of the activity C is analogous to the analysis of the activity B. The child activity of C: D will be pushed into QA. We should pay more attention to the activity D. When we get the activity D from QA, we call the function handlingActivity(a), and check the type of D, its type is “receive”, it signifies the traverse should end now, this receive activity D should be put into the queue: QNT. Figure 4.3 shows this process.

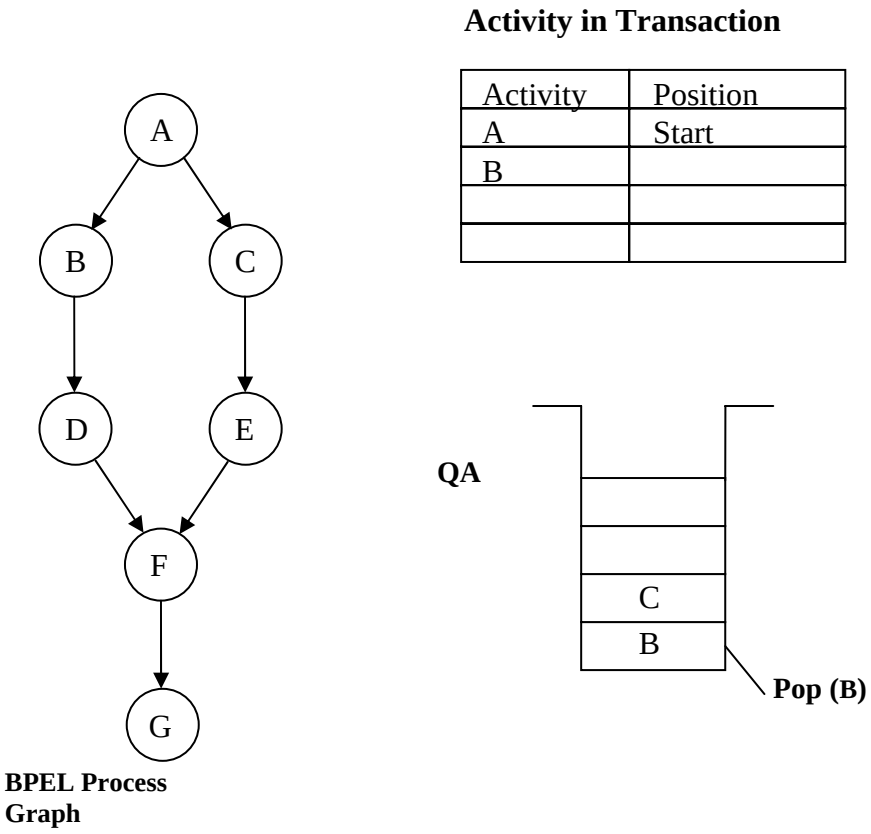


Figure 4.1 get child activity

```
1:  procedure ActivityAnalyse(a)
2:
3:     // Preconditions
       Define ActivityInTransaction, the set of all the activities in
current transaction, initialize it as empty set, and then put "a" and its
position into it.

4:   ActivityInTransaction = ActivityInTransaction U {(a, a's position)}
5:   GetChildActivity(a)    //function GetChildAcitvity
6:      while QA is not empty:
7:   //the loop begin from the top element of QA, deenqueue a and handle it
8:           Deenqueue a from QA
9:           handlingActivity(a)  //function handlingActivity
10:    end while
11:   // sub procedure createTranasction
12:   createTransaction()
13:   //begin new round of traverse
14:   while QNT is not empty:
15:       Deenqueue a from QNT
16:     // ActivityAnalyse will be rekursive called until the QNT is empty
17:       ActivityAnalyse(a)
18:   end while
19:
20: end procedure
```

Algorithm 1. procedure activity analyse

The following algorithm presents the function: GetChildActivity(a), it shows how to get the direct child activities of an activity. The idea is relative simple: as mentioned above, we just need to get the outgoing links of an activity, and follow the outgoing links to get the target activity of each link. As the algorithm 2 shows, line 9 checks, whether the current activity has only one or more outgoing links. If yes, then line 10 adds all outgoing links into the set: OL. Lines 12 to13 loop through all outgoing links in OL, get the target activity of each outgoing links and put them into the set: TA. Lines 14 to 15 loop through all target activities in set: TA and put them into the Queue: QA. In our example, we get the child activities of the activity A with this function. The activity A has two outgoing links, link AB and link AC. We get the target activity of each link and then we get the activity B and C, and put them one by one into QA (see figure 4.1).

```
1:   function GetChildActivity(a)
2:   //initialize QA, OL, TA as empty set,
     //all the child activities are saved in the set: QA
3:   Input: activity    Output: all the children activities
4:   Var: x, y
5:   // QA: Queue Activities, OL: outgoing links, TA: target activities
6:   Init: OL = null, TA = null;
7:
8:   // |Out (a)|: the number of the Outgoing Links of the current activity
9:         if | Out (a)| >=1, then
10:           OL= OL U {Out (a)}
11:        end if
12:         foreach x in OL do
13:           TA = TA U {Target (Out (a))}
14:         foreach y in TA do
15:           Enqueue y
16:
17:    end function
```

Algorithm 2: function GetChildActivity(a)

In the following we present an algorithm (Algorithm 3.) for checking activity types, different function will be run based on the different activity types. Line 4 defines a new queue: QNT to save the activity, which runs a new round of traverse, it means, the next traverse begins from the top element of the queue: QNT. This top element acts in next traverse as root activity. Lines 5 to 24 loop through all the activities in QA. Line 6 puts the top element of QA and its position into the set ActivityInTransaction. The next part determines the type of activity that is currently being handled and delegates the handling to designated procedures (line 7-22). This is the main part of this function.

```
1.  function HandlingActivity(a)
2.  Input: activity
3.  Activity:  a
4.  Init: QNT = Null
5.    foreach a in QA
6.      ActivityInTransaction = ActivityInTransaction U {a, position}
7.       if((activityTypes=Receive)  or(activityTypes=Wait)  or(activityTypes
           =Pick)),  then
8.        QNT =QNT U {a}
9:       while(QA!=Null){
10:       Deenqueue top element "top"from QA
11:       QNT= QNT U {top}
12:       QA= QA U {top}
13:      }
14:      else if (JoinActivity(a))
15:           // function JoinActivity(a) is described later
16:           then
17:              if a is not in QNT, then
18:                 QNT =QNT U {a}
19:              else skip it
20:              end if
21:      else
22:          GetChildActivity(a)
23:
24:      end if
25:  end function
```

Algorithm 3: function HandlingActivity(a)

Line 7 checks, if the activity type is “receive”, “wait” or “pick”, these three types are fest transaction boundaries. (This has already been explained in chapter 2, here I will not repeat it.) If yes, a new transaction should be added. The current activity should be put into the new Queue: QNT. The following things should be noted: we should check (Line 9), if the QA is already empty, if yes, we should not do anything, the current traverse finishes. If not, the rest activities of QA should also be put into the set: ActivityInTransaction and the Queue: QNT. It means, these activities should not only be included in the current transaction, but also act as the root activity in the subsequent transaction, so this activity should also be included in the subsequent transaction.

As the figure 4.3 shows, in our example we get the top element of QA, here is D, and check its type, it is “receive”, then the current transaction should end, the activity D should be push into QNT. Then we check the QA, it is not empty, so we go on to get the activity in QA, here is “E”, and push it in QNT. From the figure 4.2 we can see, the transaction T1 includes activity set {(A, start), (B, middle), (C, middle), (D, end), (E, end)}, and the new traverse begins from D, it means, new transaction T2 begins from activity D, and the procedure “ActivityAnalysis” will be recursive called until the QNT is empty.

Line 14 checks if the current activity has the logic relation— fork or join. About these two logic relations we have explained in chapter 2. Here we only take the “Join” relation into consideration, because the fork relation has no effect on the transaction boundary of the

ultimate transaction type. If the current activity has join relation, for short, we call these activities as join activity, then the current traverse finishes, the current transaction finishes too. It means the current Join activity and its previous activity make up the current transaction. And before the new traverse begins, the join activity should be put into QNT. But in this case, an additional step should be taken, that is to say, according to the speciality of the Join relation, that is, the join activity can be contained by two transactions. We should check at first, if the Queue QNT already contains the current join activity. It means it is possible that the join activity has already been put into QNT by last transaction. If yes, this activity should not be put into QNT, because it is already included in QNT, when we put it again, this join activity will be doubly analysed, it will be redundant and not permitted. Line 22 goes on to call the function GetChildActivity() to get the child activity of the current activity for the case that the activity type is not one of the boundary activities.

In our example, as the figure 4.4 and 4.5 show, activity F has join relation, activity D is its previous activity, then these two activities: D and F make up the transaction T2, and F should be put into QNT, and before we put it, the queue QNT will be checked, if QNT already contains F. So far it is not yet contained, so we put F in QNT. We go on traversing the BPEL process, get the activity E from QNT, and call the procedure AnalysisActivity(a) to analyse the activity E. The call procedure AnalysisActivity(E) is analogous to AnalysisActivity(D). The only difference is, the activity F is already in QNT this time, we could not push F in QNT again.

The algorithm 4 presents how to check, whether an activity has join relation or not. It is simple, we should just check, if the number of incoming links of the current activity more than one, then we can assert, the current activity has join relation. In our example, the activity F is a join activity, it has two incoming links. This function will return a Boolean value.

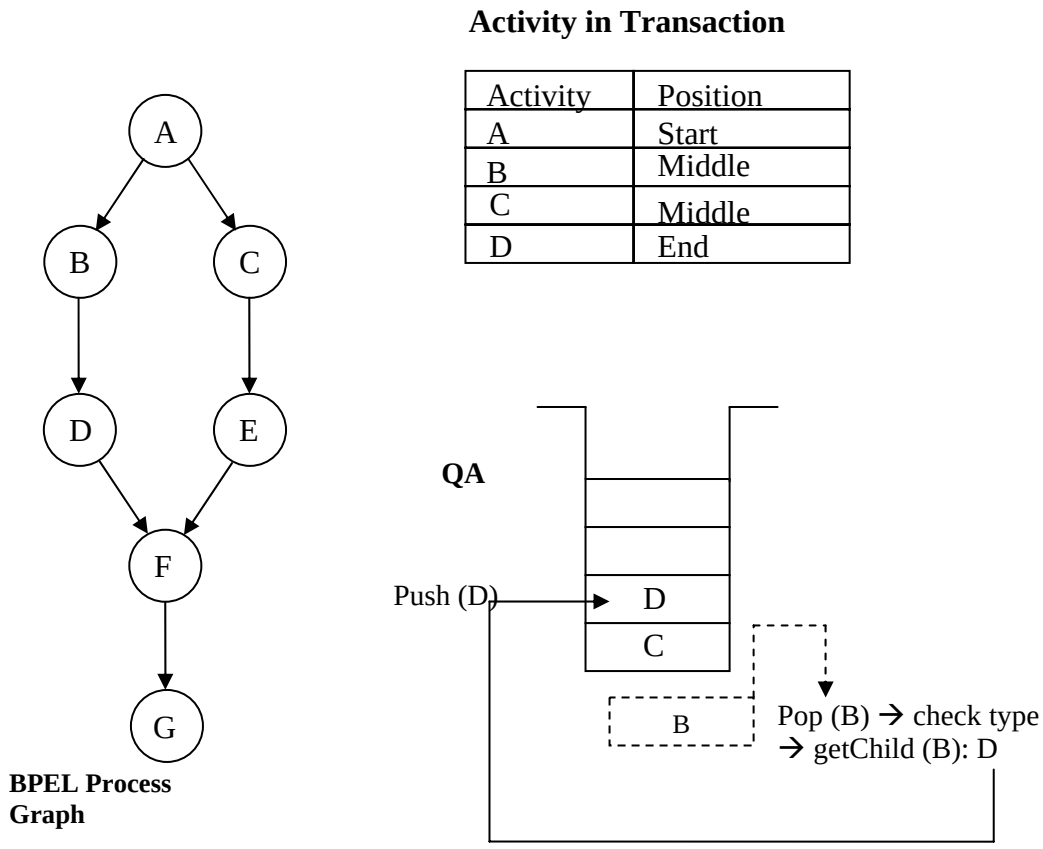


Figure 4.2 handling child activity

```
// function JoinActivity: to check an activity, whether it is Join or not
1:   JoinActivity(a)
2:      if |IN (a)| >1, then
3:         return true
4:      else
5:        return false
6:      end if
```

Algorithm 4: JoinActivity

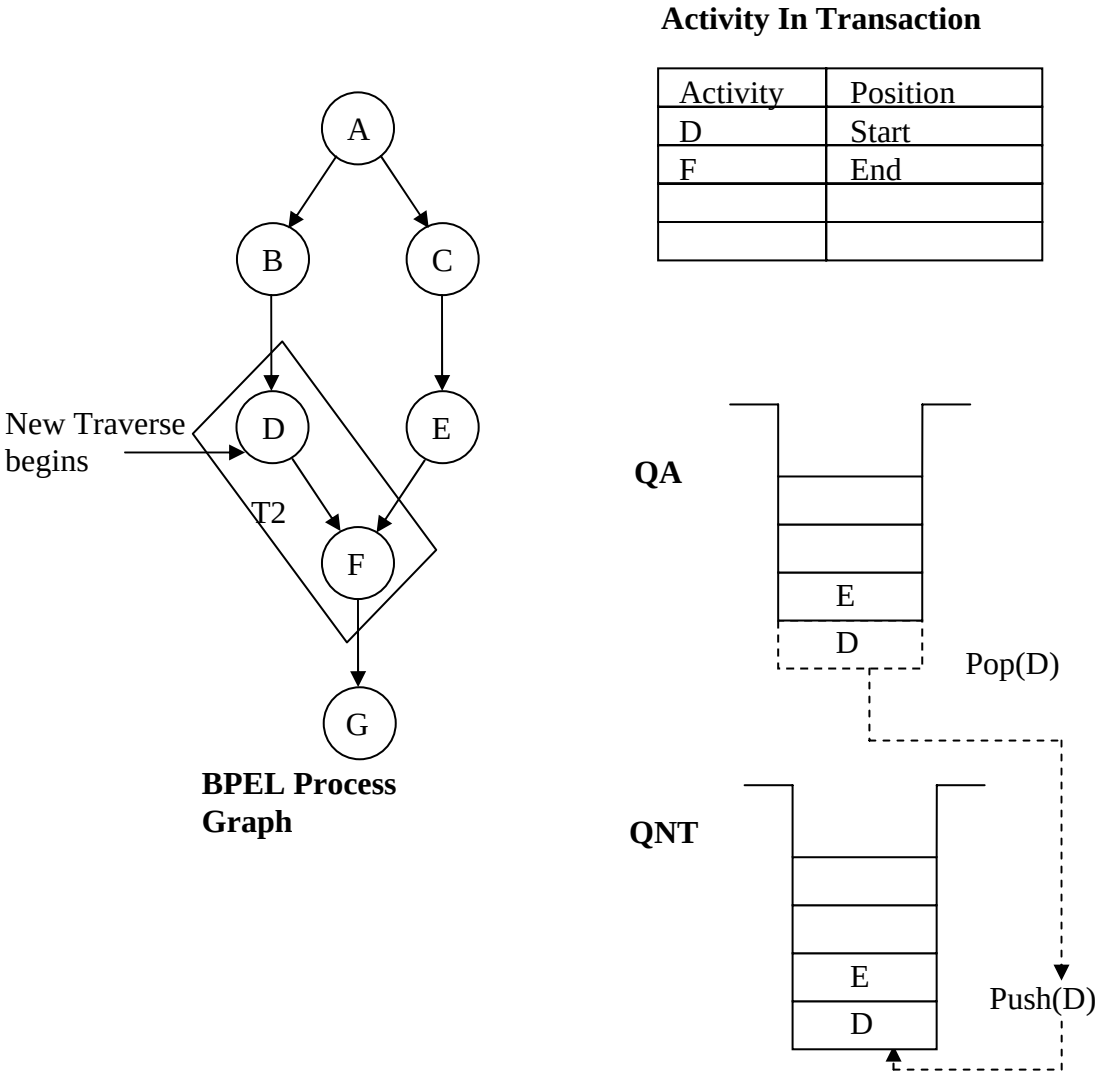


Figure 4.3 first round of traverse

The following algorithm (Algorithm 5.) shows how to create Transaction from the result of the analyse procedure. This function is called when the current traverse finishes, the current transaction should be built. It means, while a new transaction is needed to generat, the function createTransaction() should be called. From the set "ActivityInTransaction" we get all the activities in each transaction separately, and these activities will to be divided into three classes for each transaction according to their position: the start activity, the middle activities and end activities. As above mentioned, in this thesis we take no consideration into the transaction with multiple start activities, so the start activity class of each transaction contains

only one activity. These three classes play an important role in determining the transaction flow. We explain it in next section.

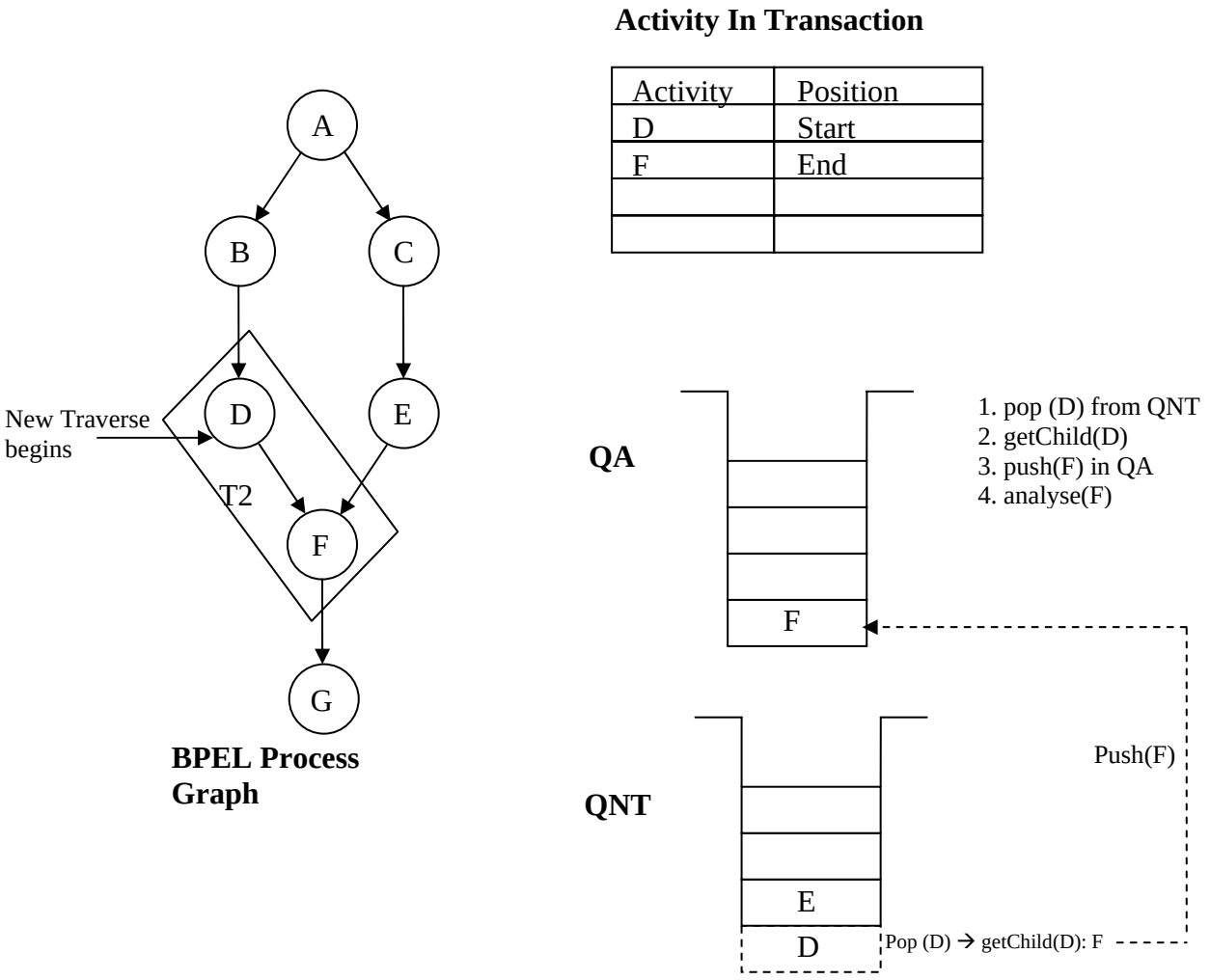


Figure 4.4 second round of traverse

In the algorithm 5, line 7 to 13 checks the activity position, and according to the position, the current transaction ID and corresponding activity name will be put into the set. After calling this function, we should clear the set: ActivityInTransaction, in order to provide space for storing the required data for next transaction. We could backup the set in another place for the further use, because our final goal is the flow execution plan, in this plan the transaction is made up of a set of activities and a set of variables, so we need to save these activities for generating the flow execution plan.

After calling the function createTransaction(), the queue QNT will be checked, if it is empty, then the traverse of the whole BPEL process finishes. At the end of the traverse, both QA and QNT must be empty. We get a set of transactions as well as start activity class and end activities class of each transaction (As figure 4.6 shows).

```
1:  Function creatTransaction()
2:   var: activityPosition
3:  Init:  TransactionStartActivity= [null,null] TransactionEndActivity=
           [null,null],  TransactionMiddleActivity=  [null,null]
4:
5:    // get each element in the ActivityInTransaction
6:         foreach element in activityInTransaction
7:            if  activityPosition= Start   then
8:               TransactionStartActivity = TransactionStartActivity U
                                          [TransactionID, activtiyName]
9:            else if   activityPosition= End  then
10:             TransactionEndActivity = TransactionEndActivity U
                                         [TransactionID, activtiyName]
11:           else if   activityPosition=Middle  then
12:              TransactionMiddleActivity = TransactionMiddleActivity U
                                           [TransactionID, activtiyName]
13:           end if
14:      end for
15:    // at the end, the activityInTransaction must be cleared,
          in order to save the activities in next transaction.
16:    clear activityInTransaction
17:  End function
```

Algorithm 5: create Transaction

Now we have discussed the general process of an analysis of an activity as well build a transaction and its corresponded activities. In next section we will specify, how to create transaction flow with these generated transactions.

`

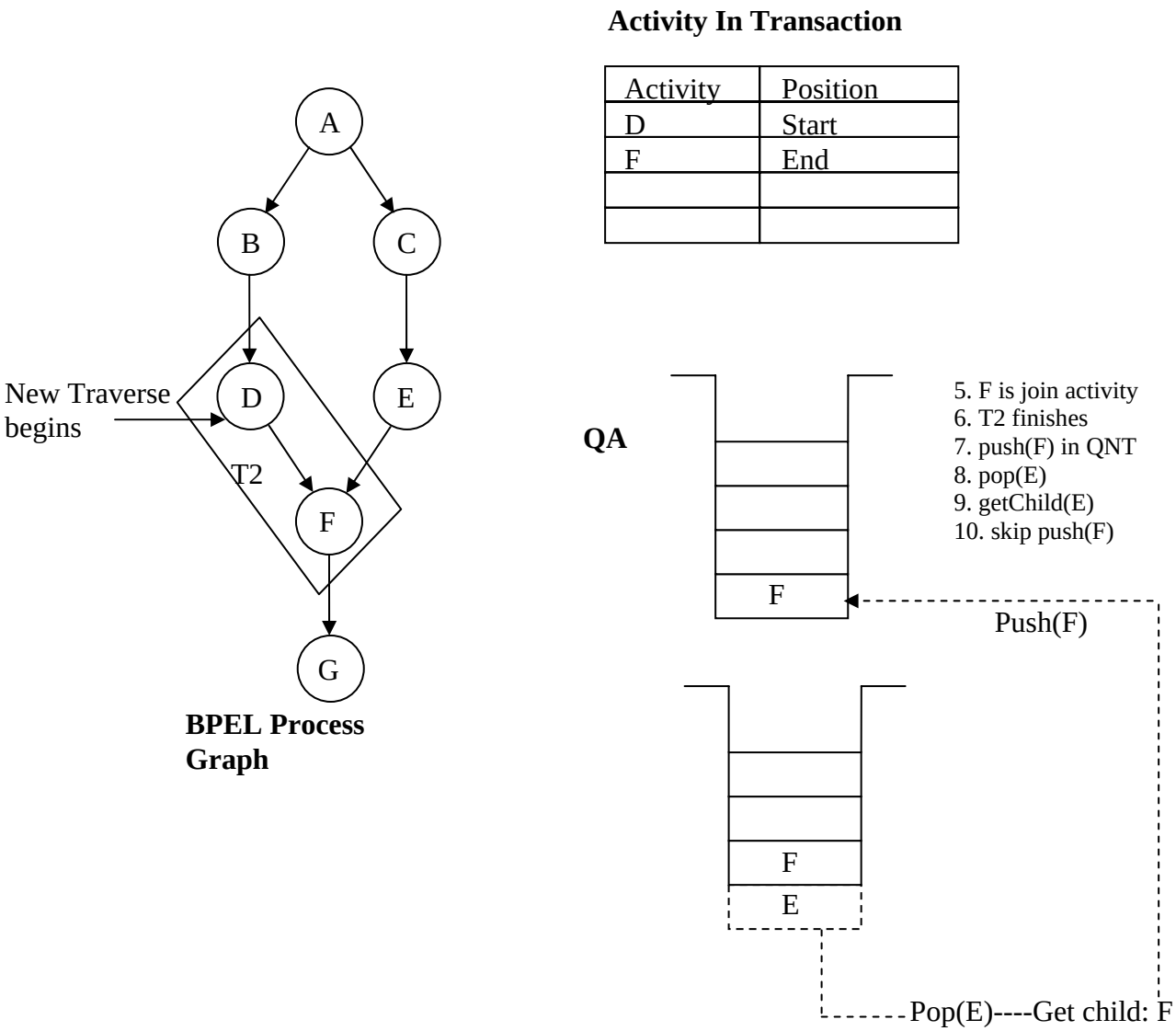
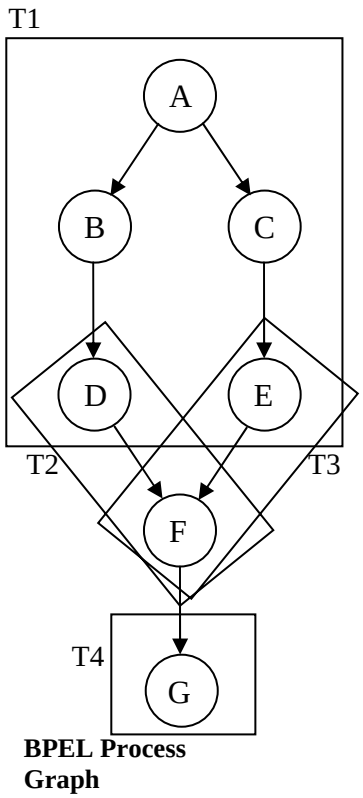


Figure 4.5 third round of the traverse



Tranasction Id	Start activity	End activity
T1	A	D,E
T2	D	F
T3	E	F
T4	F	G

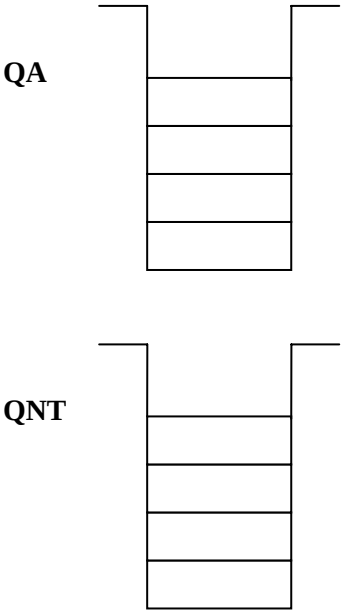


Figure 4.6 create transactions

4.3 General Variable Analyse

In this section we introduce how to analyse the variable in each transaction. There exist three activity types, which can create variable: receive, assign and synchronous invoke. As a result we just need to focus on these three activity types. Algorithm 6 shows the function variable_handling, this function handled all the variables in general, it will be called after creating a transaction. All the activities in this transaction will be looped, and special handling function will be called according to different activity types. And for a receive activity, an additional step should be taken, we should also check the position of this receive activity, only when its position is “Start”, the variable that created by this receive activity will be handled (line 10). In other words, if the receive activity is a boundary activity, its position is “End”. It will not be considered in this round of traverse. There exist three different functions for handling variable in different activities:

- VariableInAssignHandling(Activity)
- VariableInInvokeHandling(Activity)
- VariableInReceiveHandling (Activity)

```
1:  function Variable_Handling
2:  for each transaction Ti (i=1, 2, …, n) ∈Transaction set {
3:       for each activity Ai (i=1, 2, …, n)  ∈Activity set In Ti {
4:           if (Ai.type==Assign) {
5:                VariableInAssignHandling (Ai)
6:           }
7:           else if (Ai.type==Invoke) {
8:               VariableInInvokeHandling (Ai)
9:           }
10:          else if ((Ai.type==Receive)&&(Ai.position==Start)) {
11:              VariableInReceiveHandling (Ai)
12:         }
13:     end for
14:  end for
15: end function
```

Algorithm 6: Handling of Variable

In the following sections these three functions will be introduced.

4.3.1 Handling of Variable in Assign Activity

It is important to observe how an assign activity is used to transfer information between data variables. In chapter 2 we have introduced the assign activity, as we known, the standard element in assign activity is <copy>, through <copy> the data will be copied from a variable to another variable. If we want to get all the variables in an assign activity, we should focus on each <copy> element. Algorithm 7 presents the handling procedure of the variables in an assign activity. As the algorithm shows, line 2 to 39 loops through each <copy>element. As we in chapter 2 explained, there exist some variants of the sources (“from-spec”) and the destination (“to-spec”), so when we loop through the <copy> element, we should check at first, which is the type of the source or destination, and then different function will be called respectively according to the types.

4.3.2 Handling of Variable in Invoke- and Receive Activity

The handling of the variable in Invoke- and receive activity is relative simple. We just need to get the input variable and output variable in an invoke activity, and the created variable in an receive activity during traverse. So we need not extra design an algorithm. We can see the handling process in 5.4.1 and 5.4.2.

```
1:  function VariableInAssignHandling (Activity Ai)
2:   for each copy ∈assign activity {
3:
4:      // determine the types of the from and to specifications
5:       getFromSpecType();
6:       getToSpecType();
7:
8:       // get from variable
9:         if (FromSpecType== VARIABLE) {
10:             getFromVariable();
11:         }
12:       else if (FromSpectType== VARIABLE_PART) {
13:            getFromVariable();
14:            getPartNumber();
15:            getPartname();
16:            getFromVariablePart();
17:       }
18:      else if (FromSpecType== VARIABLE_PART_QUERY) {
19:      getVariable();
20:      getPartName();
21:         getVariablePart(partName);
22:         getPosition();
23:         getQuery();
24:         getNumberOfParts();
25:      }
26:      else if (FromSpecType== PARTERLINK) {
27:          getPartnerLink();
28:      }
29:      //get the to variable
30:      if (ToVariable == variable){
31:             getTovariable();
32:        }
33:       else if (Tovariable==VariablePart){
34:            get ToVariable();
35:            getPartNumber();
36:            getPartname();
37:            getToVariablePart();
38:       }
39:    end for
40:  end function
```

Algorithm 7: Handling of variable in an Assign activity

4.3.3 Set the properties of a Variable

We have specified in chapter 3 the properties of a variable in a transaction. Now we will explain how to set these properties. The following algorithm shows the process of set the state of a variable in transaction.

All the variables in a transaction can be retrieved after handling of each activity. Line 4 begins to loop through each variable for each transaction in the transaction set. We compare every variable in a transaction in the transaction set to every variable in the other transactions in the transaction set. If there is a variable, which is contained by Ti, and at the same time, it also be contained by another transaction Tj (i is unequal to j, and i is smaller as j), then we need to set the END of the variable in Ti as “KEEP”, because this variable will be used again in transaction Tj. This will be done by line 11 to 13.

The following things should be noticed for a join activity. If a variable is created by a join activity and the joinable state variable is still “false”(line 15), we can not make sure that the start state of this variable is “NEW” or ”OLD”, so we can only set the start state of this

variable to “UNKNOWN”(line 16). And because the joinable state variable is still “false”, it means, not all the incoming links of this join activity have entered, so the end state of this variable should be set to “KEEP”, this variable will be needed later (line 17). And when the joinable state variable has already been set to “true”, the start state should be set to “OLD”, and the end state should be set to “DELETE”.

For the variable, whose creation activity is an assign activity, and at the same time is not a join activity. We should set the start state of this variable to “NEW” (line 26 to 29). And if the assign activity is also a join activity, then this creation activity should be taken as a join activity instead of an assign activity. And if a variable is created by a receive activity, we should check the activity position in transaction firstly, if the activity position of this activity is “START”, so we can draw a conclusion that the start state of this variable is “NEW” (line 23), this receive activity maybe the start activity of the whole process. Otherwise if the activity position is “END”, this activity locates on the boundary of a transaction, then the variable, which created by this activity, will not be considered in this transaction. In other words, the receive activity, whose position is END, will only be read instead of being processed. So it will not appear in the flow execution plan. If the variable is not conforming to any of the case mentioned above, the end state of it will be set to “DELETE” (line 29).

```
1:    function setStateOfVariable()
2:     //for each transaction Tk in Transaction set (T1, T2,…, Tn), |Tran|
is the
       number of transaction in transaction set
3:
4:     for (int k=1; k<|Tran|;k++){
5:
6:      // for each variable Vi in Transaction Tk
7:       for (int i=1; i <n;i++){
8:        for (int m=k+1; m <|Tran|; m++) {
9:         // for each variable Va in Transaction Tm, m=k+1, "|Var|" is the
number of Variable in a transaction
10:        for (int a= 1; a<|Var|; a++) {
11:           if (Vi=Va) {
12:            Vi.endState=Keep
13:           }
14:
15:         else if ((Vi.CreationActivity=JoinActivity) &&Joinable=false) {
16:            Vi.StartState=Unknown
17:            Vi.EndState=Keep
18:          }
19:         else if ((Vi.CreationActivity=JoinActivity) &&Joinable=true) {
20:             Vi.StartState=Old
21:             Vi.EndState=Delete
22:         }
23:         else if ((Vi.CreationActivity=ReceiveActivity) &&
                   (Vi.CreationActivity.Position=Start)) {
24:             Vi.StartState= New
25:         }
26:         else if ((Vi.CreationActivity=AssignActivity) &&
                   (Vi.CreationActivity !=JoinActivity)) {
27:
28:             Vi.StartState= New
29:         }
30:         else {
31:                   Vi.endState=Delete
32:                   Vi.StartState=New
33:         }
34:       end for
35:    end for
36:   end for
37: end for
38: end function
```

Algorithm 8: set the state of variables

4.4 Create Transaction Flow

After the activity has been analysed, procedure "ActivityAnalyse" is executed, both QA and QNT are empty. We get all the necessary information about each activity, and a set of transactions, we define it as TransactionSet = (T1, T2, T3,…, Tn), and then we can generate the transaction flow with these information.

The following algorithm presents how to create transaction flow from a transaction set. Suppose we have now the transaction set (T1, T2, T3,…, Tn) as input, line 6 to 15 loop through all the element in this transaction set, for each element in this set, we compare its end activities to the start activity of all the other transactions in this transaction set. If the end activity of Ti is same as the start activity of Tj, and "i" is unequal to "j", then we can conclude that Ti is the source transaction of Tj, and Tj is exactly the target transaction of Ti.

Once the source transactions plus target transactions for each transaction are given, the transaction flow is created.

```
1:   Procedure createTransactionFlow()
2:   Init: SourceTransaction= null, TargetTransaction= null
3:   Input: < T1, T2… Tn>
4:         result from Algorithm 5: Ti.TransactionStartActivity,
                                    Tj. TransactionEndActivity
5:   // for each transaction Ti, i= 1, 2, …, n in < T1, T2… Tn>
6:      for (i=1; i<n; i++) {
7:   // for all the other transaction Tj, i!=j, if .TransactionEndActivity=
       Tj.TransactionStartActivity, then Ti is the source transaction of Tj
8:         for (j=i+1; j<=n; j++) {
9:             if (Ti.TransactionEndActivity= Tj.TransactionStartActivity)
10:            then
11:             Ti.SourceTransaction = Ti.SourceTransaction U {Tj}
12:             Tj.TargetTransaction = Tj.TargetTransaction U {Ti}
13:            end if
14:        end for
15:     end for
16:
17:  end procedure createTransactionFlow ()
```

Algorithm 9: create Transaction Flow

For example, we assume that at the end of the traverse we get a transaction set {T1, T2 …Ti… Tn}. This result will be used as the argument in the method createFlowExecutionPlan. It can be seen in section 5.6.

As the figure 4.6 shows, we find that:
T1. endActivities = {T2. stratActivity} U {T3.startActivity}, then we add T2 and T3 into the set: Source Transaction of T1, as the figure 4.6 shows: T1 is the source transaction of T2 and T3, T2 and T3 is the source transaction of T4, and at the same time T4 is the target transaction of T2 and T3. The transaction flow T1→ T2→ T3→ T4 is as the middle of the figure 4.7 shows.

All the algorithms described above base on the Transaction, which type is “long”, for the Transaction, which type is “ultimate”, is still to take some additional steps. The only difference between these two types is the handling of join activity. The join activity is no longer the boundary activity for ultimate transaction type, only the non-finished join activity will be taken as boundary activity (more detail see section 2.4). So we should pay more attention to the join activity.

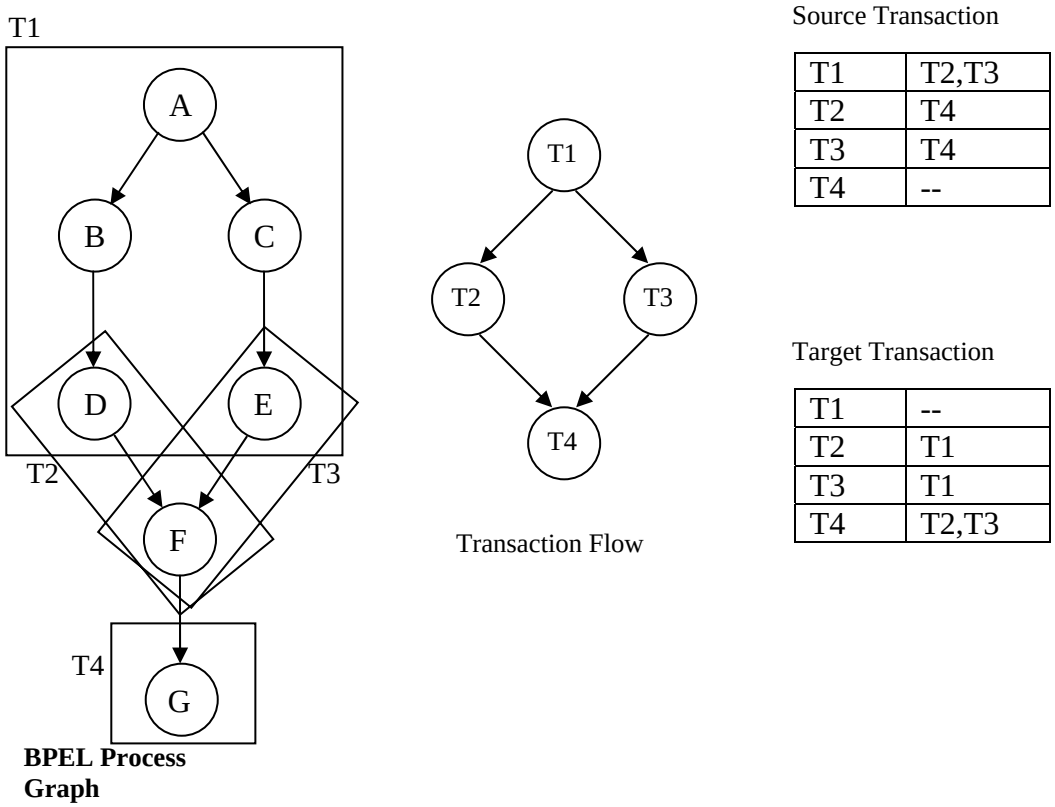


Figure 4.7 create transaction flow

The algorithm 10 shows how to deal with the join activity, when the transaction is of the type “ultimate”. The core idea of this algorithm is: find out the transaction in the transaction set, whose start activity is a join activity, then go back to the previous transaction of this transaction, a Boolean variable will be needed to indicate if a join activity is “joinable”. We define here a Boolean variable called “joinable” for the end activity of the previous transaction, and set the value of “joinable” as “true”. As the algorithm 10 shows, line 3 initialled the “joinable” as “false”, line 5 to line 7 loops through each transaction in the transaction set to get the start activity of each transaction, and store the current start activity in a temporal variable tempStartActivity. The activity type of this temporal variable will be checked, if it is a join activity, then we get the previous transaction of current transaction (line 9 to 13). The previous transaction of the current transaction will be stored in a temporal variable called “tempPreviousTransaction”, then we focus on this transaction, its end activity will be retrieved, and the joinable state variable of the end activity will be set to “true” (line15 to 16). It indicates all the incoming links of the join activity have already entered, the current transaction finishes.

```
1:   function set_Joinable
2:    // the start activity of each transaction will be checked, if it is
         a join activity, then set joinable = true
3:   Boolean joinable=false
4:   //for each transaction Ti in transaction set (T1, T2,…, Tn)
5:   for (i=1; i<n; i++) {
6:      // get the start activity of the Ti
7:        tempStartActivity =Ti.StartActivity
8:      // if the start activity is a join activity
9:      if (JoinActivity(tempStartActivity)) then
10:        //get current Transaction
11:       tempTransaction =Ti
12:       // get the previous transaction of the tempTransaction
13        tempPreviousTransaction= tempTransaction. previousTransaction
14:       // get the end activity of the target transaction
15:       joinableActivity = tempPreviousTransaction.endActivity
16:       set Joinable = ture
17:      End if
18:   }
19:   End for
20:  End function
```

Algorithm 10. set state variable of a join activity

Up to now, all the important algorithms that will be used in this work have been introduced. In next chapter the implementation of these algorithms will be provided.

5 Implementation

Implementing the algorithm is a necessary task.

For reading BPEL processes, there are libraries already at hand. We retrieve the Model Loader, and get all the information about BPEL processes.

To generate the flow execution plan, we need to construct a new package in Administration Package, it is named as Optimizer.

The implementation will be realized in the following steps:
Firstly, we construct a session bean called OptimizerBean.
We need the following code:

- Session bean class (OptimizerBean)
- Local interface(OptimizerLocal)

Secondly, we construct a new class in Optimizer package, called TransactionBuilder.
The implementation is based on the algorithms, which we have discussed in chapter 4.

The following sections cover these topics:

- Section 5.1 Implementation background and related technologies:
- Section 5.2 session bean
- Section 5.3 build transactions
- Section 5.4 activity analysis
- Section 5.5 create transaction flow
- Section 5.6 create flow execution plan
- Section 5.7 test optimizer

5.1 Implementation Background and Related Technologies

In this section a brief description of the implementation background and the related implementation technologies that are used in this thesis is given.

5.1.1 Test environment

IBM Rational Application Developer for WebSphere Software (RAD) will be used as test environment. It is a commercial Eclipse-based integrated development environment (IDE), made by IBM's Rational Software division. It can be used as visually design, construct, test, and deploy Web services, portals, and Java Enterprise Edition (JEE) applications [RAD].

The Stuttgarter Workflowmachine(SWoM) implements the basic processing structure of a set of server components.

There exist two prerequisites to install SWoM:
IBM WebSphere Application Server: running environment for SWoM
IBM DB2: the environment for databases used to run SWoM

IBM WebSphere Application Server (WAS) is a software application server. WAS is built using open standards such as Java EE, XML, and Web Services. It is supported on the following platforms: Windows, AIX, Linux, Solaris, i/OS and z/OS. It has been chosen as the foundation for the workflow management system [IBM].

The IBM DB2 Enterprise Server Edition will be used as database system for SWoM. The IBM DB2 is a relational model database server developed by IBM. In this work DB2 Express Edition V9.5 will be used. For running SWoM a set of local databases will be in DB 2 created, as the following list shows [SWoM]:

- SWOMSY: consists all the information for the operation of the WfMS, such as users and configuration properties
- SWOMAU: audit databse, storing all the audit information, such as all the activities and events of the current or finished process model instance
- SWOMBT: buildtime database, storing the process model information. This database to facilitate the import/export of a process model and the deployment of those objects so that process instance can be created.
- SWOMRT: run time database, storing all the process instance information, it consists two major component:
    - Navigator: navigates through a process instance using the information provided by the associated process model
    - Service invoker: call the web services

5.1.2 XML Schema

An XML Schema describes the structure of an XML document [W3C]. It is a language describing valid schemas of XML documents, used for message exchanges. An XML Schema:

- defines elements and attributes that can appear in a document
- defines which elements are child elements
- defines the order of child elements
- defines the number of child elements
- defines whether an element is empty or can include text
- defines data types for elements and attributes
- defines default and fixed values for elements and attributes

Elements are the primary ingredients of an XML schema and can be declared using the <xsd: element>. The element declaration defines the element name, content model, and allowable attributes and data types for each element type. The <xsd: element> element either denotes an element declaration, defining a named element and associating that element with a type, or it is a reference to such a declaration [Skonnard 02]. The element declarations that appear as immediate descendants of the <xsd: schema> element are treated as global element declarations and can be referenced from anywhere within the schema document or from other schemas. For example, Listing 5.1 depicts an XML schema fragment. In this schema FlowExecutionPlan is defined globally and in fact constitutes the root element in this schema, its type attribute defines the type of the element being declared. This attribute is a reference to a simple or complex type. Here is a complex type: tFep. The <complexType> element is used to define structured types. An element is considered to be a complex type if it contains child element and/or attributes. Complex type definitions appear as children of an <xsd: schema> element, as listing 5.1 shows, the complex type tFep. it contains child elements transactionFlow and executionOptions.

There exist other components of an element declaration in this schema, such as minOccurs and maxOccurs. These two attributes control occurrence behaviour, they specify the minimum and maximum number of times this element may appear within the context where it is declared in a valid XML document. In generally, an element is required to appear when the

value of minOccurs is 1 or more [PA06]. The default value for both the minOccurs and the maxOccurs attributes is 1. In Listing 5.1, the value of these two attributes is the default value. It means there is only one transaction flow in a Flow execution plan of a BPEL process.

```
<xsd:element name="FlowExecutionPlan" type="tns:tFep">
      <xsd:annotation>
            <xsd:documentation>
                  This is the root element for a SWoM Flow Execution
                  Plan
            </xsd:documentation>
      </xsd:annotation>
</xsd:element>

<xsd:complexType name="tFep">
      <xsd:all>
            <xsd:element name="transactionFlow" type="tns:tTransactionFlow"
                  minOccurs="1" maxOccurs="1" />
            <xsd:element name="executionOptions" type="tns:tExecutionOptions"
                  minOccurs="1" maxOccurs="1" />
      </xsd:all>
      <xsd:attribute name="processModel" type="xsd:NCName"
            use="required" />
</xsd:complexType>
```

Listing 5.1 xml schema fragment of fep

5.1.3 XML Beans

XMLBeans is a technology for accessing XML by binding it to Java types [XMLBeans]. Any XMLBeans project begins with an xml schema. And an XML schema is stored in an XSD file generally. The setup of XMLBeans is not complex, more information can be found in [XMLBeans]. When we successfully set up XMLBeans, the following steps are very simple. We just need to open a DOS prompt and change to the directory containing the .XSD file, then issue the following commond to compile the xml schema:
scomp -out fep.jar fep.xsd

The figure 5.1 shows this simply process generally.

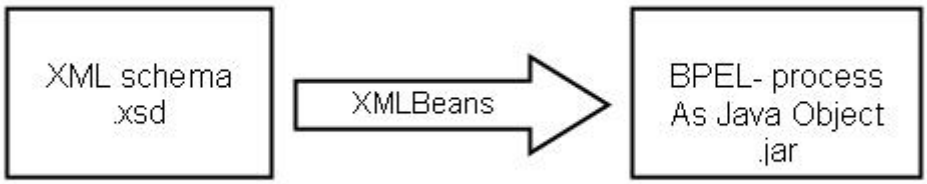


Figure 5.1 xml beans

For example, we create a “fep.jar” from a “fep.xsd” file with this command. If we see following window, it shows the compile is successfully, we have a JAR file containing the classes necessary to process.

```
Loading schema file fep.xsd
Time to build schema type system: 0.851 seconds
Time to generate code: 1.983 seconds
Compiled types to fep.jar
```

When we compile this schema: fep.xsd, for the root element FlowExecutionPlan, it is also a global element, the compiler generated an interface: “FlowExecutionPlanDocument”. For each of the complex element a separate interface will be generated.

The following listing is a fragment of the xml schema fep.xsd. This fragment describes the root element of the xml schema, the compiler can generate XMLBeans interface whose name ends with “Document”: FlowExecutionPlanDocument. The FlowExecutionPlanDocument interface represents the flow execution plan document that contains the root element – FlowExecutionPlan.

The list 5.2 shows a complex type element “tTransaction”. The compiler generates for this complex type an interface “TTransaction”.

```
<xsd:complexType name="tTransaction">
      <xsd:all>
            <xsd:element name="executionProperties" type="tns:tExecutionProperties"
                  minOccurs="1" maxOccurs="1" />
            <xsd:element name="sources" type="tns:tSources"
                  minOccurs="0" maxOccurs="1" />
            <xsd:element name="targets" type="tns:tTargets"
                  minOccurs="0" maxOccurs="1" />
            <xsd:element name="cacheHandling" type="tns:tCacheHandling"
                  minOccurs="0" maxOccurs="1" />
            <xsd:element name="activitiesInTransaction" type="tns:tActivitiesInTransaction"
                  minOccurs="0" maxOccurs="1" />
            <xsd:element name="variablesInTransaction" type="tns:tVariablesInTransaction"
                  minOccurs="0" maxOccurs="1" />
      </xsd:all>
      <xsd:attribute name="ID" type="xsd:string" use="required" />
</xsd:complexType>
```

Listing 5.2 complex type of xml schema

On the other side, we can create an XML file with XMLBeans. It will be specified in section 5.5.
The following sections focus on the implementation.

5.2 Session Bean

A session bean is an enterprise edition specification. According to [SessionBean], a session bean “represents a single client inside the J2EE server. To access an application that is deployed on the server, the client invokes the session bean's methods.”

There are two types of session beans: stateful and stateless. As its name suggests, a stateful session bean retains the state for each unique client-bean session, each client has an instance. Because the client interacts ("talks") with its bean, this state is often called the conversational state. And a stateless session bean does not need maintain a conversational state for a particular client. The state is only retained during the invocation of the method of a stateless

bean. The state disappears when the method finishes. From the view of performance, the stateful session bean need not to be written to secondary storage, so it offers better performance than stateful beans.

In this work, the bean's state has no data for a specific client, and the bean fetches from the buildtime database a set of read- only data that is used by clients, as a result we create a stateless session bean for the optimizer.

All session beans require a session bean class. The session bean that we created for this work is called "Optimizer", we create a class with the same name: OptimizerBean.class, and a local interface named "OptimizerLocal". All the methods that be defined in the interface "OptimizerLocal" should be realized. Following method will be defined in the interface "OptimizerLocal".

- optimizerProcessModel(String credentials, String PMID)
- imporFlowExecutionPlan(String credentials, String PMID)
- exportFlowExecutionPlan(String credentials, String PMID)

In this work only the first method is used. With this method the process model will be retrieved, and the class TransactionBuilder with process model will be invoked, as the listing 5.3 presents.

```
/**
 * Optimize request
 */

public void optimizeProcessModel(String credentials, String PMID)
        throws UserException, InternalErrorException {

            ....

      /*
       * Get process model
       */

      processModel = modelCache.getProcessModel(PMID) ;

      /*
       * Load transaction builder
       */

      transactionBuilder = new TransactionBuilder() ;

      /*
       * Invoke transaction builder with process model
       */

      transactionBuilder.buildTransactions(processModel) ;
  }
```

Listing 5.3 function optimizeProcessModel

5.3 Build Transactions

5.3.1 Class Queue

As chapter 4 introduced, a FIFO queue is very important, it should be used to store the child activity of each activity and the activities, which run a new round of traverse. The structure of a FIFO queue is not fresh, in a FIFO queue, the first element added to the queue will be the first one to be removed. The queue should also have the ability to pop up the top element of the queue, push an element into the queue, and this queue should also be able to check, whether it is empty.

Figure 5.2 shows the class queue. The value of the integer variables “begin” and “end” will be initialized as “-1”, they are pointer in a sense. When a new element is pushed into the queue, the value of “begin” will plus one. And when the top element of the queue is removed, the value of “end” will plus one. In this queue structure, the top element will not be really removed. As a result, when the value of “begin” and “end” is equal, the FIFO queue is already “empty”. The Boolean value “true” will be returned.

```
public class Queue {
      Queue queue;
      private Activity[] activities;
    private int begin = -1;
    private int end = -1;

   public Queue(int size){
         activities = new Activity[size];
   }

   public void push(Activity value){
         activities[++begin] = value;
   }

   public Activity pop(){
        return activities[++end];

   }

   public boolean isEmpty(){
        return begin == end;
   }
}
```

Figure 5.2 class queue

5.3.2 Class TransactionBuilder

Before we introduce the concrete process of build transactions, a class “ProcessModel” will be introduced, it plays important role in the whole implementation. There exist a class named “ProcessModels” in package “iaas.swom.admin.datamanagement”. This class provides methods to work with the process models stored in the buildtime database. And class ProcessModel is in the package “iaas.swom.sharedejbs.ModelCache.BPELConstruct”, it is the Cache representation of process models reflecting the structure and the attributes of a business process. This class provides methods to retrieve all the information about the BPEL process. For example, with the method “getProcessDeploymentDescriptor” we can get the process deployment descriptor, and with the method “getRootActivity” we can get the initial start activity of a BPEL process, it could be see in next page.

The figure 5.3 shows an overview of the class TransactionBuilder. This class contains five methods, getRootActivity(), getTransactionType(), analyseActivity(), createTransactionFlow() and createFlowExecutionPlan(). The core method is analyseActivity (), it will be described in detail in next section.

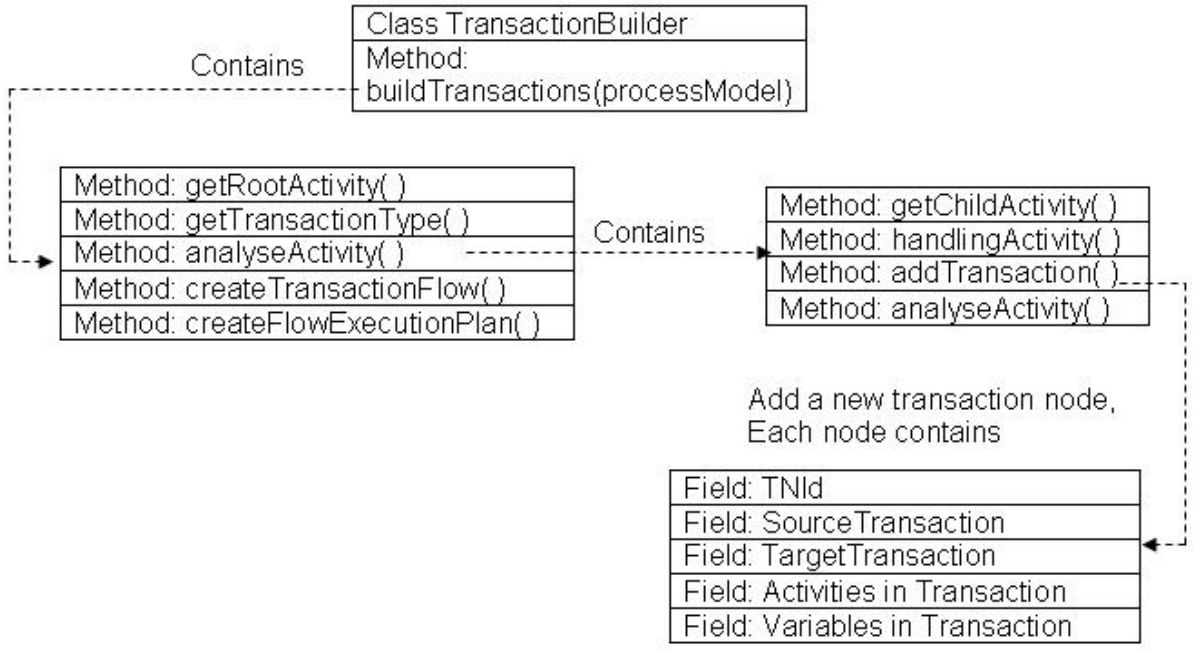


Figure 5.3 class diagram of TransactionBuilder

Each process is associated with a process deployment descriptor. After importing a process model, we can retrieve general information about the deployment descriptor. The process deployment descriptor provides the definition of the transaction type, and then the current transaction type can be retrieved via the process deployment descriptor. As mentioned in last chapter, the default transaction type is ultimate, hence we focus only on this transaction type in this implementation.

Process analysis starts at the process element. For container activities, analysis is started for all the directly contained activities. When any activity analysis has finished, successors to the activity will be analyzed: If the activity has outgoing BPEL links, the links are followed, analyzed and analysis of the target activities starts. If the activity is part of a sequence, the successor is analyzed. If the activity is contained in another activity, the container might need to do some data collecting and finish analysis.

An initial start activity is the start activity that caused a particular process instance to be instantiated. We get the initial start activity of a BPEL process through the process model class, this initial start activity is either a flow activity or a sequence activity. If the activity is a flow activity, we get the actual root activity within the first activity, and the actual root activity must be a receive activity. And if the activity is a sequence activity, we get the start activity of the first activity. As we in chapter 4 discussed, the analysis starts with this actual root activity, by expanding the root activity we can traverse the whole process. Firstly, we need to get the root activity within the initial start activity. Root activity is an activity, which is the top level activity of a process model as it can be found in the appropriate BPEL file. In other words, it really starts the process, it must be a receive activity.
Listing 5.4 shows the process of finding the root activity within the initial start activity. The initial start activity will be getting through class ProcessModel. After getting the initial start

activity, its type will be checked at first. If the activity type is “Flow”, then the first element of the start activities will be getting. If the activity type is “Sequence”, we just need to get the start activity of the initial start activity.

```
/*
 * Get deployment descriptor
 */

ProcessDeploymentDescriptor processDeploymentDescriptor =
    processModel.getProcessDeploymentDescriptor() ;
/*
 * get the root activity - either a flow or a sequence
 */
Activity initialActivity = processModel.getRootActivity();

/*
 * Get the actual first activity within the root activity
 * - must be a receive
 */

if (initialActivity.getActivityType() == ActivityTypes.FLOW) {
    rootActivity = ((FlowActivity)initialActivity).getStartActivities().firstElement() ;
    ActivityInTransaction.add(rootActivity);

} else {
    rootActivity = ((SequenceActivity)initialActivity).getStartActivity();
    ActivityInTransaction.add(rootActivity);
}
```

Listing 5.4 get root activity

From the root activity we begin to build the first transaction actually. We define a class, called “TransactionNode”. Each object of this class represents a transaction node in the future transaction flow.

Through this class, we can set for each transaction its start activity and end activities, its source transaction and target transaction, all the activities in each transaction, and all the variables in each transaction. If we need to add a new transaction element in transaction set, we should create a new TransactionNode object. The class diagram is as figure 5.4 presents. Variable TNId identifies the transaction node Id.

Class TransactionNode
+TNId: String +sources: ArrayList <String> +targets: ArrayList <String> +startActivity: Activity +endActivity: ArrayList <Activity> +activities: HashMap<Activity, String> +variableList: ArrayList<VariableNode>
+getTNId: String +getSources : ArrayList <String> +getTargets: ArrayList <String> +getStartActivity :Activity +getEndActivity: Activity +getActivities: HashMap<Activity, String>

Fgiure 5.4 TransactionNode class diagram

5.4 Activity Analysis

In this section we specify how to implement the analyse process of an activity. As figure 5.3 shows, the method analyseActivity() contains four sub methods. Before we define these sub methods we should make the following definitions:

- Queue ChildActivity: a FIFO queue, all the child activities of an activity will be put into the queue as long as it has a child activity.
- ArrayList ActivityInTransaction: an ArrayList, stores all the activities in each transaction. It means, the activity will be added in this ArrayList once it is read from the BPEL process. After storing it, the analysis can begin. As presented in chapter 4, we should get the child Activity of the activity at first. Listing 5.5 shows the method getChildActivity (Activity). If the first activity is a flow activity, then we follow the outgoing links of the activity and get the target activity of each links and put them into the queue "ChildActivity". And if the number of the outgoing links is null, it means the first activity is a sequence activity, we have no outgoing links to follow. We can only get the successor activity of the sequence activity. In this case, the number of outgoing links will be checked firstly, if it is null, then the successor activity of the current activity will be put into the queue "ChildActivity".
- Queue newTraverse: a FIFO queue, stores the activities, which run later a new round of traverse. For example, a receive activity or a join activity.
- LinkedHashMap ActivityInTransaction: a LinkedHashMap, stores all the activities in each transaction as well as their position in transaction. The key of the HashMap is activity, the value is the position.
- LinkedHashMap VariableInActivity: a LinkedHashMap, stores all the variables and its creation activity. The key is variable, and the value is the creation activity.

```
public void getChildActivity(Activity activity){
    int NumberOfOutgoingLinks=activity.getNumberOutgoingLinks();
    if(NumberOfOutgoingLinks!=0){
        for(int n=0;n<NumberOfOutgoingLinks;n++){
            ChildActivity.push(activity.getOutgoingLinks().get(n).getTargetActivity());
            }
    }
    else if((NumberOfOutgoingLinks==0)&&(activity.getSuccessorActivity()!=null)){
        ChildActivity.push(activity.getSuccessorActivity());
    }
}
```

Listing 5.5 get child activity

During the handling process of an activity, we check the activity type at first, and then special procedure will be called according to the activity type.

Before handling these activity as well as their variables, we should import the following interfaces (see Listing 5.6). These imported interfaces will be used in the following implementation.

```
Import iaas.swom.sharedejbs.ModelCache.activityTypes.AssignActivity;
Import iaas.swom.sharedejbs.ModelCache.activityTypes.FlowActivity;
Import iaas.swom.sharedejbs.ModelCache.activityTypes.InvokeActivity;
Import iaas.swom.sharedejbs.ModelCache.activityTypes.ReceiveActivity;
Import iaas.swom.sharedejbs.ModelCache.assign.Copy;
Import iaas.swom.sharedejbs.ModelCache.assign.FromSpecTypes;
Import iaas.swom.sharedejbs.ModelCache.assign.ToSpecTypes;
Import iaas.swom.sharedejbs.ModelCache.assign.variablePartSpec;
Import iaas.swom.sharedejbs.ModelCache.assign.VariableSpec;
```

Listing 5.6 imported interfaces

5.4.1 Handling Receive Activity

If an activity's type is receive, as we known, the receive activity starts a process, it creates a new instance for this process. At the same time, it is a fest transaction boundary activity, it terminates the current transaction. So for such an activity (except the start activity), we put it direct into the queue NewTraverse instead of getting its child activity.

Listing 5.7 shows the handling procedure of a receive activity.

```
if(receiveActivity(activity)){
      NewTraverse.push(activity);
      ActivityInTransaction.add(activity);

      while(!ChildActivity.isEmpty()){
            Activity notCheckedActivity = ChildActivity.pop();
            ActivityInTransaction.add(notCheckedActivity);
            NewTraverse.push(notCheckedActivity);
         }

}
```

Listing 5.7 handling procedure for receive activity

During handling the receive activity, we should also observe, what variable will be created by this activity.

There exist an interface called ReceiveActivity, it has been imported before. It provides a methods that we can get the variable of the receive activity directly. We just need to converse the type of the current activity from "Activity" into the "ReceiveActivity" type, then we can call all the methods of the interface ReceiveActivity. After calling the method getVariable(), we get the created variable. Then we put it and the receive activity into the LinkedHashMap VariableInActivity for further use. The receive activity is naturally the creation activity of the variable.

```
if(receiveActivity(thisActivity)&&(b==0)){
    receiveCreatedVariable=((ReceiveActivity)thisActivity).getVariable();
    if(receiveCreatedVariable!=null){
        //HashMap variableInActivity for storing variable and creation activity
        variableInActivity.put(receiveCreatedVariable, thisActivity);       }

}
```

Listing 5.8 variable in receive activity

5.4.2 Handling Invoke Activity

Because an invoke activity has no effect on the transaction boundary, we just need to take consideration into the input and output variables. We can get them through the interface InvokeActivity. See listing 5.9. “thisActivity” represents the current invoke activity.

```
inputVariable = (( InvokeActivity) thisActivity).getInputVariable( );
outputVariable = (( InvokeActivity) thisActivity).getOutputVariable( );
```

Listing 5.9 get variable in invoke activity

Then these variables and their creation activity will be put into the LinkedHashMap VariableInActivity, the key is the variable and the value is the creation activity. This is analogous to receive activity.

5.4.3 Handling Assign Activity

As the invoke activity, an assign activity is also not transaction boundary activity. We need only focus on its variable. But its variable could not be gotten directly, as in section 2.3 mentioned, the assign activity copies data from one location to another with a list of copy statements. An assign activity may contain one or more copy elements. As Listing 5.10 and 5.11 shows, we get each copy via the interface Copy. This interface represents the BPEL copy construct. It contains a from-spec and a to-spec attribute. The assign activity copies the value specified in the from-spec attribute to the location specified in the to-spec attribute. We loop through each copy element, determine the types of the from and to specifications and run special procedure according to the different types. In this work only two types: VARIABLE and VARIABLE_PART are considered. Two ArrayLists fromVariableInAssignActivity and toVariableInAssignActivity are defined for storing the from variable and to variable separately. Sometimes two different from- variables are copied into one to- variable, or one from- variable is copied into two different to- variables. In these cases, before we store these variables, we should check the ArrayList at first, whether the variable is already stored. If yes, then the step of “storing” should be omitted.

```
for (Copy currentCopy : ((AssignActivity)thisActivity).getCopies()) {

    /*
     * Determine the types of the from and to specifications
     */

    fromSpecType = currentCopy.getFromSpecType();
    toSpecType = currentCopy.getToSpecType();

    /*
     * Load the information that is used as the source
     */
    switch (fromSpecType) {
    case (FromSpecTypes.VARIABLE):
        fromVariable = ((VariableSpec) currentCopy.getFromSpec()).getVariable();

    if((fromVariable!=null)&&fromVariableInAssignActivity.isEmpty()){
        fromVariableInAssignActivity.add(fromVariable);
    }
        break;
    case (FromSpecTypes.VARIABLE_PART):

            fromVariable = ((VariableSpec) currentCopy.getFromSpec()).getVariable();
    if(fromVariable!=null&&fromVariableInAssignActivity.isEmpty()){
                for(int nrFrom=0;nrFrom<fromVariableInAssignActivity.size();nrFrom++){
                    if(fromVariable!=fromVariableInAssignActivity.get(nrFrom))
                        fromVariableInAssignActivity.add(fromVariable);
                }

     }
     partName = ((VariablePartSpec) currentCopy.getFromSpec()).getPartName();
     fromVariablePart = fromVariable.getVariablePart(partName);
     partPosition = fromVariablePart.getPosition();
     numberOfParts = fromVariable.getNumberOfParts();
     break;
  }
```

Listing 5.10 from variable

```
switch (toSpecType) {
case (ToSpecTypes.VARIABLE):
    toVariable = ((VariableSpec) currentCopy.getToSpec()).getVariable();
if(toVariable!=null&&(toVariableInAssignActivity.isEmpty())){
    toVariableInAssignActivity.add(toVariable);
    for(int toVar=0; toVar<toVariableInAssignActivity.size();toVar++){
        variableInActivity.put(toVariableInAssignActivity.get(toVar),thisActivity);
    }
}
    break;
case (ToSpecTypes.VARIABLE_PART):
    toVariable = ((VariableSpec) currentCopy.getToSpec()).getVariable();
if(toVariable!=null&&(!toVariableInAssignActivity.isEmpty())){
    for(int nrTo=0;nrTo<toVariableInAssignActivity.size();nrTo++){
        if(toVariable!=toVariableInAssignActivity.get(nrTo)){
            toVariableInAssignActivity.add(toVariable);
            for(int toVar=0; toVar<toVariableInAssignActivity.size();toVar++){
                toVariableInAssignActivityMap.put(toVariableInAssignActivity.get(toVar),thisActivity);
            }
        }
    }

}
partName = ((VariablePartSpec) currentCopy.getToSpec()).getPartName() ;
toVariablePart = toVariable.getVariablePart(partName);
partPosition = toVariablePart.getPosition();
numberOfParts = toVariable.getNumberOfParts();
    break;
}
```

Listing 5.11 to variable

5.4.4 Class VariableNode

Figure 5.5 shows the class diagram of the class VariableNode. During traverse, once a variable is created, a new VariableNode object will be newly created.

With this class we could have the creation activity as well as the state of the variable.

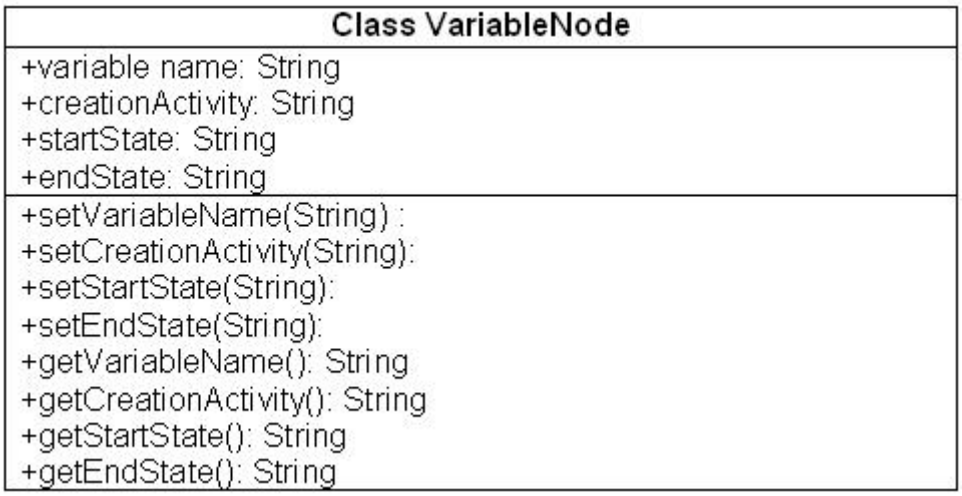


Figure 5.5 class VariableNode

5.5 Create Transaction Flow

As the transaction set has been created from a process model, we could use it to create the transaction flow. As we described in chapter 4, the transaction flow will be created by comparing of the start activity and end activities of each transaction. Base on this idea the implementation is relative simple.
Listing 5.12 presents the method createTransactionFlow(). This method will be called as long as the queue ChildActivity and NewTraverse are empty.

We have stored all the objects of the class TransactionNode in the arrayList-- TransactionList except the sources and the targets of each transaction, the value of these fields are still empty. It means we have all the transaction nodes. We just need to create edges to bind those transaction nodes. We could loop through each object in TransactionList, retrieve the start activity and end activities of each object, and compare them to determine the sources and targets.

After running this method the transaction flow is generated, and all required information about a transaction is stored in TransactionList. With this information we could create flow execution plan, it will be explained in next section.

```
public void createTransactionFlow(){

    for(int t=0;t<TransactionList.size();t++){

        TransactionNode tempTransaction=TransactionList.get(t);
        ArrayList<Activity> TEndActivities= tempTransaction.getEndActivities();


        for(int e=0;e<TEndActivities.size();e++){

            for(int s=t+1;s<TransactionList.size();s++){

                Activity TStartActivity = TransactionList.get(s).getStartActivity();
                if(TEndActivities.get(e)==TStartActivity){

                    tempTransaction.getSources().add(TransactionList.get(s).getTNID());

                        TransactionList.get(s).getTargets().add(tempTransaction.getTNID());

                }

            }
        }
    }
}
```

Listing 5.12 create transaction flow

5.6 Create and Save Flow Execution Plan

The final purpose of this thesis is to create a flow execution plan, and save it in an xml file, so it is important to converse the object of the TransactionNode into an XML instance.

In this section will be introduced, how to create the flow execution plan in xml format with XML Beans and how to store it.

5.6.1 Create Flow Execution Plan

It is easy to create an xml file with XMLBeans. We simply create instances of the object that represents the data and add them to the object that represents the document.

Before creating the flow execution plan, we need to reference following classes (see Lisitng 5.13). These classes directly map to the element names defined in the xml schema: fep.xsd. And these classes are created from the xml schema "fep.xsd", which have been explained in section 5.1.

Once we have imported the relevant classes, we can create the desired XML file. It means, we create an instance of the relevant document and the relevant section within the document with the appropriate classes, it can be see later.

```
Import iaas.swom.shared.fep.FlowExecutionPlanDocument;
Import iaas.swom.shared.fep.TActivitiesInTransaction;
Import iaas.swom.shared.fep.TActivityInTransaction;
Import iaas.swom.shared.fep.TSource;
Import iaas.swom.shared.fep.TSources;
Import iaas.swom.shared.fep.TTarget;
Import iaas.swom.shared.fep.TTargets;
Import iaas.swom.shared.fep.TTransaction;
Import iaas.swom.shared.fep.TTransactions;
Import iaas.swom.shared.fep.TVariableInTransaction;
Import iaas.swom.shared.fep.TVariableInTransaction;
Import iaas.swom.shared.fep.TVariablesInTransaction;
```

Listing 5.13 the relevant classes of flow execution plan

The FlowExecutionPlanDocument interface represents the flow execution plan document that contains the root FlowExecutionPlan element. XMLBeans creates a special "document" type for global element types. This document type provides a way to set and get the value of the underlying types, here represented by "transactionFlow", "transactions" and "transaction". The FlowExecutionPlan is considered as a global element, it can be referenced from anywhere else in the schema.

The xml file is created by the FlowExecutionPlanDocument Factory, calling it with the newInstance() method. Then we can use the addNewFlowExecutionPlan() method on this object to create a new FlowExecutionPlan section. To add a new transaction flow to the FlowExecutionPlan section, we go on to call the method addNewTransactionFlow(). We can continually use the add-series method up to addnewTransaction () section. Then we can get the following xml file, as the figure shows.

The flow execution plan will be generated in following steps:

- add new FlowExecutionPlan
- add new TransactionFlow
- add new Transactions
- add new Transaction
- set the values of child element of the transaction type
- add new activities in transaction
- add new activity in transaction
- set the values of child element of the activity type
- add new variables in transaction
- add new variable in transaction
- set the values of the child element of the variable type
- set the source and target transactions

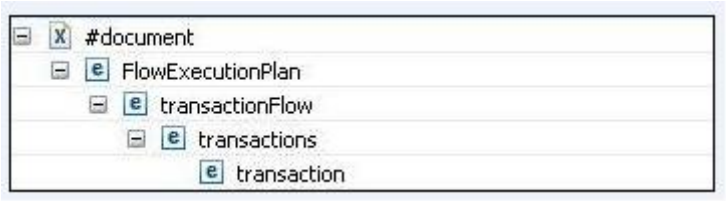


Figure 5.6 FlowExecutionPlanDocument Instance

To create more transaction section, we can continually use the add- series method.
The figure 5.7 describes the xml instance—transaction, it conforms the xml schema that we described in section 5.2. From this instance we can see that this complex type will be translated into a sequence of the underlying type: transaction, activitiesInTransaction, variablesInTransaction, sources and targets, as figure 5.7 shows. The element activitiesInTransaction stands for all the activities in current transaction, the element variablesInTransaction stands for all the variables in current transaction, the element sources stands for all the transactions, of which the current transaction is source, the element targets stands for all the transactions, of which the current transaction is target.

The interface TTransaction provides also the methods such as setSources, setTargets, setVariablesInTransaction, setActivitiesInTransaction to set the value of these elements. And also a method setId to set the attribute name: the transaction Id. And the TTransaction becomes the return type of a get method or the parameter of the set method.

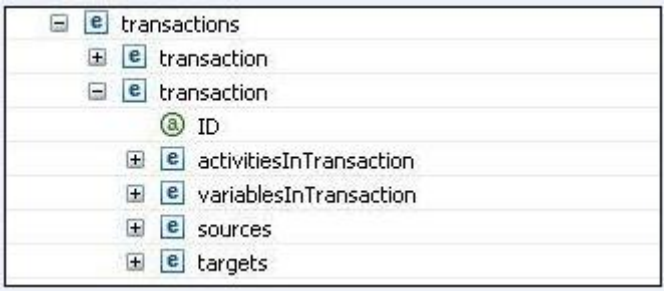


Figure 5.7 transctions

In order to add data to the Transaction section, we could simply call the appropriate ‘set’ methods that provided by the interface TTransaction.

The following bit of java code (Listing 5.14) illustrates how we might to use these interfaces to create a new instance of the flow execution plan and set the values of its child elements.

A new instance of the flow execution document: flowExecutionPlanDocument will be firstly created. And then a new instance of the transaction flow will be added into the flow execution plan instance. And now we could create our transaction flow, a new instance of the TTransaction will be added.

The method createFlowExecutionPlan takes the ArrayList-- TransactionList as argument. The list will be looped, once an element in TransactionList is read, a new instance of the TTransaction should be added. After adding a new instance of the TTransaction, we can set the value of its child elements, such as all the activities in this transaction, all the variable created and used in this transaction.

```
public void createFlowExecutionPlan(ArrayList<TransactionNode> list) {

    try{
        FlowExecutionPlanDocument flowExecutionPlanDocument = FlowExecutionPlanDocument.Factory.newInstance();
        TTransactionFlow transactionFlow  = flowExecutionPlanDocument.addNewFlowExecutionPlan().addNewTransactionFlow();
        TTransactions transactions=transactionFlow.addNewTransactions();

        for(int n=0;n<TransactionList.size();n++){
            TTransaction transaction =transactions.addNewTransaction();
            transaction.setID(TransactionList.get(n).getTNID());
            TActivitiesInTransaction activitiesInTransaction= transaction.addNewActivitiesInTransaction();

           /*
            * set activities in Transaction
            */

          for(Entry<Activity,String>activitiesInEachTransaction:TransactionList.get(n).getActivities().entrySet()){

             TActivityInTransaction activityInTransaction= activitiesInTransaction.addNewActivityInTransaction();

             activityInTransaction.setName(activitiesInEachTransaction.getKey().getName());

             activityInTransaction.setPosition(activitiesInEachTransaction.getValue().toString());
```

Listing 5.14 create flow execution plan

The whole flow execution plan is as the figure 5.8 shows.

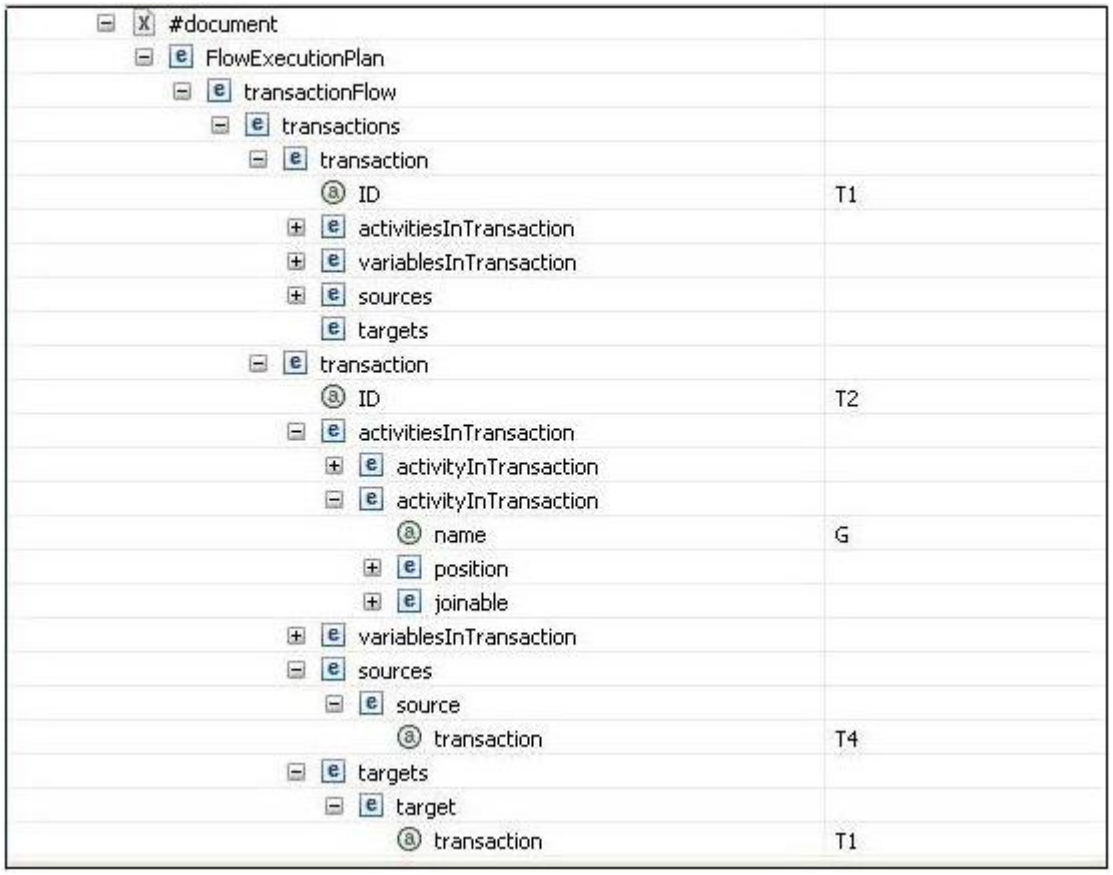


Figure 5.8 flow execution plan from the xml view

5.6.2 Save the Flow Execution Plan

When the flow execution plan has been created, we could use the method .save (String) of the FlowExecutionPlan interface to save it. When the optimization is successful, we can retrieve it in the place, where we store the flow execution plan. The following listing shows the method.

```
String fileName="E:\\fep.xml";

File xmlFile=new File(fileName);

flowExecutionPlanDocument.save(xmlFile);
```

Listing 5.15 save the flow execution plan

5.7 Test Optimizer

We can test the Optimizer direct in the administration interface of SWoM.

We start the WebSphere application server at first, when the server is started and synchronized we can open the main page of the SWoM under this address:
http://localhost:9080/AdministrationInterface/login.faces

We login as administrator in SWoM, and import a process model and deploy it. After successful deployment we can test the Optimizer. The following figures show the whole test course.

As the figure 5.9 shows, we import a process model (a valid SPAR file) into SWoM at first. All files necessary for the process model must be included in one SPAR file. The SPAR file containing the BPEL process model, the SPDD and the WSDL file. If the import successful completed, we can see a confirmation message: import successful (see Figure 5.10).

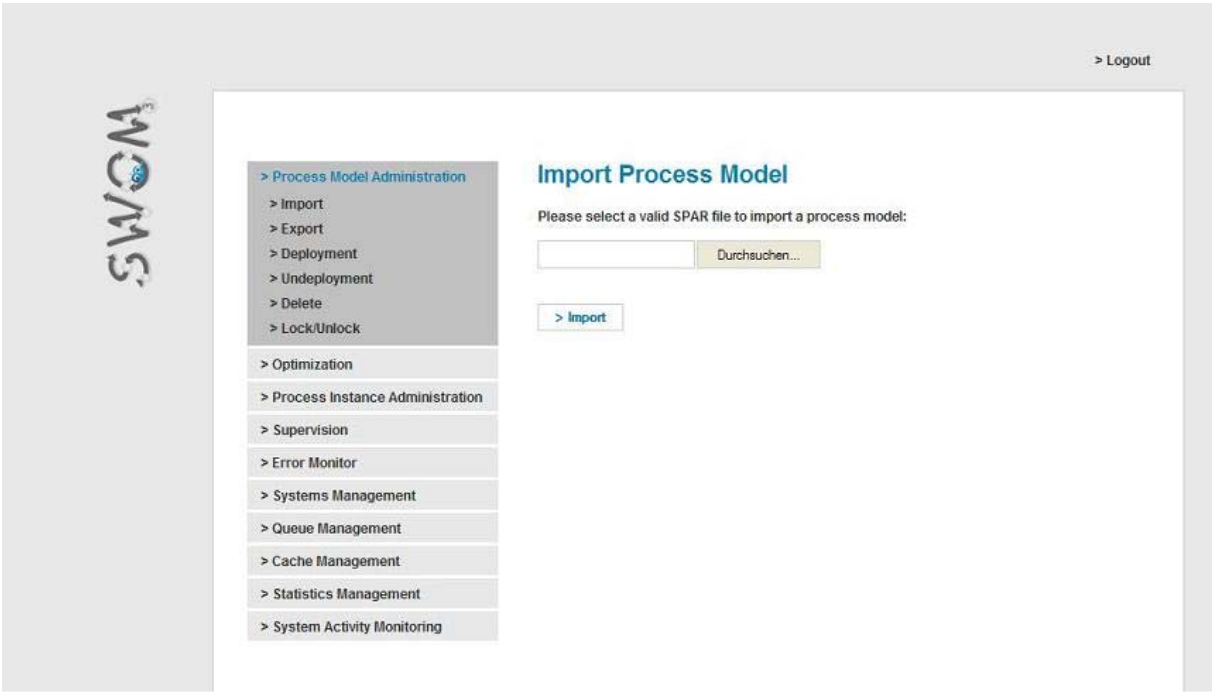


Figure 5.9 Import process model

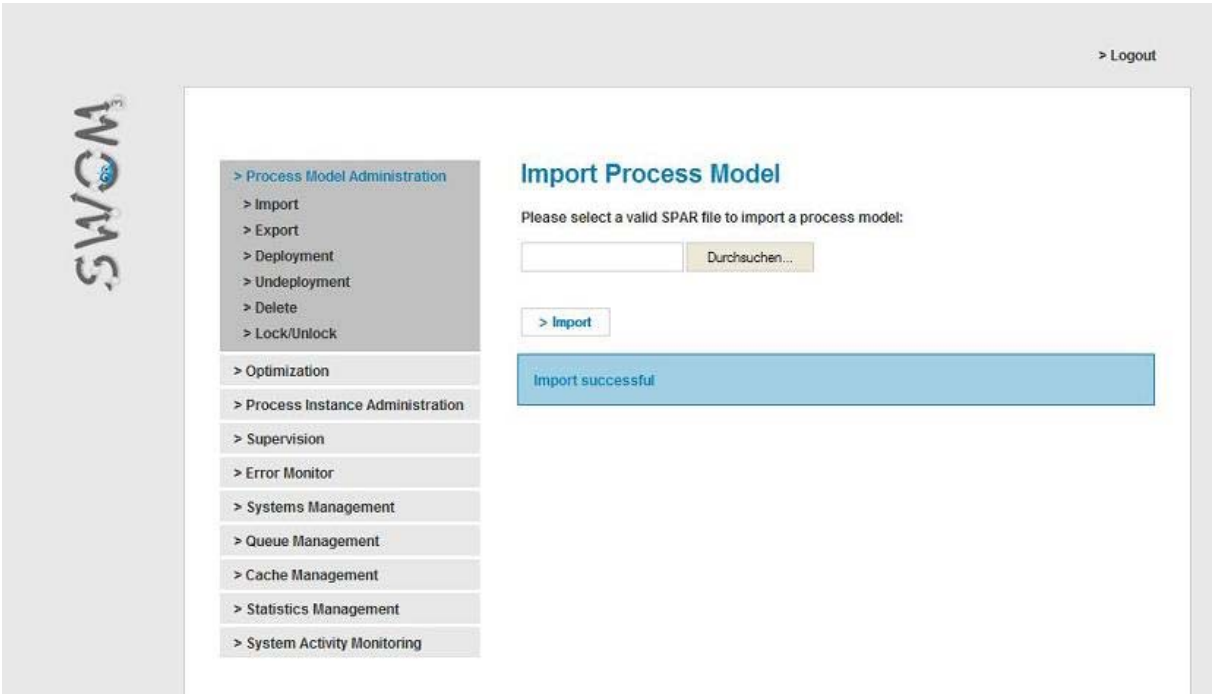


Figure 5.10 Import successful

After importing a process model we should deploy it. This can also be completed within the administration interface, as figure 5.11 shows. We just need to select a process model from the list and then click the button “Deployment” and select a process that we want to deploy. After successful deployment we can see the successful deployment, as the figure 5.12 presents.

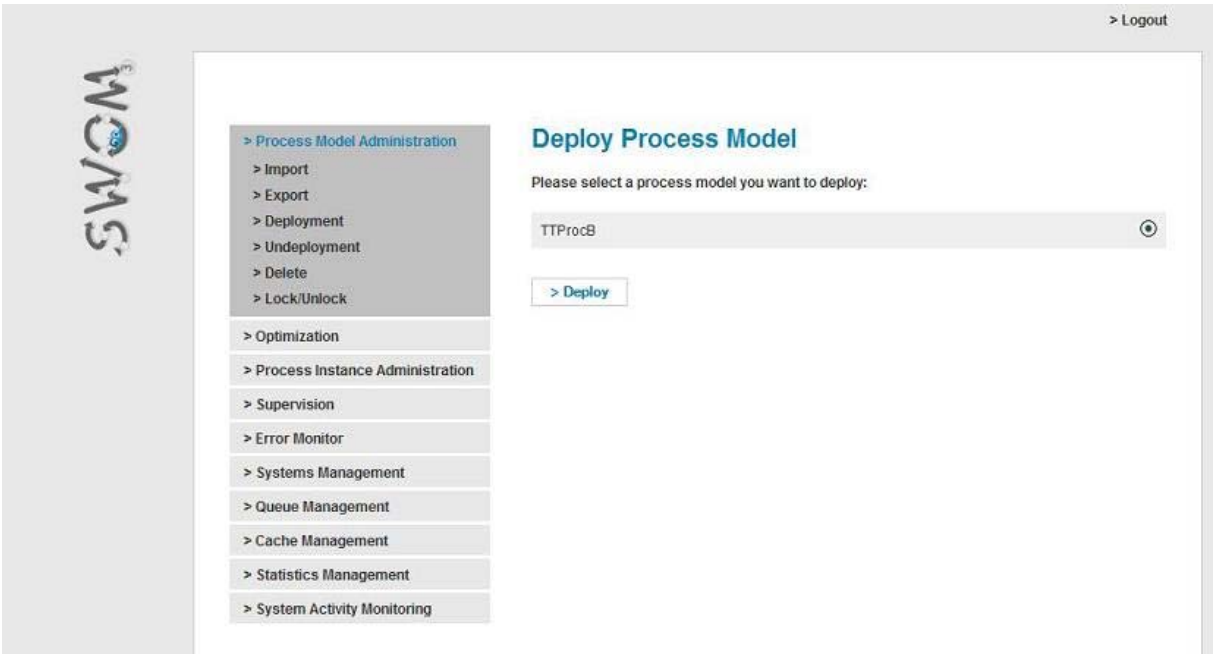


Figure 5.11 select Process to deploy

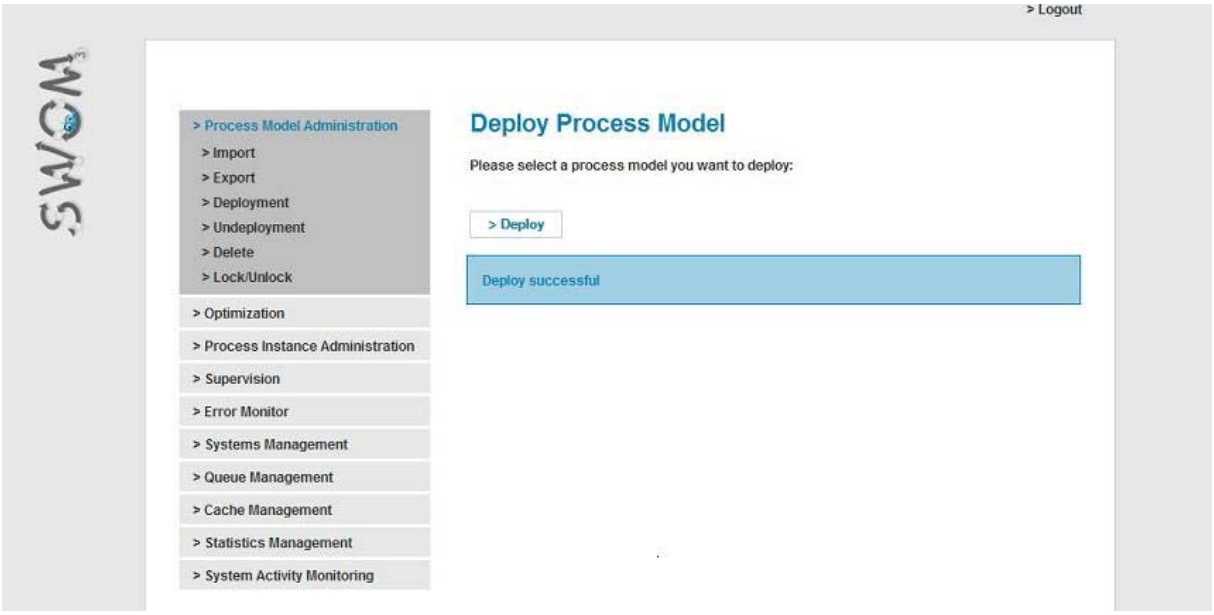


Figure 5.12 Deploy successful

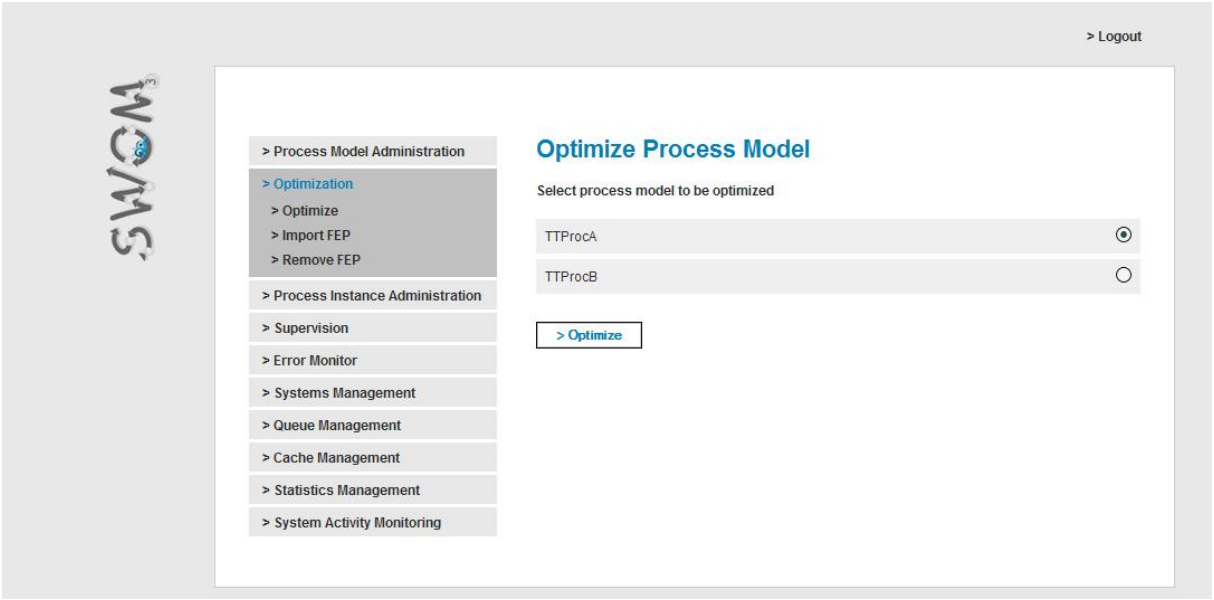


Figure 5.13 optimize process model

When a process model has been successful imported and deployed, we can this process model really optimize. We click the button “Optimization” on the left side of the administration interface to optimize the selected process model. If we can see the confirm information: optimization successful, as following figure (figure 5.14) presents. It means, the optimization succeeds, we have generated the flow execution plan. It means our task has finished. The generated flow execution plan will be saved in the place, where we previous defined. An example of the generated flow execution plan can be found in appendix.

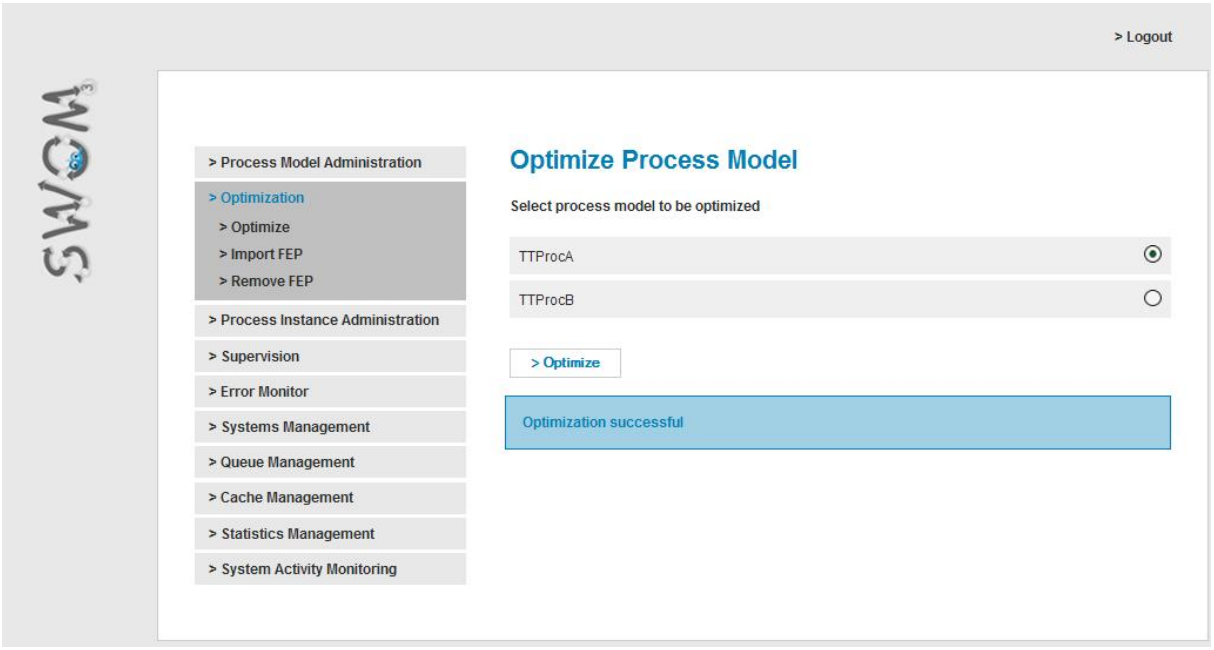


Figure 5.14 optimization successful

6 Summary and Outlook

The performance of a workflow engine can be significantly improved through appropriate optimization techniques. Appropriate test case showed that proper transaction types can improve the execution by a factor [Rol10]. This work was supposed to implement the transaction flow generator that generates transaction flows from WS-BPEL process model and stores the transaction flow execution plan as an XML file in the database. This transaction flow generator is expected to gain greater performance as a workflow optimizer.

I first gave an overview of the BPEL and the associated activities. Transaction flow and the associated transaction boundary criteria were introduced in the following chapter. Following was a general introduction of the flow execution plan. The algorithms of the transaction flow generator have been revealed completely, each procedure and function of the algorithm saw a full “walkthrough” detailing ideas and implications of each line of code. Following an implementation of the complete algorithm has also to be provided. Through the implementation the correctness of the algorithm was verified. Finally, the flow execution plan for a WS- BPEL process model could be generated with XMLBeans.

A whole flow execution plan consists two major pieces: the transaction flow and the execution options. In this work only the transaction flow has been generated, but the execution options were not considered.

The ultimate workflow optimizer can determine the best set of transactions automatically, so some additional analysis, for example, the analysis of the correlation sets, cache handling, and the execution options should still to be required in the future work.

Another interesting area of future work maybe includes the comparison of the algorithm to other control flow algorithms.

And we could apply the technology of transaction flows to another workflow engine, and compare the performance with SWoM.

Appendix: Example of Flow Execution Plan

```
<?xml version="1.0" encoding="UTF-8" ?>
<fep:FlowExecutionPlan xmlns:fep="http://shared.swom.iaas/fep/">
 <transactionFlow>
   <transactions>
     <transaction ID="T1">
        <activitiesInTransaction>
            <activityInTransaction name="A">
                 <position>START</position>
            </activityInTransaction>
            <activityInTransaction name="B">
                 <position>MIDDLE</position>
            </activityInTransaction>
            <activityInTransaction name="C">
                 <position>MIDDLE</position>
            </activityInTransaction>
            <activityInTransaction name="D">
                 <position>MIDDLE</position>
            </activityInTransaction>
            <activityInTransaction name="E">
                 <position>END</position>
            </activityInTransaction>
            <activityInTransaction name="F">
                 <position>END</position>
            </activityInTransaction>
       </activitiesInTransaction>
       <variablesInTransaction>
            <variableInTransaction name="InRequest">
                 <creationActivity>A</creationActivity>
                 <startState>NEW</startState>
                 <endState>DELETE</endState>
            </variableInTransaction>
            <variableInTransaction name="OutRequest1">
          <creationActivity>B</creationActivity>
          <startState>NEW</startState>
          <endState>DELETE</endState>
       </variableInTransaction>
            <variableInTransaction name="OutRequest2">
          <creationActivity>B</creationActivity>
          <startState>NEW</startState>
          <endState>DELETE</endState>
       </variableInTransaction>
   </variablesInTransaction>
       <sources>
       <source transaction="T2" />
       <source transaction="T3" />
   </sources>
   <targets />
  </transaction>
    <transaction ID="T2">
       <activitiesInTransaction>
          <activityInTransaction name="E">
         <position>START</position>
     </activityInTransaction>
          <activityInTransaction name="G">
         <position>END</position>
       <joinable>YES</joinable>
     </activityInTransaction>
   </activitiesInTransaction>
       <variablesInTransaction>
          <variableInTransaction name="InResponse1">
```

```
        <creationActivity>E</creationActivity>
        <startState>NEW</startState>
        <endState>KEEP</endState>
      </variableInTransaction>
         <variableInTransaction name="InInvoke">
        <creationActivity>G</creationActivity>
        <startState>UNKNOWN</startState>
        <endState>KEEP</endState>
      </variableInTransaction>
    </variablesInTransaction>
       <sources>
      <source transaction="T4" />
    </sources>
       <targets>
      <target transaction="T1" />
    </targets>
   </transaction>
     <transaction ID="T3">
<activitiesInTransaction>
  <activityInTransaction name="F">
    <position>START</position>
  </activityInTransaction>
  <activityInTransaction name="G">
    <position>END</position>
    <joinable>YES</joinable>
  </activityInTransaction>
  <activityInTransaction name="H">
    <position>END</position>
  </activityInTransaction>
</activitiesInTransaction>
<variablesInTransaction>
  <variableInTransaction name="InResponse2">
   <creationActivity>F</creationActivity>
   <startState>NEW</startState>
   <endState>KEEP</endState>
  </variableInTransaction>
  <variableInTransaction name="InInvoke">
   <creationActivity>G</creationActivity>
   <startState>OLD</startState>
   <endState>DELETE</endState>
  </variableInTransaction>
  <variableInTransaction name="OutInvoke">
    <creationActivity>H</creationActivity>
    <startState>NEW</startState>
    <endState>DELETE</endState>
   </variableInTransaction>
  </variablesInTransaction>
<sources>
    <source transaction="T4" />
</sources>
<targets>
    <target transaction="T1" />
</targets>
</transaction>
<transaction ID="T4">
 <activitiesInTransaction>
    <activityInTransaction name="G">
      <position>START</position>
      <joinable>YES</joinable>
    </activityInTransaction>
    <activityInTransaction name="H">
      <position>END</position>
    </activityInTransaction>
```

```
        <activityInTransaction name="I">
          <position>END</position>
        </activityInTransaction>
      </activitiesInTransaction>
      <variablesInTransaction>
        <variableInTransaction name="InInvoke">
          <creationActivity>G</creationActivity>
          <startState>OLD</startState>
          <endState>DELETE</endState>
      </variableInTransaction>
      <variableInTransaction name="OutInvoke">
          <creationActivity>H</creationActivity>
          <startState>NEW</startState>
          <endState>DELETE</endState>
      </variableInTransaction>
    </variablesInTransaction>
    <sources />
    <targets>
        <target transaction="T2" />
        <target transaction="T3" />
    </targets>
   </transaction>
  </transactions>
 </transactionFlow>
</fep:FlowExecutionPlan>
```

Bibliography

[FEP] SWoM Flow Execution Plan—Structure, Dieter Roller 2010, Internal Document

[IBM] Websphere Application Server, IBM
URL: http://www-01.ibm.com/software/webservers/appserv/was/

[LR00] Frank Leymann and Dieter Roller, Production Workflow. Concepts and Techniques, Prentice-Hall, 2000

[OAS07] The OASIS Committee: Web Services Business Process Execution Language (WSBPEL) Version 2.0; Apr. 2007
URL: http://www.oasis-open.org/committees/wsbpel/

[Pa06] Michael P. Papazoglou and Pieter M.A. Ribbers, e- Business: Organizational and Technical Foundations, Wiley 2006

[RAD] Rational Application Developer, IBM
URL: http://www01.ibm.com/software/awdtools/developer/application/

[Rol10] Dieter Roller, Transaction Flows, a Base for Workflow Engine Optimization, 2010, to be submitted

[SessionBean] What is a Session Bean? Sun
URL: http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts3.html

[Skonnard02] A. Skonnard, M.Gudgin. Essential XML Quick Reference. Addison Wesley, 2002

[SWoM] SWoM Administration Guide, study project, Internal Document

[XMLBeans] Welcome to XMLBeans, Apache
URL: http://xmlbeans.apache.org/

[W3C] XML Schema, W3C
URL: http://www.w3.org/XML/Schema

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[Wes07] M. Weske. Business Process Management: Concepts, Languages, Architectures. Springer-Verlag, 2007.

[Wor01] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1, 2001.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Place, Date) (Li Li)