Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstr.38
70569 Stuttgart

Diplomarbeit Nr. 2474

Optimization of the Runtime Database of a BPEL Engine

Yilian Shen

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dieter Roller
Begonnen am:	09.03.2006
Beendet am:	08.09.2006
CR Klassifikation:	C.1.3

Table of Contents

ABSTRACT 5

CREDIT 7

1 WSDL (WEB SERVICE DESCRIPTION LANGUAGE) 9

1.1 INTRODUCTION 9
1.2 XML 10
1.3 WSDL SERVICE DEFINITION 10
1.3.1 DOCUMENT NAMING AND LINKING 12
1.3.2 TYPES 13
1.3.3 MESSAGES 14
1.3.4 PORT TYPES 15
1.3.5 ONE-WAY OPERATION 16
1.3.6 REQUEST-RESPONSE OPERATION 16
1.3.7 BINDINGS 17
1.3.8 PORTS 18
1.4 EXAMPLE 18
1.5 SUMMARY 22

2 BPEL (WEB SERVICE BUSINESS PROCESS EXECUTION LANGUAGE) 23

2.1 INTRODUCTION 23
2.2 NOTATION 26
2.3 RELATIONSHIP WITH WSDL 27
2.4 DEFINE A PROCESS MODEL 28
2.5 BASIC AND COMPLEX ACTIVITIES IN BPEL 30
2.5.1 STRUCTURE OF A PROCESS 30
2.5.2 BASIC ACTIVITIES 33
2.5.3 COMPLEX ACTIVITIES 35
2.5.4 PARTNER LINKS 37
2.5.5 VARIABLE MANAGEMENT IN BPEL 37
2.6 SUMMARY 38

3 SWOM 39

3.1 OVERVIEW OF THE SYSTEM 39
3.1.1 REQUIREMENTS 39
3.1.1.1 Functional 39
3.1.1.2 Non-functional 39
3.1.2 ARCHITECTURE 39
3.2 PROCESS EXECUTION MODULE 41
3.2.1 NAVIGATOR 41
3.2.2 DATA MANAGER 43

3.2.3 AUDITING 43
3.3 SOFTWARE REQUIREMENTS 43
3.4 SUMMARY 44

4 OPTIMIZE DATABASE SCHEMAS 45

4.1 GOALS 45
4.2 DESIGN AND IMPLEMENTATION 45
4.2.1 VARIABLE STRUCTURE 45
4.2.2 CATEGORIZATION OF THE PROCESS MODELS 45
4.2.2.1 Simple Sequential Process Model 46
4.2.2.2 Other type of Process Models 69
4.3 TEST AND RESULTS 73
4.4 SUMMARY 73

5 SUMMARY AND OUTLOOK 74

5.1 CONCLUSION 74
5.2 OUTLOOK 74

APPENDIX A BPEL SCRIPTS FOR EACH TYPE OF PROCESS MODELS 75

APPENDIX B DDL FOR EACH TYPE OF PROCESS MODEL 79

LIST OF REFERENCE 87

Abstract

Nowadays E-Business turns out to be a trend. More and more business processes are executed in internet. People are fond of online banking or online booking system, which helps check banking balance, transferring money or booking flight tickets so easily and so quickly while they just need to sit cozily before their computers connected with internet, instead of going to the bank or traveling office in person where they might have to wait in a long boring queue. As more and more people prefer these web services while web resources, for example, net speed, space for storage, etc. are quite limited, thus, how to realize web services in a save and effective way becomes a hot researching field.

Web services, which are on the base of the service-oriented architecture framework, work as the foundation for modern distributed, heterogeneous business applications. The two-level programming model is used here to describe these kinds of services and it's also the characteristics of business workflow processes.

There are two distinct programming in workflow-based applications: one is programming in the large, where a process model describes the sequence which is composed of different activities being carried out; the other one is programming in the small which deals with the individual components. In the Web services environment, we use the Business Process Execution Language for Web Services (BPEL) to describe process model. The actual implementation of all activities, here so called web services, can be done in any language and programming model.

In BPEL, business processes are, in fact, the Web services, providing for a recursive aggregation model. A workflow management system (WFMS) which uses the BPEL specification is designed to manage the life cycle of business processes, to navigate through the associated process models, and to invoke the appropriate Web services, and we call it BPEL engine. The navigator is the core of the WFMS, which opens a transaction, receives an appropriate navigation request from a queue, retrieves all state information about the process instance and the associated process model from a database, carries out navigation, stores the new process state in the database, insert a message into a queue, and commits the transaction. The navigator typically works statelessly in a certain application server environment. The database operations are another main tasks by the navigator. They calculate necessary resources. And that's why we need to optimize BPEL engine's runtime database. The number of I/O operations and cycles for managing and fetching process model information should be reduces, so that the resources could be spared and optimized by caching them. The structure of the database where we store the

process state information should support all the capabilities of the BPEL language. And this works perfectly for those complicated process models which use most of the capabilities, while creating huge overhead for simple process models, for example, simple sequential process.

In this diploma thesis, a method that optimizes the storage layout for different complexities of process models is to be developed. For instance, for a simple sequential process model a single table may be sufficient. The method includes five tasks: 1. to define categories of process models; 2. to define an optimal database structure for each of the categories; 3. to develop the code for analyzing a given process model; 4. to develop the appropriate navigation code, and 4. to measure the performance and determine the performance improvements of the optimized process models.

In the first and second chapters, WSDL and BPEL are to be introduced and their main grammar structures are clarified. The third chapter describes SWoM, which contains its technical requirements and the structure of process execution module. The fourth chapter tells about the concepts of database system used here. They are XML and synthesization of relational database schema. And a general view of DB2 is presented. The fifth part is the main design for this optimization which tells the goals and the process of design and implementation.

Credit

I would like to thank Prof. Dr. Frank Leymann for his support and this interesting topic.

Special thanks go to Mr. Dieter Roller. Without his support, experience, and input this thesis would not have been possible.

I would also like to thank Ms. Stephanie Maucharther, Felipe Erias Morandeira and Dominique-Xavier Kiefner for their help and support throughout the implementation.

Moreover, I would like to thank my parents. Without them, I would not be where and what I am today.

1 WSDL (Web Service Description Language)

WSDL is an XML format for describing web services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate.

1.1 Introduction

As communications protocols and message formats are standardized in the web world, it becomes more and more possible and important to be able to describe the communications in some structured way. WSDL addresses this need by defining an XML grammar for describing web services as collections of communication endpoints able tp exchange messages.

A WSDL document defines services as collections of network endpoints, or so called ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, the abstract descriptions of the data being exchanged, and port types, the abstract collections of operations. A reusable binding is composed of the concrete protocol and data format specifications for a particular port type. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Therefore, a WSDL document uses the following seven elements to define web services:

1. Types– a container for data type definitions.
2. Message– an abstract, typed definition of the data being transmitted. And a message consists of logical parts.
3. Operation– an abstract description of an action supported by the service.
4. Port Type–an abstract set of operations supported by one or more endpoints. And each one refers to an input message and output message.
5. Binding– a concrete protocol and data format specification operations and messages defined by a particular port type.
6. Port– a single endpoint defined as a combination endpoint of a binding and a network address.

7. Service– a collection of related endpoints.

WSDL does not introduce a new type definition language. Instead, WSDL recognizes the need for rich type systems for describing message formats, and supports the XML Schemas specification (XSD) as its canonical type system. In addition, WSDL allows using other type definition languages extensibly.

Furthermore, WSDL defines a common binding mechanism. This is used to attach a specific protocol or data format or structure to an abstract message, operation, or endpoint. It allows the reuse of abstract definitions.

In addition to the core service definition framework, this specification introduces specific binding extensions for the following protocols and message formats:

1) SOAP 1.1
2) HTTP GET / POST
3) MIME

1.2 XML

The Extensible Markup Language (XML) is a general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. In other words: XML is a way of describing data and an XML file can contain the data too, as in a database. It is a simplified subset of Standard Generalized Markup Language (SGML). Its primary purpose is to facilitate the sharing of data across different systems, particularly systems connected via the Internet. Languages based on XML are defined in a formal way, allowing programs to modify and validate documents in these languages without prior knowledge of their form.

XML provides a text-based means to describe and apply a tree-based structure to information. At its base level, all information manifests as text, interspersed with markup that indicates the information's separation into a hierarchy of character data, container-like elements, and attributes of those elements.

1.3 WSDL Service Definition

This section describes the core elements of the WSDL language. WSDL Document Structure:

A WSDL document is simply a set of definitions. The grammar is as follows:

<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

```
<import namespace="uri" location="uri"/>*

<wsdl:documentation .... /> ?

<wsdl:types> ?
      <wsdl:documentation .... />?
      <xsd:schema .... />*
      <-- extensibility element --> *
</wsdl:types>

<wsdl:message name="nmtoken"> *
      <wsdl:documentation .... />?
      <part name="nmtoken" element="qname"? type="qname"?/> *
</wsdl:message>

<wsdl:portType name="nmtoken">*
      <wsdl:documentation .... />?
      <wsdl:operation name="nmtoken">*
          <wsdl:documentation .... /> ?
          <wsdl:input name="nmtoken"? message="qname">?
                <wsdl:documentation .... /> ?
          </wsdl:input>
          <wsdl:output name="nmtoken"? message="qname">?
                <wsdl:documentation .... /> ?
          </wsdl:output>
          <wsdl:fault name="nmtoken" message="qname"> *
                <wsdl:documentation .... /> ?
          </wsdl:fault>
      </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="nmtoken" type="qname">*
      <wsdl:documentation .... />?
      <-- extensibility element --> *
      <wsdl:operation name="nmtoken">*
          <wsdl:documentation .... /> ?
          <-- extensibility element --> *
          <wsdl:input> ?
                <wsdl:documentation .... /> ?
                <-- extensibility element -->
          </wsdl:input>
          <wsdl:output> ?
```

```
                <wsdl:documentation .... /> ?
                <-- extensibility element --> *
            </wsdl:output>
            <wsdl:fault name="nmtoken"> *
                <wsdl:documentation .... /> ?
                <-- extensibility element --> *
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="nmtoken"> *
        <wsdl:documentation .... />?
        <wsdl:port name="nmtoken" binding="qname"> *
            <wsdl:documentation .... /> ?
            <-- extensibility element -->
        </wsdl:port>
        <-- extensibility element -->
    </wsdl:service>

    <-- extensibility element --> *

</wsdl:definitions>
```

As follows, we describe some basic rules in WSDL languages:

1.3.1 Document Naming and Linking

WSDL documents can be assigned an optional name attribute of type NCNAME that serves as a lightweight form of documentation. Optionally, a targetNamespace attribute of type URI may be specified. The URI MUST NOT be a relative URI.

WSDL allows associating a namespace with a document location using an import statement:

```
<definitions .... >
    <import namespace="uri" location="uri"/> *
</definitions>
```

A reference to a WSDL definition is made using a QName. The following types of definitions contained in a WSDL document may be referenced:

1) WSDL definitions: service, port, message, bindings, and portType
2) Other definitions: if additional definitions are added via extensibility, they should use QName linking.

Each WSDL definition type listed above has its own name scope (i.e. port names and message names never conflict). Names within a name scope MUST be unique within the WSDL document.

The resolution of QNames in WSDL is similar to the resolution of QNames described by the XML Schemas specification.

1.3.2 Types

The types element encloses data type definitions that are relevant for the exchanged messages. For maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system, and treats it as the intrinsic type system.

```
<definitions .... >
    <types>
        <xsd:schema .... />*
    </types>
</definitions>
```

The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is actually XML, or whether the resulting XSD schema validates the particular wire format. This is especially interesting if there will be multiple bindings for the same message, or if there is only one binding but that binding type does not already have a type system in widespread use. In these cases, the recommended approach for encoding abstract types using XSD is as follows:

1. Use element form (not attribute).

2. Don't include attributes or elements that are peculiar to the wire encoding (e.g. have nothing to do with the abstract content of the message). Some examples are soap:root, soap:encodingStyle, xmi:id, xmi:name.

3. Array types should extend the Array type defined in the SOAP v1.1 encoding schema (http://schemas.xmlsoap.org/soap/encoding/) (regardless of whether the resulting form actually uses the encoding specified in Section 5 of the SOAP v1.1 document). Use the name ArrayOfXXX for array types (where XXX is the type of the items in the array). The type of the items in the array and the array dimensions are specified by using

a default value for the soapenc:arrayType attribute. At the time of this writing, the XSD specification does not have a mechanism for specifying the default value of an attribute which contains a QName value. To overcome this limitation, WSDL introduces the arrayType attribute (from namespace http://schemas.xmlsoap.org/wsdl/) which has the semantic of providing the default value. If XSD is revised to support this functionality, the revised mechanism SHOULD be used in favor of the arrayType attribute defined by WSDL.

4. Use the xsd:anyType type to represent a field/parameter which can have any type.

However, since it is unreasonable to expect a single type system grammar can be used to describe all abstract types present and future, WSDL allows type systems to be added via extensibility elements. An extensibility element may appear under the types element to identify the type definition system being used and to provide an XML container element for the type definitions. The role of this element can be compared to that of the schema element of the XML Schema language.

```
<definitions .... >
    <types>
        <-- type-system extensibility element --> *
    </types>
</definitions>
```

1.3.3 Messages

Messages consist of one or more logical parts. Each part is associated with a type from some type system using a message-typing attribute. The set of message-typing attributes is extensible. WSDL defines several such message-typing attributes for use with XSD:

1) element. Refers to an XSD element using a QName.
2) type. Refers to an XSD simpleType or complexType using a QName.

Other message-typing attributes may be defined as long as they use a namespace different from that of WSDL. Binding extensibility elements may also use message-typing attributes.

The syntax for defining a message is as follows. The message-typing attributes (which may vary depending on the type system used) are shown in bold.

```
<definitions .... >
    <message name="nmtoken"> *
```

```
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </message>
</definitions>
```

The message name attribute provides a unique name among all messages defined within the enclosing WSDL document.

The part name attribute provides a unique name among all the parts of the enclosing message.

1.3.4 Port Types

A port type is a named set of abstract operations and the abstract messages involved.

```
<wsdl:definitions .... >
    <wsdl:portType name="nmtoken">
        <wsdl:operation name="nmtoken" .... /> *
    </wsdl:portType>
</wsdl:definitions>
```

The port type name attribute provides a unique name among all port types defined within in the enclosing WSDL document.

An operation is named via the name attribute.

WSDL has four transmission primitives that an endpoint can support:

1. One-way. The endpoint receives a message.
2. Request-response. The endpoint receives a message, and sends a correlated message.
3. Solicit-response. The endpoint sends a message, and receives a correlated message.
4. Notification. The endpoint sends a message.

WSDL refers to these primitives as operations. Although request/response or solicit/response can be modeled abstractly using two one-way messages, it is useful to model these as primitive operation types because:

1. They are very common.

2. The sequence can be correlated without having to introduce more complex flow information.
3. Some endpoints can only receive messages if they are the result of a synchronous request response.
4. A simple flow can algorithmically be derived from these primitives at the point when flow definition is desired.

Although request/response or solicit/response are logically correlated in the WSDL document, a given binding describes the concrete correlation information. For example, the request and response messages may be exchanged as part of one or two actual network communications.

Although the base WSDL structure supports bindings for these four transmission primitives, WSDL only defines bindings for the One-way and Request-response primitives. It is expected that specifications that define the protocols for Solicit-response or Notification would also include WSDL binding extensions that allow use of these primitives.

Operations refer to the messages involved using the message attribute of type QName.

1.3.5 One-way Operation

The grammar for a one-way operation is:

```
<wsdl:definitions .... > <wsdl:portType .... > *
        <wsdl:operation name="nmtoken">
            <wsdl:input name="nmtoken"? message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

The input element specifies the abstract message format for the one-way operation.

1.3.6 Request-response Operation

The grammar for a request-response operation is:

```
<wsdl:definitions .... >
    <wsdl:portType .... > *
        <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
            <wsdl:input name="nmtoken"? message="qname"/>
            <wsdl:output name="nmtoken"? message="qname"/>
```

```
            <wsdl:fault name="nmtoken" message="qname"/>*
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

The input and output elements specify the abstract message format for the request and response, respectively. The optional fault elements specify the abstract message format for any error messages that may be output as the result of the operation (beyond those specific to the protocol).

1.3.7 Bindings

A binding defines message format and protocol details for operations and messages defined by a particular portType. There may be any number of bindings for a given portType. The grammar for a binding is as follows:

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *
        <-- extensibility element (1) --> *
        <wsdl:operation name="nmtoken"> *
            <-- extensibility element (2) --> *
            <wsdl:input name="nmtoken"? > ?
                <-- extensibility element (3) -->
            </wsdl:input>
            <wsdl:output name="nmtoken"? > ?
                <-- extensibility element (4) --> *
            </wsdl:output>
            <wsdl:fault name="nmtoken"> *
                <-- extensibility element (5) --> *
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

The name attribute provides a unique name among all bindings defined within in the enclosing WSDL document.

A binding references the portType that it binds using the type attribute. This QName value follows the linking rules defined by WSDL.

Binding extensibility elements are used to specify the concrete grammar for the input (3),

output (4), and fault messages (5). Per-operation binding information (2) as well as per-binding information (1) may also be specified.

An operation element within a binding specifies binding information for the operation with the same name within the binding's portType. Since operation names are not required to be unique (for example, in the case of overloading of method names), the name attribute in the operation binding element might not be enough to uniquely identify an operation. In that case, the correct operation should be identified by providing the name attributes of the corresponding wsdl:input and wsdl:output elements.

1.3.8 Ports

A port defines an individual endpoint by specifying a single address for a binding.

```
<wsdl:definitions .... >
      <wsdl:service .... > *
            <wsdl:port name="nmtoken" binding="qname"> *
                <-- extensibility element (1) -->
            </wsdl:port>
      </wsdl:service>
</wsdl:definitions>
```

The name attribute provides a unique name among all ports defined within in the enclosing WSDL document.

The binding attribute (of type QName) refers to the binding using the linking rules defined by WSDL.

Binding extensibility elements (1) are used to specify the address information for the port.

1.4 Example

Example: WSDL and BPEL script for a sequential structure

This is the WSDL and BPEL script for a particular instance of a complex sequential process model:

```
<definitions
        targetNamespace="http://example.com/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
   xmlns:expl="http://example.com/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

<import namespace="http://example.com/" location="http://example.com/"/>

<types>
      <xsd:schema targetNamespace="http://example.com/">
            <xsd:restriction base="xsd:int">
                  <xsd:minInclusive value="1"/>
                  <xsd:maxInclusive value="9999"/>
            </xsd:restriction>
      </xsd:schema>
</types>
<bpws:property name="PIID" type="xsd:int"/>
<bpws:property name="PMID" type="xsd:int"/>
<bpws:property name="AID" type="xsd:int"/>
<bpws:property name="state" type="xsd:int"/>
<bpws:property name="value" type="xsd:blob"/>
</types>

<message name="receiveMessage">
      <part name="id" type="xsd:int"/>
      <part name="name" type="xsd:string"/>
</message>

<message name="replyMessage">
      <part name="id" type="xsd:int"/>
      <part name="name" type="xsd:string"/>
</message>

<portType name="receiveAndReply">
      <operation name="requestAndReply">
          <input message="expl:receiveMessage"/>
          <output message="expl:replyMessage"/>
          <fault name="messageFault" message="expl:errorMessage"/>
      </operation>
</portType>

<plnk:partnerLinkType name="sendType">
      <plnk:role name="sendService" portType="expl:sendServicePT"/>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="invokeType">
```

```
        <plnk:role name="invokeService" portType="expl:invokeServicePT"/>
    </plnk:partnerLinkType>


    <plnk:partnerLinkType name="receiveType">
        <plnk:role name="receiveService" portType="expl:receiveServicePT"/>
    </plnk:partnerLinkType>

</definitions>

<process name="471" targetNamespace="http://example.com/"
          suppressJionFailure="yes"
          abstractProcess="no"
          xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">
  <import namespace="http://example.com/" location="http://example.com/"
           importType="http://example.com/"/>
  <partnerLinks>

        <partnerLink name="A" partnerLinkType="expl:sendType"
                      myRole="sendService"/>
        <partnerLink name="B" partnerLinkType="expl:receiveType"
                      partnerRole="sendService"/>
        <partnerLink name="C" partnerLinkType="expl:invokeType"
                      partnerRole="sendService"/>
  </partnerLinks>
  <variables>
        <variable name="(a.b)" messageType="expl:FromNavigatorMessage"/>
        <variable name="(c,d)" messageType="expl:FromNavigatorMessage"/>
  </variables>
  <sequence>
        <receive partnerLink="B" portType="receiveType"
                  operation="receiveMessage"
                     variable="(a,b)">
        </receive>
        <assign>
            <copy>
                <from>($a,$b).receiveMessages</from>
                <to>($c.$d).receiveMessages</to>
            </copy>
        </assign>
        <invoke partnerLink="C" portType="expl:invokeType"
                  operation="invokeAction"
                  inputvariable="(a,b)" outputvariable="(c,d)">
```

```
        </invoke>
        <reply parnterLink="B" portType="expl:receiveType"
              operation="replyMessage"    variable="(c.d)"/>
        </reply>
   </sequence>
</process>
```

This example shows a good combination of WSDL and BPEL. The first part of the example is the definition in WSDL while the second part is the workflow described in BPEL. In the first part, types, messages, portType and partnerlinkType are defined for the interactive partnerlink and corresponding operations in the process. In the second part, the program paragraph between the marks <sequence> and </sequence> depicts the structure of a sequence workflow.

This example describes a particular instance of a complex sequential process model (See Figure 1-1). It encompasses three activities, that is, Receive, Assign, and Reply. There are three partners A, B, C. Variable types are vector. And there are two set of variables: (a, b) and (c, d). The entire process depicts that after B receives messages with parameter (a, b) from A, the value of (a, b) is copied into (c, d). And then C is invoked with the input of (a, b). When C is completed, its result is transferred into (c, d) and moreover, C sends a reply back to B with the result (c, d)

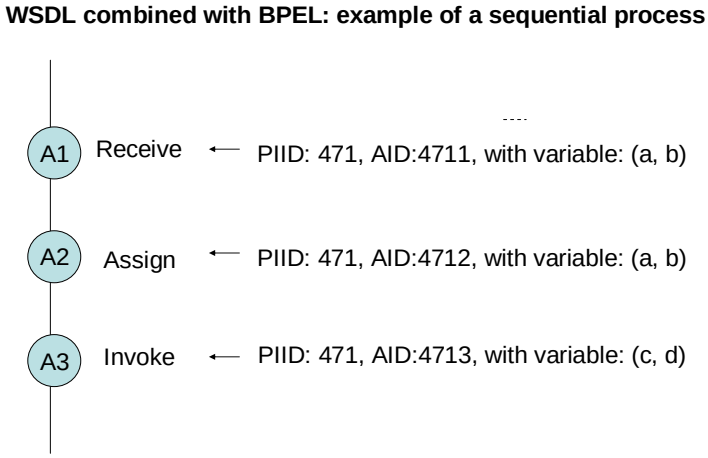


Figure 1-1 Process script example of a complex sequential process

This example is a combination of WSDL and BPEL, which is to be described in detail (See Chapter 2.3).

1.5 Summary

This chapter provided an overview of WSDL, Web Service Description Language, the definition schema in WSDL, its syntax and semantics. Furthermore, the detailed script with the combination of WSDL and BPEL for a particular instance of complex sequential process model was shown and explained as an example.

2 BPEL (Web Service Business Process Execution Language)

Business processes can be described in two ways. Executable business processes model actual behavior of a participant in a business interaction. Business protocols, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. The process descriptions for business protocols are called abstract processes. The purpose of BPEL is to model the behavior of both executable and abstract processes.

2.1 Introduction

Business Process Execution Language for Web Services provides a means to formally specify business processes and interaction protocols.

BPEL provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web Services interaction model and enables it to support business transactions. BPEL defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to- business spaces.

BPEL is an important language for the process-oriented approach to SOA. Because BPEL has been designed specifically for definition of business processes it provides good support for various specifics of business processes such as support for long running transactions, compensation, event management, correlation, etc. BPEL is well suited for use with the J2EE platform and many BPEL servers build on top of J2EE. With ideas of combing BPEL and Java (BPELJ), and WSIF, the usability of BPEL is even increasing. We should also look at the emerging JBI (Java Business Integration) specification aka JSR 208 which will give business integration and BPEL an even better documented position in the Java platform.

BPEL is an XML-based language designed to enable task-sharing for a distributed computing or grid computing environment - even across multiple organizations - using a combination of Web services. (BPEL is also sometimes identified as BPELWS or BPEL.)

Using BPEL, a programmer formally describes a business process that will take place across the Web in such a way that any cooperating entity can perform one or more steps in the process the same way. In a supply chain process, for example, a BPEL program might describe a business protocol that formalizes what pieces of information a product order consists of, and what exceptions may have to be handled. The BPEL program

would not, however, specify how a given Web service should process a given order internally.

The goal of the Web Services effort is to achieve universal interoperability between applications by using Web standards. Web Services use a loosely coupled integration model to allow flexible integration of heterogeneous systems in a variety of domains including business-to-consumer, business-to-business and enterprise application integration. The following basic specifications originally defined the Web Services space: SOAP, Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). SOAP defines an XML messaging protocol for basic service interoperability. WSDL introduces a common grammar for describing services. UDDI provides the infrastructure required to publish and discover services in a systematic way. Together, these specifications allow applications to find each other and interact following a loosely coupled, platform-independent model.

Systems integration requires more than the ability to conduct simple interactions by using standard protocols. The full potential of Web Services as an integration platform will be achieved only when applications and business processes are able to integrate their complex interactions by using a standard process integration model. The interaction model that is directly supported by WSDL is essentially a stateless model of synchronous or uncorrelated asynchronous interactions. Models for business interactions typically assume sequences of peer-to-peer message exchanges, both synchronous and asynchronous, within stateful, long- running interactions involving two or more parties. To define such business interactions, a formal description of the message exchange protocols used by business processes in their interactions is needed. The definition of such business protocols involves precisely specifying the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal implementation. There are two good reasons to separate the public aspects of business process behavior from internal or private aspects. One is that businesses obviously do not want to reveal all their internal decision making and data management to their business partners. The other is that, even where this is not the case, separating public from private process provides the freedom to change private aspects of the process implementation without affecting the public business protocol.

Business protocols must clearly be described in a platform-independent manner and must capture all behavioral aspects that have cross-enterprise business significance. Each participant can then understand and plan for conformance to the business protocol without engaging in the process of human agreement that adds so much to the difficulty of establishing cross-enterprise automated business processes today.

In thinking about the data handling aspects of business protocols it is instructive to consider the analogy with network communication protocols. Network protocols define the shape and content of the protocol envelopes that flow on the wire, and the protocol

behavior they describe is driven solely by the data in these envelopes. In other words, there is a clear physical separation between protocol-relevant data and "payload" data. The separation is far less clear cut in business protocols because the protocol-relevant data tends to be embedded in other application data.

BPEL uses a notion of message properties to identify protocol-relevant data embedded in messages. Properties can be viewed as "transparent" data relevant to public aspects as opposed to the "opaque" data that internal/private functions use. Transparent data affects the public business protocol in a direct way, whereas opaque data is significant primarily to back-end systems and affects the business protocol only by creating non-determinism because the way it affects decisions is opaque. We take it as a principle that any data that is used to affect the behavior of a business protocol must be transparent and hence viewed as a property.

The implicit effect of opaque data manifests itself through non-determinism in the behavior of services involved in business protocols. Consider the example of a purchasing protocol. The seller has a service that receives a purchase order and responds with either acceptance or rejection based on a number of criteria, including availability of the goods and the credit of the buyer. Obviously, the decision processes are opaque, but the fact of the decision must be reflected as behavior alternatives in the external business protocol. In other words, the protocol requires something like a switch activity in the behavior of the seller's service but the selection of the branch taken is nondeterministic. Such non-determinism can be modeled by allowing the assignment of a nondeterministic or opaque value to a message property, typically from an enumerated set of possibilities. The property can then be used in defining conditional behavior that captures behavioral alternatives without revealing actual decision processes. BPEL explicitly allows the use of nondeterministic data values to make it possible to capture the essence of public behavior while hiding private aspects.

The basic concepts of BPEL can be applied in one of two ways. A BPEL process can define a business protocol role, using the notion of abstract process. For example, in a supply-chain protocol, the buyer and the seller are two distinct roles, each with its own abstract process. Their relationship is typically modeled as a partner link. Abstract processes use all the concepts of BPEL but approach data handling in a way that reflects the level of abstraction required to describe public aspects of the business protocol. Specifically, abstract processes handle only protocol-relevant data. BPEL provides a way to identify protocol-relevant data as message properties. In addition, abstract processes use nondeterministic data values to hide private aspects of behavior.

It is also possible to use BPEL to define an executable business process. The logic and state of the process determine the nature and sequence of the Web Service interactions conducted at each business partner, and thus the interaction protocols. While a BPEL process definition is not required to be complete from a private implementation point of

view, the language effectively defines a portable execution format for business processes that rely exclusively on Web Service resources and XML data. Moreover, such processes execute and interact with their partners in a consistent way regardless of the supporting platform or programming model used by the implementation of the hosting environment.

Even where private implementation aspects use platform-dependent functionality, which is likely in many if not most realistic cases, the continuity of the basic conceptual model between abstract and executable processes in BPEL makes it possible to export and import the public aspects embodied in business protocols as process or role templates while maintaining the intent and structure of the protocols. This is arguably the most attractive prospect for the use of BPEL from the viewpoint of unlocking the potential of Web Services because it allows the development of tools and other technologies that greatly increase the level of automation and thereby lower the cost in establishing cross-enterprise automated business processes.

2.2 Notation

BPEL defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. The interaction with each partner occurs through Web Service interfaces, and the structure of the relationship at the interface level is encapsulated in what we call a partner link. The BPEL process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination. BPEL also introduces systematic mechanisms for dealing with business exceptions and processing faults. Finally, BPEL introduces a mechanism to define how individual or composite activities within a process are to be compensated in cases where exceptions occur or a partner requests reversal.

BPEL is layered on top of several XML specifications: WSDL 1.1, XML Schema 1.0, and XPath1.0. WSDL messages and XML Schema type definitions provide the data model used by BPEL processes. XPath provides support for data manipulation. All external resources and partners are represented as WSDL services. BPEL provides extensibility to accommodate future versions of these standards, specifically the XPath and related standards used in XML computation.

The structure of the main processing section is defined by the outer <sequence> element, which states that the three activities contained inside are performed in order. The customer request is received (<receive> element), then processed (inside a <flow> section that enables concurrent behavior), and a reply message with the final approval status of the request is sent back to the customer (<reply>). Note that the <receive> and <reply> elements are matched respectively to the <input> and <output> messages of the

operation invoked by the customer, while the activities performed by the process between these elements represent the actions taken in response to the customer request, from the time the request is received to the time the response is sent back (reply).

The processing taking place inside the <flow> element consists of <sequence> blocks running concurrently. The synchronization dependencies between activities in the concurrent sequences are expressed by using "links" to connect them. The links are defined inside the flow and are used to connect a source activity to a target activity. (Note that each activity declares itself as the source or target of a link by using the nested <source> and <target> elements.) In the absence of links, the activities nested directly inside a flow proceed concurrently.

Information is passed between the different activities in an implicit way through the sharing of globally visible data variables.

Certain operations can return faults, as defined in their WSDL definitions. For simplicity, it is assumed here that the two operations return the same fault. When a fault occurs, normal processing is terminated and control is transferred to the
corresponding fault handler, as defined in the <faultHandlers> section.

Finally, it is important to observe how an assignment activity is used to transfer information between data variables.

2.3 Relationship with WSDL

BPEL depends on the XML-based specifications: WSDL 1.1, XML Schema 1.0,XPath 1.0 and WS-Addressing. Among these, WSDL has the most influence on the BPEL language. The BPEL process model is layered on top of the service model defined by WSDL 1.1. At the core of the BPEL process model is the notion of peer-to-peer interaction between services described in WSDL; both the process and its partners are modeled as WSDL services. A business process defines how to coordinate the interactions between a process instance and its partners. In this sense, a BPEL process definition provides and/or uses one or more WSDL services, and provides the description of the behavior and interactions of a process instance relative to its partners and resources through Web Service interfaces. That is, BPEL defines the message exchange protocols followed by the business process of a specific role in the interaction.

The definition of a BPEL business process also follows the WSDL model of separation between the abstract message contents used by the business process and deployment information (messages and portType versus binding and address information). In particular, a BPEL process represents all partners and interactions with these partners in

terms of abstract WSDL interfaces (portTypes and operations); no references are made to the actual services used by a process instance. However, the abstract part of WSDL does not define the constraints imposed on the communication patterns supported by the concrete bindings. Therefore a BPEL process may define behavior relative to a partner service that is not supported by all possible bindings, and it may happen that some bindings are invalid for a BPEL process definition.

A BPEL process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them. Note that the description of the deployment of a BPEL process is out of scope for this specification. The dependency on WS-Addressing is meant to avoid inventing a private BPEL mechanism for web service endpoint references—such references are obviously a very general requirement in the usage of web services.

2.4 Define a Process Model

Before describing the structure of processes in detail, this section presents a simple example of simple parallel process model. The operation of the process is very simple, and is represented in the following figure (See Figure 2-1).

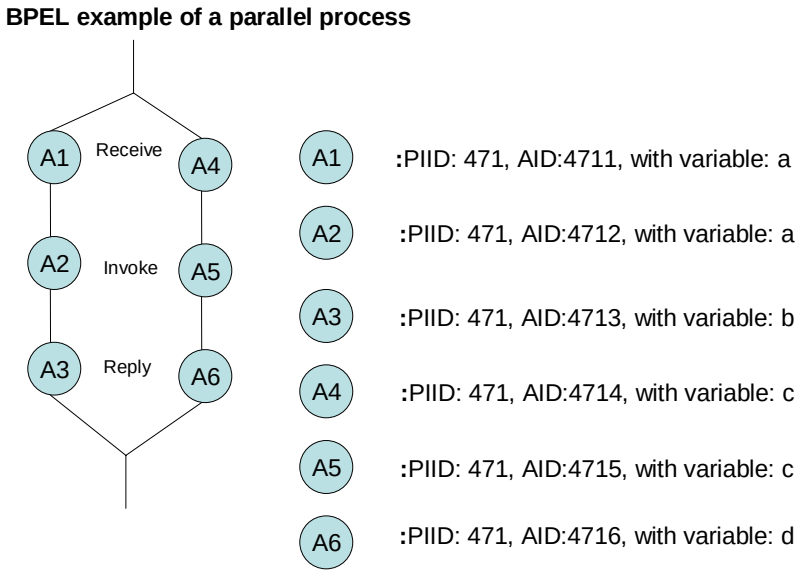


Figure 2-1 BPEL example of a parallel structure

This is the BPEL script for a simple parallel process model:

```
<process name="471" targetNamespace="uri"
            queryLanguage=" "
            expressionLanguage=" "
            suppressJionFailure="yes"
            abstractProcess="no"
            xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">
  <import namespace="uri" location="uri" importType="uri"/>
  <partnerLinks>
        <partnerLink name="A" partnerLinkType=" " myRole=" "/>
        <partnerLink name="B" partnerLinkType=" " partnerRole=" "/>
        <partnerLink name="C" partnerLinkType=" " />
        <partnerLink name="D" partnerLinkType=" " myRole=" "/>
        <partnerLink name="E" partnerLinkType=" " partnerRole=" "/>
        <partnerLink name="F" partnerLinkType=" " />
  </partnerLinks>
  <variables>
        <variable name="a" messageType=" "/>
        <variable name="b" messageType=" "/>
        <variable name="c" messageType=" "/>
        <variable name="d" messageType=" "/>
  </variables>
  <flow>
    <sequence>
        <receive partnerLink="B" portType=" " operation="receiveMessage"
                variable="a">
        </receive>
        <invoke partnerLink="C" portType=" " operation="invokeAction"
                inputvariable="a">
        </invoke>
        <reply parnterLink="B" portType=" " operation="replyMessage"
                variable="b"/>
        </reply>
    </sequence>
    <sequence>
        <receive partnerLink="E" portType=" " operation="receiveMessage"
                variable="c">
        </receive>
        <invoke partnerLink="F" portType=" " operation="invokeAction"
                inputvariable="c">
        </invoke>
        <reply parnterLink="E" portType=" " operation="replyMessage"
                variable="d"/>
        </reply>
```

```
    </sequence>
  </flow>
</process>
```

In this example, the program paragraph between the marks <flow> and </flow> presents the structure of a parallel workflow. And this parallel workflow consists of two sequential sub-workflows which are marked by <sequence> and </sequence> respectively. This example describes a particular instance of a simple parallel process model (See the above figure). It encompasses three activities in each parallel branch, that is, Receive, Assign, and Reply. There are six partners A, B, C and D, E, F which stay in each parallel branch respectively. Here, simple variables are used, namely, a, b, c and d. The entire process depicts that B and E starts receiving messages with parameter a and c from A and D simultaneously, and then C and F invoke another action with input variable of a and c. And finally B and E send replies back with the results saved in b and d after C and F are completed.

2.5 Basic and Complex Activities in BPEL

This section presents some important basic and complex BPEL-activities used for this thesis.

2.5.1 Structure of a Process

This section gives a brief picture of the BPEL syntax.

The basic structure of BPEL is:

```
<process name="NCName" targetNamespace="anyURI"
         queryLanguage="anyURI"?
         expressionLanguage="anyURI"?
         suppressJoinFailure="yes|no"?
         exitOnStandardFault="yes|no"?
         xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">

<extensions>?
    <extension namespace="anyURI" mustUnderstand="yes|no"/>*
</extensions>


<import namespace="anyURI"? location="anyURI"? importType="anyURI"/>*

<partnerLinks>?
```

```
<!-- Note: At least one role must be specified. -->
      <partnerLink name="NCName" partnerLinkType="QName"
            myRole="NCName"? partnerRole="NCName"?>+
      </partnerLink>
</partnerLinks>

<messageExchanges>?
      <messageExchange name="NCName"/>+
</messageExchanges>

<variables>?
       <variable name="NCName" messageType="QName"?
            type="QName"? element="QName"?/>+
</variables>

<correlationSets>?
         <correlationSet name="NCName" properties="QName-list"/>+
</correlationSets>

<faultHandlers>?
      <!-- Note: There must be at least one faultHandler -->
      <catch faultName="QName"? faultVariable="NCName"?
                                          faultMessageType="QName"?
                                          faultElement="QName"?>*
            activity
      </catch>
      <catchAll>?
            activity
      </catchAll>
</faultHandlers>

<eventHandlers>?
<!-- Note: There must be at least one onEvent or onAlarm eventHandler. -->
            <onEvent partnerLink="NCName" portType="QName"?
               operation="NCName" (messageType="QName" | element="QName")?
               variable="NCName"?
               messageExchange="NCName"? >*
                  <correlations>?
                     <correlation set="NCName"
                                 initiate="yes|join|no"?/>+
                  </correlations>
                  <fromPart part="NCName" toVariable="NCName"/>*
                  <scope ...> ... </scope>
```

```
            </onEvent>
            <onAlarm>*
           <!-- Note: There must be at least one expression. -->
               ( <for expressionLanguage="anyURI"?>duration-expr</for> |

                   <until expressionLanguage="anyURI"?>deadline-expr</until> )?
                   <repeatEvery expressionLanguage="anyURI"?>
                      duration-expr
                   </repeatEvery>?


                     <scope ...> ... </scope>
            </onAlarm>
</eventHandlers>
            activity
</process>
```

The whole process is depicted inside the marks <process> and </process>. To define a process, the following attributes for <process> must be defined correctly at first:

1. The attribute of queryLanguage specifies the query language used in the process to select nodes in assignment and define properties, etc. The default value of it is: "urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0".

2. The attribute of expressionLanguage specifies the expression language used in the process. The default value of it is: "urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"

3. The attribute of suppressJoinFailure determines if the joinFailure fault will be suppressed for all activities in the process. The effect of the attribute at the process level can be overridden by an activity using a different value for the attribute. The default value of it is "no" at the process level.

4. The attribute of abstractProcess decribes whether the process to be defined is abstract or not. The default value of it is “no”. The syntax of an abstract process has its own distince target namespace.

The value of the queryLanguage and expressionLanguage attributes on the <process> element are global defaults and can be overridden on specific constructs, such as <condition> of a <while> activity.

Note that: <documentation> construct may be added to virtually all WS-BPEL constructs as the formal way to annotate processes definition with human documentation.

And BPEL activities are listed as follows:

<receive>, <reply>, <invoke>, <assign>, <throw>, <exit>, <wait>, <empty>, <sequence>, <if>, <while>, <repeatUntil>, <forEach>, <pick>, <flow>, <scope>, <compensate>, <compensateScope>, <rethrow>, <validate> and <extensionActivity>

2.5.2 Basic Activities

1. <receive>

In the <receive> activity, the process waits for a certain message to arrive. The <receive> activity is completed after the message arrives. The portType attribute for the <receive> activity is optional. If the portType attribute is included for readability, the value of the portType attribute must be in accordance with the portType value implied by the combination of the specified partnerLink and the role implicitly specified by the activity The optional messageExchange attribute is used to associate a <reply> activity with a <receive> activity.

```
<receive partnerLink="NCName" portType="QName"? operation="NCName"
    variable="NCName"? createInstance="yes|no"?
     messageExchange="NCName"?
    standard-attributes>
    standard-elements
    <correlations>?
       <correlation set="NCName" initiate="yes|join|no"?/>+
    </correlations>
    <fromPart part="NCName" toVariable="NCName"/>*
</receive>
```

2. <reply>

In the <reply> activity, the process sends a message to reply to a message that was received by an inbound message activity (IMA), that is, <receive>, <onMessage>, or <onEvent>. The combination of an IMA and a <reply> forms a request-response operation. The portType attribute on the <reply> activity is optional. If the portType attribute is included for readability, the value of the portType attribute must be in accordance with the portType value implied by the combination of the specified partnerLink and the role implicitly specified by the activity. The optional messageExchange attribute is used to associate a <reply> activity with an IMA.

```
<reply partnerLink="NCName" portType="QName"? operation="NCName"
```

```
    variable="NCName"? faultName="QName"?
    messageExchange="NCName"?
    standard-attributes>
    standard-elements
    <correlations>?
       <correlation set="NCName" initiate="yes|join|no"?/>+
    </correlations>
    <toPart part="NCName" fromVariable="NCName"/>*
</reply>
```

3. <invoke>

In the <invoke> activity, the process invokes a one-way or request-response operation on a portType offered by a partner. In the request-response case, the invoke activity is completed after the response is received. The portType attribute on the <invoke> activity is optional. If the portType attribute is included for readability, the value of the portType attribute must be in accordance with the portType value implied by the combination of the specified partnerLink and the role implicitly specified by the activity .

```
<invoke partnerLink="NCName" portType="QName"? operation="NCName"
    inputVariable="NCName"? outputVariable="NCName"?
    standard-attributes>
    standard-elements
    <correlations>?
       <correlation set="NCName" initiate="yes|join|no"?
          pattern="request|response|request-response"/>+
    </correlations>
    <catch faultName="QName"? faultVariable="NCName"?
                        faultMessageType="QName"?
                        faultElement="QName">*
       activity
    </catch>
    <catchAll>?
       activity
    </catchAll>
    <compensationHandler>?
       activity
    </compensationHandler>
    <toPart part="NCName" fromVariable="NCName"/>*
    <fromPart part="NCName" toVariable="NCName"/>*
</invoke>
```

4. <assign>

In the <assign> activity, the main task is to update the values of variables with new data. An <assign> construct can contain any number of elementary assignments, including <copy> assign elements or data update operations defined as extension under other namespaces.

```
<assign validate="yes|no"? standard-attributes>
    standard-elements
    (<copy>
       from-spec
        to-spec
    </copy> |
    <extensibleAssign>
       ...assign-element-of-other-namespace...
    </extensibleAssign>) +
</assign>
```

2.5.3 Complex Activities

1. <sequence>

The <sequence> activity is used to define a collection of activities to be performed sequentially. The <sequence> activity is completed after the last activity in the sequence has completed.

.

```
<sequence standard-attributes>
     standard-elements
     activity+
</sequence>
```

2. <flow> and the links

The <flow> activity is used to define one or more activities to be performed concurrently. Links can be used within a flow to define explicit control dependencies between nested child activites.

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

activity+
</flow>

The basic semantics of grouping a set of activities in a <flow> is to enable concurrency. A <flow> is completed after all of the activities enclosed by the <flow> have been completed. If its performing condition evaluates to be false then this activity is skipped and also considered completed

The <flow> activity creates a set of concurrent activities directly nested within it. It enables synchronization dependencies between activities that are nested within it to any depth. The <link> construct is used to express these synchronization dependencies. The links for which activity A is the source will be referred to as A's outgoing links, and the links for which activity A is the target will be referred to as A's incoming links. If activity X is the target of a link that has activity Y as the source, then X has a synchronization dependency on Y.

Every activity that is the target of a link has a join condition related with it, even if the activity has only one incoming link. A join condition is a Boolean expression. The expression for a join condition must be constructed by using Boolean operators and the activity's incoming links' status values.

If an activity ready to start has incoming links, then it can only start after the status of all its incoming links has been determined and the join condition has been evaluated.

A link is of three states: true, false, or unset. The lifetime of the status of a <link> is exactly the lifetime of the <flow> activity within which it is declared. Whenever a <flow> activity is activated, the status of all the links declared in that activity is unset.

The semantics of link status evaluation are described as follows:

When activity A completes, the following steps must be performed to determine the effect of the links on other activities:

1) Determine the status of all outgoing links for activity A. The status will be either true or false. To determine the status for each link its <transitionCondition> is evaluated. If some of the variables referenced by the <transitionCondition> are modified in a concurrent path, the result of the transition condition evaluation may depend nondeterministically on the timing of behavior among concurrent activities (See Chapter 4.2.2.1.8).

2) For each activity B that has a synchronization dependency on A, check whether the status of all incoming links for B has been determined. If the condition is true, then evaluate the <joinCondition> for B, if it evaluates to be true, activity B is started. And

thus, it could result in a dead path which is to be eliminated (See Chapter 4.2.2.1.9).

The associated source activity must be completd before the <transitionCondition> of a link is evaluated.

2.5.4 Partner Links

In BPEL cross enterprise business interactions in which the business processes of each enterprise interact through web service interfaces by using partner links. Therefore, in BPEL it's able to model the required relationships between partner processes.

The relationship of a business process to a partner is typically peer-to-peer, requiring a two-way dependency at the service level. That's to say, a partner represents both a consumer of a service provided by the business process and a provider of a service to the business process. This is especially the case when the interactions are based on one-way operations rather than on request-response operations.

The notion of partnerLinks is used to directly model peer-to-peer conversational partner relationships. PartnerLinks define the shape of a relationship with a partner by defining the portTypes used in the interactions in both directions. However, the actual partner service may be dynamically determined within the process.

Each partnerLink is characterized by a partnerLinkType.

```
<partnerLinks>
    <partnerLink name="NCName" partnerLinkType="QName"
       myRole="NCName"? partnerRole="NCName"?
       initializePartnerRole="yes|no"?/>+
</partnerLinks>
```

Each partnerLink is named, and this name is used for all service interactions via that partnerLink.

2.5.5 Variable Management in BPEL

While processes interact with each other, data, for example, variables, is exchanged between partners.

Variables are used to hold messages that are a part of the state of a process. The messages held are often those that have been received from partners or are to be sent to partners.

Variables can also hold data that are needed for holding state related to the process and never exchanged with partners.

The syntax of the <variables> declaration is:

```
<variables>
      <variable name="NCName" messageType="QName"?
           type=”QName”? element=”QName”?>+
         from-spec?
      </variable>
</variables>
```

Each variable follows the analogous variable scope rules to those in other programming languages to deal with the local variables and global variables.

The name of a variable must be unique amongst the names of all variables defined within the same immediately enclosing scope.

The messageType, type or element attributes are used to specify the type of a variable. One of these attributes must be used. Attribute messageType refers to a WSDL message type definition.

2.6 Summary

This chapter gave a brief overview of programming language BPEL, Business Process Execution Language, including its syntax structure, its basic activities as well as their semantics.

For each type of process model, a BPEL script was given out for the respective instance. (See Appendix A) However, these processes are only abstract processes which depict the structure of different process running models, instead of executable processes. In the future, they would be made executable and included into the actual running BPEL engine.

3 SWoM

In the section, SWoM is to be introduced briefly. The main topics of SWoM are its functional and non-functional requirements, its architecture and the software requirements for installation and running.

3.1 Overview of the System

SWoM is the abbreviation of Stuttgarter Workflow Machine. This project is organized by the IAAS institute at the University of Stuttgart.

The SWoM is developed using a phases model, which contains six phases as follows: specification, design, implementation, module testing, integration and integration test. This thesis put an emphasis on the optimizing in the phase of design. To reduce the complete yet complicated system into a simple running model according to different kinds of process models.

3.1.1 Requirements

This parts depicts the functional, non-functional and software requirements in the project SWoM as well as the architecture of it.

3.1.1.1 Functional

In this project, the main functionality of BPEL compliant workflow machine which is described in the last chapter is to provide BPEL process models as web services. BPEL executes process instances which are created by the process models against the request of a certain web service client. In addition, BPEL still administrate some functionalities.

3.1.1.2 Non-functional

The non-functional requirements contain the quantity structure, performance and scalability requirements, design constraints and requirements for availability and security. A detailed description about them is given in the SWoM Specification 1.2.

3.1.2 Architecture

The SWoM is composed of three modules: the process execution module, the

administration module and the gateway module. These three modules comprise different components respectively. They connect with each other by messaging middleware.

The following figure (See Figure 3-1) tells the architecture of the SWoM:

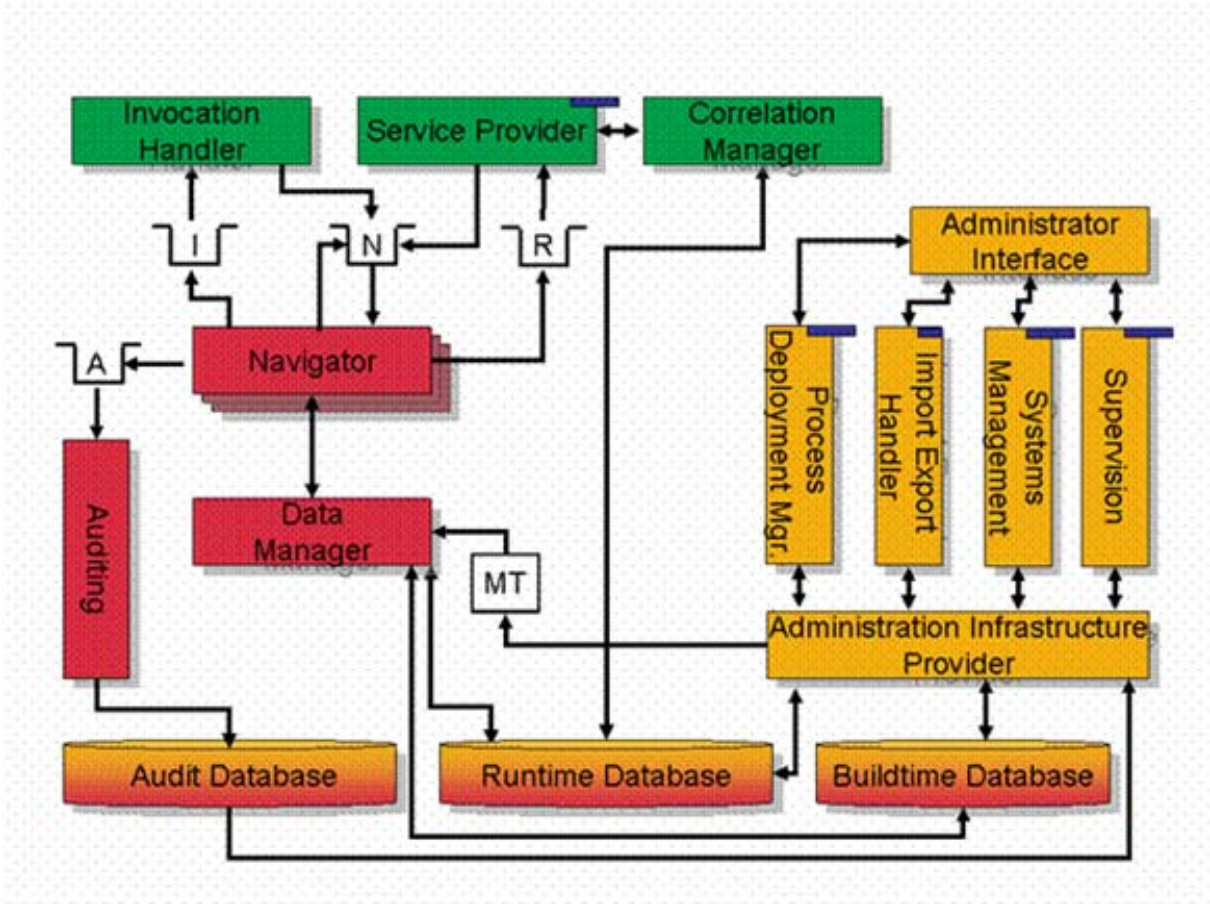


Figure 2-1 Architecture of the SWoM

The green boxes show the elements in the gateway module, the yellow boxes show the elements in the administration module and the red boxes depict the elements in the process execution module. The little blue boxes indicate web service interfaces. The little blue box of the import export handler is smaller then the others, because only the deletion of process models is exposed as a web service, while the import and export are not ‘A’ stands for the auditing queue, ‘I’ stands for the invocation queue, ‘N’ stands for the navigation queue and ‘R’ stands for the reply queue. ‘MT’ stands for the manager topic. The arrows indicate communication dependencies. The cylinders at the bottom are the databases.

3.2 Process Execution Module

The main task in this module is to navigate through process instances and handle the activities. And the main components here are the navigator, data manager and auditing.

The internal dependencies and their relationship with the databases are shown in the following figure (See Figure 3-2):

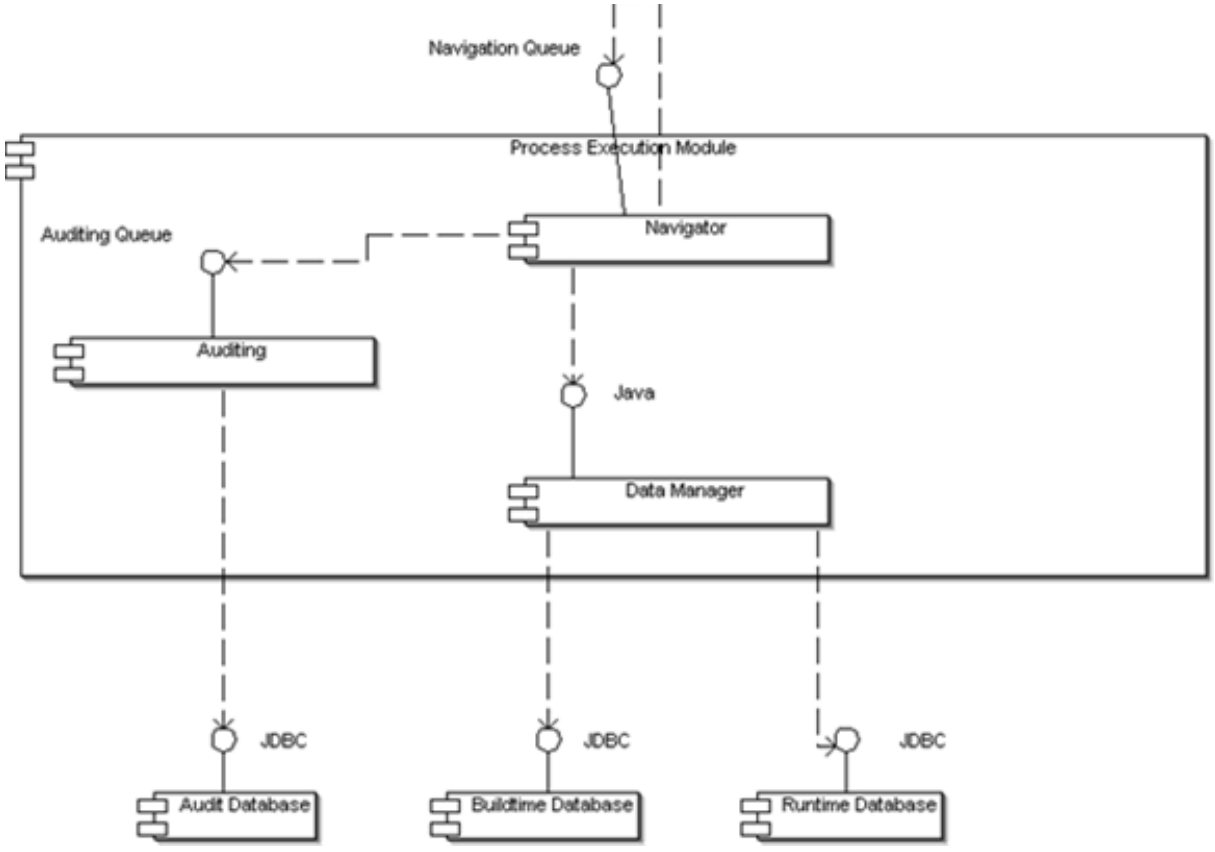


Figure 3-2 Internal dependencies and the relationship with databases

3.2.1 Navigator

Navigator performs the navigation through process instance, which is a center in the process execution model. The concrete navigation includes following tasks:

1. Execute activities;
2. Evaluate join conditions (By a simple sequential process model this evaluation phase can be simplified);
3. Compute the next activities;

4. Evaluate transition conditions.

Navigator listens on the navigation queue for messages which contain tasks for the navigator. And it's either the navigator or an invocation handler or a service provider put those messages into the queue. The navigator uses invocation queue to invoke remote web services, while reply queue is designed to send replies back to the web service clients in a synchronous invocation. The navigator controls the workflow of process models as well as reading from and writing to the runtime database by the data manager. And it puts messages into auditing queue to inform auditing components of the execution of activities..

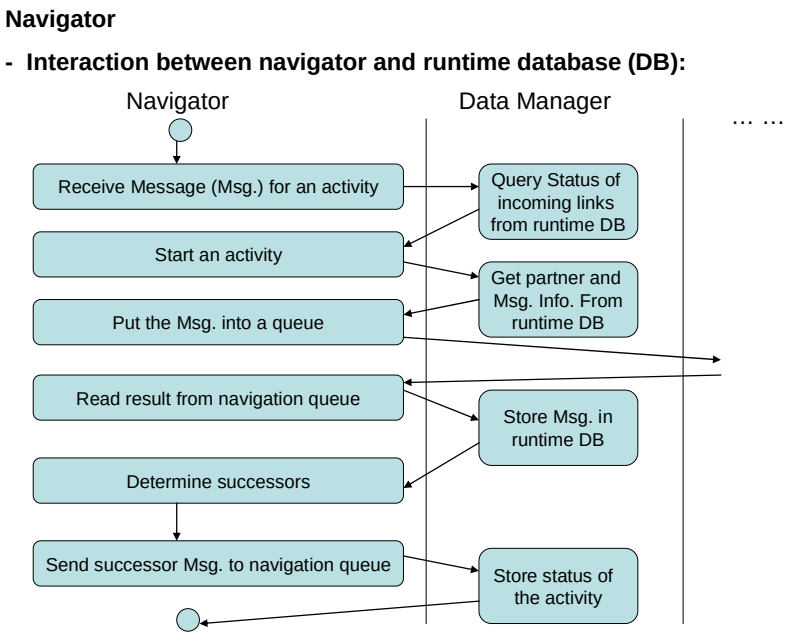


Figure 3-3 Main workflow of Navigator and its relationship with Data Manager

The navigator performs the navigation through a process instance. Within the navigator three kinds of queues are to be dealt with. They are Navigation Queue, Invocation Queue and Reply Queue. As the above figure (See Figure 3-3) has described, the Navigator works as follows: Firstly it receives message for activity from the navigation queue; then it starts to execute the activity and put the message into a queue, for example, an invocation queue. After performing the activity, the navigator reads result message from navigation queue and then determine a successor which is an activity following up. And finally the navigator sends this successor message to navigation queue.

In each phase the navigator has closed relationship with the data manager. And at most

time it communicates with runtime database. Before starting an activity, the navigator needs the status of incoming links from the runtime database, and then it gets partner and message information again from the runtime database. On finishing the execution of an activity, the message and results are stored in the runtime database and finally the status of the activity is to be saved there.

In a word, navigation is done by executing the activities, computing the successors and evaluating transition conditions. With the help of the data manager, the navigator works on process models and reads from and writes to the runtime database where all information about the process instance at runtime, namely, the communicating messages, the status of activities and the execution results, is stored. In the meanwhile, the data manager provides process models with cache. And the cache manager takes charge of the management of the cache as well as the access to runtime databases. In addition, it contains the transformation component for transforming the process model from the representation stored in the buildtime database where all information about process models imported into the SWoM is stored, to an executable one. Therefore the data manager has access to both of the runtime and the buildtime database. In the meanwhile, the navigator uses the data manager to get process model specific information from the cache and to store process instance specific information in the Runtime Database.

3.2.2 Data Manager

The data manager manages the process models so that no access to the buildtime is no needed when a process instance is being carried out or created. Furthermore it determines the set of next activities when the navigator needs the next activity. Additionally, the data manager has access to the runtime database.

3.2.3 Auditing

The auditing element writes the audit data into the audit database. The auditing listens on the auditing queue.

3.3 Software Requirements

SWoM is installed on Websphere Application Server. And the minimum requirements of software to install and run SWoM are as follows:

1) WAS(Websphere Application Server) is the running environment for SWoM

IBM WebSphere Application Server (WAS) is the flagship product within IBM's WebSphere brand. WAS is built using open standards such as J2EE, XML, and Web Services. Multiple IBM labs around the world participate in creating WebSphere run-time products and development tools.

2) DB2, which is the environment for three databases used to run SWoM, namely, the Buildtime, the Runtime and the Audit Database.

Three SQL scripts (See Table 3-1) exist for creating the table structure needed by the SWoM, one for each database:

Database	**Script**
Buildtime Database	bt.sql
Runtime Database	rt.sql
Audit Database	au.sql

Table 3-1 SQL Scripts

3.4 Summary

This Chapter gave an overview of the project SWoM, Stuttgarter Workflow Machine. It presented the architecture of the entire system and depicts the center module, project execution module in detail, which comprises three components, navigator, data manager and auditing. In this thesis, navigator and data manager are to be optimized. Furthermore, functional as well as non-functional requirements and software requirements were stated, in order to clarify the running environment of SWoM..

4 Optimize Database Schemas

This section firstly sets a research goal for this thesis, then depicts the design and implementation of optimizing of table schema in runtime database and the related navigator, and finally gives out the test and its result.

4.1 Goals

The general navigation model is surely suited for all kinds of process models. However, for a very simple process model, for example, a very simple sequential process model with just three activities. It should have been proceeded very quickly and easily. But it takes more resources and time to run while the general navigation model which is complete and so complicated is used. Therefore the process model is not effective for simple process models. In order to optimize, we need to give a certain kind of process model a matched navigation model, so that the related process instance could run effectively and efficiently.

4.2 Design and Implementation

This section presents the design and implementation for the optimization of the runtime database with BPEL engine.

4.2.1 Variable Structure

Here we use vector to describe variables so that we can handle all the variables required in a process instance as a tuple.

4.2.2 Categorization of the Process Models

We categorize all the process models into five categories, according to the data amount they refer to, namely simple or complicated, and the styles of the workflow of process models, namely sequential or parallel:

1. Simple sequential process model (SS)
2. Complicated sequential process model (CS)
3. Simple parallel process model (SP)
4. Complicated parallel process model (CP)
5. Others

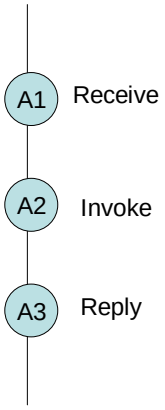
In order to avoid naming conflicts, the abbreviation of SS, CS, SP and CP are added before the original names as prefix. We categorize process model types in order to categorize the navigation types. For each type of the process models, a related optimized navigation type is to be used. For example, for simple sequential process model, there is a special navigation model instead of the original complete process model, so that the resources and time used to proceed the process instance would be reduced to the maximum extent.

4.2.2.1 Simple Sequential Process Model

Activities in this process model are carried out sequentially. And this workflow refers to only small amount of data. The following figure (See Figure 4-1) shows an example of a simple sequential model.

In this example, it's a very simple sequential process. All the three activities to be taken are: to receive a message, to invoke an operation and then to give a reply, As an abstract

I. Example for simple sequential process



PIID	PMID	AID	State	Variables
471	PM1	4711	1	(a, b)
471	PM1	4712	0	(a, b)
471	PM1	4713	0	(c, d)

I. Field *state* contains the current state of an activity.
0 for ready,
1 for running,
2 for suppressingJoinFailure,
3 for compensating,
4 for finishedNormally,
5 for finishedAbnormally.
II. The variables are kept in a tuple in the field Variables.
III. *AID 4711 for A1, 4712 for A2,4713 for A3, etc.

Figure 4-1 An example of simple sequential process model

process model, the process instance contains three activities, that is, A, B and C. And

their AID (ActivityID) are respectively 4711, 4712 and 4713. All of them belong to the process instance with PIID (ProcessInstanceID) 471 which is submitted to the process model with PMID (ProcessModelID) PM1.

Here variables are scored in a tuple, which is of type BLOB(Binary Large Object) in DB2. The Blob data type allows storing all the variables in a tuple, thus taking advantage of the different mechanisms of integrity and control provided by the database management system. It allows seeing all the variables within a set as a unit to be handled. And thus to keep all the variables in a tuple helps improve the organization of the information. Furthermore, the set of variable stored in the field of Variables are organized in the form so that access to the individual variables can be done via some direct access without the need to permanently serialize and deserialize all variables.

Tables in the Runtime Database

At runtime all information about a simple sequential process instance is stored in the Runtime Database. Furthermore the Runtime Database contains information about the Compensation Log for processing the compensation as well as runtime errors and message delivery errors occurring at runtime. Following tables show information that is stored in the Runtime Database.

1) SIMPLESEQUENTIAL_PROCESS

In this table (See Table 4-1) all information about a simple sequential process instance is stored. For a simple sequential process instance, we can use only one table to store activities and variables.

Name	Data	Mandatory	Description
PIID	Character (8) for bit data	Mandatory	It contains the unique identifier which represents the process instance in the database. With AID together it constitudes the primary key.
PMID	Character (8) for bit data	Mandatory	It contains the identifier of the associated process model in the Buildtime Database. This is a foreign key.
AID	Character	Mandatory	It contains the identifier

	(8) for bit data		of the associated activity definition in the Buildtime Database.
State	Smallint	Mandatory	It contains the current state of a process instance this can be 0 = creating, 1 = running, 2 = terminating, 3 = terminated, 4 = fault handling, 5 = finished abnormally, 6 = finished normally, 7 = suspended or 8 = deleted
Variables	Blob (4 K)	Mandatory	It contains the current value of the variable.

Table 4-1 Simplesequential_process

2) SIMPLESEQUENTIAL_COMPENSATION

In this table (See Table 4-2) all information about the compensation log is stored. Zero or more CompensationLog entries are related to one ProcessInstance.

Name	Data	Mandatory	Description
CompensationLogID	Character (8) for bit data	Mandatory	It contains the identifier of the compensation log entry, which is the primary key.
PIID	Character (8) for bit data	Mandatory	It contains the identifier of the associated process instance and is a foreign key.
AID	Character (8) for bit data	Mandatory	It contains the identifier of the activity instance that was completed. It is a foreign key.
CompensationSeq	Bigint	Mandatory	It contains an autoincremented value. The compensationSeq is used to create an order in the compensation log of a process instance.
BPELVariableID	Character	Mandatory	It contains the identifier

	(8) for bit data		of the associated variable definition in the Buildtime Database. It is a foreign key.
State	Smallint	Mandatory	It contains the state of the CompensationVariable. This can be “0” for unintialized or “1” for initialized.
Value	Blob (4 K)	Mandatory	It contains the value of the CompensationVariable.

Table 4-2 Simplesequential_compensation

DDL(Data Definition Language)

In this DDL script, four tables for the simple sequential process model are to be created:

```
UPDATE COMMAND OPTIONS USING c OFF ;

CREATE TABLE SIMPLESEQUENTIAL_PROCESS
{
   PIID              CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   PMID              CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   AID               CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   STATE             SMALLINT           NOT NULL WITH DEFAULT 0 ,
   VARIABLES    SMALLINT          NOT NULL WITH DEFAULT 0 ,
   CONSTRAINT PK_PROCESS PRIMARY KEY (PIID,AID)
};

CREATE TABLE SIMPLESEQUENTIAL_COMPENSATION
{
   COMPENSATIONLOGID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   PIID                          CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   AID                           CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   COMPENSATIONSEQ      BIGINT                NOT NULL WITH DEFAULT 0 ,
   BPELVARIABLEID          CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   STATE                          SMALLINT          NOT NULL WITH DEFAULT 0 ,
   VALUE                          BLOB (4K)       NOT NULL LOGGED NOT COMPACT,
   CONSTRAINT PK_COMPENSATIONLOG PRIMARY KEY
                                        (COMPENSATIONLOGID)
```

};

COMMIT;

In the simple sequential process model, the new table of Simplesequential_Process is a combination of original tables of Process and of Activity, which includes all the information about the process model ID, the process instance ID, its related activities' ID, the corresponding state and the variables.

Furthermore, the new table of Simplesequential_Compensation is a combination of original tables of Compensationlog and of Compensationvariables, which simplifies the structure and includes all the compensation related information inside one table.

Compare with the structure of the original and the optimized Navigator

The figure (See Figure 4-2) gives out the original structure of a workflow machine. It tells about how all the data (information) flows by navigation. Here, buildtime database, runtime database, cache and navigator are four main nodes. The Navigators access cache via appropriate Cache Manager. Complete process models get cached from buildtime database to executable component. Then the navigator can go through the process model with the access to cache instead of buildtime database.

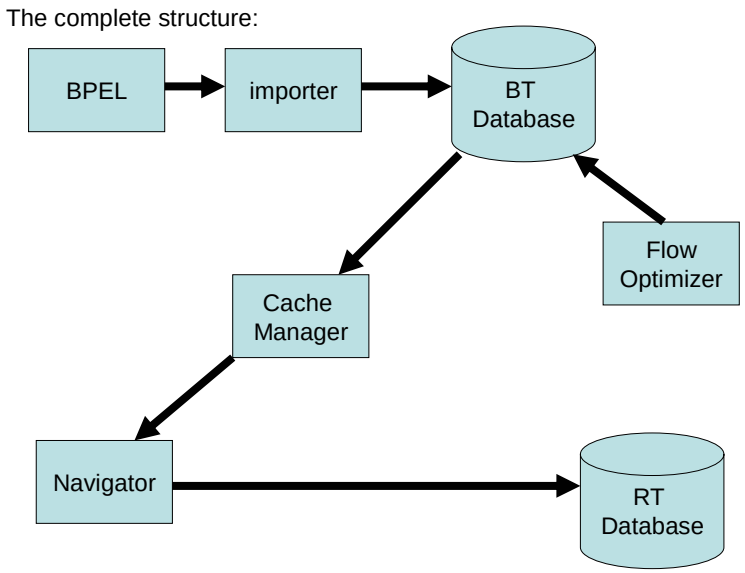


Figure 4-2 Structure of Original Navigator and Data Manager

For the optimization, we must take types of process models into account. As we've discussed above, process models are categorized into five types: simple sequential process model, complicated sequential process model, simple parallel process model, complicated parallel process model and others. Thus, a flow optimizer is used to distinguish all the process models into five types, and then record the type into the buildtime database. Therefore a new field is added into the buildtime table BPEL (See Table 4-3), named PMType which contains the information about the types of process models.

BPEL table in buildtime database

Name	Data Type	Mandatory	Description
PMID	Character (8) for bit data	Mandatory	It contains the primary key which uniquely identifies the BPEL process model in the database.
PMType	Character (8) for bit data	Alternative	It contains the information about the types of process models: 0: simple sequential PM 1: complicated sequential PM 2: simple parallel PM 3: complicated parallel PM 4: others
RootAid	Character (8) for bit data	Mandatory	It contains the identifier of the activity which is the very first activity in the process model. This should reasonably be a <flow> activity. It is a foreign key pointing to BPELActivity.
Name	Varchar (255)	Mandatory	It contains the name of the process model as given in the BPEL file.
TargetNamespaceName	Varchar (255)	Mandatory	It contains a URI which is the target namespace name of the process model.

Querylanguage	Smallint	Mandatory	It contains the foreign key to the name of the language used for queries in the language table. The default value is 0, which points to the name "http://www.w3.org/TR/1999/REC-xpath-19991116".
Expressionlanguage	Smallint	Mandatory	It contains the foreign key to the name of the language used for expressions in the language table. The default value is 0, which points to the name "http://www.w3.org/TR/1999/REC-xpath-19991116".
SuppressJoinFailure	Smallint	Mandatory	It contains the value of the suppressJoinFailure attribute of the BPEL process. Only 0 ("no") and 1 ("yes") are legal values. The default value is 0.
EnableInstanceCompensation	Smallint	Mandatory	If the value is set to "yes" the process instance as a whole can be compensated. Only 0 ("no") and 1 ("yes") are legal values. The default value is 0.
AbstractProcess	Smallint	Mandatory	If the value is set to "yes" the process model is abstract. Only 0 ("no") and 1 ("yes") are legal values. The default value is 0.
DefaultNamespaceName	Varchar (255)	Mandatory	It contains the default namespace name of the process model.The default value is "http://schemas.xmlsoa

			p.org/ws/2003/03/business-process/".
State	Smallint	Mandatory	It contains the state of the process model. "0" means "deployable", "1" means "deployed" and "2" means "undeploying".
Filename	Varchar (255)	Mandatory	This contains the filename of the BPEL process file inside the SPAR file.
IsSynchronous	Smallint	Mandatory	This is 1 if the process model is synchronous and 0 otherwise.

Table 4-3 Optimized table in buildtime database

Because navigator is directly connected with cache manager, in order to notify navigator what kind of navigation is to be followed, the stored information about the process model in the cache is changed concurrently by adding the type of the process model.

Inside the navigator, a if-then structure is used. At first it judges which navigation type is to be followed according to the process model type recorded in the cache. The algorithm is described as follows:

If PMType = 0 then run Navigation Model for Simple Sequential PM;
If PMType = 1 then run Navigation Model for complicated Sequential PM;
If PMType = 2 then run Navigation Model for Simple Parallel PM;
If PMType = 3 then run Navigation Model for complicated Parallel PM;
If PMType = 4 then run general Navigation Model;

After choosing the required navigation model, the related changed runtime database schema is to be selected. For instance, if we run the navigation model for simple sequential process model, a integrated table with all information about process model, its activities and all the variables to be used, as we've told above, is to be imported.

Implementation of Optimized Navigator

In the optimization model, the simple navigator is reconstructed into a "fork" structure (See Figure 4-3), that is, a center navigator with a switch structure which leads to

individual sub-navigator for corresponding type of process model.

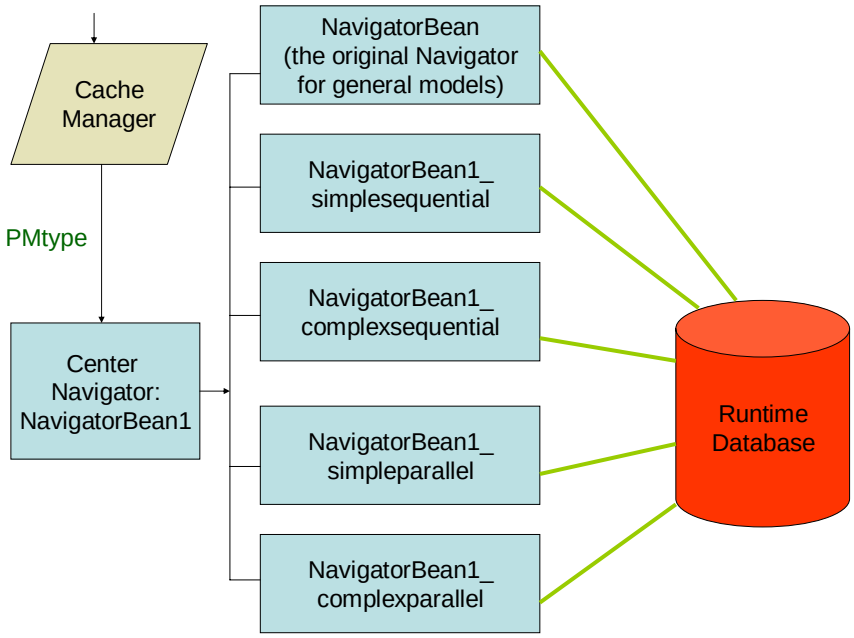


Figure 4-3 Center navigator and its “fork” structure

In the new model, it is the center navigator which is directly connected with the cache manager, instead of a simple navigator. And the center navigator reads the type of process model from cache manager while cache manager gets the PMType from the BPEL-table (See Table 4-4) in buildtime-database.

The original simple navigator turns to be a branch as a sub-navigator. The implementation is realized by pointing back to the original navigator directly from the center navigator. And it is used for all the normal process models except the four specific types of process models, named general process model in the thesis. And this branch is also a default branch. If there is no type of process model specified or the processType is 4, then the center navigator switches directly to this branch.

And the new navigator center, compared to the original simple navigator, instead of getting the start activity after providing respective access to cache manager and runtime-database and getting the basic information about the process model to be executed, for example, the process model ID(PMID), the process instance ID(PIID) and the type of process model(PMType), runs the following “fork” structure firstly, in order to simplify the whole process for a pre-defined type of process model. And the individual type of process model accesses the runtime-database respectively.

And the running procedure is analogous for the complex sequential process model, simple parallel process model and complex parallel process model. The difference of the actual execution among these three types is the process instance to be created in the runtime-database which should follow the different table schema for each of the process model.

The following code shows the navigator process in a simple sequential process model:

```
1 SELECT AID
      INTO :activityId
      FROM Simplesequential_process
      WHERE (AID = :aid)
2 activityState = finished
  UPDATE Simplesequential_process
      SET state = :activityState
      WHERE AID = :aid
3 DECLARE targetActivities AS CURSOR FOR
      SELECT TargetAct FROM Controlconnector
         WHERE (SourceAct = :activityId)
4 OPEN targetActivities
5 FETCH targetActivities
        INTO :activityId
6 Check start condition
7 IF TRUE THEN
      EXECUTE staff resolution and workitem assignment
8 ELSE
      Abort
9 CLOSE targetActivities
```

Statement1 determines the activity in the process model that is associated with the activity instance that just completed by accessing the process table with the AID which compared to the original model, the tables of process and activities are subsumed into one table.

Statement2 updates the state of the activity to finished. The activity instance cannot be deleted since it may be needed for process monitor queries.

Statement3 defines a cursor that allows the control connector (See the section “Judge a Sequential Process”) having the completed activity as its source to be retrieved. The cursor is opened in Statement4.

Statement5 determines the target activity, then goes on with the navigation.

Statement6 checks the start condition. If the start condition evaluates to TRUE, then

Statement7 performs staff resolution and generates workitems. If the start condition evaluates to false, then

Statement8 causes the activity to be skipped and process is aborted.

Statement9 closes the cursor that was used to retrieve the target activities.

Judge Whether It's a Simple or a Complex Sequential Process

There are two methods to judge whether it's a simple sequential process or a complex one. Either it's judged by the user himself. The User decides and chooses which type of process model is to be followed; Or it's judged by the machine. Instead of the user, the machine is in charge of the choice-making automatically.

1 Judged by the user

The idea is that: Firstly, to design a checkbox in the interface and the user will then make a choice: either simple or complex before he input all the application data. This choice is to be written into a table which is newly created and designed to record the choice separately. It is stored in the BT-database, and to be read in the cache manager before the switch command in the navigator.

It will bring two advantages. It shows what the user wants and reflects the user's requirement and willing perfectly. And thus, it's very clearly and obviously to tell the program which process model to follow up. And no need to run any other program to help do this judgment.

However, there're still some disadvantages. Firstly, a user is not a professional expert who is not familiar with any technical stuff. Therefore, his judgment is very subjective. And thus, the choice the user's ever made could be a mistake, which might lead to an unnecessary resource wasting. For example, if he took a simple process for a complex process model, then it would take more time and occupy more resources while running the program.

2 Judged by the machine

Here, the machine will offer a judgment, instead of the user.

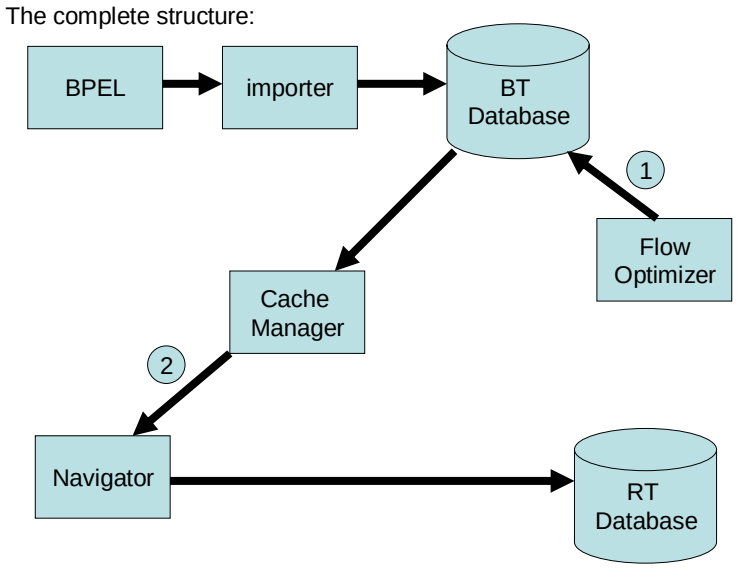


Figure 4-4 Place where there is the structure to be optimized

The figure above (See Figure 4-4) tells the complete structure of our model. There are five types of process model. Therefore, before running a process model, we must judge whether it's sequential or parallel, and whether it's simple or complex, so that a correct type of process model can be followed.

- In place 1: judge whether it's sequential or parallel
- In place 2: judge whether it's simple or complex

The key to judgment in place 2 is data amount. After the user input all the required data into the database, a way to get the data amount is to calculate the total size of tables. Usually, tables used in a complex process model have large size, and small size for simple ones. However, which size is a limit, so that we can say, inside this limit, we judge the process as a simple one, and out of the limit, as a complex one?

However, how to get a data amount limit for a simple sequential process?

For each process model there exists a data amount limit. The ideal way to get this limit is to make a researching survey. Here, we still need the cooperation by the user.

The user should firstly tell, to his mind which kind of process could be complex. According to that, ten or twenty (the more the better) complex samples are to be listed.

And then, use these samples to do testing. For each sample, its total table-size is to be recorded. And testing results will to be averaged and set as a limit finally.

And there are following advantages. Compared to the first way, the judgment is objective. It's independent from the user. The machine can do it automatically and correctly on the condition of a good survey.

Judge a Sequential Process

In this section, the problem of how to judge a process instance as a sequential process model is to be dealt with.

Important Definition

There are three important concepts here in order to clarify the whole procedure to judge a sequential process clearly. They are PM-Graph, control connector and data connector.

1) PM-Graph

The syntax and semantics of the workflow model are based on graph theory. Suppose that $\mathcal{P}$ stands for the set of all process models. A particular process model $P \in \mathcal{P}$ is represented as a special kind of directed graph G which is called "PM-Graph". The note set N of G comprises the set of all activities of the associated process model P. The edge set E of G provides all possible sequences of activities within P. A set of conditions $\mathcal{C}$, for example, the business rules, preconditions and post-conditions of a certain activity, etc, determines the actual activity sequence of a particular instance of P.

Here G is the mathematical representation of the model P of a given business process. The instance of G consists of the activities which are the components of a process instance, conditions which determine the successors to be executed which are based on the actual output data, and the data flow itself within the process instance which is changed by performing the workflow.

Therefore, a process model can be taken as a map, each activity, the node, affects the actual path taken through the graph which determines the sequence of nodes, which conditions are blocks and determine whether the workflow can go further or not.

2) Control Connector

Suppose that the set $E \subseteq N \times N \times \mathcal{C}$ is called the set of control connectors of a process model $P \in \mathcal{P}$. For a control connector $(A, B, p) \in E$, the predicate $p \in \mathcal{C}$ is called a transition condition. And each transition condition p is considered as a Boolean function in its input container and results in $\{0, 1\}$ in its output container. If no transition condition is explicitly specified, a transition condition then always evaluates to be true (!) by default, and the control always flows along such a control connector to its target after its source completes.

However control connectors cannot bind activities in an arbitrary manner. There are still two restrictions for the set of control connectors of any process model:

One is called unified, that is, between two given activities exists at most one control connector. Two activities can be either separated or connected with an only link (See the following examples in the Figure 4-5). There's not a third case.

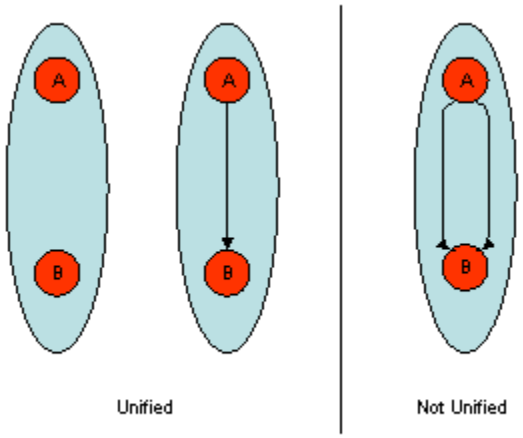


Figure 4-5 Unified control connectors

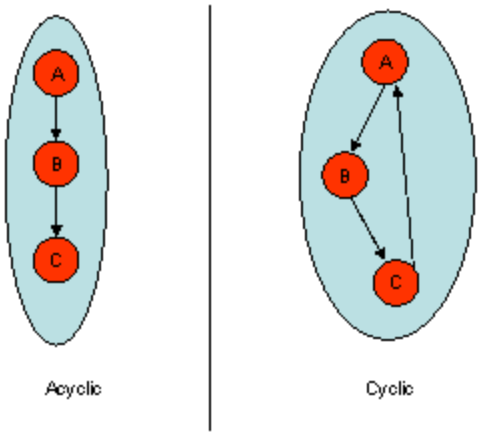


Figure 4-6 Acyclic control connectors

The other restriction is called acyclic, that is, control connectors must not build loops and one activity has no path to get back to itself. The workflow starts from a certain activity A along one of its outgoing control connectors to B, one of its successors, and then from B to its possible successor C, ... , and C will never come back to the original activity A. (See the above examples in the Figure 4-6)

Thus, the set of control connectors of a process model is both unified and acyclic.

Suppose that E is the set of control connectors of a process model P, and $e_1, \ldots, e_n \in E$. The sequence $(e_1, \ldots, e_n)$ is called a path from the activity $A \in N$ (A is the start node of the path) to activity $B \in N$ (B is the end node of the path).

3) Data Connector

Data plays a great role in the workflow. Therefore, a business process model contains the definition of the process data flow of a certain process instance. The definition is divided into two parts. One is about the collection of input and output containers of all activities and conditions of the process model, which is called data dependencies; the other one is about how data is exchanged between the activities in the process model, which is called data mapping.

Suppose A is an activity and B is either another activity or a predicate. If B requires data as input that is expected to be produced by A, then B is data dependent on A. And data flows from A to B. The data dependency of B on A is expressed by a data connector, a directed edge from A to B. Here data elements of B's input container expect values from data elements of A's output container.

Examples

Here are two examples to tell how to judge a sequential process model:

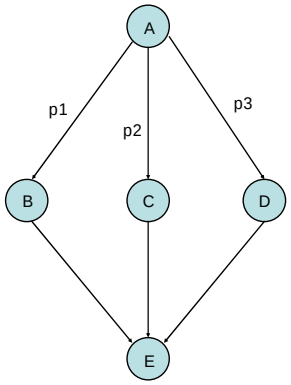


Figure 4-7 Example 1(1)

Example 1:

Suppose that there is a process model P with five activities A, B, C, D, E. The control connectors existing among them are as follows: (A, B, p_1), (A, C, p_2), (A, D, $p_{3)}$, etc. that is :

CONTROL FROM A TO B
WHEN p_1 IS TRUE
CONTROL FROM A TO C
WHEN p_2 IS TRUE
CONTROL FROM A TO D
WHEN p_3 IS TRUE

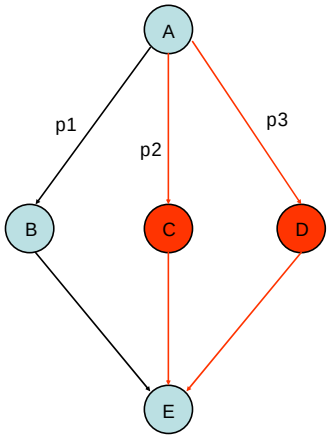


Figure 4-8 Example 1(2)

To be a sequential process model, following conditions have to be met simultaneously:
$p_1 \cap p_2 = \varnothing$
$p_1 \cap p_3 = \varnothing$
$p_2 \cap p_3 = \varnothing$

That is to say:
1. At most one path at one time is to be selected and will be followed: (A, B, E) or (A, C, E) or (A, D, E). From the above figure, obviously, the process model is then sequential.

2. Assumed that is p_1 true and p_2, p_3 are both false, then only (A, B, E) is executable, while (A, C, E) and (A, D, E) are both dead path. For variable control, all the variables for C and D are thus regardless.

Example 2:

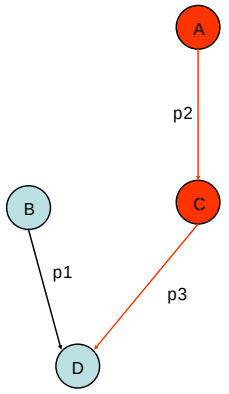


Figure 4-9 Example 2

Suppose that there is a process model P with four activities: A, B, C and D. And the control connectors are as follows: (B, D, p_1), (A, C, p_2) and (C, D, p_3). Assume that is p_1 true and p_2, p_3 are both false, then only (A, D) is executable while (A, C, D) is a dead path. For variable control, the variables for C are regardless and the variables for A can be released, because they are no use any more.

And the topics of dead path, dead successors and variable control are to be discussed in the following section.

Dead Path, Dead Activity, Successor and Variable Control

In the PM-Graph there is a normal structure, join. At a join normally join condition should be judged. Join condition is defined as the set of all Boolean conditions at the join in disjunctive normal form of transition conditions of control connectors pointing to the activity. In the second example above, assume that (A, C, D) hasn't been judged as a dead path yet, then (B, D), (C, D) build a join and p_1, $p_3$are join conditions.

Waiting at a join until all incoming transition conditions have been evaluated would increase the complexity level for implementation of workflow management systems. Therefore a workflow management system must automatically resolve situations by means of evaluation of dead path. That is, before all paths reach the join activity, the workflow management system judge one path as executable, the rest as dead path, so that the level of complexity at the join would be degraded.

In the second example above, the figure presents a very simple process model P with the join activity D. Assume that in a particular instance of the model P, Activity A and B are completed, p_1 has been evaluated to be true and p_2 has been evaluated to be false. Thus, Activity C will never be scheduled, for its only incoming transition condition p_2 is false.

As a result, the path (A, C, D) is “dead”, called a dead path. It stopped at C, and p_3 will never be evaluated so that the join condition of D will never be evaluated. D starts since its join condition is known to be true because of p_1.

In this example, regular node C is a dead activity, because it has an incoming control connector with the transition condition p_2 to be evaluated to be false. In addition, a join activity can also be a dead activity, if its join condition is evaluated to be false, that is, all its incoming control connectors with the transition conditions to be evaluated to be false.

As the figure shows, in edge AC, activity A is the start node while activity C is the end node. Therefore, C is the successor of A. And so are the edges of BD and CD. D is the successor for both B and C. To judge the successor is one of the important tasks by navigator.

For the variable management, since (A, C, D) has already been judged as a dead path. C is a dead activity. In this instance, the variables only used for A and C can be regardless and thus during the next update, those variables can be released.

Causal Relationship Can Help Judge a Sequential Process?

Last section tells that it is a good way to help judge a sequential process by using graph theory, PM-Graph. In this section, causal relationship of input and output between linked activities is to be discussed. Can it also help judge a sequential process? If so, how to deal with the variable control?

Here, all variables used by the activity, or input- and output containers are to be observed.

Assume that there exist two activity nodes A_l, A_m. For A_l the input container is I_l while the output container is O_l. For A_m the input container is I_m while the output container is O_m.

The evaluation process is as follows:
1. If $((O_l \cap I_m \neq \varnothing) \cap (I_l \cap O_m = \varnothing))$, then $A_l \Rightarrow A_m$, they are sequential;
2. If $((O_m \cap I_l \neq \varnothing) \cap (I_m \cap O_l = \varnothing))$, then $A_m \Rightarrow A_l$, they are sequential;
3. If $(((O_m \cap I_m \neq \varnothing) \cap (O_m \cap I_l \neq \varnothing)) \cup ((O_l \cap I_m = \varnothing) \cap (O_m \cap I_l = \varnothing)))$, then A_l and A_m are parallel. (See Figure 4-10)

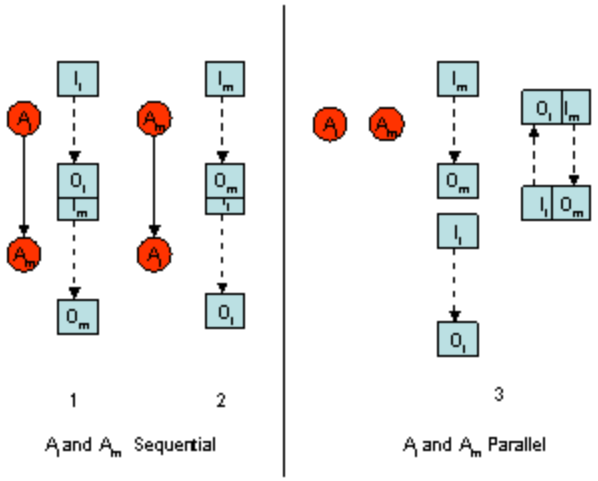


Figure 4-10 Causal relationship

Either the output container of A_l and the input container of A_m have some elements in common, that is, A_l, A_m might have causal relationship and A_l is ahead of A_m; or vice versa, the output container of A_m and the input container of A_l have some elements in

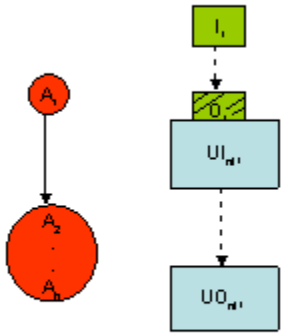


Figure 4-11 Variable control according to causal relationships

common, and A_m is ahead of A_l, which is shown in the left side of the above figure. If both the input- and output containers of A_m have nothing to do with both the input- and output containers of A_l or the output container of A_m and the input container of A_l are

related while the input container of A_m and the output container of A_l are related, and then A_m and A_l are parallel, which is shown in the right side of the above figure.

Then, how to control the variables according to these causal relationships?

Assume that there exists a sequential process instance with activities A_1, A_2, …, An in the order of (A_1, A_2, …, An). (See Figure 4-11) A_1 has input container I_1 and output container O_1 while the rest activities have input containers expressed as $UI_{n/1}$ and output containers expressed as $UO_{n/1}$. Therefore, after is completed, its updated output container is $O_1-(O_1\cap(UI_{n/1}))$, and all the elements(variables) in the set $O_1-(O_1\cap(UI_{n/1}))$ (The shade part in the following figure) can be released.

Possible to Simplify a Parallel Process into a Sequential Process

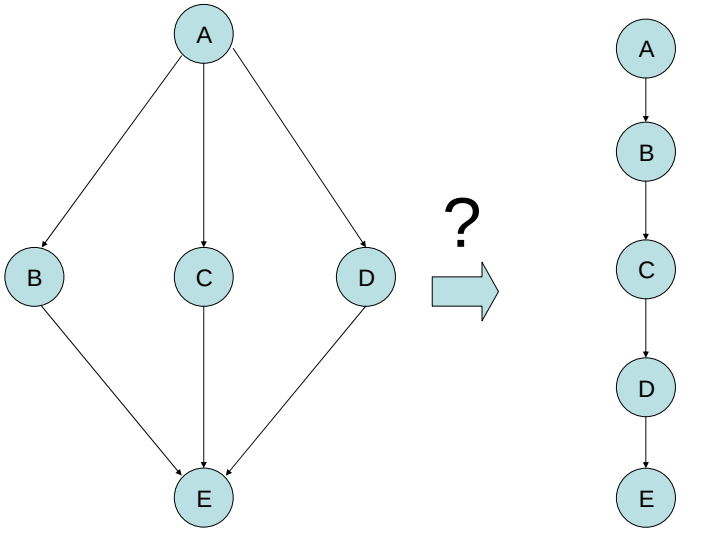


Figure 4-12 Example of the conversion from a parallel process model into a sequential process model

It's possible to reduce the parallel process model into a sequential process model. And two points are needed to take into consideration.

1) Data amount

If it is a process with large data amount, the originally parallel process model can runs

obviously faster then a sequential process model, then it's no need to simplify the process. An important advantage of parallel process model to its equivalent sequential process model is to reduce the running time.

With a small amount of data, the simplification would be worth. Against a parallel process model, the running process of a sequential process model is far easier.

In a word, we should balance two points: running time and the complex degree of running process models. And the key is the data amount. How to get a data amount limit? We can do the same kind of research survey to get an experienced result set as a limit.

2) Potential variable conflicts

To simplify a parallel process model to a sequential one might leads to variable conflicts. For example, in the above figure, the left side is a parallel process. There are three paths ABE, ACE and ADE. And there is no logical relation among them. That's to say, except the same start and the same end, each path has its own running process and the variables and is to be taken as an independent and complete unit. Input variables for B have nothing to do with input variables for C and D. And there is no relationship between the input variables for C and D. And so are the output variables. However, after the simplification, the output of B becomes the input of C and the output C becomes the input of D. Thus, variable conflicts might occur.

How to solve this problem?

Firstly calculate how many paths the parallel process model has before the simplification. In this example (See Figure 4-14) there're three. And then, define 3 sets of input container (IC) and output container (OC): (IC1, OC1), (IC2, OC2), (IC3, OC3). Copy the original data input from the start into these three input containers to get ready for B, C and D, to

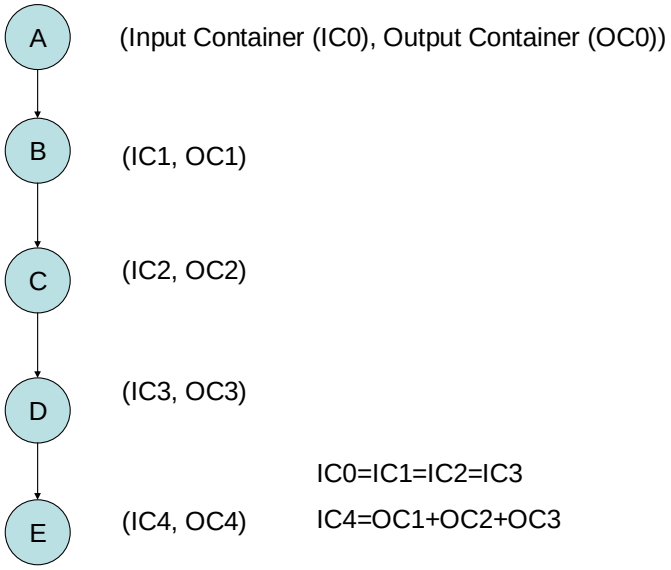


Figure 4-13 The solution of the conversion in the example

make sure running B has no influence on the change of variables in C and D. Therefore from the very beginning, IC1, IC2 and IC3 have the same input data. And after running B, C and D, the results are written into OC1, OC2, and OC3 separately. And OC1 has no relation with IC2, OC2 has no relation with IC3. And input variable for E is the join of OC1, OC2 and OC3.

In the end, record the first activity after the start on each path. Here we record B, C and D. Before running each activity, we make a judgment whether it's B or C or D.

If B, call (IC1, OC1)
If C, call (IC2, OC2)
If D, call (IC3, OC3)
If other activities, go on with the variable container being used.

Record the Entire Running Process

How to record the history depends how precise history the user wants. If the user just wants to follow the data change. In a sequential process, the easiest way is:

- Create a log file before the start and write the original input and the start in it.

- Record the name of the activity when the next activity is called.
- Record the input of the called activity.
- Whenever the program print variables, record these variables in the log as well.
- Record the output of the activity when it's over.
- Do the recording till the end.

If the running commands in the program also need to be followed, that's to say, after running each command, to write the log file is a must, to record the name of the command and the result of it, then it's much more complicated.

Tell the System Whether to Use Optimizer

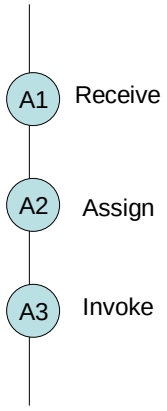
Actually, we don't need to judge whether we should use optimizer or not. Suppose that our program model is optimized. And all the processes are based on this optimized program model. All the business processes are categorized into five types: simple sequential, complex sequential, simple parallel, complex parallel and the general process model. As we have said, before running the program, a judgment must be made to decide which process model to follow. If the fifth type, the general process model is chosen, then automatically no optimization is to be used. Besides, for the other four types, as long as they are chosen, optimizer is called.

4.2.2.2 Other type of Process Models

In this section, other types of process models, namely complex sequential process model, simple parallel process model, complex parallel process model and general process model are to be depicted.

Complex Sequential Process Model

II. Example for complex sequential process



PIID	PMID	AID	Status
471	PM1	4711	
471	PM1	4712	
471	PM1	4713	

AID	VID	Variable
4711*	1	a
4712	2	b
4713	2	b

*AID 4711 for A1, 4712 for A2,4713 for A3, etc.

Figure 4-14 An example of complex sequential process model

The only difference between a simple sequential process model and a complex sequential process model is the data amount the process instance deals with. Within a complex sequential process model there are some activities which are more complicated and refer to large amount of data. For example, “Assign” shown in the above example (See Figure 4-14).

Here VID which stands for ValueID is intended for identifying each value so that this large amount of data, namely, each value is able to be well managed.

Due to data amount, to use a single table to describe both the process instance and the activities as depicted in the simple sequential process model is irrational. Therefore we divide this complete table into two parts, one is for the process instance; the other is for the activities which are to be executed in the process instance. Otherwise, it would result in data redundancy in the schema, if an only table were used for both of the process instances and the activities. Data redundancy costs more space to store the information and is not good for the management of database system. Obviously, it isn’t in accordance with the optimizing standards.

It is a normalization process from a composite table to two smaller tables. Here, the tables are in the third normal form. There are no non-trivial functional dependencies of non-key attributes on something other than a superset of a candidate key. All non-key attributes are mutually independent.

In the example of simple sequential model, PIID and AID is composite primary key. While in this example, VID and Value are fully dependent on AID, because the same activity there may exist many VID and Value, AID, VID, Value are derived from the composite table and build up therefore a separated table in the complex sequential process model.

Simple Parallel Process Model

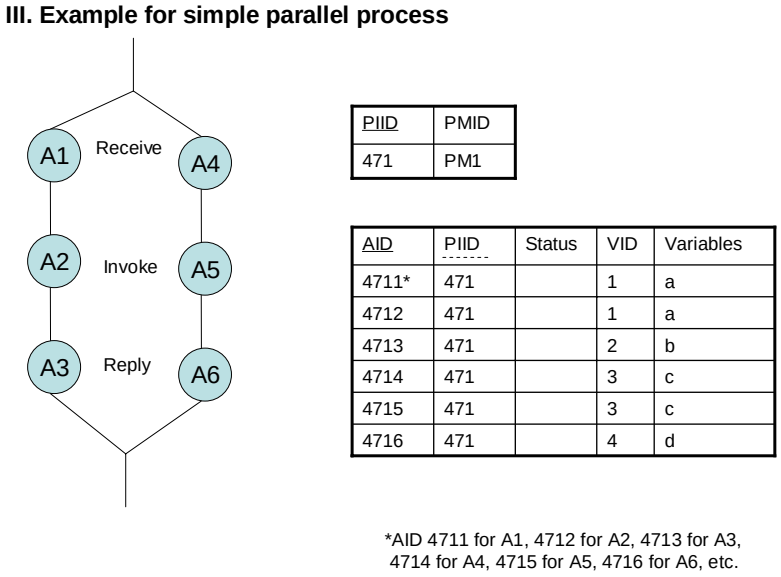


Figure 4-15 An example of simple parallel process model

Parallel process model is a very typical model type of all the processes. In a sequential process model the activities are to be executed one by one, which makes the structure simplified. Especially in a simple sequential process model, the complicated schema can be reduced into one table to describe the process instance as well as the activities, which makes the administration much easier. Compared with a sequential process model, a parallel structure helps reduce the running time for the entire process. Through a fork structure, several activities which have no dependencies on each other can be executed

simultaneously.

In the above example (See Figure 4-15), AID is fully derived from the first table for the description of process instance, compared with last example of a complicated sequential process model. If AID were still kept in the table of process instance, it would lead to the problem of data redundancy.

It is also possible to reduce a parallel process model into a sequential process model in order to get a simplified structure and make the workflow easy to be controlled. (See 4.2.2.1.8 for details)

Complex Parallel Process Model

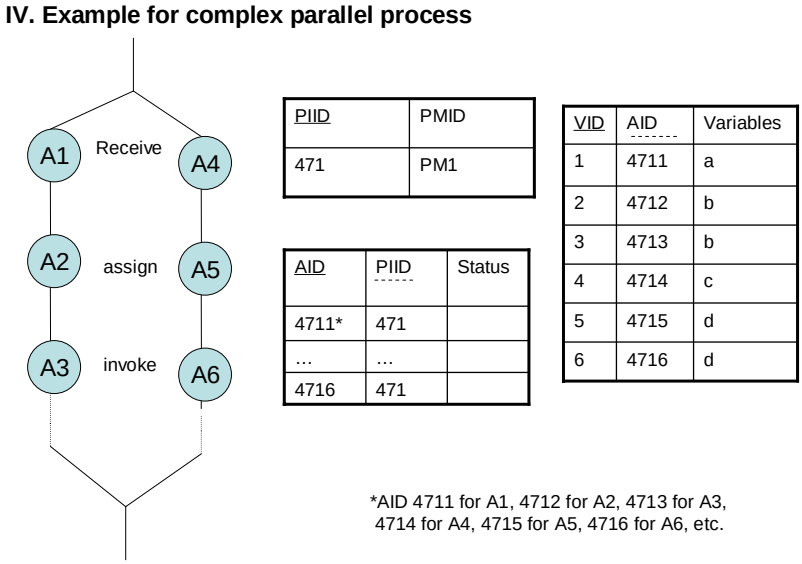


Figure 4-16 An example of complex parallel process model

The difference between a simple parallel process model and a complex parallel process model still depends on the amount of data to be dealt with. In the above example (See Figure 4-16), obviously, we derive VID and Value from the second table because of the great amount of the data. If VID and Value were still kept in the second table, it would result in the problem of data redundancy.

General Process Model

Actually general process model is right a normal process model. No simplification and optimization will be taken into consideration. For this process model, the original navigator before optimizing, which is set to be a default, is to be pointed to und used. The workflow of the navigator and the table schema in the database are the same as the process model before.

4.3 Test and Results

The J2EE codes for the navigator are programmed in the IBM Rational Application Develop Version 6.0 which runs in the environment of WebSphere Application Server. And they are deployed in the runtime database of SWoM using DB2.

Compared with the original process model, the newly optimized the process model for simple sequential process is simplified a lot. The number of the tables is reduced obviously. The length of SQL calls by the navigator in the runtime database is shortened. And the records of the running history are thus shortened. Therefore the simple sequential process model occupies less space not only by the reduced tables in the database but also by the shortened codes and records when the process is running. Furthermore, the running time of navigator when it proceeds in the simple sequential process is also reduced by few seconds.

4.4 Summary

This section gave an overview of the entire procedure of design and implementation and finally the corresponding tests for the optimization of the runtime database and concentrated on the structure of runtime database as well as the corresponding navigator.

The tables in the runtime database were optimized for different types of process models respectively. And the concept of center navigator was designed to figure out a picture of a switch structure which controls the switching of different types of process models. And then a specific navigator for the simple sequential process model was developed.

Furthermore, the topic of the control of variables, the judgment of a sequential process model and the conversion between a sequential process model and a parallel process model.

5 Summary and Outlook

This section provides a short conclusion of the optimization process for the runtime database and then gives an outlook in the future.

5.1 Conclusion

The goal to optimize the runtime database of the BPEL engine for SWoM is to simplify the whole workflow process of navigator so that the schema structure of tables are simplified which is easy for good administration, the execution time of the whole process can be reduced by designing individual navigator for each type of process models respectively, and the resources being occupied are able to be reduced by means of variable control.

Process models can be classified into five types: simple sequential process model, complex sequential process model, simple parallel process model, complex parallel process model and general process model. Here, to differentiate a sequential process model from a parallel process model depends on the detailed structure of the process model itself, while to distinguish a simple process model from a complex process model depends on the data amount involved in the execution of a process instance. Except all the process models of the first four types, the rest normal complicated process instances belong to the general process model. And navigator of a general process model is right the navigator developed before the optimization.

In this thesis, the simple sequential process model was concentrated on, including the optimizing of its table structures, DDL, the individual navigator for the process model, variable control, how to judge a simple sequential process model, whether it is possible to convert a sequential process model into a parallel process model, etc.

5.2 Outlook

This thesis emphasize on the simple sequential process model. For the other types only the database schemas and DDL are optimized. Thus, the future work is to develop ahead the individual navigators for these four types of database schema, and to complete the switch structure of the center navigator.

Appendix A BPEL Scripts for Each Type of Process Models

A-1.Process script for a simple sequential model:

```
<process name="471" targetNamespace="uri"
            queryLanguage=" "
            expressionLanguage=" "
            suppressJionFailure="yes"
            abstractProcess="no"
            xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">
  <import namespace="uri" location="uri" importType="uri"/>
  <partnerLinks>
        <partnerLink name="A" partnerLinkType=" " myRole=" "/>
        <partnerLink name="B" partnerLinkType=" " partnerRole=" "/>
  </partnerLinks>
  <variables>
        <variable name="a" messageType=" "/>
        <variable name="b" messageType=" "/>
        <variable name="c" messageType=" "/>
  </variables>
  <sequence>
        <receive partnerLink="B" portType=" " operation="receiveMessage"
                  variable="a">
        </receive>
        <assign>
            <copy>
                <from>$a.receiveMessages</from>
                <to>$b.receiveMessages</to>
            </copy>
        </assign>
        <invoke partnerLink="C" portType=" " operation="invokeAktion"
                  inputvariable="b">
        </invoke>
        <reply parnterLink="B" portType=" " operation="replyMessage"
                  variable="a"/>
        </reply>
  </sequence>
</process>
```

PS:

variable a works just like a bridge for variable b. The result has no change. And variable a is useless.

A-2.Process script for a complex sequential model:

```
<process name="471" targetNamespace="uri"
            queryLanguage=" "
            expressionLanguage=" "
            suppressJionFailure="yes"
            abstractProcess="no"
            xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">
  <import namespace="uri" location="uri" importType="uri"/>
  <partnerLinks>
        <partnerLink name="A" partnerLinkType=" " myRole=" "/>
        <partnerLink name="B" partnerLinkType=" " partnerRole=" "/>
        <partnerLink name="C" partnerLinkType=" " />
        ...
  </partnerLinks>
  <variables>
        <variable name="a" messageType=" "/>
        <variable name="b" messageType=" "/>
        <variable name="c" messageType=" "/>
        ...
  </variables>
  <sequence>
        <receive partnerLink="B" portType=" " operation="receiveMessage"
                  variable="a">
        </receive>
        <assign>
            <copy>
                <from>$a.receiveMessages</from>
                <to>$b.receiveMessages</to>
            </copy>
        </assign>
        <invoke partnerLink="C" portType=" " operation="invokeAktion"
                  inputvariable="b">
        </invoke>
        ...
  </sequence>
</process>
```

A-3.Process script for a simple parallel process model (See Chapter 2.4)

A-4.Process script for a complex parallel process model:

```
<process name="471" targetNamespace="uri"
            queryLanguage=" "
            expressionLanguage=" "
            suppressJionFailure="yes"
            abstractProcess="no"
            xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">
  <import namespace="uri" location="uri" importType="uri"/>
  <partnerLinks>
        <partnerLink name="A" partnerLinkType=" " myRole=" "/>
        <partnerLink name="B" partnerLinkType=" " partnerRole=" "/>
        <partnerLink name="C" partnerLinkType=" " />
        <partnerLink name="D" partnerLinkType=" " myRole=" "/>
        <partnerLink name="E" partnerLinkType=" " partnerRole=" "/>
        <partnerLink name="F" partnerLinkType=" " />
        ...
  </partnerLinks>
  <variables>
        <variable name="a" messageType=" "/>
        <variable name="b" messageType=" "/>
        <variable name="c" messageType=" "/>
        <variable name="d" messageType=" "/>
        ...
  </variables>
  <flow>
    <sequence>
        <receive partnerLink="B" portType=" " operation="receiveMessage"
                    variable="a">
        </receive>
        <assign>
            <copy>
                <from>$a.receiveMessages</from>
                <to>$b.receiveMessages</to>
            </copy>
        </assign>
        <invoke partnerLink="C" portType=" " operation="invokeAktion"
                  inputvariable="b">
        </invoke>
        ...
    </sequence>
```

```
    <sequence>
        <receive partnerLink="E" portType=" " operation="receiveMessage"
                 variable="c">
        </receive>
        <assign>
            <copy>
                <from>$c.receiveMessages</from>
                <to>$d.receiveMessages</to>
            </copy>
        </assign>
        <invoke partnerLink="F" portType=" " operation="invokeAktion"
                  inputvariable="d">
        </invoke>
        ...
    </sequence>
  </flow>
</process>
```

Appendix B DDL for Each Type of Process Model

B-1. DDL for simple sequential process model (See Chapter 4)

B-2. DDL for complex sequential process model:

Figure B-1 Complex sequential process table

```
CREATE TABLE COMPLEXSEQUENTIAL_PROCESS
{
  PIID      CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PMID      CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  AID       CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  STATUS VARCHAR (255)                     NOT NULL ,
  CONSTRAINT PK_PROCESS PRIMARY KEY (PIID,AID)
};
```

Figure B-2 Complex sequential activity table

```
CREATE TABLE COMPLEXSEQUENTIAL_ACTIVITY
{
  AID            CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  VID            CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  VARIABLES SMALLINT             NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_ACTIVITY PRIMARY KEY (AID)
};
```

Figure B-3 Complex sequential correlation set instance table

```
CREATE TABLE COMPLEXSEQUENTIAL_CORRELATIONSETINSTANCE
{
  CORRELATIONSETINSTANCEID
            CHARACTER (8)   FOR BIT DATA   NOTNULL ,
  PIID      CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  VALUE VARCHAR (1023),
  CONSTRAINT PK_CORRELSETINCE PRIMARY KEY
                    (CORRELATIONSETINSTANCEID)
};
```

Figure B-4 Complex sequential link instance table

```
CREATE TABLE COMPLEXSEQUENTIAL_LINKINSTANCE
{
  LINKINSTANCEID CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID           CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  LINKID         CHARACTER (8)   FOR BIT DATA   NOT NULL ,
```

```
  STATE                  SMALLINT    NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_LINKINSTANCE PRIMARY KEY (LINKINSTANCEID)
};
```

Figure B-5 Complex sequential partner link instance table

```
CREATE TABLE COMPLEXSEQUENTIAL_PARTNERLINKINSTANCE
{
  PARTNERLINKINSTANCEID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  PIID                              CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  BPELPARTNERLINKID        CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  PARTNEREPR                  VARCHAR (1023),
  CONSTRAINT PK_PNTERINCE PRIMARY KEY (PARTNERLINKINSTANCEID)
};
```

Figure B-6 Complex sequential compensation log table

```
CREATE TABLE COMPLEXSEQUENTIAL_COMPENSATIONLOG
{
  COMPENSATIONLOGID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  PIID                            CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  AID                             CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  COMPENSATIONSEQ BIGINT NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_COMPENSATIONLOG PRIMARY KEY
                                    (COMPENSATIONLOGID)
};
```

Figure B-7 Complex sequential compensation variable table

```
CREATE TABLE COMPLEXSEQUENTIAL_COMPENSATIONVARIABLE
{
  COMPENSATIONVARIABLEID
                           CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  AID                      CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  BPELVARIABLEID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  STATE                    SMALLINT           NOT NULL WITH DEFAULT 0 ,
  VALUE                    BLOB (4K)          NOT NULL LOGGED NOT COMPACT,
  CONSTRAINT PK_COMPENSATIONVAR PRIMARY KEY
                               (COMPENSATIONVARIABLEID)
};
```

Figure B-8 Complex sequential received variable table

```
CREATE TABLE COMPLEXSEQUENTIAL_RECEIVEVARIABLE
{
  RECEIVEVARIABLEID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
  AID                            CHARACTER (8)    FOR BIT DATA    NOT NULL ,
```

```
   STATE                SMALLINT        NOT NULL WITH DEFAULT 0 ,
   VALUE                BLOB (4K)       NOT NULL LOGGED NOT COMPACT,
   CONSTRAINT PK_RECEIVEVARIABLE PRIMARY KEY
                                (RECEIVEVARIABLEID)
};

COMMIT;
```

B-3. DDL for simple parallel process model:

```
UPDATE COMMAND OPTIONS USING c OFF ;
```

Figure B-9 Simple parallel process table

```
CREATE TABLE SIMPLEPARALLEL_PROCESS
{
   PIID    CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   PMID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   CONSTRAINT PK_PROCESS PRIMARY KEY (PIID)
};
```

Figure B-10 Simple parallel activity table

```
CREATE TABLE SIMPLEPARALLEL_ACTIVITY
{
   AID              CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   PIID             CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   STATUS           VARCHAR (255)                    NOT NULL ,
   VID              CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   VARIABLES     SMALLINT            NOT NULL WITH DEFAULT 0 ,
   CONSTRAINT PK_ACTIVITY PRIMARY KEY (AID)
};
```

Figure B-11 Simple parallel control connector table

```
CREATE TABLE SIMPLEPARALLEL_CONTROLCONNECTOR
{
   PMID          CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   SOURCEAID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   TARGETAID CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   TRANSCOND VARCHAR (255)                    NOT NULL ,
   CONSTRAINT PK_CTRLCONNECTOR PRIMARY KEY
                           (PMID,SOURCEAID,TARGETAID)
};
```

Figure B-12 Simple parallel correlation set instance table

```
CREATE TABLE SIMPLEPARALLEL_CORRELATIONSETINSTANCE
{
  CORRELATIONSETINSTANCEID
                  CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID            CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  VALUE          VARCHAR (1023),
  CONSTRAINT PK_CORRELSETINCE PRIMARY KEY
                       (CORRELATIONSETINSTANCEID)
};
```

Figure B-13 Simple parallel link instance table

```
CREATE TABLE SIMPLEPARALLEL_LINKINSTANCE
{
  LINKINSTANCEID CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID                CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  LINKID              CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  STATE               SMALLINT          NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_LINKINSTANCE PRIMARY KEY (LINKINSTANCEID)
};
```

Figure B-14 Simple parallel partner link instance table

```
CREATE TABLE SIMPLEPARALLEL_PARTNERLINKINSTANCE
{
  PARTNERLINKINSTANCEID CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID                          CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  BPELPARTNERLINKID       CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PARTNEREPR                VARCHAR (1023),
  CONSTRAINT PK_PNTLINKINCE PRIMARY KEY
                       (PARTNERLINKINSTANCEID)
};
```

Figure B-15 Simple parallel compensation log table

```
CREATE TABLE SIMPLEPARALLEL_COMPENSATIONLOG
{
  COMPENSATIONLOGID CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID                          CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  AID                           CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  COMPENSATIONSEQ BIGINT NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_COMPENSATIONLOG PRIMARY KEY
                                    (COMPENSATIONLOGID)
};
```

Figure B-16 Simple parallel compensation variable table

```
CREATE TABLE SIMPLEPARALLEL_COMPENSATIONVARIABLE
{
  COMPENSATIONVARIABLEID
                          CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  AID                     CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  BPELVARIABLEID   CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  STATE                   SMALLINT          NOT NULL WITH DEFAULT 0 ,
  VALUE                   BLOB (4K)         NOT NULL LOGGED NOT COMPACT,
  CONSTRAINT PK_COMPENSATIONVAR PRIMARY KEY
                            (COMPENSATIONVARIABLEID)
};
```

Figure B-17 Simple parallel received variable table

```
CREATE TABLE SIMPLEPARALLEL_RECEIVEVARIABLE
{
  RECEIVEVARIABLEID CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  AID                       CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  STATE                     SMALLINT          NOT NULL WITH DEFAULT 0 ,
  VALUE                     BLOB (4K)         NOT NULL LOGGED NOT COMPACT,
  CONSTRAINT PK_RECEIVEVARIABLE PRIMARY KEY
                                  (RECEIVEVARIABLEID)
};

COMMIT;
```

B-4. DDL for complex parallel process model:

```
UPDATE COMMAND OPTIONS USING c OFF ;
```

Figure B-18 Complex parallel process table

```
CREATE TABLE PROCESS
{
  PIID          CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PMID          CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  CONSTRAINT PK_PROCESS PRIMARY KEY (PIID)
};
```

Figure B-19 Complex parallel activity table

```
CREATE TABLE ACTIVITY
{
```

```
  AID             CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID            CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  STATUS          VARCHAR (255) NOT NULL ,
  CONSTRAINT PK_ACTIVITY PRIMARY KEY (AID)
};
```

Figure B-20 Complex parallel variable instance

```
CREATE TABLE VARIABLEINSTANCE
{
  VID             CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  AID             CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  VARIABLES   SMALLINT            NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_VARIABLEINSTANCE PRIMARY KEY (VID)
};
```

Figure B-21 Complex parallel

```
CREATE TABLE CONTROLCONNECTOR
{
  PMID              CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  SOURCEAID         CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  TARGETAID         CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  TRANSCOND         VARCHAR (255)                  NOT NULL ,
  CONSTRAINT PK_CONTROLCONNECTOR PRIMARY KEY
                              (PMID,SOURCEAID,TARGETAID)
};
```

Figure B-22 Complex parallel correlation set instance table

```
CREATE TABLE CORRELATIONSETINSTANCE
{
  CORRELATIONSETINSTANCEID
                    CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID              CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  VALUE             VARCHAR (1023)
  CONSTRAINT PK_CORRELATIONSETINSTANCE PRIMARY KEY
                                  (CORRELATIONSETINSTANCEID)
};
```

Figure B-23 Complex parallel link instance table

```
CREATE TABLE LINKINSTANCE
{
  LINKINSTANCEID      CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID                CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  LINKID              CHARACTER (8)   FOR BIT DATA   NOT NULL ,
```

```
  STATE                    SMALLINT          NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_LINKINSTANCE PRIMARY KEY (LINKINSTANCEID)
};
```

Figure B-24 Complex parallel partner link instance table

```
CREATE TABLE PARTNERLINKINSTANCE
{
  PARTNERLINKINSTANCEID CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID                  CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  BPELPARTNERLINKID     CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PARTNEREPR            VARCHAR (1023)
  CONSTRAINT PK_PARNTERLINKINSTANCE PRIMARY KEY
                                (PARTNERLINKINSTANCEID)
};
```

Figure B-25 Complex parallel compensation log table

```
CREATE TABLE COMPENSATIONLOG
{
  COMPENSATIONLOGID   CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  PIID                CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  AID                 CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  COMPENSATIONSEQ     BIGINT                NOT NULL WITH DEFAULT 0 ,
  CONSTRAINT PK_COMPENSATIONLOG PRIMARY KEY
                                (COMPENSATIONLOGID)
};
```

Figure B-26 Complex parallel compensation variable table

```
CREATE TABLE COMPENSATIONVARIABLE
{
  COMPENSATIONVARIABLEID
                          CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  AID                     CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  BPELVARIABLEID          CHARACTER (8)   FOR BIT DATA   NOT NULL ,
  STATE                   SMALLINT          NOT NULL WITH DEFAULT 0 ,
  VALUE                   BLOB (4K)         NOT NULL LOGGED NOT COMPACT,
  CONSTRAINT PK_COMPENSATIONVARIABLE PRIMARY
                        KEY (COMPENSATIONVARIABLEID)
};
```

Figure B-27 Complex parallel received variable table

```
CREATE TABLE RECEIVEVARIABLE
{
  RECEIVEVARIABLEID CHARACTER (8)   FOR BIT DATA   NOT NULL ,
```

```
   AID                    CHARACTER (8)    FOR BIT DATA    NOT NULL ,
   STATE                  SMALLINT          NOT NULL WITH DEFAULT 0 ,
   VALUE                  BLOB (4K)         NOT NULL LOGGED NOT COMPACT,
   CONSTRAINT PK_RECEIVEVARIABLE PRIMARY KEY
                                    (RECEIVEVARIABLEID)
};

COMMIT;
```

List of Reference

[1] Dieter Roller, Frank Leymann: *Production Workflow, Concept and Techniques*

[2] SWoM team at the University Stuttgart: *SWoM Specification Version 1.2*, Christoffer Bromberg as contact person, July,13.2005

[3] SWoM team at the University Stuttgart: *SWoM Design Version 1.2*, Mirko Sonntag as contact person, Nov, 18.2005

[4] SWoM team at the University Stuttgart: *Administration Guide, Version 1*, Daniel Tertilt as contact person, May, 14.2005

[5] Chamberlin, Donald Dean: *DB2 Universal Database* , 1999

[6] Pürner, Heinz Axel; Pürner, Beate: *DB2 Common Server*, 1997

[7] Lim, Pacifico A.: *DB2 for application programmers* , 1990

[8] Molinaro, Anthony : *SQL cookbook* , 2006

[9] Achilles, Albrecht: *SQL* , 1995

[10] David C. Fallside, Priscilla Walmsley: XML Schema Part 0: Primer Second Edition W3C Recommendation Oct.28, 2004 http://www.w3.org/TR/xmlschema-0/

[11] Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling Commerce, Canyang Kevin Liu, Satish Thantte, Prasad Yendluri, Alex Yiu: *Web Services Business Process Execution Language Version 2.0 Working Draft*, Aug. 23, 2005
http://www.oasis-open.org/apps/org/workgroup/wsbpel

[12] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana: *Web Services Description Language(WSDL) 1.1, W3C Note Mar.15, 2001*
http://www.w3.org/TR/2001/NOTE-wsdl-20010315