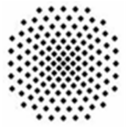


Institut für Architektur von Anwendungssystemen (IAAS)

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 2316

Abstract Business Process Monitoring

Sumadi Lie

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. David Schumm
begonnen am:	21.12.2010
beendet am:	22.06.2011
CR-Klassifikation:	C.2.4, D.2.2, H.4.1, H.5.2, H.5.3

Abstract

Business process monitoring provides the means to monitor the executing activities of process instance and it allows checking the resulting state of each activity. This information provides users knowledge about which activities have successfully been performed and which ones need to be fixed by an administrator or technical operator. However, modeling and executing of business processes are carried out on different levels of abstraction, i.e., the process model that is designed on high-level by the business users might not be directly executed, but it needs to be either decomposed into several small additional steps or translated into low-level executable codes for example *Business Process Execution Language (BPEL)* by the technical users, so that the process engine can understand how to execute the business processes. In the end the business users who are interested in viewing the resulting business process instance can only have a low-level view, i.e., the status of the high-level view is unknown.

In this student thesis business processes based on the language *BPEL* will be used in the low-level view, while the high-level process model is proposed and realized as Chevron-like processes (used in Microsoft PowerPoint). The Chevron process model might be defined by tagging some useful information such as name and picture to each activity to reflect the business purposes, and also small indicator for the activity status.

The problem described above can be dealt with the assistance of *process views* [Schumm2] and *state propagation patterns* [Schumm3]. *Process views* allow given process model to be customized, e.g., by removing a particular activity or by augmenting additional information to activities which can be used during visualization. In business process monitoring, *process views* enable the mapping between activities on different levels of abstraction and they also visualize the current state of running activity instances.

State propagation patterns [Schumm3] define how states of low-level view can be projected into the high-level view. The resulting states of activities from the *BPEL*

business process should be propagated back into the activities of Chevron process. Some basic patterns will be presented and each of them contributes a solution to a particular case. At the end an example scenario is introduced and a test of the projection from low-level model into high-level model will be conducted.

.

Table of Contents

1. Introduction

1.1. Motivation

Business process is a vital guideline for a company or an organization to successfully run its business. Each business part is reflected as activities and they are connected by a connector to show their dependencies (process model). For an example the activities of the process model of a book publisher could be book ordering, book availability checking, book delivery, etc. They must be well-ordered so that the process model as a whole can be understood and executed properly.

In order to ensure that the process model is carried out correctly there should be a mechanism to check the current state of the activities. Herein, business process monitoring comes into play because it is not easy to track each activity as neither the process model is typically simple, nor small. The status of each activity is contained in a database, called *audit trail.* By augmenting this information into activities it is possible for a monitoring tool to provide functionality which allows the users to visually inspect the current state of running process instance.

The *Business Process Execution Language* (BPEL) allows business process designer to design a process model, execute it in a process engine, and monitor the resulting status. It means that the process model that is being executed and monitored is identical. However, this is not always the case. The high-level process model that is designed with business in mind needs to be refined or translated into a low-level format which is understood by the process engine, and the resulting model structure is somehow quite different to the original one.

For example it is possible for a business user to design its own, high-level process model using for instance the *Decidr* (http://www.decidr.eu) platform. The process model is described in the *Decidr Workflow Definition Language* (DWDL). Before the process is executed the high-level DWDL will be translated into BPEL format based on the pre-defined mapping rules. For an instance the activity “book delivery” in high-level will be mapped into a sequence of BPEL activities, *invoke* and *receive* activity. *Invoke* means the books are ready to be sent to the customer and after successfully being delivered, a notification is received by the *receive* activity. However, all states

related to “book delivery” are visible only on the BPEL activities. The business user who designed the process model also wants to monitor the activity status, but it is not available on the high-level view because it has not been propagated back to the origin.

1.2. Task

The purpose of this student thesis is to implement the *state propagation* approach [Schumm3] to support monitoring a given business process instance based on BPEL [4] in a higher level language. For such language, in this thesis we use the abstract process language “Chevron”. For this task, an XML-based serialization format and an SVG-based visualization for the Chevron process language have to be defined. Furthermore, a rule language for *state propagation* has to be developed for defining projections of execution states from BPEL to Chevron. A prototype to perform such projections is being developed as a proof of concept.

1.3. Structure of Work

Chapter 2 introduces the basic concept of business process monitoring, *process views* and *state propagation patterns*.

Chapter 3 summarizes the technologies used in this work. They include Extensible Markup Language (XML), Java APIs for XML Parsers, Scalable Vector Graphics (SVG), and Business Process Execution Language (BPEL).

Chapter 4 introduces an example scenario including a low-level process model based on BPEL and Chevron process model. Chapter 4 also explains how they are related to each other.

Chapter 5 shows the architecture which bounds the technologies and the *state propagation patterns* together to produce a SVG-based visualization for the Chevron process instance.

Chapter 6 discusses the architecture of the prototype.

Chapter 7 examines the implementation of this architecture.

Chapter 8 concludes the work and gives outlook how this work could be further extended.

2. Basic Concepts

2.1. Business Process Monitoring

Business process monitoring is a general term for techniques that provide information about the status of a process instance [Freund]. A process instance represents a single executing process model. The process instance that is being monitored can be either still running or already completed. Besides each status activity of the process instance can also be observed so the administrator has complete knowledge about the execution progress.

Either process or activity instance follows a life-cycle whenever the execution takes place. In the Workflow Management Coalition Specification (WfMC) for Application Programming Language the status of process and activity instances is presented as a nested state [WAPI]. In the context of this student thesis only activity state will be considered. The details of the states are described in the following table.

State	Description
Open	The activity instance is active.
open.running	Indicate that the activity instance is executing.
open.notRunning	The activity instance has not been started yet.
open.suspended	The activity instance might be temporarily suspended.
Closed	Indicate that the instance has been completed.
closed.aborted	The activity instance has been aborted. (stop the activity if it is possible)
closed.terminated	The activity instance has been terminated. (stop the activity when it is completed)
closed.completed	The activity instance has completed normally.

Table 2.1 States of activity instance [WAPI]

The *Audit trail* is a database which records the state changes for process instances, activity instances, and work items. It plays an important role in business process monitoring [zurMuehlen]. For recovery purposes this database keeps the states of process and activity instances. If the system crashed, the last known status of the

instances can be retrieved and the execution can also be resumed. Moreover, because of the precious information contained in *audit trail,* process evaluation can be carried out more effectively and accurately.

The users who can take benefits from the business process monitoring are classified into three groups [zurMuehlen]:

1) Workflow participants

   The participants perform their jobs based on the work list created for them, i.e., they can choose a task from the list and work on it. By using the monitoring tool they can evaluate the history of a process instance. For example they can see the pending tasks or identify their colleagues who encounter a difficulty from a particular task.

2) Workflow administrators and process managers

   They can evaluate the overall performance of the process engine and provide the result to either technical or organizational level. Using this monitoring information they might also balance the workload to other departments or participants by reassigning the tasks to them.

3) Workflow customers

   Normally, customers interact with the system via a process invocation interface. Once it is invoked the process instance will be created and some functionality like current state monitoring and even state manipulation are possible. An example is a customer who buys goods from the online store can take a look of its current orders status.

According to [zurMuehlen], process monitoring can be divided into two categories: technical and organizational process monitoring. The technical process monitoring is used to observe the system response time and workload. It also gives the administrator and process manager information about the number of active participants, pending activities, faulted activities, etc. If the process instance is executed on business partner, the details of an internal process model are usually hidden or abstracted. This is on one side to give freedom to modify the internal business details and on another side the company that owns the process model does not want to expose its internal business processes. For instance, one coarse-grained activity will appear instead of some fine-granular ones and in this case the

organizational process monitoring allows the business partner to monitor only that activity.

2.2. Process Views

One particular business process might contain hundreds of activities and the effort to maintain the complex business process becomes even harder. In addition, most of the users only want to view some particular activities instead of the whole process, i.e., the business process can be personalized based on user interests [Bobrik] because presenting all activities to them would not make much sense. With help of *process viewing patterns* [Schumm2] it is possible to transform a given business process in order to decrease the complexity of the original business process and also presents the users a better view experience in terms of abstracting the internal process details [Polyvyanyy].

Herein the basic *process viewing patterns* in [Schumm2] that are related to business process monitoring will be briefly presented. *Omission* pattern shows a removal of activities and the related connectors, i.e., information can be filtered. *Aggregation* combines some activities into one coarse-grained activity (summarizing information) and *alteration* makes it possible to change a property of an activity or connector. The properties could be a name, identifier, transition condition or status related to the activity. *Theme* pattern also plays its part by determining which information should be explicitly visible. *Augmentation* pattern with *runtime information* describes the augmentation of information e.g. monitoring information (current state, workload) to the process. All of these patterns can be used together to assist and solve different task with different complexity. For instance *runtime information* might augment the process with the monitoring information and then present it to users using the *theme* pattern. By applying *process view* patterns to original process model may produce a structurally different resulting process model. For this thesis, the augmentation with runtime information is of fundamental importance as it is a basic prerequisite for state propagation.

2.3. State Propagation Patterns for Business Process Monitoring

The concept of *process viewing patterns* plays a fundamental role for the *state propagation patterns*. *Process views* in business process monitoring describe the projection of activities of low-level to high-level view, and based on this projection *state propagation patterns* define a way how information, i.e., the status of running activity instances for monitoring purposes, should be mapped onto elements at different levels of abstraction.

Following are some basic *state propagation patterns* as described in [Schumm3] that will be used in this work.

1) *Direct state propagation pattern*

   This pattern describes a one-to-one mapping of the low-level to high-level view i.e. a status of one activity on the low-level will be propagated to exactly one activity on the high-level view. The status of the low-level is exactly the same one as on high-level (See figure 2.1). For an example the completed *assign* activity of BPEL will be presented on the activity of the Chevron process with state completed.

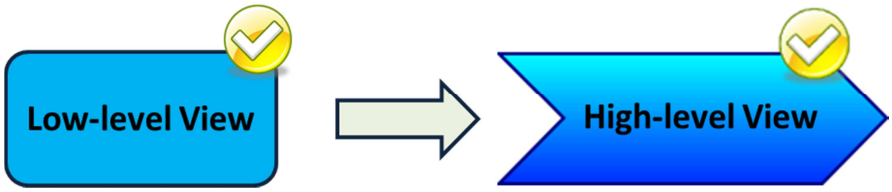


Figure 2.1 Direct state propagation

2) *State alteration pattern*

   Before mapping the state of low-level to high-level view the corresponding state needs to be adapted. This is because the states set from the low-level may be different from the states set from high-level. As for another case the state of low-level has different semantic compared to the one on high-level view (See figure 2.2). Example: a faulted process should not be presented as a fault on high-level view. It should be adapted to the state sets from the high-level e.g. as a running process.

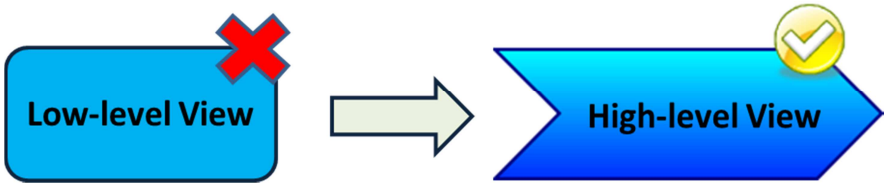


Figure 2.2 State alteration

3) *State combination pattern*

As the name already implied the *state combination* pattern aggregates the states from activities of the low-level and propagates it to one single activity on high-level view. The combination can be carried out from consecutive directly connected low-level activities, but it is not necessary, it can be combined from arbitrary low-level activities (See figure 2.3). Example: the sequence of *invoke* and *receive* of BPEL activities represents one activity on the high-level view. The states from *invoke* and *receive* will be combined before the mapping takes place.

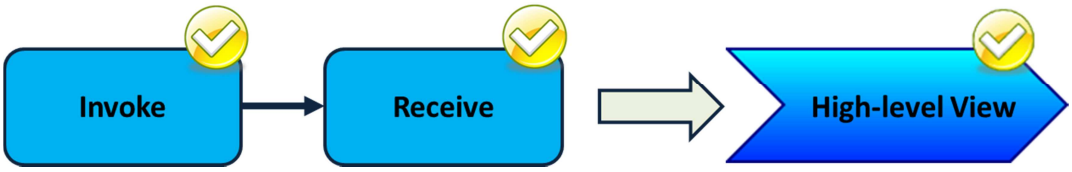


Figure 2.3 State combination

3. Technologies

3.1. Extensible Markup Language (XML)

XML was originally designed to transport and store data [w3s1] so it is mainly used to describe information rather than display information. XML is standardized as data representation by *The World Wide Web Consortium* (W3C). XML Document and XML Schema will be briefly presented in the following sub-chapters.

3.1.1. XML Document

Below is a simple XML document (Listing 3.1). XML document is structured hierarchically and has the form of a tree. It starts with the root element ("book" element in the example) and branches until the leaves (the "title" and "price" element). The relationship between elements can also be described as parent – children and sibling. Book element is parent of title and price element, and title and price element are sibling. An element in XML has a start and end tag, and can contain text or other elements. An element might also have additional information which is described by an attribute, e.g., "id" on the "title" element. The following XML document is also well-formed [w3org] because it begins with the XML declaration *<?xml version ="1.1"?>*, it has exactly one root element, and the elements are nested properly.

Listing 3.1 A Simple XML Document

```
<?xml version = "1.1"?>
<book>
      <title id ="1">Introduction to XML</title>
      <price>49.99</price>
</book>
```

3.1.2. XML Schema

XML documents that have been created must follow a set of rules regarding the structure. It will be difficult to process an XML document which does not have a legal building block as we expected. XML Schema Definition (XSD) is used to describe the structure of an XML document and to validate the data correctness. It defines which elements and attributes can appear on the document, which elements are child

elements and how they are being ordered, default value for the elements and attributes. XML Schema has the root element <schema>.

XSD defines two types of elements that can be used in XML document, namely the simple element and the complex element.

1) XSD simple element

   A simple element can contain only text (integer, string, boolean, etc.), and it cannot contain any other elements or attributes. The simple elements definition for previous example is described in Listing 3.2.

Listing 3.2 An Example of Simple Element Definition

```
<xs:element name="title" type="xs:string"/>
<xs:element name="price" type="xs:decimal"/>
<xs:attribute name="id" type="xs:integer"/>
```

2) XSD complex element

   A complex element can contain other elements and/or attributes. From previous example in Listing 3.1 the element “book” is a complex element because it contains two other elements i.e. the “title” element and the “price” element. The schema definition is as followed (Listing 3.3):

Listing 3.3 An Example of Complex Element Definition

```
<xs:element name="book">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="price" type="xs:decimal"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:integer"/>
    </xs:complexType>
</xs:element>
```

3.2. Java APIs for XML Parsers

An XML parser is needed to read XML document and provides an application with contents of the document so that the related data can be extracted and processed. There are varieties of parser that can be employed, and the Java Programming Language offers the Java APIs for reading XML documents like Document Object Model (DOM) and Streaming APIs for XML (StAX).

3.2.1. Document Object Model (DOM)

DOM is a high-level parsing API which interprets the XML document in a tree-like structure and it allows for random access. Before any data is fetched into the application an XML document will be entirely parsed and created as objects, and these objects are then kept in the memory. If the XML document is very large a lot of memory will be consumed.

DOM specification defines a set of objects and by using the objects a program can access the information in the XML document and also modify or update it if necessary. So the main benefits of DOM are its ease of use [Griffith] and it allows data modification.

The specification of DOM in Java is implemented as the package *org.w3c.dom*. The package provides interface for the DOM. The interfaces are nodes in the DOM parse tree. Some interface definitions that are relevant to this student thesis are listed in Table 3.1. The complete list of the interface definitions can be found in [oracle1].

Name	Description
Document	This interface represents the entire XML document.
Element	Element interface represents an XML element.
Node	The Node interface is the primary data-type for the entire DOM.
NodeList	An ordered list of nodes that is accessible via index.
Text	This interface represents the textual content of XML element or attribute.

Table 3.1 The *org.w3c.dom* package [oracle1]

3.2.2. Streaming APIs for XML (StAX)

StAX is a bi-directional API for reading and writing XML. It is often referred as *pull parsing* i.e. users only get or pull XML information when it explicitly requests for it. So, *pull parsing* allows users to have full control of the application. One drawback of StAX is there is no *Create, Read, Update, Delete* (CRUD) capabilities [JAXP].

The cursor API of StAX is a cursor that can walk through the entire XML document. The cursor can point to one thing at a time and always moves forward, never

backward. It has two interfaces, namely *XMLStreamReader* for reading from and *XMLStreamWriter* for writing to an XML document.

An XML document is broken down into a set of *events* and the cursor API will iterate through these *events*. *Events* are for example *START ELEMENT, END ELEMENT, ATTRIBUTE* that indicates the start element, end element, and attribute of XML document, respectively. The cursor API uses the iterator *hasNext()* and *next()* to iterate through the *events*. The *hasNext* iterator will return true if there is an *event* to process, while the *next* iterator moves the cursor one step forward to the next *event* [JWSPT].

When the *XMLStreamReader* is first created, the current *event* is *START DOCUMENT* and *XMLStreamReader.next* method moves the cursor to the next *event*. Other common used methods in *XMLStreamReader* are *getLocalName()* which return local name of the current event, and *getElementText()* that reads the textual content of an element.

The interface *XMLStreamWriter* specifies how to write an XML document, but it does not validate the input against the well-formed XML. The methods *writeStartDocument, writeStartElement, writeAttribute* are commonly used to compose an XML document. The complete methods summary for *XMLStreamReader* and *XMLStreamWriter* can be found in [oracle2] and [oracle3], respectively.

3.3. Scalable Vector Graphics (SVG)

SVG defines 2D-vector graphics in XML format. The language is simple and intuitive for example the use of *ellipse* and *polygon* make the language easier to learn. Because it is an XML-based language, it can be generated and parsed using standard XML tools, e.g., DOM. It is an open standard and a W3C recommendation.
SVG offers some advantages over other image formats such as JPEG and GIF. SVG files can be read and modified using a large range of tools like notepad. An SVG file size is smaller and more compressible than JPEG and GIF formats. SVG images are scalable, i.e., the images can be printed in high quality at any resolution and they can be zoomed without losing the image quality [w3s2].

SVG has specified predefined shape elements that can be directly used. The shapes are:

- Rectangle, it is defined using the <rect> tag.
- Circle, it is defined using the <circle> tag.
- Ellipse, it is defined using the <ellipse> tag.
- Line, it is defined using the <line> tag.
- Polyline, it is defined using the <polyline> tag.
- Polygon, it is defined using the <polygon> tag.
- Path, it is defined using the <path> tag.

Listing 3.4 shows a simple drawing of a rectangle. The x and y attributes describe the position of the rectangle, and width and height attributes specifies the width and height of the rectangle. The rectangle is red and has a text written on it.

Listing 3.4 An Example of Drawing Simple Rectangle

```
<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">
    <rect x="20" y="20" width="400" height="200" style="fill:red"/>
    <text x="25" y="100" font-size="15" style="fill:black">Simple
Rectangle</text>
</svg>
```

3.4. Business Process Execution Language (BPEL)

BPEL was originated from the combination of graph-based language Web Service Flow Language (WSFL) from IBM [ibm] and calculus-based language XML Business Process Language (XLANG) from Microsoft [microsoft]. The first version of BPEL was known as BPEL4WS and on 2007 the second version WS-BPEL was released and standardized by OASIS.

BPEL is an XML-based language for describing the behavior of a business processes between Web Services and as Web Services, i.e., it recursively aggregates the Web Services. BPEL can be classified into abstract and executable processes. Abstract processes are not intended to be executed and they might represent the internal operational details. The executable processes define the

process execution and interaction between Web Services in a consistent way in heterogeneous environments [oasis].

3.4.1. Activities in BPEL

BPEL activities describe the process logic, i.e., the functional implementations of a business task will be executed on each activity. There are two classes of BPEL activities [oasis]: basic and structured activities. The basic activity describes the basic task of the business logic. The structured activity specifies the control-flow logic. They contain a set of basic or structured activities. Each activity can have an optional attribute *name* so that the activity can be identified by parser. The lists of basic and structured activities are summarized into Table 3.2 and Table 3.3, respectively.

Activity Name	Description
<invoke>	It allows the business process to invoke Web services offered by business partners.
<receive>	It allows the business process to wait until a matching message is arrived and then it completes..
<reply>	It sends a reply message after it received a message from a corresponding <receive> activity.
<assign>	It updates values of variables with new data.
<throw>	It generates a fault from inside the business process.
<empty>	It represents the “no-op” in business process.
<wait>	It waits until a certain period of time has been reached.
<rethrow>	It re-throws the fault that was originally caught by the immediately enclosing fault handler.
<exit>	It ends the business process instance.
<compensate>	It starts compensation on all inner scopes that have already completed successfully, in default order.
<compensateScope>	It starts compensation on a specified inner scope that has already completed successfully.
<validate>	It validates the values of variables against their associated XML and WSDL data definition
<extensionActivity>	It allows the extension of new activity type.

Table 3.2 List of BPEL basic activities [oasis]

Activity Name	Description
<sequence>	It defines activities that are sequentially performed.
<if>	It is used to select exactly one activity from a set of activities.
<while>	A loop of child activity if the given condition is true.
<repeatUntil>	A loop of child activity until the condition evaluates to true.
<pick>	It waits for one of several possible messages to arrive or for a time-out to occur.
<flow>	It allows one or more activities to be performed concurrently.
<forEach>	A loop of child scope activity of N+1 times where N is the difference between <finalCounterValue> and <startCounterValue>.
<scope>	It defines nested activity which has transactional semantic.

Table 3.3 List of BPEL structured activities [oasis]

3.4.2. BPEL Event Model

BPEL *event model* [KKS+] specifies events related to the life-cycle of processes, activities, scopes, loops, and links. It is independent of any BPEL processor implementation. Events are produced by a processor and are used to determine the state transitions (from one state to another) of the artifacts specified above. In the context of the student thesis only the *event model* for activity will be presented.

The life-cycle of activity event is applicable for all BPEL activities, including the scopes and loop as activities. Following are the activity events definition [KKS+]:

- Activity_Ready
  This event is fired when an activity is ready to execute.
- Activity_Executing
  This event is fired when an activity begins to execute. For example <receive> activity is executing when it starts waiting for a message to arrive not when it receives a message.
- Activity_Executed
  This event is fired when the execution of an activity has finished, but it needs to wait for a signal from external source before completing the activity.

- Activity_Complete

  This event is fireed when the activity is completed and received a signal from external source.
- Activity_Terminated

  This event is fired when an activity is terminated because the process instance is terminating.
- Activity_Faulted

  This event is fired when an activity is aborted because of a fault within the activity.

3.4.3. BPEL Extension

Extensions in BPEL are desirable to introduce new concepts and capabilities that are not available on BPEL standard. BPEL is designed to be extensible and the extensions can range from new attributes to new elements. BPEL for Java (BPELJ) [Blow] is one example of the extensions which combine BPEL with Java code so that it is more convenient to program the BPEL process because functionalities written in Java can be integrated into the processes.

BPEL specification [oasis] defines a general way how extensions are carried out (see Listing 3.5). The <extension> element is used to describe the namespaces of BPEL extension of elements or attributes. The *mustUnderstand* attribute indicates whether the extension must be understood by the process engine or it may be ignored. In [compas] some specific extensions such as element and activity extensions are presented.

Listing 3.5 Formality of Element Extension [oasis]

```
<process ...>
...
   <extensions>
      <extension namespace="anyURI" mustUnderstand="yes|no"/>+
   </extensions>
...
</process>
```

4. Motivating Example

In this chapter a simple example scenario will be introduced and used in the rest of the student thesis. This example represents a business process of book ordering which can appear in different levels of abstraction i.e. the low-level view and high-level view, and the mapping between them to show their relationship is required.

4.1. Motivating Scenario

A business process of book ordering represents every business task (activity) that is related to book ordering from the book publisher. In this case the business process should at least be able to deal with three fundamental activities i.e. ordering, checking book stock, and book shipment. The sequence flow might be started when an order for books has been received. The flow can then continue checking the warehouse whether the ordered book exemplars are still in stock and it is ended when a shipment notification that indicates the book has been delivered to the customer is received.

How much details are available on the design and implementation of the business process depends strongly on the designer of the business process, available services, and the business itself. In the following there are two possible scenarios which depict how a business process is designed in terms of different levels of abstraction:

1) Business process is designed on high-level.

   In this sense the business process is designed by the business consultants. Normally the resulting process model is more or less straightforward. Some particular activity may have sub-activities to represent more details about business tasks. If the process model needs to be automatically executed in a process engine some structural transformation are required from the high-level business view to low-level implementation view. This transformation is then done by the technical users, so-called “IT refinement” (see Figure 4.1).

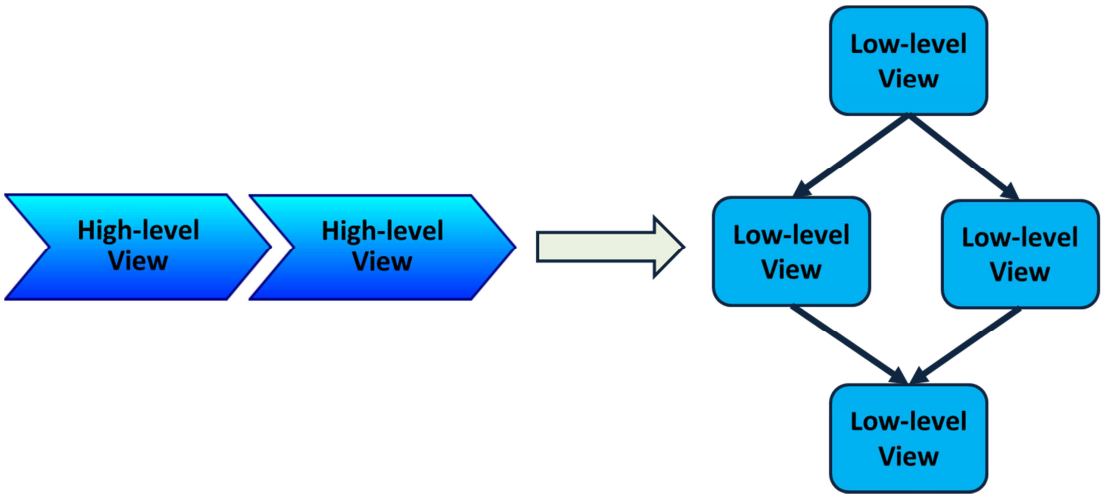


Figure 4.1. Transformation from high-level to low-level View

2) Business process is designed on low-level.

The BPEL language allows business processes to be designed and executed in a low-level, though directly executable view. However, the structure of a process model is more complex and more difficult to understand. Another example is when a company wants to provide limited process transparency to the customer. In this case the company on one hand does not want to expose its private business logic, and on the other hand a customer only needs information that is relevant to track her book ordering. Thus, she also does not want see any detail implementations. That is why a transformation (with help of *process views*) is also needed from low-level to high-level view (see Figure 4.2).

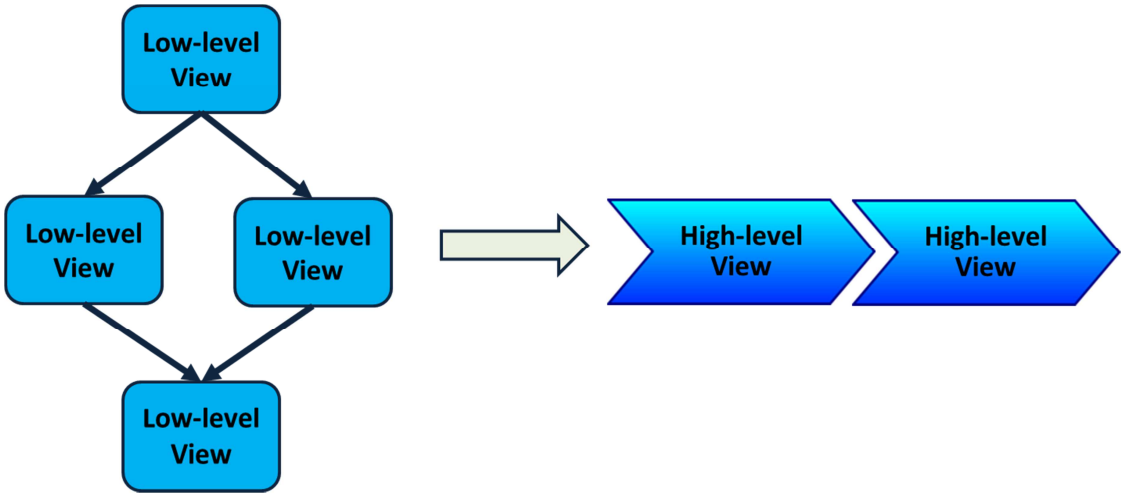


Figure 4.2. Transformation from low-level to high-level View

In either ways of design and implementation the states of the activities on high-level view are the main concern. In the context of this student thesis the status from executed activities needs to be retrieved and propagated back to business users. So the goal is to visualize the state of activities for monitoring purposes, though on high level process models. On the next sub-chapters, i.e., Sub-chapter 4.2 and 4.3, an example of the process model book ordering in BPEL and Chevron process are presented, respectively.

4.2. Example of BPEL as Low-level View

In this section, the business process for book ordering will be derived. The three main business tasks: book ordering, book availability checking, and book shipment should be analyzed so that the semantic translation into BPEL activities can be performed properly and understood by the process engine.

Listing 4.1 describes the BPEL activities as the translation result from book ordering. Book ordering will be interpreted into two BPEL activities, namely <*receive*> and <*invoke*> activities. The <*receive*> activity waits until it receives an order from a customer and it is followed by <*invoke*> *activity* to take care of the administrative details.

Listing 4.1 BPEL Activities Translation from Book Ordering

```
...
   <bpws:receive createInstance="yes" name="receive Order"
            operation="initiate" partnerLink="client"
            portType="tns:Ordering-Process" variable="input"/>

      <bpws:invoke name="prepare Order Processing"/>
...
```

Book availability checking is realized into a structured <*flow*> activity wherein all availability checking related activities are carried out. The first child activity of <*flow*> is the *prepare Availability Check* activity and it is represented by the <*assign*> construct. It provides information e.g. how many copies of a book are ordered. Based on this value *check Availability* is invoked via <*invoke*>. If the book exemplars are still available the <*flow*> activity is finished, otherwise the <*sequence*> activity of

book printing will be executed. Below is the BPEL code snippet for book availability checking (Listing 4.2).

Listing 4.2 BPEL Activities Translation from Book Checking

```
...
   <bpws:flow name="Flow">
                <bpws:links>
                    <bpws:link name="link1"/>
                    <bpws:link name="link2"/>
                </bpws:links>
                <bpws:invoke name="check Availability">
                    <bpws:targets>
                        <bpws:target linkName="link1"/>
                    </bpws:targets>
                    <bpws:sources>
                        <bpws:source linkName="link2"/>
                    </bpws:sources>
                </bpws:invoke>
                <bpws:assign name="prepare Availability Check"
                             validate="no">
                    <bpws:sources>
                        <bpws:source linkName="link1"/>
                    </bpws:sources>
                </bpws:assign>
                <bpws:if name="if Product Available">
                    <bpws:targets>
                        <bpws:target linkName="link2"/>
                    </bpws:targets>
                    <bpws:empty name="do Nothing"/>
                    <bpws:else>
                        <bpws:sequence name="Sequence">
                            <bpws:assign name="Prepare Production"
                                         validate="no"/>
                            <bpws:invoke name="Invoke Production Process"/>
                      <bpws:receive name="wait for completion of production"/>
                        </bpws:sequence>
                    </bpws:else>
                </bpws:if>
      </bpws:flow>
...
```

After book availability checking has been performed the control flow will continue to the book shipment. It begins with an <*assign*> activity to prepare all shipping details. A shipping service is then invoked by <*invoke*> activity and <*receive*> activity waits for a notification to denote that the book has been delivered. Then the system records this notification as an archive. The whole process instance is ended after <*reply*> activity has sent a reply message to the *receive order* activity.

Listing 4.3 BPEL Activities Translation from Book Shipment

```
...
    <bpws:assign name="prepare Shipping" validate="no"/>
        <bpws:invoke name="shipping"/>
        <bpws:receive name="shipping completion notice"/>
        <bpws:assign name="prepare order log" validate="no"/>
        <bpws:reply name="order completion"/>
...
```

The process model realized as BPEL is shown in Figure 4.3.

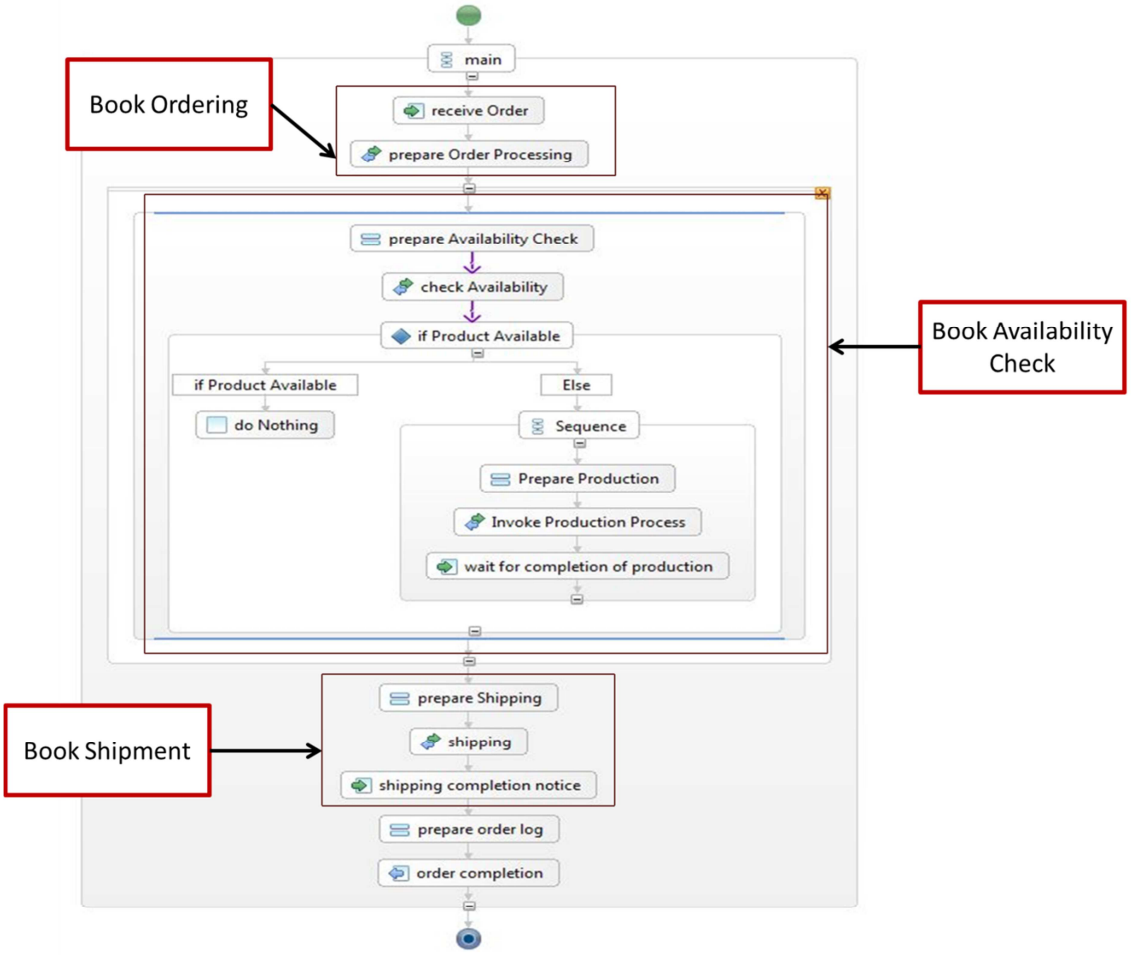


Figure 4.3 BPEL Process Model for Book Ordering

4.3. Example of Chevron Process as High-level View

The simple book purchasing order will now be designed in business in mind. In the high-level view Chevron-like processes will be adopted because it shows a simple and straightforward representation for interpreting and understanding the business

tasks. On the Chevron process it is possible to visualize the process by augmenting text description and icon to depict a particular business task.

The three basic business tasks book ordering, book availability check and book shipment will be examined again in order to produce the Chevron process model. Table 4.1 describes the translation, while the complete Chevron is presented in Figure 4.4.

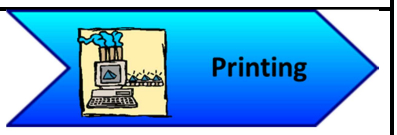
Main Task	Chevron Translation	Chevron Process
Book Ordering	*Book Ordering* (No modification is needed).	
Book Availability Check	New activity: *Printing* will be created and it has sub-activities: *check availability, production, and check product* which take care of every detail implementation of *Printing.*	   
Book Shipment	*Book Shipment* (No modification is needed)	

Table 4.1 Business Tasks Mapping to Chevron Process

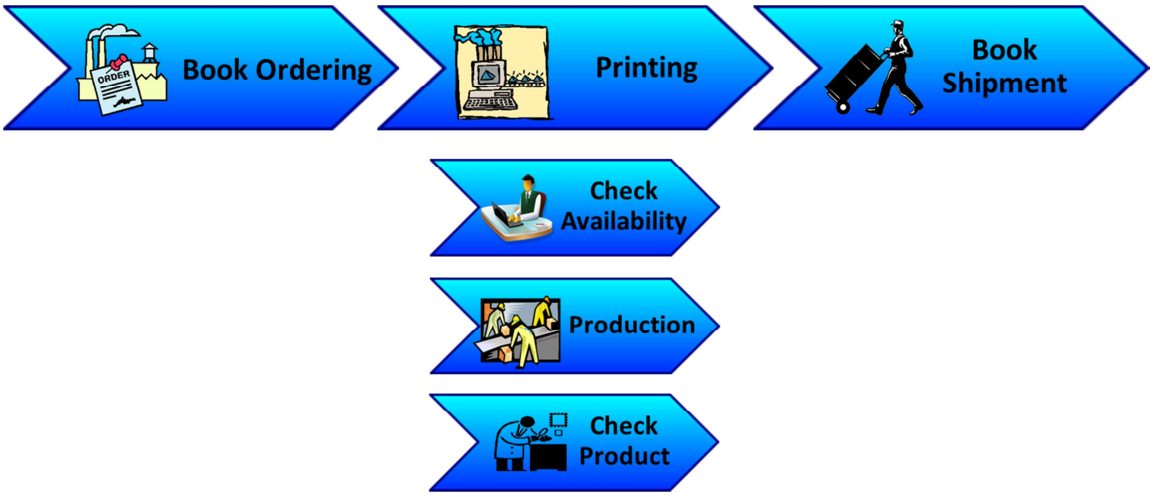


Figure 4.4 Chevron Process Model for Book Ordering

4.4. The Mapping between BPEL and Chevron Process

The process models in form of BPEL and Chevron were already depicted in the last sub-chapters. Now the mapping from BPEL to Chevron is required. This mapping is not only to indicate the semantic relationship between them but also to obtain the states of BPEL activities and augment the states to already mapped Chevron process elements.

There are some approaches that can be adopted to perform the mapping. The most straightforward way is directly plotting one BPEL activity to exactly one Chevron activity. The *process view patterns* for instance the *aggregation* (e.g. *shipping* and *shipping completion notice* to *book shipment*) and *omission pattern* (e.g. omission of *prepare order log* activity) could also be employed. Because the mapping is from a detailed to an abstract process model the relationship between the activities is either one-to-one or many-to-one. In addition to the approaches described above the semantic similarity between the activities of both process models is very helpful to perform a proper mapping. The mapping between BPEL activities to Chevron activities is defined on Table 4.2 and presented in Figure 4.5.

The second activity (*Printing*) of Chevron model is not mapped by any BPEL activities. The mapping is rather made to its sub-activities i.e. *Check Availability, Production, Check Product.*

BPEL Activities	Chevron Activity
receive Order prepare Order Processing	Book Ordering
check Availability	Check Availability
Prepare Production Invoke Production Process wait for completion of production	Production
Flow	Check Product
shipping shipping completion notice	Book Shipment

Table 4.2 The Mapping between BPEL Activities to Chevron Activity

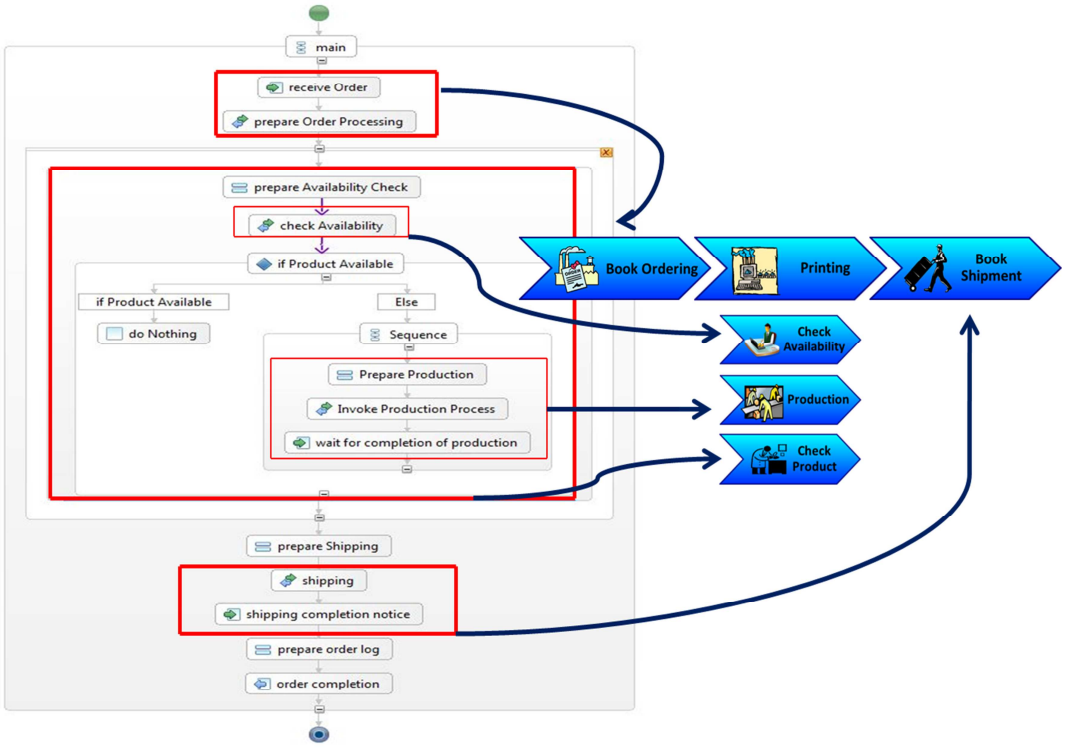


Figure 4.5 The Presentation of the Mapping between BPEL Activities and Chevron Activities

5. Concepts and Architecture

5.1. Concepts

In this section, the concepts related to Chevron process model as a high-level representation will be defined. These concepts include the BPEL and Chevron states, and technical definitions of the Chevron process and the activities mapping.

5.1.1. BPEL States

BPEL activities presented in chapter 4.2 are stateless i.e. the activities have no status information attached. These states are prerequisite for the states of Chevron activities i.e. BPEL states are required so that the *state propagation patterns* can be used to map the states to Chevron activities.

Based on the states of activity instance described in [WAPI] the states of BPEL activities in this work can be deduced. The BPEL states are defined in Table 5.1.

BPEL States	Description
No Status	The activity has not started yet or there is no information provided.
Active	The activity is still executing.
Completed	The activity has successfully completed.
Suspended	The activity can be temporarily suspended and started again later.
Failed	The activity is faulted and it will stop.
Terminated	The activity is terminated by user and it will stop after the execution has completed.

Table 5.1 States of BPEL activity

Figure 5.1 shows the state transition of the BPEL activities. The state transition is defined based on BPEL events [KKS+]. When an activity instance is first instantiated it has no status because it has not started. If the activity starts executing it is in active state and can go to one of four states: completed, suspended, terminated or failed. A suspended activity can be resumed and go to active state again or can be

terminated by the user. In the end of its execution the activity instance will go to the completed state.

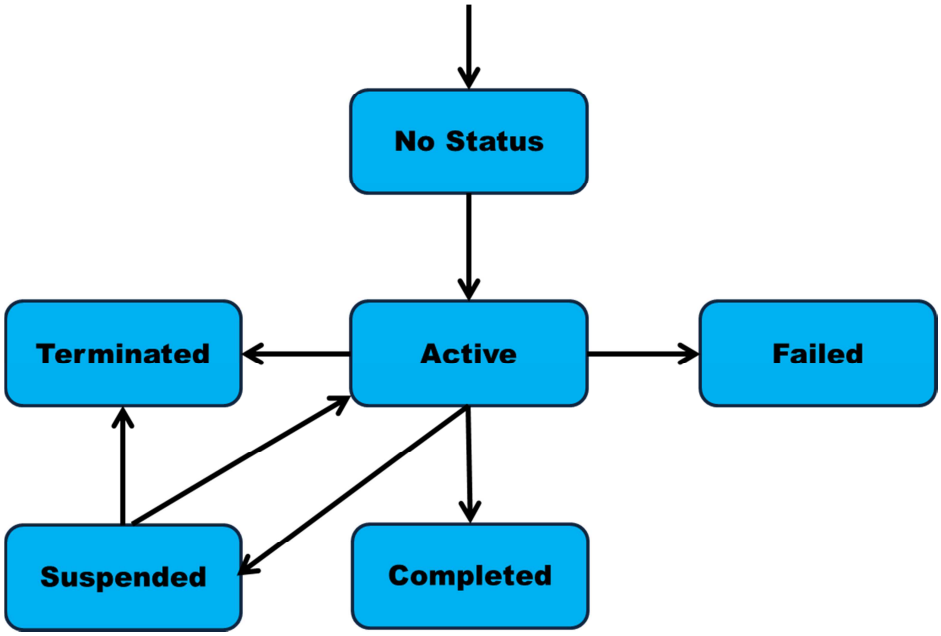


Figure 5.1 State Transition of BPEL activities

5.1.2. Chevron States

Because Chevron process model is intended for business audiences the Chevron states are not as detail as in the BPEL. The states like terminated or failed will be neglected because it describes technical related information and are interesting only for the administrator. The Chevron states are also concluded from activity states described in [WAPI] and BPEL events [KKS+]. The resulting states are presented (see Table 5.2) and the state transition is presented in Figure 5.2.

Chevron States	Description
Outstanding	This state is equivalent to No Status state of BPEL activity i.e. the activity has not yet started or no information state is provided.
Running	The activity is still executing.
Completed	The activity has successfully completed.

Table 5.2 States of Chevron Activity

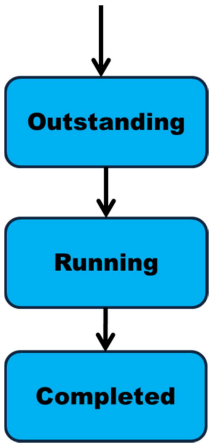


Figure 5.2 State Transition of Chevron Activity

5.1.3. Chevron Process Model Definition

After a short illustration of Chevron process model example in sub-chapter 4.3 the technical definition of the process model will be covered in this section. A Chevron activity is regarded as a regular activity. The activity might contain sub-activities to reflect the refinement of the corresponding parent activity. The parent activity is referred as a *complex activity*, and the sub-activities are just other regular activities. Moreover the activity might also contain a number of properties (see Table 5.3). The activities of Chevron process model is read from left to right and the children of *complex activity* are read from top to bottom. The technical definitions of the Chevron activities are illustrated in Figure 5.3.

Chevron Properties	Description
Activity ID	It is used to identify the activity
Activity Caption	It reflects the business task.
Activity Status	It describes the current state or behavior of corresponding activity.
Activity Icon for the status	It is used to visualize the activity status.
Activity Icon for the caption	It is used to visualize the business task.
Activity Color	It is used to draw the activity in particular color.
Activity Documentation	It gives a short description of corresponding activity.

Table 5.3 Properties of Chevron Activity

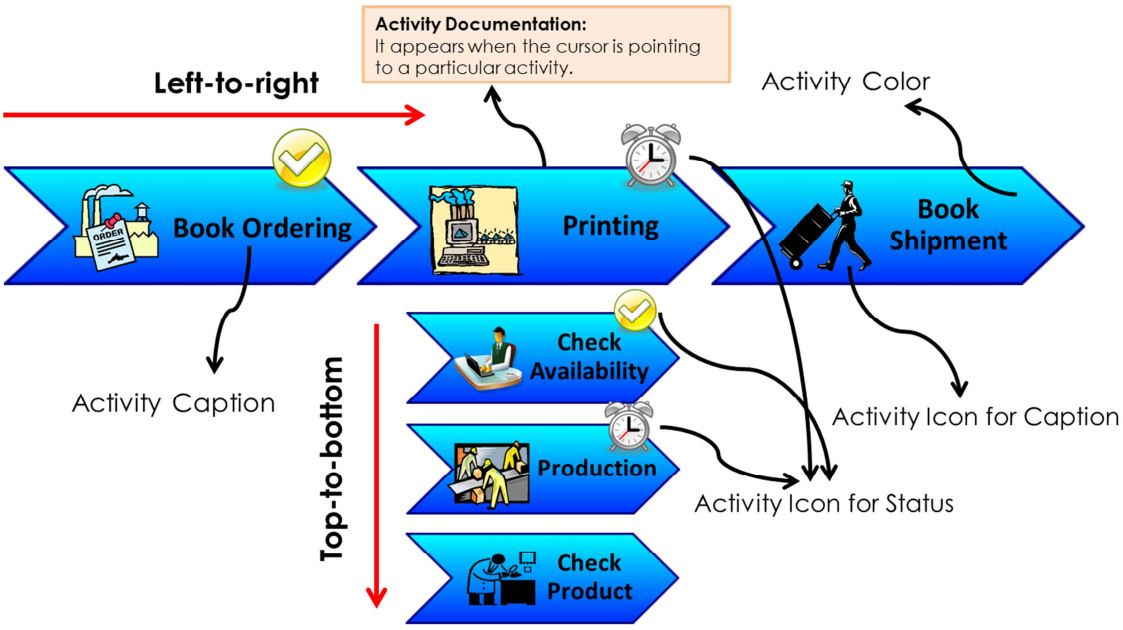


Figure 5.3 Properties Presentation of Chevron Activities

5.1.4. The Mapping Definition

In this section the mapping between BPEL activities to Chevron activities will be formally defined and it is described as follows:

1) The BPEL and Chevron activities are referred to as *source* activities and *target* activities, respectively. Each activity of the *source* and *target* has an *identifier* and a *value* that allows defining which source activity shall be mapped to which target activity.
2) Before the state propagation can take place a set of different state mappings needs to be defined. The mapping set comprises the *state propagation patterns* in [schumm3]. Each mapping is actually represented by one of the *state propagation patterns* and variants of one particular pattern can be defined to describe under which conditions the source activities should be mapped to target activities. For an example, the *alteration pattern* can defined into two variants that describe different conditions for a state alteration.
3) Based on the two conditions above the *propagation rules* can now be carried out by specifying the source activity, target activity and state mapping set. The collection of propagation rules is called the *state projection*.

5.2. Architecture

In the following section (sub-chapter 5.2.1) an architecture for the state projection is introduced. It shows how each element is connected to the Java processor and which output the processor produces. In the next sub-chapter 5.2.2 the functions that make up the processor will be examined.

5.2.1. The main Building Blocks

The architecture shown in Figure 5.4 represents building blocks of all related files for state projection purpose. Java Programming Language as the core of this architecture will take the stateful BPEL process, the stateless Chevron model, and the mapping set and propagation rule files as inputs. These artifacts will be processed by the Java program to produce a corresponding stateful Chevron process model. At the same time an SVG representation of the corresponding Chevron will also be generated based on the SVG template which describes the graphical rendering of a Chevron process.

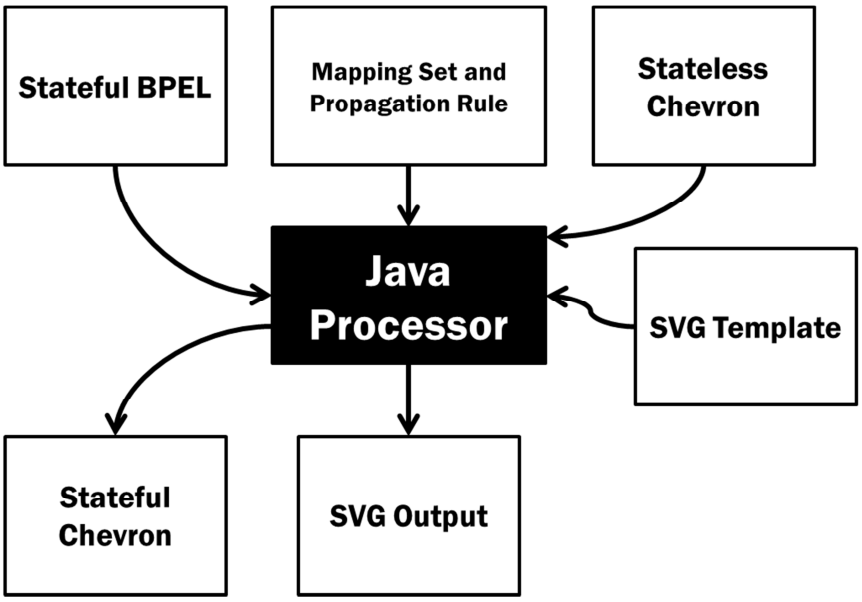


Figure 5.4 Architecture of the State Projection

Following are the descriptions of each building block:

1) Stateful BPEL: a BPEL file which contains the status of each BPEL activity (see sub-chapter 6.2).

2) Mapping set and the propagation rules: an XML document which implements the formal mapping definition (see sub-chapter 6.3.2).
3) Stateless Chevron: an XML document for Chevron process model but the status of the activities is still unknown (see sub-chapter 6.3.1).
4) SVG template: an SVG file that represents and visualizes one activity of Chevron process model, but it does not contain any properties related to Chevron activity (see sub-chapter 6.1).
5) Stateful Chevron: an XML document for Chevron process model whose states of the activities is already defined as results from the mapping implementation.
6) SVG output: it produces the SVG file from the stateful Chevron.

5.2.2. Functions of the Java Processor

The Java processor contains a collection of Java functions whose goal is to produce an XML file of stateful Chevron and an SVG representation of the corresponding Chevron process model. There are six functions inside the processor and they are executed sequentially (see Figure 5.5) i.e. it begins from reading stateful BPEL file and ends after generating the SVG file. The detail implementation of each function is explained in sub-chapters 6.3.

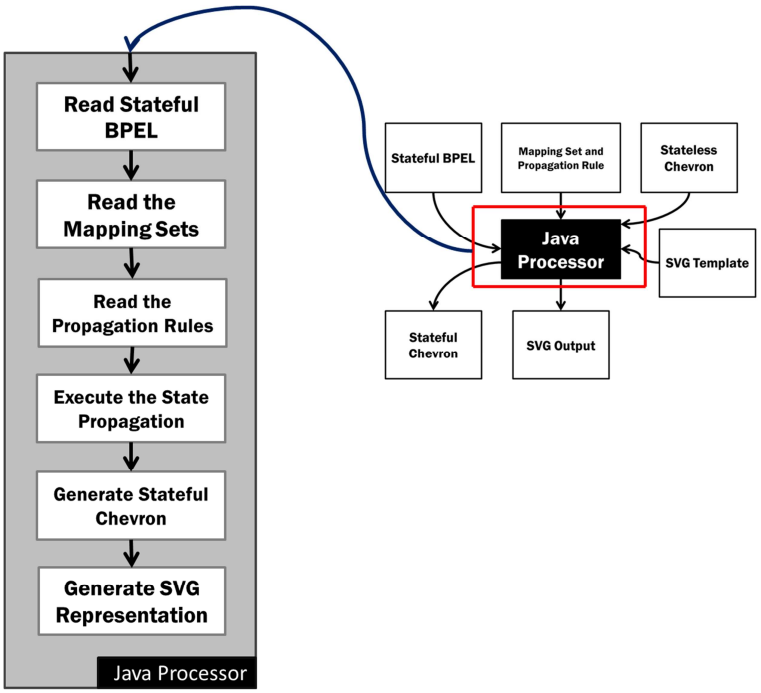


Figure 5.5 Functions of Java Processor

6. Implementation of State Propagation

This chapter deals with all implementation details. In first sub-chapter an SVG template will be proposed. In sub-chapter 6.2 the BPEL will be extended to support the stateful BPEL. Furthermore, the XML Scheme of Chevron process model and state projection will be discussed so that a valid XML document for each of them can be generated. In the last sub-chapter the Java processor takes the SVG template and the XML documents for creating SVG representation based on the stateful Chevron process model.

6.1. SVG Template

SVG template is an SVG file that represents an activity of Chevron process model. The purpose of using the template is to generate a number of Chevron activities effectively based on the corresponding template. The template declares a base position in the coordinate system and it can be manipulated by introducing a running variable whose value will be multiplied by a counter value for every new activity.

There are two templates defined: one for a regular activity and another one for the child activity. The regular activity has characteristics of bigger dimension and horizontally ordered, while the child activity is smaller and vertically ordered. Some placeholders for both templates are needed in order to augment the properties of the activity such as the activity caption, status, color, and so on.

Listing 6.1 SVG Template for Regular Activity

```
<!-- definition of Chevron Activity using Polygon -->
<polygon points="X+40,80 X+240,80 X+280,120 X+240,160 X+40,160 X+80,120"
         style="fill:url(#COLOR);stroke:#STROKE_COLOR;stroke-width:2"/>

  <!-- definition of activity caption -->
  <text x=" X+160" y="130" style="fill:black"
           font-family="Franklin Gothic Heavy"
           font-size="20">CAPTION</text>

  <!-- definition of activity status and icon -->
  <image x="X+210" y="55" width="40" height="40"
           xlink:href="STATUS"/>
  <image x="X+85" y="90" width="55" height="65"
           xlink:href="ICON"/>
```

Listing 6.1 above shows an SVG template for regular activity. The activity is best drawn by using <*polygon*> shape element because it allows drawing a close shape by defining a set of connected straight line elements [w3org2]. Because this activity always moves forward, it is only necessary to manipulate the x coordinate by summing the "X" variable with x value of base position. This method is also valid for the text and images definition. All variables are highlighted with red color.

The element <*polygon*> allows color to be defined by using the *fill* property which refers to variable *COLOR* for defining a certain color. The color used in this thesis is gradient color. Gradient enables a smooth transition from one color to another color [w3s3]. Listing 6.2 below presents a gradient color of aqua to blue.

Listing 6.2 Gradient Color for Chevron Process Model

```
<defs>
   <linearGradient id="aqua_blue" x1="0%" y1="0%" x2="0%" y2="120%">
      <stop offset="0%" style="stop-color:#00ffff;stop-opacity:1"/>
      <stop offset="100%" style="stop-color:#0000ff;stop-opacity:1"/>
   </linearGradient>
</defs>
```

Listing 6.3 SVG Template for Child Activity

```
<!-- definition of Chevron Activity using Polygon -->
<polygon points="X,Y+185 X, Y+185 X, Y+215 X, Y+245 X, Y+245 X, Y+215"
         style="fill:url(#COLOR);stroke:#STROKE_COLOR;stroke-width:2"/>

  <!-- definition of activity caption -->
  <text x=" X" y=" Y+130" style="fill:black"
           font-family="Franklin Gothic Heavy"
           font-size="20">CAPTION</text>

  <!-- definition of activity status and icon -->
  <image x="X" y=" Y+165" width="40" height="40"
           xlink:href="STATUS"/>
  <image x="X" y=" Y+195" width="55" height="65"
           xlink:href="ICON"/>
```

The template definition for child activity is more or less similar to the regular activity definition, but it needs more consideration in terms of position manipulation. The child elements do not only move vertically but also moves forward implicitly because they follow the position of their parent. In this case all values of x coordinate of corresponding parent activity must be retrieved, and based on these values the y

coordinate can be calculated by summing the “Y” variable with the default value of base position to reflect the vertical movement (see Listing 6.3).

6.2. BPEL Extension for Stateful BPEL

This chapter shows how the BPEL file of book ordering is extended for the activity status (see Listing 6.4). The extension is first of all done by a declaration of new namespace with “state” as the prefix. Then the extension formality takes place by specifying the value of *mustUnderstand* attribute to “no”. In the end the activity status can be defined by using the element <*state:activityStatus*> (see Listing 6.5).

Listing 6.4 BPEL Extension for Activity Status

```
<bpws:process exitOnStandardFault="yes" name="Ordering-Process"
    suppressJoinFailure="yes"
    targetNamespace="http://iaas.orderingProcess.com"
    xmlns:bpws=http://docs.oasis-open.org/wsbpel/2.0/process/executable
    xmlns:tns="http://iaas.orderingProcess.com"
    xmlns:state="http://iaas.orderingProcess.com/states">

    <!-- definition of the states extension-->
    <extensions>
            <extension namespace=http://iaas.orderingProcess.com/states
                       mustUnderstand="no"/>
    </extensions>
...
</process>
```

Listing 6.5 Example of Activity Status

```
...
    <bpws:invoke name="prepare Order Processing">
        <!-- activity status of prepare Order Processing -->
        <state:activityStatus>Completed</state:activityStatus>
    </bpws:invoke>
...
    <bpws:reply name="order completion">
        <!-- status of check order completion -->
        <state:activityStatus>No Status</state:activityStatus>
    </bpws:reply>
...
```

6.3. XML Scheme and XML Documents

In sub-chapter 6.3.1 the XML Schema for validating the XML document of Chevron process model are discussed, while in next sub-chapter the schema and the corresponding XML document for state projection are also explained.

.

6.3.1. XML Schema and XML Document for Chevron Process Model

First, an example of XML Schema for the Chevron process model will be proposed and discussed. The schema is used to define the structure of the Chevron process model based on the model definition proposed in sub-chapter 5.1.3.

The complete schema definition is listed in Listing 6.6. The schema allows only two types of activity elements to be declared, namely <*Activity*> and <*complexActivity*>. Both elements are defined as complex type elements because each of them has corresponding status: <*activityStatus*> and <*complexActivityStatus*>, and also properties, e.g., activity id, caption, etc. The schema definition also defines the order of each element, for example <*complexActivityStatus*> element must proceed before the child <*activity*> element, but the <*Activity*> and <*complexActivity*> elements can appear in any order. The states in Chevron process model is pre-defined based on the Chevron states in sub-chapter 5.1.2. They are *Outstanding, Running, Completed*, and also an empty string. The empty string is intended to represent the stateless information of an activity. If status of all activities is not declared, the Chevron process model is stateless.

Listing 6.6 XML Schema Definition for Chevron Process Model

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
	<!-- definition of simpleActivity status -->
	<xs:element name="activityStatus">
		<xs:simpleType>
			<xs:restriction base="xs:string">
				<xs:enumeration value="Outstanding"/>
				<xs:enumeration value="Running"/>
				<xs:enumeration value="Completed"/>
				<xs:enumeration value=""/>
			</xs:restriction>
		</xs:simpleType>
	</xs:element>
	<!-- definition of complexActivity status -->
	<xs:element name="complexActivityStatus">
		<xs:simpleType>
			<xs:restriction base="xs:string">
				<xs:enumeration value="Outstanding"/>
				 <xs:enumeration value="Running"/>
				<xs:enumeration value="Completed"/>
				<xs:enumeration value=""/>
			</xs:restriction>
		</xs:simpleType>
	</xs:element>
	<!-- definition of attributes of chevron process model -->
	<xs:attribute name="processModelName" type="xs:string"/>
	<xs:attribute name="pid" type="xs:string"/>
	 <xs:attribute name="piid" type="xs:string"/>
	<xs:attribute name="processModelVersion" type="xs:long"/>
```

```
	<xs:attribute name="processModelDocumentation" type="xs:string"/>
	<!-- definition of attributes of activities -->
	<xs:attribute name="activityName" type="xs:string"/>
	<xs:attribute name="activityID" type="xs:string"/>
	<xs:attribute name="activityCaption" type="xs:string"/>
	<xs:attribute name="activityDocumentation" type="xs:string"/>
	<xs:attribute name="activityIconForCaption" type="xs:anyURI"/>
	<xs:attribute name="activityIconForComplexActivity" type="xs:anyURI"/>
	<xs:attribute name="activityColor" type="xs:string"/>
	<!-- definition of chevron process model -->
	<xs:element name="chevronProcessModel" type="cPM"/>
	<xs:complexType name="cPM">
		<xs:choice maxOccurs="unbounded">
			<xs:element ref="Activity" maxOccurs="unbounded"/>
			<xs:element ref="complexActivity"
				minOccurs="0" maxOccurs="unbounded"/>
		</xs:choice>
		<xs:attribute ref="processModelName" use="required"/>
		<xs:attribute ref="pid" use="required"/>
		<xs:attribute ref="piid" use="required"/>
		<xs:attribute ref="processModelVersion" use="required"/>
		<xs:attribute ref="processModelDocumentation" use="required"/>
	</xs:complexType>
	<!-- definition of complex activity -->
	<xs:element name="complexActivity">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="complexActivityStatus"/>
				<xs:element ref="Activity" maxOccurs="unbounded"/>
			</xs:sequence>
			<xs:attribute ref="activityName"/>
			<xs:attribute ref="activityID" use="required"/>
			<xs:attribute ref="activityCaption" use="required"/>
			<xs:attribute ref="activityDocumentation" use="required"/>
			<xs:attribute ref="activityIconForCaption" use="required"/>
			<xs:attribute ref="activityColor" use="required"/>
		</xs:complexType>
	</xs:element>
	<!-- definition of simple activity -->
	<xs:element name="Activity">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="activityStatus"/>
			</xs:sequence>
			<xs:attribute ref="activityName"/>
			<xs:attribute ref="activityID" use="required"/>
			<xs:attribute ref="activityCaption" use="required"/>
			<xs:attribute ref="activityDocumentation" use="required"/>
			<xs:attribute ref="activityIconForCaption" use="required"/>
			<xs:attribute ref="activityColor" use="required"/>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Listing 6.7 An example of XML Document for stateless Chevron Process Model

```
<chevronProcessModel processModelVersion="1" piid="15" pid="15.1"
		processModelDocumentation="documentation"
		processModelName="Book Ordering"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:noNamespaceSchemaLocation="chev_rev_stateless.xsd">

<Activity activityColor="aqua_blue"
		activityIconForCaption="order.png"
		activityDocumentation="ordering"
```

```
                activityCaption="Book Ordering" activityID="1">
      <activityStatus></activityStatus>
</Activity>

<complexActivity activityColor="aqua_blue"
                 activityIconForCaption="product.png"
                 activityDocumentation="production"
                 activityCaption="Printing" activityID="2">
      <complexActivityStatus></complexActivityStatus>
      <Activity activityColor="aqua_blue"
                activityIconForCaption="check.png"
                activityDocumentation="checking book availability"
                activityCaption="Check Availability" activityID="2.1">
            <activityStatus></activityStatus>
      </Activity>
      <Activity activityColor="aqua_blue"
                activityIconForCaption="produce.png"
                activityDocumentation="producing"
                activityCaption="Production" activityID="2.2">
            <activityStatus></activityStatus>
      </Activity>
      <Activity activityColor="aqua_blue"
                activityIconForCaption="product.png"
                activityDocumentation="checking product"
                activityCaption="Check Product" activityID="2.3">
            <activityStatus></activityStatus>
      </Activity>
</complexActivity>

<Activity activityColor="aqua_blue"
             activityIconForCaption="deliver.png"
             activityDocumentation="delivery"
             activityCaption="Book Shipment" activityID="3">
      <activityStatus></activityStatus>
</Activity>
</chevronProcessModel>
```

An example of XML document based on the XML Schema in Listing 6.6 is listed above (see Listing 6.7). The XML document represents a stateless Chevron process model and this is the document that refers to the stateless Chevron file described in sub-chapter 5.2.1.

The Chevron process model contains two regular activities and one complex activity, and the complex activity contains yet another three regular activities. The process model has a process version, process id, process instance id, process name that associated to it. While a Chevron activity contains properties (see Table 5.3) such as activity id, caption, icon, color, documentation, and also a status as an element.

6.3.2. XML Schema and XML Document for State Projection

The XML Schema and XML document for state projection are the implementation of the formal mapping definition (see sub-chapter 5.1.4). The schema for state projection is divided into two parts: the *statePropagationSets*, i.e., the mapping sets, and the *statePropagationRules* which specify the source, target activities, and the state mapping set being used. Both parts are defined loosely-coupled to each other. The schema is presented in Listing 6.8.

The *statePropagationSets* might contain one or more state mapping sets. Each state set must have a set name as an identifier so that it is clear which state set is being utilized. One state set contains one or more conditions and each condition must specify one source state and one target state. The source state represents a list of possible states from source activities, i.e., the BPEL states, while target state is the result state, i.e., either in *Outstanding*, *Running*, or *Completed* state. The <*else*> element is used if all conditions are not met, and it has a pre-defined state of *Outstanding*.

The *statePropagationRules* can contain one or more state propagation rules. Every state propagation rule must be identified by a name and it defines the source activities, one target activity, and one state mapping set. Each activity has an identifier and value. These attributes describe which source activity shall be mapped to which target activity.

Listing 6.8 XML Schema Definition for State Projection

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
      <!-- attributes definition -->
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="identifier" type="xs:string"/>
      <xs:attribute name="value" type="xs:string"/>
      <xs:attribute name="pattern" type="xs:string"/>
      <!-- low level activity status definition-->
      <xs:simpleType name="low-levelActivityStatus">
            <xs:list itemType="xs:string"/>
      </xs:simpleType>
      <!-- containState element definition -->
      <xs:element name="containState" type="stateList"/>
      <!-- low level activity states as a list -->
      <xs:simpleType name="stateList">
            <xs:restriction base="low-levelActivityStatus"/>
      </xs:simpleType>
      <xs:element name="targetState">
```

```
            <xs:simpleType>
                  <xs:restriction base="xs:string">
                        <xs:enumeration value="Outstanding"/>
                        <xs:enumeration value="Running"/>
                        <xs:enumeration value="Completed"/>
                  </xs:restriction>
            </xs:simpleType>
      </xs:element>
      <!-- allStateEqual element definition -->
      <xs:element name="allStatesEqual" type="xs:string"/>
      <!--  element definition -->
      <xs:element name="else">
            <xs:simpleType>
                  <xs:restriction base="xs:string">
                        <xs:enumeration value="Outstanding"/>
                  </xs:restriction>
            </xs:simpleType>
      </xs:element>
      <!-- definition of rule element -->
      <xs:element name="condition">
            <xs:complexType>
                  <xs:sequence>
                        <xs:choice>
                              <xs:element ref="containState"/>
                              <xs:element ref="allStatesEqual"/>
                        </xs:choice>
                        <xs:element ref="targetState"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
      <!-- definition of the statePropagationSet -->
      <xs:element name="statePropagationSet">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="condition" maxOccurs="unbounded"/>
                        <xs:element ref="else" minOccurs="1" maxOccurs="1"/>
                  </xs:sequence>
                  <xs:attribute ref="name" use="required"/>
            </xs:complexType>
      </xs:element>
      <!-- definition of the statePropagationSets -->
      <xs:element name="statePropagationSets">
            <xs:complexType>
               <xs:sequence>
                  <xs:element ref="statePropagationSet" maxOccurs="unbounded"/>
               </xs:sequence>
            </xs:complexType>
      </xs:element>
      <!-- definition of the activities -->
      <xs:element name="activity">
            <xs:complexType>
                  <xs:attribute ref="identifier" use="required"/>
                  <xs:attribute ref="value" use="required"/>
            </xs:complexType>
      </xs:element>
      <!-- definition of the fromActivities -->
      <xs:element name="fromActivities">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="activity" maxOccurs="unbounded"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
      <!-- definition of the toActivity -->
```

```
	<xs:element name="toActivity">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="activity"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<!-- definition of the stateSet -->
	<xs:element name="stateSet">
		<xs:complexType>
			<xs:attribute ref="pattern" use="required"/>
		</xs:complexType>
	</xs:element>
	<!-- definition of the statePropagationRule -->
	<xs:element name="statePropagationRule">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="fromActivities"/>
				<xs:element ref="toActivity"/>
				<xs:element ref="stateSet"/>
			</xs:sequence>
			<xs:attribute ref="name" use="required"/>
		</xs:complexType>
	</xs:element>
	<!-- definition of the statePropagationRules -->
	<xs:element name="statePropagationRules">
		<xs:complexType>
		   <xs:sequence>
		    <xs:element ref="statePropagationRule" maxOccurs="unbounded"/>
		   </xs:sequence>
		</xs:complexType>
	</xs:element>
	<!-- definition of state Projection -->
	<xs:element name="stateProjection" type="sP"/>
	<xs:complexType name="sP">
		<xs:sequence>
			<xs:element ref="statePropagationSets"/>
			<xs:element ref="statePropagationRules"/>
		</xs:sequence>
	</xs:complexType>
</xs:schema>
```

Following shows an example of the XML document for the scenario of Book Ordering. This document contains three state mapping sets: combination-1, alteration-1, and alteration-2. The definition of each mapping set will be discussed in chapter 7. *<containState>* and *<allStatesEqual>* elements are variants of source state. The *<containState>* means the corresponding activity is on one of the specified states, while *<allStatesEqual>* is a special source state that describes the source activities are on the same state.

The *statePropagationRules* are a list of mappings between source and target activities. The source activities and target activity are represented by the *<fromActivities>* and *<toActivity>* elements, respectively. Because of the mapping

relationship many-to-one or one-to-one <*fromActivities*> might have one or more activities. The element <*stateSet*> specifies which state mapping set will be employed. For an instance, the source activity with identifier “name” and value “prepare Order Processing” shall be mapped onto target activity with identifier “id” and value “1”. The source state is propagated to target state by using the pattern “alteration-1”.

Listing 6.9 An Example of XML Document for the State Projection

```
<stateProjection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="rules_schema.xsd">
<statePropagationSets>
            <statePropagationSet name="combination-1">
                  <condition>
            <containState>Active;Failed;Suspended;Terminated</containState>
                        <targetState>Running</targetState>
                  </condition>
                  <condition>
                        <allStatesEqual>Completed</allStatesEqual>
                        <targetState>Completed</targetState>
                  </condition>
                  <else>Outstanding</else>
            </statePropagationSet>
            <statePropagationSet name="alteration-1">
                  <condition>
                        <containState>Active;Suspended;Failed</containState>
                        <targetState>Running</targetState>
                  </condition>
                  <condition>
                        <containState>Completed;Terminated</containState>
                        <targetState>Completed</targetState>
                  </condition>
                  <condition>
                        <containState>No Status</containState>
                        <targetState>Outstanding</targetState>
                  </condition>
                  <else>Outstanding</else>
            </statePropagationSet>
            <statePropagationSet name="alteration-2">
                  <condition>
                        <containState>Completed</containState>
                        <targetState>Completed</targetState>
                  </condition>
                  <else>Outstanding</else>
            </statePropagationSet>
      </statePropagationSets>
      <statePropagationRules>
            <statePropagationRule name="Ordering">
                  <fromActivities>
                        <activity identifier="name"
                              value="prepare Order Processing"/>
                  </fromActivities>
                  <toActivity>
                        <activity identifier="id" value="1"/>
                  </toActivity>
                  <stateSet pattern="alteration-1"/>
            </statePropagationRule>
            <statePropagationRule name="Check availability">
                  <fromActivities>
                        <activity identifier="name"
```

```
                              value="check Availability"/>
                  </fromActivities>
                  <toActivity>
                        <activity identifier="id" value="2.1"/>
                  </toActivity>
                  <stateSet pattern="alteration-1"/>
            </statePropagationRule>
            <statePropagationRule name="Produce">
                  <fromActivities>
                        <activity identifier="name"
                              value="Prepare Production"/>
                        <activity identifier="name"
                              value="Invoke Production Process"/>
                        <activity identifier="name"
                              value="wait for completion of production"/>
                  </fromActivities>
                  <toActivity>
                        <activity identifier="id" value="2.2"/>
                  </toActivity>
                  <stateSet pattern="combination-1"/>
            </statePropagationRule>
            <statePropagationRule name="Check Product">
                  <fromActivities>
                        <activity identifier="name" value="Flow"/>
                  </fromActivities>
                  <toActivity>
                        <activity identifier="id" value="2.3"/>
                  </toActivity>
                  <stateSet pattern="alteration-2"/>
            </statePropagationRule>
            <statePropagationRule name="Shipping Completion">
                  <fromActivities>
                        <activity identifier="name"
                              value="shipping"/>
                        <activity identifier="name"
                              value="shipping completion notice"/>
                  </fromActivities>
                  <toActivity>
                        <activity identifier="id" value="3"/>
                  </toActivity>
                  <stateSet pattern="combination-1"/>
            </statePropagationRule>
      </statePropagationRules>
</stateProjection>
```

6.4. Projection of Low-level to High-level View of Process Models

After examining all input files, i.e., stateful BPEL, stateless Chevron, mapping set and propagation rule, and SVG template, the Java functions of the Java processor described in sub-chapter 5.2.2 will be discussed. There are six functions and each of them will be presented in the following sub-chapters.

6.4.1. Read Stateful BPEL

The following code works as follows. First, the value of the *eventBPEL* will be copied into *bpelElement* and *bpelSubElement*, so both have the same event value. The value of *bpelSubElement* is tested against the *activityStatus*. If it is not the

activityStatus element, the parser will continue iterating until it finds the *activityStatus* element, and if it finds one then the activity name, value, and state of *bpelElement* will copied into an array list. The BPEL information will be stored in *listBPELandStates*.

Listing 6.10 Java Code Snippet for Reading the Stateful BPEL

```
while(parserForBPEL.hasNext())
{
   eventBPEL = parserForBPEL.next();
   if(eventBPEL == XMLStreamConstants.START_ELEMENT)
   {
       bpelElement = parserForBPEL.getName().getLocalPart();
       bpelSubElement = parserForBPEL.getName().getLocalPart();
       bpelAttributeName = parserForBPEL.getAttributeValue(0);
       if(!(bpelSubElement.equals("activityStatus")))
       {
          while(parserForBPEL.hasNext())
          {
              eventBPEL = parserForBPEL.next();
              if(eventBPEL == XMLStreamConstants.START_ELEMENT)
              {
                   bpelSubElement = parserForBPEL.getName().getLocalPart();
                   if(bpelSubElement.equals("activityStatus"))
                   {
                        bpelSubElement = parserForBPEL.getName().getLocalPart();
                        bpelStatus = parserForBPEL.getElementText();
                    //adding BPEL activity name | BPEL attribute name | State
                    //e.g.               invoke |       receive order | Completed
                        listBPELandStates.add(new ArrayList<String>());
                        listBPELandStates.get(rowBPEL).add(bpelElement);
                        listBPELandStates.get(rowBPEL).add(bpelAttributeName);
                        listBPELandStates.get(rowBPEL).add(bpelStatus);
                        rowBPEL++;
                        break;
                   }
                   else
                   {
                        bpelElement = bpelSubElement;
                        bpelAttributeName = parserForBPEL.getAttributeValue(0);
                   }
              }
          }
       }
    }
}
```

6.4.2. Read the Mapping Sets

This function fetches all the state mapping sets (see Listing 6.11). The parser will move forward until it meets a start element whose local name is *statePropagationSet*. The parser will return each condition with its corresponding source state (*containState* or *allStatesEqual*), and the *targetState* defined by the mapping set. The parser stops processing a particular mapping set until it encounters *else* element which means that is the end of construct of one particular

mapping set. The parser then processes another *statePropagationSet*, if any. The output of this function is stored in *propagationSets*.

Listing 6.11 Java Code Snippet for Reading the Mapping Sets

```
...
while(parserForSets.hasNext())
{
  eventSets = parserForSets.next();
  if(eventSets == XMLStreamConstants.START_ELEMENT)
  {
    if(eventSets == XMLStreamConstants.START_ELEMENT &&
       parserForSets.getLocalName().equals("statePropagationSet"))
    {
       rowCondition = 0;
       conditionSets = new ArrayList<ArrayList<String>>();
       conditionSets.add(new ArrayList<String>());
       propagationSets.add(new ArrayList<ArrayList<String>>());
       String setName;
       setName = parserForSets.getAttributeValue(0);
       conditionSets.get(rowCondition).add("setName");
       conditionSets.get(rowCondition).add(setName);
       rowCondition++;
       conditionSets.add(new ArrayList<String>());
       while(parserForSets.hasNext())
       {
         eventSets = parserForSets.next();
         if(eventSets == XMLStreamConstants.START_ELEMENT ||
             eventSets == XMLStreamConstants.END_ELEMENT)
         {
           if(eventSets == XMLStreamConstants.START_ELEMENT &&
              parserForSets.getLocalName().equals("condition"))
          {
            while(parserForSets.hasNext())
            {
               eventSets = parserForSets.next();
               if(eventSets == XMLStreamConstants.START_ELEMENT ||
                   eventSets == XMLStreamConstants.END_ELEMENT)
               {
                 if(eventSets == XMLStreamConstants.START_ELEMENT &&
                     parserForSets.getLocalName().equals("containState"))
                 {
                       String containState;
                       containState = parserForSets.getElementText();
                       conditionSets.get(rowCondition).add(containState);
                 }
                 else if(eventSets == XMLStreamConstants.START_ELEMENT &&
                          parserForSets.getLocalName().equals("targetState"))
                 {
                       String targetState;
                       targetState = parserForSets.getElementText();
                       conditionSets.get(rowCondition).add(targetState);
                 }
                 else if(eventSets == XMLStreamConstants.START_ELEMENT &&
                           parserForSets.getLocalName().equals("allStatesEqual"))
                 {
                       String allStatesEqual;
                       allStatesEqual = parserForSets.getElementText();
                       conditionSets.get(rowCondition).add(allStatesEqual);
                 }
                 else if(eventSets == XMLStreamConstants.END_ELEMENT &&
                          parserForSets.getLocalName().equals("condition"))
                 {
                       rowCondition++;
```

```
                        conditionSets.add(new ArrayList<String>());
                         break;
                    }
                  }
               }
            }
          else if(eventSets == XMLStreamConstants.START_ELEMENT &&
                    parserForSets.getLocalName().equals("else"))
          {
               String elseCondition;
               elseCondition = parserForSets.getElementText();
               conditionSets.get(rowCondition).add("else");
               conditionSets.get(rowCondition).add(elseCondition);
               propagationSets.get(rowPropagation).addAll(conditionSets);
               rowPropagation++;
               break;
          }
...
```

6.4.3. Read the Propagation Rules

This function retrieves all information about the source activities, target activity, and the state set (see Listing 6.12). The parser move forward until it meets a start element *statePropagationRule*. The *eventMapping* will be checked whether it is a start element whose local name is *fromActivities*, *toActivity*, or *stateSet*. For an instance if the parser encounters *fromActivities* element, the parser will return all activities the element has until it the value of the *eventMapping* is an end element and the local part equals to *fromActivities*. It applies also to the *toActivities*. Once the parser returns *stateSet* element together with its property, the parser will stop processing the corresponding propagation rule, and starts processing another propagation rule, if any. The propagation result is kept in *mappingList*.

Listing 6.12 Java Code Snippet for Reading the Propagation Rules

```
while(parserForMapping.hasNext())
{
  eventMapping = parserForMapping.next();
  if(eventMapping == XMLStreamConstants.START_ELEMENT &&
       parserForMapping.getLocalName().equals("statePropagationRule"))
  {
       mappingList.add(new ArrayList<String>());
       while(parserForMapping.hasNext())
       {
          eventMapping = parserForMapping.next();
          if(eventMapping == XMLStreamConstants.START_ELEMENT)
          {
             if(parserForMapping.getLocalName().equals("fromActivities"))
             {
                 fromActivities = new ArrayList<String>();
                 while(parserForMapping.hasNext())
                 {
                     eventMapping = parserForMapping.next();
                     if(eventMapping == XMLStreamConstants.START_ELEMENT &&
                            parserForMapping.getLocalName().equals("activity"))
                     {
```

```
                    String activityName;
                    activityName = parserForMapping.getAttributeValue(1);
                    fromActivities.add(activityName);
                }
                else if(eventMapping == XMLStreamConstants.END_ELEMENT &&
                      parserForMapping.getLocalName().equals("fromActivities"))
                {
                    mappingList.get(rowList).addAll(fromActivities);
                    break;
                }
            }
          }
        else if(parserForMapping.getLocalName().equals("toActivity"))
        {
            while(parserForMapping.hasNext())
            {
                eventMapping = parserForMapping.next();
                if(eventMapping == XMLStreamConstants.START_ELEMENT &&
                     parserForMapping.getLocalName().equals("activity"))
                {
                     String activityName;
                     activityName = parserForMapping.getAttributeValue(1);
                     mappingList.get(rowList).add(0,activityName);
                }
                else if(eventMapping == XMLStreamConstants.END_ELEMENT &&
                       parserForMapping.getLocalName().equals("toActivity"))
                      break;
                }
            }
        else if(parserForMapping.getLocalName().equals("stateSet"))
        {
                String stateSetsName;
                stateSetsName = parserForMapping.getAttributeValue(0);
                mappingList.get(rowList).add(0,stateSetsName);
                break;
        }
    }
  }
  rowList++;
 }
}
```

6.4.4. Execute the State Propagation

By processing the output lists from the previous functions the states for Chevron activities can be mapped. For each state set in the *mappingList* the state of source activities will be fetched and kept into a temporary list. If all source states from corresponding source activities have been collected, the source states will be transform by applying the corresponding state set, and the result is stored in list of result state. The resulting state then added to a list of Chevron process (see Listing 6.13).

Listing 6.13 Pseudo Code for the Propagation Rules Execution

```
Input:
List propagationSets
List mappingList
List listBPELandStates
```

```
Output:
List toStates

  FOR each stateSet in mappingList
      FOR each fromActivities in stateSet
      tmp-statelist += read state of fromActivities in listBPELandStates
      NEXT
      END FOR
  resultState = apply propagationsSet that is referenced in mappingList
  add (resultState, to) to toStates
  NEXT
  END FOR
```

6.4.5. Generate Stateful Chevron

The listing below describes how states as a result from state projection can be augmented into Chevron activities. Each activity in stateless Chevron will be verified with the activity in the *toStates* list. The activity status of the qualified activity will be augmented with the state in *toStates*.

Listing 6.14 Pseudo Code for Generating Stateful Chevron

```
Input:
toStates
statelessChevron

Output:
statefulChevron

  FOR each activity in statelessChevron
      IF the ID of activity in statelessChevron =
         the ID activity in toStates THEN
             get the state from toStates
             augment the state from toStates to statelessChevron
      END IF
  NEXT
  END FOR
```

6.4.6. Generate SVG Representation

The following pseudo code shows how the Java function generates the stateful Chevron into an SVG-based visualization. The function will check whether the activity is a regular or is a complex activity. If it is a regular activity then fetch all attributes for corresponding activity and supply the SVG template for regular activity with the attributes. If it is a complex activity then perform the same task as a regular activity but with some additional task to define the child elements. So each child in the complex element will be retrieved including its properties, and for each child implements the SVG template for child activity.

Listing 6.15 Pseudo Code for Generating SVG Representation

```
Input:
statefulChevron

Output:
SVGRepresentation

  IF activity is a regular activity THEN
     get all attributes of the activity including its status
     apply svg template for regular activity
  ELSE IF activity is a complex activity THEN
     get all attributes of the complex activity including its status
     apply svg template for regular activity
     FOR each child activity in complex activity
         get all attributes of the activity including its status
        apply svg template for child activity
     NEXT
     END FOR
   END IF
```

7. Testing

In the previous chapter the implementation of concepts and architecture has been discussed. In this chapter the stateful Chevron and SVG-based visualization for the corresponding Chevron process model will be generated by using the state mapping sets described in sub-chapter 6.3.2. The first sub-chapter will briefly describe the state mapping sets definition. In the last sub-chapter the output files will be presented.

7.1. Example of State Mapping Sets

There are three state mapping sets declared in sub-chapter 6.3.2: combination-1, alteration-1, and alteration-2. The state set of combination-1 is a variant and implementation of *state combination pattern* [schumm3], while alteration-1 and alteration-2 are a variant and implementation of *state alteration pattern* [schumm3].

The state set combination-1 has two conditions. The conditions can be understood like a decision-making statement in Java program e.g. *if-else* statement. Therefore, the first condition will be checked whether the source states of the source activities are on the list of states (*Active, Failed, Suspended, Terminated*). If this is the case, then the resulting target state is *Running*. The second condition specifies whether all source states of the source activities equals to *Completed*. If this is also the case, the *Completed* state will be propagated to the target activity. If all conditions are not met then the target state is Outstanding.

State set alteration-1 contains three conditions. The first condition defines the state mapping from either *Active*, *Suspended*, or *Failed* to *Running* state. The second condition maps state *Completed* or *Terminated* to *Completed*. The last condition associates state *No Status* to *Outstanding*. If all conditions are not met the target state is Outstanding.

State set alteration-2 has only one condition which checks whether the source state is *Completed*. If it is the case the resulting state is also *Completed*. If it is not the case the resulting state is *Outstanding*. This kind of state mapping is adopted to

verify the activities states of BPEL structured activity. In this test case the alteration-2 is applied to <*flow*> activity.

7.2. Stateful Chevron and SVG Representation

The Java processor takes all input files and generates two output files (see Figure 5.4). In this section each output generated from each function in the Java processor until the XML-based stateful Chevron and SVG-based visualization are produced, will be presented.

1) Reading the stateful BPEL file

   First result is the list of BPEL activities, name, and corresponding status from reading the stateful BPEL file. Because there are 60 activities in the file, only the activities related to the test case are presented. The list will be shown in Table 7.1.

BPEL Activity	Activity Name	Status
invoke	prepare Order Processing	Completed
flow	Flow	Active
invoke	check Availabilty	Active
assign	Prepare Production	Completed
invoke	Invoke Production Process	Completed
receive	wait for completion of production	Completed
invoke	shipping	Active
receive	shipping completion notice	No Status

Table 7.1 Result from Reading the Stateful BPEL file

2) Reading the mapping sets

   The result of reading the state mapping sets are presented in Table 7.2, Table 7.3, Table 7.4. The tables are read from left to right, e.g. the set name is combination-1, source state *Completed* and target state *Completed*, and so on.

setName	combination-1
Active;Failed;Suspended;Terminated	Running
Completed	Completed
Else	Oustanding

Table 7.2 State Set for Combination-1

setName	alteration-1
Active;Failed;Suspended	Running
Completed;Terminated	Completed
No Status	Outstanding
Else	Oustanding

Table 7.3 State Set for Alteration-1

setName	alteration-2
Completed	Completed
Else	Oustanding

Table 7.4 State Set for Alteration-2

3) Reading the propagation rules

Below is the output table (Table 7.5) from the third function of Java processor. The information obtained here are the state set being used, and the identifier values of target activity and source activities.

State Set	Target Activity	Source Activities
alteration-1	1	prepare Order Processing
alteration-1	2.1	check Availability
combination-1	2.2	Prepare Production Invoke for Production Process wait for completion notice
alteration-2	2.3	Flow

combination-1	3	shipping shipping completion notice

Table 7.5 Result from Reading the Propagation Rules

4) Executing the state propagation

The resulting states from low-level to high-level view are already propagated (see Table 7.6).

Target Activity	Result State
1	Completed
2.1	Running
2.2	Completed
2.3	Outstanding
3	Outstanding

Table 7.6 Result from Executing the State Propagation

5) Generate Stateful Chevron

Based on the result states above an XML-based file for the Stateful Chevron can be generated (see Listing 7.1 and compare to Listing 6.7). The status of complex activity is derived from the states of sub-activities, and in this case the state is *Running*.

Listing 7.1 The XML Document for Stateful Chevron Process Model

```
<chevronProcessModel processModelVersion="1" piid="15" pid="15.1"
                processModelDocumentation="documentation"
                processModelName="Book Ordering"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="chev_rev_stateless.xsd">

<Activity activityColor="aqua_blue"
               activityIconForCaption="order.png"
               activityDocumentation="ordering"
               activityCaption="Book Ordering" activityID="1">
      <activityStatus>Completed</activityStatus>
</Activity>

<complexActivity activityColor="aqua_blue"
```

```
                activityIconForCaption="product.png"
                activityDocumentation="production"
                activityCaption="Printing" activityID="2">
        <complexActivityStatus>Running</complexActivityStatus>
        <Activity activityColor="aqua_blue"
                  activityIconForCaption="check.png"
                  activityDocumentation="checking book availability"
                  activityCaption="Check Availability" activityID="2.1">
              <activityStatus>Running</activityStatus>
        </Activity>
        <Activity activityColor="aqua_blue"
                  activityIconForCaption="produce.png"
                  activityDocumentation="producing"
                  activityCaption="Production" activityID="2.2">
              <activityStatus>Completed</activityStatus>
        </Activity>
        <Activity activityColor="aqua_blue"
                  activityIconForCaption="product.png"
                  activityDocumentation="checking product"
                  activityCaption="Check Product" activityID="2.3">
              <activityStatus>Outstanding</activityStatus>
        </Activity>
</complexActivity>

<Activity activityColor="aqua_blue"
             activityIconForCaption="deliver.png"
             activityDocumentation="delivery"
             activityCaption="Book Shipment" activityID="3">
        <activityStatus>Outstanding</activityStatus>
</Activity>
</chevronProcessModel>
```

6) Generate SVG representation

Based on the stateful Chevron process model it is now possible to generate the SVG-based visualization (see Figure 7.1).

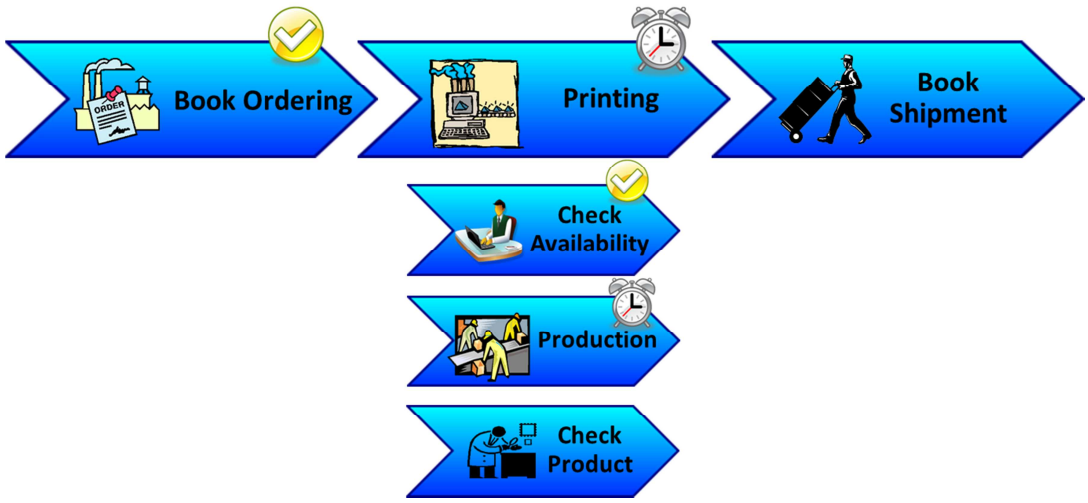


Figure 7.1. SVG Representation of the Stateful Chevron

8. Conclusion and Outlook

A business process describes a well-defined structure how the business is done. However, modeling and executing the business process are performed on different level of abstractions. For an instance, the business process which is designed in high-level business view by the business consultants needs to be refined into an executable construct by the technical operator in the low-level executable view, so that it can be understood and executed by the process engine. Hence, information about current status of the activity instances is only available in low-level view. For monitoring purposes the executing states of the activities should be propagated back to the business users because they are also interested in such information.

In this thesis a prototype to perform such state propagations has been developed. The prototype takes the low-level executable code, in this case based on the BPEL language, and extracts it to obtain all information related to monitoring purposes. Based on state mapping sets the projection from low-level to a high-level view can be carried out and an SVG-based visualization for high-level view, i.e., the Chevron language is being generated.

The prototype can be reused by exporting it as a Java library, so that it can be utilized by any business monitoring tools. The developed prototype supports only basic implementations of state propagation patterns [schumm3], and it can be further extended to fully support all the patterns. Moreover, to provide business users a better view experience the Chevron process model can also be extended in terms of the activity properties (activity color, type of font, etc.). For example, the business users could customize the process model by specifying a particular color palette.

References

[Blow]

M. Blow, Y. Goland, M. Kloppmann, F. Leymann, G. Pfau, D. Roller and M. Rowley:
BPELJ: BPEL for Java, White Paper, 2004,
http://download.boulder.ibm.com/ibmdl/pub/software/dw/webservices/wsbpelj/ws-bpelj.pdf

[Bobrik]

Ralph Bobrik, Thomas Bauer, Manfred Reichert:
Personalized and Configurable Visualizations of Business Processes
Proceedings of the 7th International Conference on Electronic Commerce and Web Technologies
(EC-Web), Springer, 2006

[compas]

COMPAS:
BPEL Extensions for Compliant Services
22 December 2009

[Decidr]

http://www.decidr.eu

[Freund]

J. Freund, K. Götzer:
Vom Geschäftsprozess zum Workflow - Ein Leitfaden für die Praxis
Carl Hanser, 2008

[Griffith]

Arthur Griffith:
Java, XML, and JAXP
Wiley Computer Publishing

[ibm]

IBM:
Web Services Flow Language Version 1.0 (WSFL 1.0)
2001
http://xml.coverpages.org/wsfl.html

[JAXP]

Java API for XML Processing (JAXP) Tutorial

http://java.sun.com/webservices/reference/tutorials/jaxp/html/stax.html

Sun Microsystems, July 2008

[JWSPT]

Java Web Services Performance Team:

Streaming APIs for XML Parsers – White Paper

Sun Microsystems, August 2005

[Karastoyanova]

Karastoyanova, D.; Khalaf, R.; Schroth, R.; Paluszek, M.; Leymann, F.:

BPEL Event Model

Institut für Architektur von Anwendungssystemen, Universität Stuttgart, 2006

[Leymann]

Frank Leymann, Dieter Roller:

Production Workflow: Concepts and Techniques

Prentice Hall, 1999

[microsoft]

Microsoft:
XML Business Process Language (XLANG)
2001
http://xml.coverpages.org/xlang.html

[oasis]

OASIS:

Web Services Business Process Execution Language Version 2.0

OASIS Standard, 11 April 2007

[oracle1]

Oracle:

Oracle on DOM package summary

http://download.oracle.com/javase/1.5.0/docs/api/org/w3c/dom/package-summary.html

[oracle2]

Oracle:
Oracle on Interface XMLStreamReader
http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/1.5/api/javax/xml/stream/XMLStreamReader.html

[oracle3]
Oracle:
Oracle on Interface XMLStreamWriter
http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/1.5/api/javax/xml/stream/XMLStreamWriter.html

[Polyvyanyy]
A. Polyvyanny, S. Smirnov, M. Weske:
Process Model Abstraction: A Slider Approach
Proceeding of the 12th IEEE Enterprise Distributed Object Conference (EDOC)
IEEE Computer Society, 2008

[Schumm1]
David Schumm, Gregor Latuske, Frank Leymann:
A Prototype for View-based Monitoring of BPEL Processes
March 2011

[Schumm2]
David Schumm, Frank Leymann, Alexander Streule:
Process Viewing Patterns
October 2010

[Schumm3]
David Schumm, Gregor Latuske, Frank Leymann, Ralph Mietzner, Thorsten Scheibler
State Propagation for Business Process Monitoring on Different Levels of Abstraction
June 2011

[w3org]
W3C:
W3C on well-formed XML.
http://www.w3.org/TR/2006/REC-xml11-20060816/#dt-wellformed

[w3s1]
W3School:
W3School for Basic XML
http://www.w3schools.com/xml/xml_whatis.asp

[w3s2]
W3School:
W3School on Basic SVG
http://www.w3schools.com/svg/svg_intro.asp

[WAPI]
Workflow Management Application Programming Interface (Interface 2 & 3) Specification.
Document Number WFMC-TC-1009. Version 2.0e.
Workflow Management Coalition, Winchester 1998

[zurMuehlen]
Michael zur Muehlen:
Workflow-based Process Controlling: Foundation, Design, and Application of Workflow-driven Process Information System
Logos Verlag Berlin, 2002

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Stuttgart, 21.06.2011

_____

(Sumadi Lie)