Verification of Business Processes for Web Services

Mariya Koshkina

A thesis submitted to the Faculty of Graduate Studies
in partial fulfilment of the requirements
for the degree of

Master of Science

Graduate Programme in Department of Computer Science
York University
Toronto, Ontario
October 2003

ABSTRACT

The Business Process Execution Language for Web Services (BPEL4WS or simply BPEL) is a recently developed language, which is used to specify interactions between web services. Among its features it allows specification of concurrent behavior. Erroneous specification can lead to such problems as deadlock. In our research we focus on the concurrency mechanism in BPEL. Our main goal is to analyze processes in order to detect possible deadlocks. To achieve this we introduce a process algebra called the BPE-calculus. It is a small language which captures all the BPEL features relevant to the analysis. This process algebra is modelled using a labeled transition system. An existing verification tool called the Concurrency Workbench is customized to use our BPE-calculus. This tool allows us to verify many properties of BPE-calculus processes specified in a logic called the μ-calculus, including deadlock freedom.

ACKNOWLEDGEMENTS

I would like to especially thank my supervisor Franck van Breugel for his tremendous contributions to my thesis, for his constant guidance, support, and encouragement. I am grateful for his many usefull comments on this work and for the many things that I have learned from him.

I also want to thank Bill O'Farrel and Jon Bennett at the IBM Toronto Lab. I thank Bill for suggesting this interesting research topic and for providing some insightful comments. I am grateful to Jon for helping me understand BPEL and web services technology.

I want to express my gratitude to the Center for Advanced Studies at the IBM Toronto Lab, in particular to Marin Litou, for providing me with the comfortable working environment and the financial support throughout my research.

Thanks to Rance Cleaveland for his much appreciated help with the Concurrency Workbench.

I would also like to thanks members of the committee Yves Lesperance, Parke Godfrey, and Markus Biehl for their time and for their helpful comments.

Last but not least, I thank Computer Science Department of York University for giving me this great opportunity to study, first at the Bachelor, and then at the Master level.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 From HTML to Web Services

In the beginning, the information highway mainly carried hypertext markup language (HTML) documents. Such a document contains a mixture of formatting data and factual data. For example, in the HTML document

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
   <HEAD>
      <TITLE>Example</TITLE>
   </HEAD>
   <BODY>
      <CENTER>
         <I>Hello world!</I>
      <CENTER>
   </BODY>
</HTML>
```

`Example` and `Hello world!` is the only factual data. The rest is mainly formatting data. Although the formatting data is usually essential for the use of HTML documents by us humans, for their use by computer programs it is often a burden. Several tools, like for example TeSS [CCD+03], have been developed to parse HTML documents and extract the factual data. However, this so-called screen scraping only has had limited success.

To address this and some related issues, the extensible markup language (XML) [BPSMM00] was created. In an XML document, the factual data is

structured using tags. However, unlike in HTML, in XML the tags are not used for formatting the factual data. The tags are only used to structure the data. Moreover, unlike HTML that uses a number of predefined tags, XML allows users to specify their own tags. For example, the factual data of the above HTML document can be captured by the XML document

```
<?xml version="1.0"?>
 <greeting name="Example"/>Hello world!</greeting>
```

In the above example, `greeting` is a user-defined tag that contains text and has an attribute `name`.

The name and structure of XML tags are defined using XML schema. For instance the `greeting` tag can be defined as

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="greeting">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="name" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

An XML Schema [Fal01] can be as simple as defining a greeting tag. It can also be as complex as defining a whole language. For example, WSDL, which is the language discussed below, is defined in terms of an XML schema. There are a variety of XML parsers available for different platforms. They allow one to parse, validate and edit XML documents. Because of its simplicity and flexibility XML has become very popular. It inspired several new technologies including a number of XML-based languages that are becoming standards.

Web services make heavy use of XML. A web service is a way for programs to interact with web sites. A web service is a program available over the Internet. TerraService [BGE+02] is an example of such a web service. It provides programmable access to the TerraServer database. This database contains 3.3 terabytes of high resolution aerial images of North America. Another example is the Google web service. It allows us to query more than 3 billion web documents.

At the heart of web services technology lies the XML-based language WSDL (Web Services Description Language). It provides the description of the web

services interface. The main purpose of this language is to enable platform-independent communication between a web service and its clients. A WSDL document provides all the information a client needs in order to use the web service. WSDL was initially proposed by Ariba, IBM and Microsoft and is now developed by a W3C working group. The latest working draft for WSDL 1.2 specifications was published in July 2002 [CGMW02].

For example, consider a web service that provides current weather conditions. Part of its WSDL document will look as follows:

```
<message name="City">
    <part name="body" element="xsd:string"/>
 </message>

 <message name="WeatherCondition">
    <part name="body" element="xsd:weatherConditionType"/>
 </message>

 <portType name="weatherWebService">
    <operation name="getWeather">
       <input message="City"/>
       <output message="WeatherCondition"/>
    </operation>
 </portType>
```

A web service communicates with its clients using messages. The WSDL document provides the description of the messages used. A message can contain several parts. Each part is either of a simple type (i.e., string, int, real) or of a complex user-defined type. Type declarations (omitted from our example above) specify user-defined data types such as the `weatherConditionType`. A port type defines the actual operations performed by the web service. In our example, the `weatherWebService` port type contains one operation called `getWeather`. This operation takes as its input the message `City` and replies with the message `WeatherCondtions`.

Web services are posted on the web, similarly to web pages. Alike regular HTML pages, there are web directories of web services. For example, if we are looking for a web service that provides current weather conditions we can browse a web service directory to find such a service with its WSDL description. Several weather web services from different providers can have the same interface. This

is a significant advantage, since a client program can be designed to dynamically find an available weather web service and invoke it.

1.2 Introducing BPEL

Web services can be exploited in many different ways. For example, they can be used to order an airline ticket over the Internet. In such an example, the customer communicates with (the web service of) a travel agent, who sends a request to one or more (web services of) airlines, waits for confirmations, returns a reply to the customer, and starts a payment process. This business process involves several interactions between a number of partners over some period of time. It is important to be able to specify such business processes in a clear and platform-independent way. To accomplish that BEA, IBM and Microsoft introduced the Business Process Execution Language for Web Services (BPEL). The initial public draft release of the BPEL specification can be found in [CGK+02]. Like WSDL, BPEL is XML-based. That is, its syntax is defined in terms of an XML schema. For example, the BPEL snippet

```
<invoke partner="AirCanada"
   portType="reservationsPT"
   operation="makeReservation"
   inputContainer="request"
   outputContainer="confirmation">
</invoke>
```

invokes a flight reservation web service.

BPEL is used to specify the partners involved in the process, the content of messages being passed between them, the flow of messages, process logic, and fault handling. In BPEL, the basic activities include invoking web service operations, receiving and replying to requests, and assigning data to messages. These basic activities are combined into structured activities using sequencing, switch constructs, while loops and selective communication, all of which be will discussed in more detail below. BPEL also has mechanisms to handle exceptions and to roll back the process in case of an error. This is accomplished by fault and compensation handlers.

We present an example of a BPEL process for a travel agency (some details are omitted). It starts with some definitions:

```
<process name = "travelAgentExample"
   xmlns = "http://schemas.xmlsoap.org/business-process/"
   xmlns:tns="urn:travelAgentExample"
   suppressJoinFailure="yes">

   <partners>
       <partner name = "client"/>
       <partner name = "AmericanAirlines"/>
       <partner name = "BritishAirways"/>
       <partner name = "AirCanada"/>
       <partner name = "weatherService"/>
       <partner name = "carRental"/>
   </partners>

   <containers>
      <container name="request"
         messageType="tns:RequestMessageType"/>
      <container name="confirmation"
         messageType="tns:ConfirmationMessageType"/>
      <container name="forecast"
         messageType="tns:WeatherMessageType"/>
      <container name="reply"
         messageType="tns:ReplyMessageType"/>
      <container name="rentalConfirmation"
         messageType="tns:ConfirmationMessageType"/>
   </containers>
```

A BPEL document usually contains partner and data container definitions. The former specifies partners participating in the given process, for example, `client`, `carRental` and `AirCanada`. The latter specifies variables used to contain the messages exchanged between web services, for example, `request` and `reply`. Each container is defined by its `messageType`, which is specified in the WSDL document.

The most interesting part of the BPEL document is the description of the process logic. In this example, the following takes place. First, the client request is received. Then, depending on the destination, a ticket is reserved from a particular airline.

```
   <sequence>
```

```
  <receive partner="client"
     portType="clientPT"
     operation="makeTravelArrangements"
     container="request"/>
  <flow>
     <links>
        <link name="travel-canada"/>
        <link name="travel-us"/>
<link name="rent-to-assign"/>
     </links>
     <switch>
        <case condition =
           "bpws:getContainerData('request',
              'destination_country')='Canada'">
           <invoke partner="AirCanada"
              portType="reservationsPT"
              operation="makeReservation"
              inputContainer="request"
              outputContainer="confirmation">
              <source linkName="travel-canada">
           </invoke>
         </case>
        <case condition =
           "bpws:getContainerData('request',
              'destination_country')='US'">
           <invoke partner="AmericanAirlines"
              portType="reservationsPT"
              operation="makeReservation"
              inputContainer="request"
              outputContainer="confirmation">
              <source linkName="travel-us"
               transitionCondition =
               "bpws:getContainerData('request',
               'destination_city')!= 'New York'"/>
           </invoke>
        </case>
        <otherwise>
           <invoke partner="BritishAirways"
```

```
            portType="reservations"
            operation="makeReservation"
            inputContainer="request"
            outputContainer="confirmation"/>
        </otherwise>
    </switch>
```

At the same time as the reservation takes place, the weather service is contacted. It provides the weather forecast for the client's destination.

```
<invoke partner="weatherService"
   portType="weatherPT"
   operation="getForecast"
   inputContainer="request"
   outputContainer="forecast"/>
```

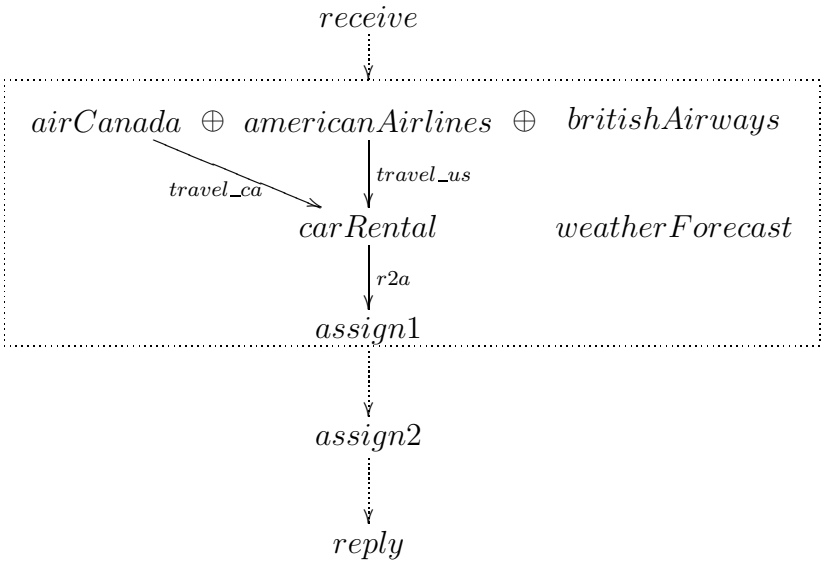
After the reservation has been completed, the car rental service might be invoked. This happens only if the city of destination is in Canada or US and is not New York City.

```
<invoke partner="carRental"
   portType="carRentalPT"
   operation="rent"
   inputContainer="request"
   outputContainer="rentalConfirmation"
   joinCondition = "bpws:getLinkStatus('travel-canada')
      or bpws:getLinkStatus('travel-us')">
   <target linkName="travel-canada"/>
   <target linkName="travel-us"/>
   <source linkName="rent-to-assign"/>
</invoke>
<assign>
   <target linkName="rent-to-assign"/>
   <copy>
      <from container="rentalConfirmation"/>
      <to container="reply" part="part2"/>
   </copy>
 </assign>
</flow>
```

Finally, the `reply` container is constructed by collecting confirmation(s) information and the weather forecast. This container is sent back to the client.

```
      <assign>
         <copy>
            <from container="confirmation"/>
            <to container="reply" part="part1"/>
         </copy>
         <copy>
            <from container="forecast"/>
            <to container="reply" part="part3"/>
         </copy>
      </assign>
      <reply partner="client"
        portType="clientPT"
        operation="makeTravelArrangements"
        container="reply"/>
   </sequence>
</process>
```

Here is the graphical representation of the above process:



The flow is represented here as a dashed box. The sequential composition is shown using dashed arrows. The links are represented as solid arrows.

The process logic in BPEL is described using activities. In BPEL specifications they are classified as basic activities and structured activities. An example of a basic activity is receiving a request from the client. Another example would be invoking a reservation web service of an airline. Basic activities also include replying to requests, assigning data from one container to another, terminating the process, waiting for some period of time, and doing nothing.

Structured activities define the flow control of the process. They include basic programming constructs as sequencing,

```
<sequence>
    activity1
    activity2
       ...
</sequence>
```

while loops,

```
<while condition="bool-expr">
    activity
</while>
```

and switch statements (as in the travel agent example presented above).

BPEL also includes a structured activity called pick. The pick construct allows for selective communication. Consider, for example,

```
 <pick>
    <onMessage partner="client" operation="buyTicket">
        <invoke partner="airline" operation="buy"/>
    </onMessage>
    <onMessage partner="client" operation="cancelReservation">
        <invoke partner="airline" operaton="cancel"/>
    </onMessage>
 </pick>
```

On the one hand, if a message `buyTicket` is received from the `client` then the activity `buy` is executed. In that case, the `cancelReservation` activity will not be performed. On the other hand, the receipt of a message `cancelReservation` from the `client` triggers the execution of the `cancel` activity and discards the `buy` activity. In the case that both messages are received almost simultaneously, the choice of activity to be executed depends on the implementation of BPEL.

This pick construct is similar to the nondeterministic choice construct found in process algebras like the π-calculus [Mil99] and is also reminiscent to the choose construct of Concurrent ML [Rep99].

Concurrency in BPEL is provided by the flow construct. For example, in the travel agent process the switch activity, the invocation of a weather service, the car rental service invocation, and the assignment activity are executed concurrently.

Synchronization is provided by specifying directed links between activities with an optional transition condition for each such link. If two activities are linked together then the target of the link can be executed only after the source activity has completed. The transition condition is a boolean expression that depends on data values of the input containers. The transition condition is evaluated once the source activity completes. For example, in travel agent example there is a link between the car rental activity and the assignment activity for rental confirmation. The rent operation is the source of a link `rent-to-assign`. The assignment activity is the target. There is no transition condition specified. Therefore the default value, `true`, is used. Once a car rental operation completes its outgoing link is activated and the assignment activity can start its execution.

Every activity that has incoming links also has a join condition associated with it. A join condition is a boolean expression. It can only consist of statuses of incoming links connected by boolean operators. A join condition can only be evaluated if all of its incoming link are defined. That is, if all of the source activities of those links have completed. If the join condition of an activity evaluates to true then this activity is executed. If not, the activity is skipped. For example, the car rental activity is executed only when one of its incoming links has value `true`.

To make the use of links safer, some restrictions are applied. All links have to be declared at the beginning of the corresponding flow. Each link has to have exactly one source activity and one target activity. Moreover, two activities can be connected by exactly one link. But each activity can have several outgoing and incoming links.

In the travel agent example, if the client's destination is neither Canada nor US or if client is traveling to New York City, then the join condition for the car rental activity becomes `false`. This activity is not executed, but what happens to its outgoing links? If this link is left undefined then the assignment activity will be waiting forever. The process will never terminate. In order to solve this problem, a technique called dead path elimination (DPE) is introduced. This technique sets all the outgoing links of skipped activities to `false`. This triggers propagation of

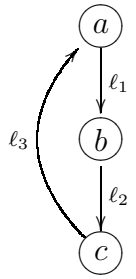
`false` link status. As a result of that, all skipped activities are garbage-collected and the process can continue. In our example, once the car rental activity is skipped its outgoing link is set to `false`, causing the assignment activity for the rental confirmation to be skipped as well. After that the flow completes and the next activity in sequence is executed. DPE is not used exclusively for join condition failure. It is also used whenever an activity has to be skipped due to pick or switch statements.

1.3 Verification of BPEL

It is well-known that concurrent programs are notoriously difficult to get right [Sch98]. The fact that concurrency is one of the key ingredients of BPEL makes specifying business processes in BPEL a far from trivial task. Therefore, the need to build tools to support the development of BPEL documents is pressing.

Our initial goal was to develop a tool that can detect deadlocks in BPEL specifications. Since it is desirable that BPEL specifications are deadlock free, such a tool would be valuable for those who specify business processes in BPEL.

Deadlock can be caused by a control cycle in linking. Links form a control cycle when a source activity cannot start until its target completes. This is illustrated by the following example:



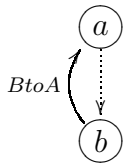
where a, b, and c are activities and ℓ_1, ℓ_2, and ℓ_3 are links. In this example, a cannot start until c has completed, but c cannot be executed until both a and b have completed.

Another less obvious example of a control cycle is

```
<sequence>
   <invoke partner="partnerA" operation="a">
      <target linkName="BtoA"/>
   </invoke>
```

```
    <invoke partner="partnerB" operation="b">
       <source linkName="BtoA"/>
    </invoke>
 </sequence>
```

This example can be depicted graphically as follows:



In this diagram the dashed arrow represents sequential composition.

In this example, although there is no link from a to b, activity b will only start if a completes, since they are sequentially composed. Yet because of the link $BtoA$, a cannot start unless b has completed.

As we will outline below, the tool we developed takes as input a BPEL process and a property specified in a logic and checks if the BPEL process satisfies the property. One can express deadlock freedom in the logic and, hence, the tool can detect deadlocks. However, many other interesting properties can be captured by the logic and, therefore, can be checked by our tool.

1.4 Overview

In order to efficiently analyze BPEL processes we need to considerably simplify the model. All the information that is irrelevant for the analysis needs to be omitted. To accomplish that we developed a small process algebra called the BPE-calculus that is similar in flavour to calculi like, for example, CCS [Mil89]. It accurately models all of the BPEL process behavior related to control flow. In Chapter 2 we present the BPE-calculus and describe the simplification steps taken in order to derive it.

The BPE-calculus is modelled by means of a structural operational semantics [Plo81]. In Chapter 3 we present a labelled transition system for the BPE-calculus. The transitions of the systems are defined by a collection of rules. Which rules apply to a BPE-process depends on its structure.

Having developed a process algebra and a labelled transition system, we reuse an existing tool, the Concurrency Workbench (CWB), to do the analysis [CS96]. The CWB is a flexible verification tool that can be extended to support different

process description languages. It provides a capability to verify process properties, such as, for example, deadlock freedom, specified by μ-calculus [Koz83] formulas. The CWB can be extended to support the BPE-calculus. In order to do that we use a tool called the Process Algebra Compiler (PAC) [CS02]. The PAC is designed to generate front-ends for verification tools, in particular, the CWB. In Chapter 4 we describe how the PAC is used to generate the BPE-calculus front-end for the CWB.

Once the CWB has been extended to support the BPE-calculus we can use it to verify BPE-processes. Chapter 5 describes how the CWB can be used to detect deadlock. It also demonstrates other verification methods provided by the CWB.

1.5 Alternative Approaches

The above diagram of the BPEL process looks like a directed graph that contains a cycle. Therefore the first solution that comes to mind is to represent processes as directed graphs. In order to find the deadlock due to a control cycle it suffices to find cycles in the graph. These can easily be detected using existing graph algorithms. An advantage of this approach is its simplicity. On the other hand, our approach gives rise to an analysis tool that allows us to verify many useful properties (beside deadlock freedom) of business processes. The tool also provides other verification techniques such as preorder checking and equivalence checking that allow to verify if the process satisfies its specifications.

UML [BRJ98] statecharts and activity diagrams are often used to model business processes. But although UML is a useful modelling language, it does not provide the formal semantics necessary to perform analysis and verification as pointed out in [DH01].

Another approach is to model processes using Petri nets [Rei85]. Constructs of business processes that support sequencing, choice, concurrency, and iteration can be easily represented by means of Petri nets. In fact, there has been a considerable amount of research done in this area, for example, [Aal03, Aal00, DE00, NM02]. But such aspect of BPEL as DPE cannot be captured easily using this approach as has been argued in [AH02].

We chose to apply the results of process algebra theory for several reasons. First, it is a well-studied area which provides solid theoretical background. Second, it allows us to model all of the behavior of BPEL relevant to the analysis. Finally, there are customizable tools available for verification. There are a few

studies that chose the same approach. We discuss them next.

In [Sch99] Schroeder presents a translation of business processes specifications into CCS. Subsequently, the CWB can be used for verification. The business process language that is studied in that paper is considerably simpler than BPEL. It is not clear how to capture DPE in CCS (it can be done though, since CCS is Turing complete). Furthermore, if the CWB detects, for example, that two processes are not bisimilar, then it will not produce a counterexample in terms of the business processes but in terms of the underlying CCS-processes.

Also Piccinelli and Williams [PW03] present a calculus for business processes that is similar to CCS. That calculus is much simpler than our BPE-calculus and does not include advanced synchronization patterns like DPE.

Nakajima [Nak02] describes how to use the SPIN model checker to verify web services flows. The language used to specify flows is WSFL (Web Services Flow Language) which is one of BPEL's predecessors. In order to do the verification using SPIN, processes are first translated into the Promela specification language provided by SPIN. The level of abstraction in this case is lower than in our approach. The advantage of the CWB over SPIN is that the former provides other functionalities in addition to model checking. SPIN does not support preorder and equivalence checking. In our opinion the BPE-calculus is better suited to describe business processes since it was designed for this purpose. The disadvantage of translating business processes into a generic language for verification is that it is not easy to relate the diagnostic information returned by the verification tool to the original process. In case of the BPE-calculus the trace returned by the CWB is closely related to the trace in the original BPEL process. The approach in [KGMW00] that uses the LTSA toolkit and the FSP process algebra suffers from the same weaknesses.

Chapter 2

The BPE-Calculus

The process algebra for BPEL has to accurately model the control flow of business processes. Yet it should be simple and concise. In order to derive such an algebra we start by simplifying BPEL. We throw away features of the language that are irrelevant for the analysis.

We make a decision to disregard data in our analysis. This allows us to look at the potential control flow in the process without concern about the unlimited possibilities of data values. We also ignore fault and compensation handlers, since we would like to look at the main process without concerning ourselves with error handling. Such elements as partner definitions and process attributes can also be ignored since they are irrelevant to the control flow.

After this initial abstraction step we end up with a set of activities that can be combined to build a process. The attribute list of the activities is kept to a minimum. Only the names of the partner, the port, and the operation are required to identify a basic activity. Each activity has the optional target and source elements, since any activity can have a number of incoming and outgoing links. In the BPEL definition they are called standard elements:

standard-elements ::=
`<source linkName=`*name* `transitionCondition=`*bool-expr*`?/>*`
`<target linkName=`*name* `/>*`

Also every activity has an optional join condition associated with it:

join-condition ::= `joinCondition=`*bool-expr*

These features are related to linking and therefore are very important for the

analysis. The grammar of the language resulting from the first simplification step looks as follows.[1]

activity::=*basic-activity* | *structured-activity*

basic-activity::=
<receive partner=*name* portType=*name* operation=*name* *join-condition*?>
    standard-elements
</receive>
| <reply partner=*name* portType=*name* operation=*name* *join-condition*?>
    standard-elements
</reply>
| <invoke partner=*name* portType=*name* operation=*name* *join-condition*?>
    standard-elements
</invoke>
| <terminate *join-condition*?>
    standard-elements
</terminate>
| <wait for=*duration-expr*? until=*deadline-expr*? *join-condition*?>
    standard-elements
</wait>
| <assign *join-condition*?>
    standard-elements
</assign>
| <empty *join-condition*?>
    standard-elements
</empty>

The structured activities are as follows:
structured-activity ::=
<sequence *join-condition*?>
    standard-elements
    activity+

[1]Here we use the same notation as used in [CGK+02] to describe the structure of BPEL. Roughly, | indicates choice, ? indicates that the element can be omitted, * indicates that the element can be repeated an arbitrary number of times, and + indicates that the element is used at least once.

```
</sequence>
| <switch join-condition?>
    standard-elements
    <case condition=bool-expr>+
       activity
    </case>
    <otherwise>?
       activity
    </otherwise>
</switch>
| <while condition=bool-expr join-condition?>
    standard-elements
    activity
</while>
| <pick join-condition?>
    standard-elements
    <onMessage partner=name portType=name operation=name >+
       activity
    </onMessage>
    <onAlarm for=duration-expr?  until=deadline-expr?>*
       activity
    </onAlarm>
</pick>
| <flow join-condition?>
    standard-elements
    <links>?
       <link name=name>+
    </links>
    activity+
</flow>
```

In the above description we omitted the specification of *name*, *bool-expr*, *duration-expr*, and *deadline-expr*. Their description can be found in [CGK+02].

The original BPEL XML schema takes roughly 15 pages. In comparison, the above language is much simpler. But our final goal is to derive a process algebra like, for example, CCS [Mil89]. In order to achieve this we have to make a number of other simplifications. Among other things, we would like to get rid of the XML

tags and introduce a more concise notation. We will call the resulting language the BPE-calculus.

2.1 Basic Activities

In the simplified version of BPEL there are several different basic activities. They are: invoke, receive, reply, terminate, wait, assign, and empty. Invoke, receive, and reply are used for the interactions with the environment. They are observable (external) basic activities. In order to simplify our model we would like to abstract from the particulars of each of these activities. We know that whenever one of these activities is invoked some interaction with the environment takes place. We don't need to distinguish between invoke, reply and receive. We introduce a set $\mathbb{A}_e$ of external basic activities. The concept of external basic activity in the BPE-calculus is analogous to that of action in other known process algebras such as CCS [Mil89].

The basic activity that deserves some special treatment is termination. Execution of this activity effects the whole process. Once it is executed the whole process stops. In our process algebra it is represented by the activity *end*.

The empty activity performs no function. Nevertheless, it is important in the analysis of control flow since it can be the source and/or the target of a link. The empty activity is not observable (internal). We will use τ to denote the internal activities in our process algebra.

The only two basic activities that we have not discussed so far are assign and wait. The concept of time is not a part of our model. Therefore, the wait activity performs no function. It is important for the analysis because it might contain incoming and/or outgoing links. Therefore, the wait activity is dealt with in the same way as the empty activity. The same applies to the assign activity. We have abstracted from all the data in our model. Therefore the assign activity serves no purpose, except that it might be the source and/or the target of a link. Empty, wait, and assign are all internal basic activities and are denoted by τ.

Therefore, the basic activities are divided into external (or observable) basic activities and internal (or unobservable) basic activities. Note, that when a basic activity in BPEL is translated to the BPE-calculus as an internal basic activity it looses its name and becomes simply a τ-activity. But when an activity is external, it is uniquely identified by its name in the BPE-calculus process. The conversion mechanism (from BPEL to the BPE-calculus) has a freedom to choose which activities are external and which are internal. Deadlock detection is not

effected by this choice. That is, if there is a deadlock in the process it will be detected in any case.

There are advantages and disadvantages for each classification of activities. Defining all of the basic activities as external has an advantage of maintaining more information about the original process. This is useful when we want to relate back to the original BPEL process. For example, we found a deadlock and we want to know the exact place it occurs. If we know that deadlock happened after executing some external activity a_i we can find the corresponding activity in the original BPEL process. If, on the other hand, deadlock happened after internal activity τ we might not know what is the corresponding activity in the BPEL process. The advantage of defining some of the basic activities as internal is that it provides a higher level of abstraction. It allows to use verification methods such as equivalence checking. We refer the reader to Chapter 5 for more details on verification methods.

As was mentioned above, we choose to consider receive, reply, and invoke to be external activities. All other basic activities (excluding termination, which is a special case) are considered to be internal basic activities.

2.2 Sequence

Sequence is a structured activity. It is one of the activities that defines the flow of control. Instead of using XML tags we represent it in our process algebra using a semicolon. For example,

$$buyTicket\,;rentCar$$

translates as the activity $buyTicket$ followed by the activity $rentCar$.

2.3 Pick and Switch

In BPEL a pick construct can contain one or more `onMessage` clauses and an optional `onAlarm` clause. Each `onMessage` clause specifies what should be done if a particular message is received. The alarm option can be used to set a timer that triggers some action if no message is received within the specified period of time. Only one of the clauses of the pick statement gets executed. If there are multiple message clauses then the first message to arrive is picked. The alarm clause gets picked if the timer expires. In the case that several messages arrive

simultaneously it is up to the implementation to decide which one is chosen. We assume that in this case the choice is non-deterministic. Therefore the behavior of pick is guided by the environment of the process. Considering this, pick resembles the choice construct present in many different process algebras including CCS.

The arrival of the message is an interaction with the environment. It is modelled as an external basic activity. As we mentioned before, the concept of time is not a part of our model. Therefore, the expiration of an alarm is an internal activity, similar to the wait activity. We represent it as a τ-action. To model the fact that the clause of the pick activity is chosen after the message arrives or the timer expires we represent each clause as: $\alpha\,;P$, where α is either a basic activity or τ, and P is a process. This is a special case of sequence. As often done, a plus sign is used to represent choice. For example, in

$$onBuy\,;buyTicket + onCancel\,;cancelReservation + \tau\,;sendReminder$$

either one of the messages is received or the timer expires and the corresponding activity is executed.

Another construct of BPEL that offers a choice is the switch statement. The switch construct contains a list of cases. The boolean expressions in the case conditions are based on data values. Since in our analysis we disregard all the data, these boolean expressions cannot be evaluated. All that is known to us at this level of abstraction is that evaluation of an expression takes place and that one of the cases is chosen. This can be modelled in the BPE-calculus as a non-deterministic choice. There is a difference between the choice construct for pick and the one for switch. The former depends on the environment, whereas the latter does not. We represent the latter with the symbol $\oplus$. For example, in

$$flyAirCanada \oplus flyBritishAirways$$

the activity will be chosen randomly. This simplification of a switch statement results from our decision to ignore data in the analysis. Notice that, the resulting construct still carries the information about the possible flow of control in the process.

Therefore, the BPE-calculus has two choice constructs also known as external (influenced by environment) and internal (non-deterministic) choice. It is not uncommon for a process algebra to include both. For example, CSP [Hoa85] has both flavours of choice constructs as well.

2.4 While Loop

Among other constructs, BPEL includes a typical while loop: while b do $activity$. The boolean expression b depends on the data. Since we made the decision to disregard the data in our analysis, the loop construct becomes non-deterministic (similar to the switch statement described above). That means that the decision to exit the loop or to continue the iteration is made randomly. In the BPE-calculus we represent a loop as follows:

$$getWeather*$$

where the activity $getWeather$ is performed an arbitrary number of times.

2.5 Flow

Flow is the construct that we are most interested in, since it defines concurrency in BPEL. The function of the flow construct is to list activities that are executed in parallel. In our notation it is represented with $\|$. As in, for example,

$$getWeather \parallel reserveTicket$$

where activities $getWeather$ and $reserveTicket$ are executed concurrently. The synchronization between activities in a flow is achieved using links. We examine them next.

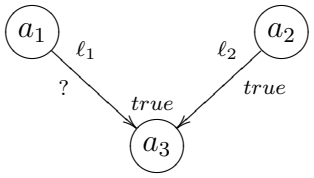
2.6 Links and Join Condition

In BPEL the links are used inside of a flow to provide synchronization between activities. Links have to be declared at the beginning of the flow. Each link can be used only once in its scope. In order to simplify the model we omit scoping and link declarations. We make the assumption that link names in a process are unique. We can enforce this by renaming duplicate links when translating a process from BPEL to the BPE-calculus.

Each link has a source and a target activity. When an activity is the source of a link we say that it has an outgoing link. Outgoing links are denoted with $\uparrow$ in the BPE-calculus. Every outgoing link has a transition condition associated with it. The transition condition is a boolean expression based on the process data. The default value is $true$. In our analysis we made the decision to disregard data.

Therefore we only consider three transition conditions. In the BPE-calculus, a transition condition is either $true$, $false$, or ?. The latter represents the transition condition that is either true or false. In this case, the choice (between $true$ or $false$) is nondeterministic. Therefore, we model the outgoing links as follows: $\uparrow \ell \ tc \ activity$, where tc can be either $true$, $false$, or ? and ℓ is a link name.

If an activity is the target of a link, we say it has an incoming link. Each such activity has a join condition associated with it. The join condition is a boolean expression formulated in terms of incoming links and boolean operators. The default join condition is a disjunction of all incoming links. The operators allowed in join conditions, according to [CGK+02], are: and, or, =, !=. The boolean constants $true$ and $false$ are also allowed. In the BPE-calculus we extend the list of boolean operators allowed in join conditions with negation operator $\neg$. The expressive power of join conditions is not affected by this addition, since negation can be expressed in BPEL as $expr = false$. Let us illustrate the behavior of the target activity by means of the following example:



The activity a_3 has two incoming links ℓ_1 and ℓ_2. The transition condition of ℓ_1 is non-deterministic and the transition condition of ℓ_2 is $true$. The activity a_3 has a join condition $true$. Initially all links are undefined. The target activity a_3 waits for all its incoming links to become defined before it can proceed. A link becomes defined once its source activity has completed and its transition condition has been evaluated. As a result of that, the link has been assigned a value $true$ or $false$. Once all of the incoming links have been defined, the join condition can be evaluated. The target activity can proceed only if its join condition evaluates to true.

In BPEL the join condition and the incoming links are specified separately. In cases where all of the incoming links are used in the join condition this is redundant. In order to simplify the model we will omit separate specification of the incoming links. Instead we will re-write the join conditions to contain all of the incoming links. For all incoming links of the activity, if some link ℓ is not used in the join condition c, we change the join condition to be $c \wedge (\ell \vee \neg \ell)$. For instance, in the above example the join condition will be changed to $true \wedge (\ell_1 \vee \neg \ell_1) \wedge (\ell_2 \vee \neg \ell_2)$. The behavior of the process remains the same. In our model the join condition

will not be evaluated unless all links used in it (all the incoming links) are defined. In the BPE-calculus we represent a join condition as follows:

$$c \Rightarrow activity$$

This process denotes that *activity* can be executed only if the join condition c evaluates to true. For example,

$$\uparrow \ell_1 \ ?\ a_1 \parallel \uparrow \ell_2\ true\ a_2 \parallel true \wedge (\ell_1 \vee \neg\ell_1) \wedge (\ell_2 \vee \neg\ell_2) \Rightarrow a_3$$

expresses the above example in the BPE-calculus.

2.7 Syntax

Before defining BPE-processes, we first fix

- a set $\mathbb{A}_e$ of external basic activities,
- a set of basic activities $\mathbb{A} = \mathbb{A}_e \cup \{\tau\}$,
- an infinite set $\mathbb{L}$ of links.

DEFINITION 1 The set $\mathbb{P}$ of BPE-processes is defined by

activity ::=	
act	(External Basic Activity)
$\mid \tau$	(Internal Basic Activity)
$\mid$ *end*	(Terminate)
$\mid$ *activity* ; *activity*	(Sequence)
$\mid$ *choice*	(Pick – external choice)
$\mid$ *activity* $\oplus$ *activity*	(Switch – internal choice)
$\mid$ *activity*$*$	(While)
$\mid$ *activity* $\parallel$ *activity*	(Flow)
$\mid\uparrow \ell$ *tc activity*	(Outgoing Link)
$\mid c \Rightarrow$ *activity*	(Join Condition)

$choice ::= \alpha\ ;\ activity \mid choice + choice$

$where\ act \in \mathbb{A}_e, \alpha \in \mathbb{A}, \ell \in \mathbb{L}, c \in \mathbb{C}, and\ tc \in \{true,\ false,\ ?\}$

The set $\mathbb{C}$ of join conditions is defined as:
$c ::= true \mid false \mid \ell \mid \neg c \mid c \wedge c \mid c \vee c \mid c = c \mid c \neq c$

It is evident that the resulting process algebra is considerably simpler and hence more manageable than BPEL. It captures all the features of the language related to control flow. To illustrate this point, consider the travel agent example presented in the introduction. It can be represented in the BPE-calculus as follows:

$$
\begin{array}{l}
receiveRequest; \\
(\uparrow ca\ true\ reserveAC \oplus \uparrow us\ ?\ reserveAA \oplus reserveBA \\
\parallel getWeatherForecast \\
\parallel ca \vee us \Rightarrow \uparrow r2a\ true\ rentCar \\
\parallel r2a \Rightarrow \tau); \\
replyRequest
\end{array}
$$

Note, that when translating BPEL process into the BPE-calculus we map any transition conditions that are expressed in terms of data to the non-deterministic condition ?. Transition conditions specified as boolean constants $true$ and $false$ are left the same. Refer to Section 2.6 for further details about transition conditions in the BPE-calculus.

In BPEL, each link should have a unique source and a unique target. We capture this restriction by means of the following simple type system. Only if a process satisfies this restriction, it can be typed. The type of a process a is a pair: the set of links that are used as incoming in a, and the set of links that are used as outgoing in a. One of the restrictions placed on links in BPEL is that they cannot cross boundaries of a while activity. Therefore the while loop can have only internal links. That is, activities inside of the while loop cannot be linked to any outside activities. Therefore the sets of incoming and outgoing links in the while loop have to be the same.

The notation used to describe rules in the type system is:

$$\frac{premises}{conclusion}$$

where if the premises hold then we may infer that the conclusion also holds. In the case when there are no premises we just state conclusion.

$$conclusion$$

DEFINITION 2 The relation $\Updownarrow \subseteq \mathbb{P} \times 2^{\mathbb{L}} \times 2^{\mathbb{L}}$ is defined by

(ACT) $\alpha \Updownarrow (\emptyset, \emptyset)$

$$\text{(END)}\quad end \Updownarrow (\emptyset, \emptyset)$$

$$\text{(OUT)}\quad \frac{a \Updownarrow (I, O) \quad \ell \notin O}{\uparrow \ell\ tc\ a \Updownarrow (I, O \cup \{\ell\})}$$

$$\text{(JOIN)}\quad \frac{a \Updownarrow (I, O) \quad links(c) \cap I = \emptyset}{c \Rightarrow a \Updownarrow (I \cup links(c), O)}$$

$$\text{(SEQ)}\quad \frac{a \Updownarrow (I_a, O_a) \quad b \Updownarrow (I_b, O_b) \quad I_a \cap I_b = \emptyset \quad O_a \cap O_b = \emptyset}{a\ ;\ b \Updownarrow (I_a \cup I_b, O_a \cup O_b)}$$

$$\text{(PICK)}\quad \frac{a \Updownarrow (I_a, O_a) \quad b \Updownarrow (I_b, O_b) \quad I_a \cap I_b = \emptyset \quad O_a \cap O_b = \emptyset}{a + b \Updownarrow (I_a \cup I_b, O_a \cup O_b)}$$

$$\text{(SWITCH)}\quad \frac{a \Updownarrow (I_a, O_a) \quad b \Updownarrow (I_b, O_b) \quad I_a \cap I_b = \emptyset \quad O_a \cap O_b = \emptyset}{a \oplus b \Updownarrow (I_a \cup I_b, O_a \cup O_b)}$$

$$\text{(FLOW)}\quad \frac{a \Updownarrow (I_a, O_a) \quad b \Updownarrow (I_b, O_b) \quad I_a \cap I_b = \emptyset \quad O_a \cap O_b = \emptyset}{a \parallel b \Updownarrow (I_a \cup I_b, O_a \cup O_b)}$$

$$\text{(WHILE)}\quad \frac{a \Updownarrow (I_a, O_a) \quad I_a = O_a}{a* \Updownarrow (I_a, O_a)}$$

In the above definition we use $links(c)$ to denote the set of links that occur in the join condition c. For the cases when the process is constructed by combining two subprocesses, we need to ensure that the resulting set of outgoing links and the resulting set of incoming links have only unique links. In order ensure that, we specify the condition that corresponding sets of links of subprocesses should not have any links in common ($I_a \cap I_b = \emptyset, \quad O_a \cap O_b = \emptyset$).

Not every process can be typed. For example, the process $\ell \uparrow true\ a \parallel \ell \uparrow true\ b$ cannot be typed. However, if a process is well-typed then its type is unique. That is,

PROPOSITION 1 *If $a \Updownarrow (I_1, O_1)$ and $a \Updownarrow (I_2, O_2)$ then $I_1 = I_2$ and $O_1 = O_2$.*

PROOF We proof this by induction.

- Consider $\alpha \Updownarrow (I_1, O_1)$ and $\alpha \Updownarrow (I_2, O_2)$. Then $I_1 = I_2 = \emptyset$ and $O_1 = O_2 = \emptyset$.
- Consider $\uparrow \ell\ tc\ a \Updownarrow (I_1, O_1)$ and $\uparrow \ell\ tc\ a \Updownarrow (I_2, O_2)$. According to Definition 2, $a \Updownarrow (I_1, O_1')$ for some O_1' such that $\ell \notin O_1'$ and $O_1 = O_1' \cup \{\ell\}$ and $a \Updownarrow (I_2, O_2')$ and for some O_2' such that $\ell \notin O_2'$ and $O_2 = O_2' \cup \{\ell\}$. By induction, $I_1 = I_2$ and $O_1' = O_2'$, and, hence $O_1 = O_1' \cup \{\ell\} = O_2' \cup \{\ell\} = O_2$.

- Consider $c \Rightarrow a \Updownarrow (I_1, O_1)$ and $c \Rightarrow a \Updownarrow (I_2, O_2)$. According to Definition 2, $a \Updownarrow (I'_1, O_1)$ for some I'_1 such that $I'_1 \cap links(c) = \emptyset$ and $I_1 = I'_1 \cup links(c)$ and $a \Updownarrow (I'_2, O_2)$ for some I'_2 such that $I'_2 \cap links(c) = \emptyset$ and $I_2 = I'_2 \cup links(c)$. By induction, $O_1 = O_2 = O$ and $I'_1 = I'_2$, and, hence, $I_1 = I'_1 \cup links(c) = I'_2 \cup links(c) = I_2$.
- Consider $a \,;\, b \Updownarrow (I_1, O_1)$ and $a \,;\, b \Updownarrow (I_2, O_2)$. According to Definition 2, $a \Updownarrow (I^a_1, O^a_1)$ and $b \Updownarrow (I^b_1, O^b_1)$, where $I_1 = I^a_1 \cup I^b_1$, $I^a_1 \cap I^b_1 = \emptyset$, $O_1 = O^a_1 \cup O^b_1$, $O^a_1 \cap O^b_1 = \emptyset$ and $a \Updownarrow (I^a_2, O^a_2)$ and $b \Updownarrow (I^b_2, O^b_2)$, where $I_2 = I^a_2 \cup I^b_2$, $I^a_2 \cap I^b_2 = \emptyset$, $O_2 = O^a_2 \cup O^b_2$, $O^a_2 \cap O^b_2 = \emptyset$. By induction, $I^a_1 = I^a_2$, $I^b_1 = I^b_2$, $O^a_1 = O^a_2$, and $O^b_1 = O^b_2$. Therefore, $I^a_1 \cup I^b_1 = I^a_2 \cup I^b_2$ and $O^a_1 \cup O^b_1 = O^a_2 \cup O^b_2$. It follows that $I_1 = I_2$ and $O_1 = O_2$.

Furthermore, each type is finite. That is,

PROPOSITION 2 *If $a \Updownarrow (I, O)$ then I and O are finite sets of links.*

PROOF We proof this by induction.

- Consider $\alpha \Updownarrow (I, O)$. By Definition 2, $I = O = \emptyset$. Therefore, I and O are finite sets.
- Consider $\uparrow \ell \; tc \; a \Updownarrow (I, O)$. By Definition 2, $a \Updownarrow (I, O \setminus \{\ell\})$. By induction, I and $O \setminus \{\ell\}$ are finite sets. Therefore, O is a finite set.
- Consider $c \Rightarrow a \Updownarrow (I, O)$. According to Definition 2, $a \Updownarrow (I \setminus links(c), O)$. By induction, O and $I \setminus links(c)$ are finite sets. Since $links(c)$ is a finite set of links used in the join condition, I is a finite set.
- Consider $a \,;\, b \Updownarrow (I, O)$. According to Definition 2, $a \Updownarrow (I_a, O_a)$ and $b \Updownarrow (I_b, O_b)$, where $I = I_a \cup I_b$ and $O = O_a \cup O_b$. By induction, I_a, O_a, I_b, and O_b are finite sets. Therefore, $I_a \cup I_b = I$ and $O_a \cup O_b = O$ are finite sets.

An activity a satisfies the restriction that each link should have a unique source and a unique target if $a \Updownarrow (I, O)$ for some I and O such that $I = O$. The condition $I = O$ captures that each link should have a source and a target. For example, in the activity $\uparrow \ell \; true \; \alpha$ which has type $(\emptyset, \{\ell\})$ the link ℓ has a source but no target. In the activity $\ell \Rightarrow \alpha$ which has type $(\{\ell\}, \emptyset)$ link ℓ has a target, but no source.

2.8 Conclusion

In this chapter we have introduced the BPE-calculus – a small and manageable language that models the behavior of BPEL. We will use this language to analyze BPEL processes. But before we can do that, we need to define the precise semantics of the BPE-calculus. We present it next.

Chapter 3

Labelled Transition System

3.1 Preliminaries

In order to describe precisely the semantics of the BPE-calculus we use the Structured Operational Semantics (SOS) approach introduced by Plotkin [Plo81]. This approach describes the semantics of a process in terms of the possible transitions the process can make. A transition is a step in the execution of a process:

$$P \xrightarrow{action} P'$$

This means that P makes a transition to P' by performing *action*. The SOS rules, defining the transitions, have the following format:

$$\frac{premises}{conclusion}\ (side\ condition)$$

where if the premises and the side condition hold then we may infer that the conclusion also holds. If the premises or the side condition are missing they are assumed to be true. Consider, for example, the simplified SOS rule for sequence:

$$\frac{a \xrightarrow{\alpha} a'}{a\,;b \xrightarrow{\alpha} a'\,;b}$$

It states that if a is capable of performing action α to become a', then the sequence $a\,;b$ is also capable of engaging in this action to become $a'\,;b$. The SOS approach describes the behavior of the process based on the syntactic structure of that process. We are going to describe the behavior of all the BPE-calculus constructs using this type of rules.

The SOS rules define a labelled transition system. A labelled transition system is a structure $\langle \Gamma, \mathbb{A}, \longrightarrow \rangle$, where Γ is a set of configurations, $\mathbb{A}$ is a set of basic

activities, and $\longrightarrow \subseteq \Gamma \times \mathbb{A} \times \Gamma$ is the transition relation. We use $\alpha \in \mathbb{A}$ to denote a basic activity.

In the execution of a BPE-process, the state of the process depends on the structure of the process and also on the values of the links used in the process. The value of a link can be $true$, $false$, or undefined. We denote the last one as $\perp$. We introduce a set of link statuses $\Sigma = \mathbb{L} \to \{true, false, \perp\}$, where $\mathbb{L}$ is the set of links.

We introduce a nil process denoted $\emptyset$ which is not capable of any transitions. We will use it to model successful termination.

We define the set of configurations (or states) $\Gamma = \mathbb{P} \times \Sigma$, where $\mathbb{P}$ is the set of BPE processes (including $\emptyset$), and Σ is the set of link statuses. We denote $\sigma \in \Sigma$, a and $b \in \mathbb{P}$, $\ell \in \mathbb{L}$, and $\langle a, \sigma \rangle \in \Gamma$. Initially all links are undefined.

In the definition of labelled transition system we use the notation $\sigma[v/_L]$ to denote the substitution defined by $\sigma[v/_L](\ell) = v$ if $\ell \in L$ and $\sigma(\ell)$ otherwise. Instead of $\sigma[v/_{\{\ell\}}]$ we often write $\sigma[v/_\ell]$.

Next we describe the labelled transition relation $\longrightarrow \subseteq (\mathbb{P} \times \Sigma) \times \mathbb{A} \times (\mathbb{P} \times \Sigma)$ using SOS-style rules.

3.2 Basic Activity

The process that consists of a basic activity is capable of making just one transition. This transition models the execution of the activity. After the basic activity has been executed the process can no longer make any transitions. In other words, it becomes the nil process. This is expressed by the following rule.

$$(Act) \qquad \langle \alpha, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma \rangle$$

3.3 Sequence

In the process composed of a sequence of two activities, the second activity can start executing only after the first one has finished. The process of the form $a\,;b$ can proceed only if the first activity, a, can make some transition. If as a result of this transition activity a completes (becomes the nil process), then activity b can start executing. If not, then process $a\,;b$ is transformed into the process $a'\,;b$ where a' is an activity resulting from a executing action α.

$$(Seq_1) \qquad \frac{\langle a, \sigma \rangle \xrightarrow{\alpha} \langle a', \sigma' \rangle}{\langle a\,;b, \sigma \rangle \xrightarrow{\alpha} \langle a'\,;b, \sigma' \rangle}(a' \neq \emptyset)$$

$$(Seq_2) \qquad \frac{\langle a, \sigma\rangle \xrightarrow{\alpha} \langle \emptyset, \sigma'\rangle}{\langle a\,;b, \sigma\rangle \xrightarrow{\alpha} \langle b, \sigma'\rangle}$$

The side condition ($a' \neq \emptyset$) is required in order to ensure that if a becomes the nil process as a result of executing α only the Seq_2 rule is used. That is, the transition $\langle a\,;b, \sigma\rangle \xrightarrow{\alpha} \langle \emptyset\,;b, \sigma'\rangle$ is not possible. Note also, that transition α may lead to a change in the link values, and therefore σ' denotes the new link status.

3.4 Switch

The switch activity performs a non-deterministic choice. The environment has no influence on this choice. The process $a \oplus b$ can make a transition to either a or b. The choice is made internally. Therefore, we label this transition with τ. The activity that is not chosen is simply discarded. Recall BPEL's garbage collecting mechanism, dead path elimination (DPE), described in Chapter 1. This mechanism sets all of the outgoing links of a discarded activity to false. This is done in order to eliminate execution paths that will never be taken. Therefore, after the choice has been made the outgoing links of a discarded activity are set to false.

$$(Switch_1) \qquad \frac{b \Updownarrow (I_b, O_b)}{\langle a \oplus b, \sigma\rangle \xrightarrow{\tau} \langle a, \sigma[^{false}/_{O_b}]\rangle}$$

$$(Switch_2) \qquad \frac{a \Updownarrow (I_a, O_a)}{\langle a \oplus b, \sigma\rangle \xrightarrow{\tau} \langle b, \sigma[^{false}/_{O_a}]\rangle}$$

3.5 Pick

Pick is an external choice construct. This means that the choice of the activity is dictated by the environment. The activity that is capable of engaging in some action is chosen. If both activities are equally capable of making progress then the choice is made randomly. As a result of a pick, one activity is chosen, the other one is discarded. DPE is then triggered and sets all the outgoing links of the discarded activity to false.

$$(Pick_1) \qquad \frac{\langle a, \sigma\rangle \xrightarrow{\alpha} \langle a', \sigma'\rangle, \quad b \Updownarrow (I_b, O_b)}{\langle a + b, \sigma\rangle \xrightarrow{\alpha} \langle a', \sigma'[^{false}/_{O_b}]\rangle}$$

$$(Pick_2) \qquad \frac{\langle b, \sigma\rangle \xrightarrow{\alpha} \langle b', \sigma'\rangle, \quad a \Updownarrow (I_a, O_a)}{\langle a + b, \sigma\rangle \xrightarrow{\alpha} \langle b', \sigma'[^{false}/_{O_a}]\rangle}$$

Note, that execution of an action α can change σ to σ'. DPE is then applied to a changed link assignment σ'.

The following rule defines the behavior of each individual clause of the pick construct. This rule is a special case of Seq_2 rule.

$$(Pick_3) \qquad \langle \alpha\,;\, a, \sigma\rangle \xrightarrow{\alpha} \langle a, \sigma\rangle$$

3.6 Flow

The flow construct is analogous to the parallel composition in calculi like, for example, CCS [Mil89], and has a very similar set of rules. The activities a and b are executed in parallel.

$$(Flow_1) \qquad \frac{\langle a, \sigma\rangle \xrightarrow{\alpha} \langle a', \sigma'\rangle}{\langle a \parallel b, \sigma\rangle \xrightarrow{\alpha} \langle a' \parallel b, \sigma'\rangle}(a' \neq \emptyset)$$

$$(Flow_2) \qquad \frac{\langle b, \sigma\rangle \xrightarrow{\alpha} \langle b', \sigma'\rangle}{\langle a \parallel b, \sigma\rangle \xrightarrow{\alpha} \langle a \parallel b', \sigma'\rangle}(b' \neq \emptyset)$$

$$(Flow_3) \qquad \frac{\langle a, \sigma\rangle \xrightarrow{\alpha} \langle \emptyset, \sigma'\rangle}{\langle a \parallel b, \sigma\rangle \xrightarrow{\alpha} \langle b, \sigma'\rangle}$$

$$(Flow_4) \qquad \frac{\langle b, \sigma\rangle \xrightarrow{\alpha} \langle \emptyset, \sigma'\rangle}{\langle a \parallel b, \sigma\rangle \xrightarrow{\alpha} \langle a, \sigma'\rangle}$$

3.7 While

In the while loop, $a*$, activity a can be repeated an arbitrary number of times. After every iteration there is a non-deterministic choice to either stop or go on for another iteration. This choice is an internal step, and therefore the transition is labelled with τ.

$$(While_1) \qquad \frac{a \Updownarrow (I_a, O_a)}{\langle a*, \sigma\rangle \xrightarrow{\tau} \langle a; (a*), \sigma[^{\perp}/_{O_a}]\rangle}$$

$$(While_2) \qquad \langle a*, \sigma \rangle \xrightarrow{\tau} \langle \emptyset, \sigma \rangle$$

Recall that there is a restriction placed on linking in while loops. If a process is well-typed there are no incoming or outgoing links that cross the boundary of a while loop. Since there cannot be any outgoing links from the loop, DPE does not have to be triggered. The loop can have a number of internal links. Before entering the loop all of these links are undefined. During the execution of the body of the loop the link values will change. In order for each iteration to use links properly we need to reset them back to $\perp$ after each iteration of the loop. Recall also that the type of a while loop is (I_a, O_a) where $I_a = O_a$, therefore setting O_a to $\perp$ sets all of the links used inside of a while loop to $\perp$.

3.8 Outgoing link

The outgoing link of an activity is represented by $\uparrow \ell \; tc \; a$, where tc is a transition condition that can be $true$, $false$ or ?. The link becomes activated only when the source activity a completes. At this point, the link status of link ℓ changes from undefined to either $true$ or $false$. If the transition condition is $true$ or $false$ then the link status is set to the value of the transition condition. If the transition condition is ? then the link status is chosen non-deterministically. This reflects the fact that the transition condition depends on data values (which we ignore) and can evaluate to $true$ in some cases and to $false$ in other cases.

$$(OutLink_1) \qquad \frac{\langle a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma' \rangle}{\langle \uparrow \ell \; true \; a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma'[^{true}/_{\ell}] \rangle}$$

$$(OutLink_2) \qquad \frac{\langle a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma' \rangle}{\langle \uparrow \ell \; false \; a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma'[^{false}/_{\ell}] \rangle}$$

$$(OutLink_3) \qquad \frac{\langle a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma' \rangle}{\langle \uparrow \ell \; ? \; a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma'[^{true}/_{\ell}] \rangle}$$

$$(OutLink_4) \qquad \frac{\langle a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma' \rangle}{\langle \uparrow \ell \; ? \; a, \sigma \rangle \xrightarrow{\alpha} \langle \emptyset, \sigma'[^{false}/_{\ell}] \rangle}$$

An additional rule for the outgoing link specifies that the source activity can make any number of steps before it completes.

$$(OutLink_5) \qquad \frac{\langle a, \sigma \rangle \xrightarrow{\alpha} \langle a', \sigma' \rangle}{\langle \uparrow \ell \; tc \; a, \sigma \rangle \xrightarrow{\alpha} \langle \uparrow \ell \; tc \; a', \sigma' \rangle} (a' \neq \emptyset),$$

where $tc \in \{true,\ false,\ ?\}$.

3.9 Join Condition and Incoming Link

In BPEL the join condition can only be evaluated when all the incoming links of the target activity are defined. Note that in the BPE-calculus all of the incoming links are included in the join condition. Therefore the join condition can be evaluated only after all of its links are defined. We will consider the join condition evaluated only if its value is true or false. Recall that the join conditions are defined as:

$c ::= true \mid false \mid \ell \mid \neg c \mid c \wedge c \mid c \vee c \mid c = c \mid c \neq c$

To model the evaluation of join conditions, we introduce a function $\mathcal{C} : \mathbb{C} \to \Sigma \to \{true, false, \bot\}$. We only give the interpretation of the operators $\neg$, $\wedge$ and $\vee$. These are defined as

$\vee$	$true$	$false$	$\bot$
$true$	$true$	$true$	$\bot$
$false$	$true$	$false$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\wedge$	$true$	$false$	$\bot$
$true$	$true$	$false$	$\bot$
$false$	$false$	$false$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\neg$	$true$	$false$	$\bot$
	$false$	$true$	$\bot$

Note, that if at least one of the links in the join condition is undefined then the join condition is also undefined. This models the behaviour of BPEL, since in BPEL the target activity waits for all the incoming links to become defined (either $true$ or $false$) before evaluating the join condition.

In case that the join condition evaluates to true the target activity will be executed. In case that the join condition is false, the activity is skipped. In addition to skipping the activity, DPE is triggered in that case. It sets all the outgoing links of this activity to false. The τ-transition is used in both cases, since the evaluation of a join condition and the execution of DPE are both internal actions.

$$(Join_1) \qquad \frac{\mathcal{C}(c)(\sigma) = true}{\langle c \Rightarrow a, \sigma\rangle \xrightarrow{\tau} \langle a, \sigma\rangle}$$

$$(Join_2) \qquad \frac{\mathcal{C}(c)(\sigma) = false, \quad a \Updownarrow (I_a, O_a)}{\langle c \Rightarrow a, \sigma\rangle \xrightarrow{\tau} \langle \emptyset, \sigma[^{false}/_{O_a}]\rangle}$$

3.10 Termination

The terminate activity, denoted *end* in the BPE-calculus, stops the whole process. When this activity is encountered all branches of the execution are abandoned and the process stops. Below, we introduce a set of rules to model this behavior. Note, that the activity *end* by itself is incapable of performing any transitions.

$(Seq_3) \qquad \langle end\,;a,\sigma\rangle \xrightarrow{\tau} \langle end,\sigma\rangle$

$(Flow_5) \qquad \langle end \parallel a,\sigma\rangle \xrightarrow{\tau} \langle end,\sigma\rangle$

$(Flow_6) \qquad \langle a \parallel end,\sigma\rangle \xrightarrow{\tau} \langle end,\sigma\rangle$

$(While_3) \qquad \langle end*,\sigma\rangle \xrightarrow{\tau} \langle end,\sigma\rangle$

$(OutLink_6) \qquad \langle \uparrow \ell\ tc\ end,\sigma\rangle \xrightarrow{\tau} \langle end,\sigma\rangle,\ where\ tc\ \in\ \{true, false, ?\}$

3.11 Conclusion

We have defined the behavior of all the BPE-calculus constructs. We can now simulate the execution of a process according to the above rules. Consider this sample process:

$$\uparrow \ell_1\ true\ get \oplus \uparrow \ell_2\ ?\ put \parallel \ell_1 \vee \ell_2 \Rightarrow print$$

It has a switch construct in parallel with a join activity. Activities *get*, *put* and *print* are basic activities. Basic activities *get* and *put* both have outgoing links. Activity *get* has the outgoing link ℓ_1 with the default transition condition *true*. Activity *put* has the outgoing link ℓ_2 with the non-deterministic transition condition ?. Activity *print* is the target of those links. Recall, that when one of the clauses of a switch construct is chosen and all others are discarded the garbage collection mechanism DPE is triggered. DPE assigns $false$ to all the outgoing links of the discarded activities. Therefore, if *put* activity is chosen then *get* is discarded and its outgoing link ℓ_1 is set to $false$. If activity *get* is chosen then ℓ_2 is set to false. Note that initially all links are undefined. We denote the initial link status as $\sigma_\perp$. There are several possible execution paths for this process. We show one of them.

$$\langle \uparrow \ell_1\ true\ get \oplus \uparrow \ell_2\ ?\ put \parallel \ell_1 \vee \ell_2 \Rightarrow print, \sigma_\perp\rangle$$

$$\xrightarrow{\tau} \langle \uparrow \ell_2 \ ? \ put \parallel \ell_1 \vee \ell_2 \Rightarrow print, \sigma_\perp[^{false}/_{\ell_1}] \rangle \qquad (Switch_2), (Flow_1)$$

$$\xrightarrow{put} \langle \ell_1 \vee \ell_2 \Rightarrow print, \sigma_\perp[^{false}/_{\ell_1}][^{true}/_{\ell_2}] \rangle \qquad (Act), (OutLink_3), (Flow_3)$$

$$\xrightarrow{\tau} \langle print, \sigma_\perp[^{false}/_{\ell_1}][^{true}/_{\ell_2}] \rangle \qquad (Join_1)$$

$$\xrightarrow{print} \langle \emptyset, \sigma_\perp[^{false}/_{\ell_1}][^{true}/_{\ell_2}] \rangle \qquad (Act)$$

The final state is $\langle \emptyset, \sigma_\perp[^{false}/_{\ell_1}][^{true}/_{\ell_2}] \rangle$. It is not capable of any transitions.

In this section we have modelled the behavior of BPEL using a labelled transition system. Therefore, the BPE-calculus is defined both syntactically and semantically. Now we are ready to analyze BPEL-processes.

Chapter 4

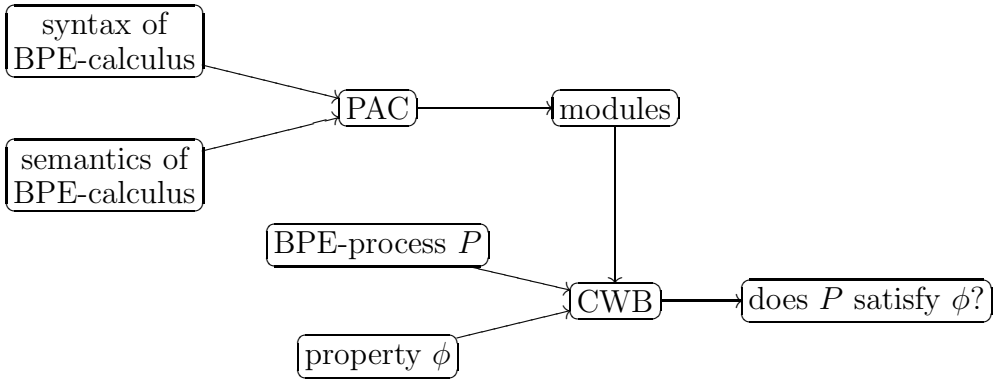
Process Algebra Compiler

In order to analyze BPE-calculus processes we are going to use a tool called the Concurrency Workbench of New Century (CWB) [CS96]. This tool provides automatic verification of finite-state concurrent systems specified using a process algebra such as CCS. The CWB supports several verification methods. It allows to verify the properties of the system specified using μ-calculus formulas. It also supports different types of behavioral equivalences and preorders between systems. A key feature of the CWB is its flexibility. Due to its modular design it can be extended to support another specification language. Extending the CWB to support the BPE-calculus will allow us to use various verification techniques to analyze BPE-calculus processes.

Implementing a module for supporting a new language for the CWB would be a time consuming task if not for the Process Algebra Compiler (PAC) [CS02]. The PAC is a tool designed to generate front-ends for verification tools, in particular the CWB. Given the syntax and the semantics of the language, the PAC produces routines for the language support. These routines include parsers, unparsers, and semantic routines. The generated code can then be incorporated in the CWB to allow verification of processes specified in that language.

We use the PAC in order to generate a BPE-calculus front-end for the CWB. The PAC takes as its input two files. One file contains the description of the syntax of the language, the other file describes the semantics of the language. The syntax is specified using a yacc-like grammar. The semantics is specified in the form of SOS rules presented in the previous section. Running the PAC generates source code in Standard ML. This generated code along with a few user-defined routines is incorporated into the CWB. The resulting version of the CWB can verify BPE-process specifications. The following picture shows the

overall architecture.



The above picture only shows model checking. Note, however, that our tool also supports equivalence checking and preorder checking.

Below, we will describe the two files that the PAC will use to generate a front-end for the CWB to adapt it to the BPE-calculus.

4.1 Syntax Description File

The syntax description file contains the definition of the syntax of the BPE-calculus. The file is logically divided into three parts. The first part presents the abstract syntax(structural essence of a language), the second describes the concrete syntax (specific strings used in the language), and the third contains some additional syntactic constructs that are used to specify the rules defining the semantics.

4.1.1 Abstract Syntax

The description of the abstract syntax consists of a `sorts` section, a `cons` section and a `funs` section. The `sorts` section lists all the types of entities of the language. For the BPE-calculus, these are

```
sorts
  id, link, act, activity, agent, boolean, join,
  new_bool, status, env, automaton, state, trans, int
```

The sort `activity` corresponds to the nonterminal *activity* in Definition 1. The sorts `act`, `link`, and `join` correspond to the sets $\mathbb{A}$, $\mathbb{L}$, and $\mathbb{C}$. We also

declare the sort `new_bool` that extends the Boolean type with $\perp$. This sort will be used to represent the value of a link. Status (`status`) of a link is a pair of a link name and its `new_bool` value. The sort `env` corresponds to the set Σ that keeps track of link statuses. We also introduce a sort `agent` which represents the set of configurations $\Gamma = \mathbb{P} \times \Sigma$. These configurations were denoted as $\langle a, \sigma \rangle$ in the previous chapter. The rest of the sorts are `id`, `boolean`, `automaton`, `state`, `trans`, and `int`. The first two of those are simple auxiliary types. The rest are required by the PAC for any language definition in order to define automata for that language. We refer the reader to [Sim99] for more details on the automata definition.

The next section of the file defines constructors for the sorts introduced. Constructor signatures are specified in the following format:

$$\text{Name:} \quad \text{domain} \rightarrow \text{codomain}$$

Based on this description the PAC will generate the actual source code for each constructor. Here are the constructors for the activities:

```
cons
  Nil:          unit -> activity
  End:          unit -> activity
  Action:       act -> activity
  Flow:         activity * activity -> activity
  Sequence:     activity * activity -> activity
  Pick:         activity * activity -> activity
  Switch:       activity * activity -> activity
  While:        activity -> activity
  Outgoing:     link * boolean * activity -> activity
  OutgoingVar:  link * activity -> activity
  Join:         join * activity -> activity
```

This definition corresponds closely to Definition 1. There is a constructor defined for each type of activity (flow, sequence, switch, etc.). For example, the `Nil` constructor takes no argument, represented by the Standard ML type `unit`, and produces an activity. The `Flow` constructor takes two activities and produces another activity. There are two constructors for an outgoing link activity: one for the outgoing link with boolean transition condition (`Outgoing`), another with the transition condition ? (`OutgoingVar`).

In order to conveniently define the pick construct in the PAC we represent it simply as $activity + activity$. Note, that this is a more general definition than

the one given in Definition 1. The resulting version of CWB will handle the pick construct correctly as well as allow a more general definition of choice.

The constructors for join are as follows:

```
SimpleJoin:       link -> join
BoolJoin:         boolean -> join
NotJoin:          join -> join
And:              join * join -> join
Or:               join * join -> join
Equal:            join * join -> join
NotEqual:         join * join -> join
```

They are the same as specified in Definition 1.

We also specify constructors for the `new_bool` sort.

```
True:         unit -> new_bool
False:        unit -> new_bool
Undef:        unit -> new_bool
```

This sort consists of three constants: true, false, and undef. Each of the constructors takes no parameters and returns a `new_bool`.

As was mentioned above `status` is a pair of a link and its `new_bool` value. The constructors for link statuses are specified as follows:

```
Status:       link * new_bool -> status
Env:          (status list) -> env
```

We represent `env` as a list of link statuses. We will keep track of the link statuses during the execution of the process by updating this list.

Constructors for some simple sorts are not defined in the `cons` section. Instead, they are listed in the `funs` section. Constructors in this section are not generated by the PAC and have to be written by the user. This is done in order to give the user an opportunity to program simple types efficiently or to re-use existing code.

```
funs
  % constructors
  link_parse:       id -> link
  id_parse:         string -> id
  boolean_parse:    string -> boolean
```

```
act_parse:        id -> act
tau:              unit -> act
delta:            unit -> act
gamma:            unit -> act
```

We provide constructors for such simple sorts as `link`, `id`, `boolean`, and `act`. We introduce a couple of actions *delta* and *gamma*. These actions is going to be used for transitions involving the *end* activity and *nil* activity. We need this new actions for verification purposes, in order to distinguish between normal termination and termination due to the *end* activity. Note that the `act` sort has four constructors: one for external basic activities (`act_parse`), one for internal basic activities (`tau`), one for *delta* (`delta`), and one for *gamma* (`gamma`).

The user of the PAC should also provide some functions for the semantics description. We include such functions as join conditions evaluation, dead-path elimination, and link environment functions. The `funs` section specifies the signatures for all the user-defined functions.

```
% environment functions
insert:       env * link * new_bool -> env
defined:      env * join -> bool
eval_join:    env * join * new_bool -> bool
trans_cond:   boolean -> new_bool
dpe:          env * activity -> env
reset:        env * activity -> env

% misc
not_nil:  activity -> bool
not_end:  activity -> bool
```

We provide Standard ML code for these routines. The function `insert` inserts a link with its `new_bool` value into the environment. The function `defined` is used for the join condition evaluation. It checks if all of the incoming links of the given activity are defined. The `eval_join` function takes care of the join condition evaluation. It takes three parameters: an environment, a join condition and a `new_bool` value. It then evaluates the join condition and returns *true* if it is equal to the given `new_bool` value. It returns $false$ otherwise. The function `trans_cond` is used to translate a boolean transition condition value into a `new_bool` value of a link. The function called `dpe` implements the dead-path-elimination. It assigns $false$ to all the outgoing links of the given activity. The function `reset` is used in

the while loop to reset the values of the links after each iteration. The functions `not_nil` and `not_end` check whether the given activity is not a nil activity and not an end activity (respectively). We will describe some of these functions in more detail later.

4.1.2 Concrete Syntax

The next part of the syntax description file specifies the concrete syntax of the language using a yacc-like grammar. This description is used by the PAC to generate parsers for the BPE-calculus. This part of the file consists of a `tokens` section, an optional `priorities` section, a `nonterminals` section, and a `grammar` section. Here are some of the tokens used by the BPE-calculus:

```
tokens
   "t"    => TAU
   "nil"  => NIL
   "end"  => END
   "\;"   => SEMICOLON
   "\?"   => QMARK
   "\=\>" => ARROW
   "\|\|" => PAR
   "out"  => OUT
```

They are used in the description of the grammar.

The `priorities` section allows to specify associativity and priority values for tokens. For example,

```
priorities
     left 1 PAR
     left 3 SEMICOLON
```

means that ";" has a higher priority than "||". Therefore, the expression $(a\,;b) \parallel c$ can be written without parentheses as $a\,;b \parallel c$.

The `nonterminals` section declares every nonterminal used in the grammar description. The declaration is of the form:

name `of` *sort*

where the *name* is the name of the nonterminal and the *sort* is its type which is one of the sorts declared in the `sorts` section. For example,

```
nonterminals
   activity of activity
   link of link
   join of join
   act of act
```

The `grammar` section describes the concrete syntax for the BPE-calculus. It uses tokens and nonterminals declared previously. For example, this is a part of the syntax description for the activities:

```
grammar
   activity:
     NIL                        (Nil())
     | END                      (End())
     | act                      (Action (act))
     | activity PAR activity    (Flow(activity1, activity2))
     | activity PLUS activity   (Pick (activity1, activity2))
     | join ARROW activity      (Join (join, activity))
     | activity STAR            (While(activity))
```

Every production of the grammar is associated with a corresponding constructor defined in the `cons` or `funs` section. For example, two activities with a "$\|$" in between correspond to the `Flow` constructor.

The BPE-syntax defined for the PAC is for the most part the same as defined in Definition 1. There is one important difference. In the definition of the BPE-calculus the link statuses (σ) and, hence, the state ($\langle activity, \sigma \rangle$) were semantic entities. We need to keep track of the link statuses during the execution of the BPE-calculus processes. Unfortunately, due to the limitation of the PAC we cannot accomplish that with σ being a semantic entity. This limitation originates from the manner in which the CWB calculates the possible transitions for the process. We can easily overcome this problem if we make σ and, consequently, $\langle activity, \sigma \rangle$ syntactic entities. As mentioned before, we call σ an environment and $\langle activity, \sigma \rangle$ an agent. We introduce the syntactic representation for the agent:

$$env : activity$$

where *env* is a list of statuses separated by commas. The list is enclosed in square brackets. The concrete syntax for `agent`, `env`, and `status` is defined below.

```
agent: env COLON activity                    (Agent(env,activity))
env: status_list                             (Env(status_list))
status: LPAREN link COMMA new_bool RPAREN (Status(link,
                                                    new_bool))
```

where status_list is defined as follows:

```
lists
   status_list is empty_list LSQUARE COMMA RSQUARE of status
```

4.1.3 Rules Syntax

After defining the syntactic structure of the language we introduce some additional syntax for the SOS rules in the RULES SYNTAX section. We describe the syntax of those constructs that are used in the BPE-calculus semantics description but that are not part of the language. This includes semantic functions such as dead-path-elimination (dpe) and join condition evaluation (eval_join). We also describe the syntax of the labelled transition relation. The RULES SYNTAX section of the file has a format similar to the concrete syntax description. It also has tokens and grammar sections. Here is a fragment of the RULES SYNTAX section.

```
RULES SYNTAX

tokens
  "-"       => DASH
  "->"      => SHORTARROW
  "dpe"     => DPE

grammar
  bool: env COLON join EQUALS new_bool      (eval_join(env,
                                                join, new_bool))

  env: DPE LPAREN env COMMA activity RPAREN (dpe(env, activity))

  relation: agent DASH act SHORTARROW agent (transition(agent1,
                                                    act, agent2))
```

This description is used by the PAC to parse the semantics description file. According to the above fragment, when the PAC encounters a statement of the

format `e:j = b`, where `e` is an environment, `j` is a join condition and `b` is a new_bool value, the PAC will invoke the user-defined function `eval_join`. The syntax for the dead-path-elimination function is simply `dpe(e, a)` where `e` is an environment and `a` is an activity. To denote the transition relation we use the labelled arrow, as defined in Chapter 3, `a - opr -> b`, where `a` and `b` are agents and `opr` is a basic activity.

4.2 Semantics Description File

The semantics of the language is defined in the PAC by means of SOS rules similar to those introduced in Chapter 3. Their format is very similar except that in the PAC the side conditions are written on the same line as the premises. We use the syntax introduced in the syntax description file to re-write the SOS rules introduced in Chapter 3. First, all the variables used in the file are defined:

```
RULE_SET transition

vars
    a, b, a', b': activity
    l: link
    opr: act
    e, e': env
    j: join
    c: boolean
```

The type of a variable is one of the sorts defined in syntax description file.

The following is a rule for the basic activity. It corresponds to the *Act* rule in Chapter 3.

```
rules

act_rule

----------------------------------------
e: opr - opr -> e: nil
```

One of the properties we would like to be able to check is deadlock-freedom. The activity is in a deadlocked state if it is incapable of any transitions. According to the semantics of the BPE-calculus described in Chapter 3, when the

process terminates it is incapable of any transitions. Therefore, in order to check deadlock-freedom it is necessary to distinguish between termination and deadlock. There are a couple of solutions to this problem. First we discuss the approach that we have chosen.

In order to distinguish between a terminated state and a deadlocked state we introduce additional rules for activities in the terminated state. We add a transition from the terminated state to itself. Introduction of this loop ensures that a terminated state always has a transition leaving it. Therefore, it is semantically different from deadlock. There are two possibilities: the activity can successfully complete, becoming the *nil* activity, or it can be terminated with the *end* activity, becoming the *end* activity. We introduce rules for each case. In order to distinguish termination from deadlock it would be enough to add a τ-transition from *nil* to *nil* and from *end* to *end*. But we also might want to distinguish if the process completes successfully or is terminated using an *end* activity. In order to make this verification possible, we introduce special transitions *delta* and *gamma*. Transition *delta* signals that the process was terminated using the *end* activity. Transition *gamma* signals that the process has completed successfully. Here are the two rules:

```
nil_rule

--------------------------------
    e: nil - gamma -> e: nil

end_rule

--------------------------------
    e: end - delta -> e: end
```

This ensures that if an activity terminates then it is not considered a deadlock, since it is capable to making either a *gamma*-transition or a *delta*-transition. It also provides us with the capability to distinguish between successful completion and termination using the *end* activity.

There is an alternative solution. Instead of changing the original semantics of the termination by adding a loop, we can introduce an additional activity `stop`. It will represent the terminated state. This state will have no transitions leaving it, therefore being semantically the same as the deadlocked state. The difference between the two will be that before ending up in the `stop` state a process has to perform either a *delta* or a *gamma* transition. Based on this difference we

can distinguish between the state right before the terminated state and the state right before the deadlocked state. These states are different because in order to terminate successfully the process executes either a *delta* or a *gamma* and then has no transitions. If the process deadlocks then it performs some other action and then has no transitions. Using this approach we can also express deadlock-freedom. There is one major drawback in this approach: we cannot identify a deadlocked state since it is semantically the same as terminated state. Therefore if we want to find where exactly deadlock happens in the process, we can only find the state before the deadlocked state. This is an unpleasant limitation.

Both approaches presented introduce a slight change to the BPE-calculus syntax and semantics. We choose the approach of introducing a loop in a terminated state, since it allows to directly identify a deadlocked state if needed. It is also, in our opinion, a more natural way to approach the problem.

The rules for the flow look as follows:

```
flow_1
    e: a - opr ->  e': a', not_nil(a'), not_end(a)
-------------------------------------------------------
    e: a || b - opr -> e': a' || b

flow_2
     e: b - opr -> e': b', not_nil(b'), not_end(b)
-------------------------------------------------------
     e: a || b - opr -> e': a || b'

flow_3
    e: b - opr -> e': nil
---------------------------------
    e: a || b - opr -> e': a

flow_4
    e: a - opr -> e': nil
---------------------------------
    e: a || b - opr -> e': b

flow_5

---------------------------------
```

```
    e: end || a - t -> e: end

flow_6

-----------------------------------
    e: a || end - t -> e: end
```

The rules `flow_1` and `flow_2` correspond to the rules $Flow_1$ and $Flow_2$. The side condition ($a' \neq \emptyset$) is expressed using the user-defined function `not_nil`. There is also an additional side condition ($a \neq end$) which is expressed by the function `not_end`. This condition became needed since we added the `end_rule` which states that the end activity can make a *delta*-transition. Therefore, this side condition is necessary in order to disallow the transition $end \parallel a \xrightarrow{delta} end \parallel a$ when the `flow_5` rule should be applied.

Note, that the `not_nil` function is applied to `a'` and the `not_end` function is applied to `a`. The reason is that in the first case we would like to eliminate the scenario when `a` becomes `nil` as a result of a transition. In case of end we want to disallow use of the `flow_1` rule whenever `a` is itself the end activity.

The code for the `not_nil` and `not_end` functions is written in Standard ML. It uses some of the PAC-generated functions. The `not_nil` function looks as follows:

```
 fun not_nil (a) = not(is_Nil(a))
```

The function `is_Nil` is one of the PAC-generated functions. The rest of the flow rules are exactly as they are defined in Chapter 3.

The rules for sequence, switch, pick, and while are as they are defined in Chapter 3.

Next we look at the rules for the outgoing links. Recall that in the syntax description we have defined two kinds of outgoing links. One with the boolean transition condition and another with the transition condition ?. We specify a separate set of rules for each type of outgoing link.

```
out_link_1
    e: a - opr -> e': nil
-----------------------------------------------------
    e: out l c a - opr -> insert (e', l, trans_cond(c)): nil

out_link_2
```

```
    e: a - opr -> e': a', not_nil(a'), not_end(a)
-----------------------------------------------------
    e: out l c a - opr -> e': out l c a'

out_link_3

-----------------------------------------------------
    e: out l c end - t -> e: end

out_link_var_1
    e: a - opr -> e': nil
-----------------------------------------------------
    e: out l ? a - opr -> insert (e', l, tt): nil

out_link_var_2
    e: a - opr -> e': nil
-----------------------------------------------------
    e: out l ? a - opr -> insert (e', l, ff): nil

out_link_var_3
    e: a - opr -> e': a', not_nil(a'), not_end(a)
-----------------------------------------------------
    e: out l ? a - opr -> e': out l ? a'

out_link_var_4

-----------------------------------------------------
    e: out l ? end - t -> e: end
```

The rule `out_link_1` represents the rules $OutLink_1$ and $OutLink_2$ defined in Chapter 3. The rules `out_link_var_1`, and `out_link_var_2` correspond to the rules $OutLink_3$, and $OutLink_4$ respectively. The rules `out_link_2` and `out_link_var_3` correspond to the rule $OutLink_5$. The other two rules correspond to the rule for termination $OutLink_6$. The user-defined function `trans_cond` used in the rules translates a boolean transition condition into a `new_bool` value. Another user-defined function used in this set of rules is `insert`. It is used to insert the new link and value pair into the environment. The environment is represented as a list of such pairs. All the link names in the list have to be unique. Therefore,

if the link to be inserted is already in the list then its value should be updated to a new one. Here is the code for the recursive implementation of the `insert` function:

```
insert (e, l, b) =
  let fun insert_list([], l, b) = Status(l, b)::[]
  | fun insert_list (hd::tl, l, b) =
    let val (temp, _) = Status_inv(hd)
    in
     if BpecUserDecls.link_eq(temp, l) then Status(temp, b)::tl
     else hd::insert_list(tl, l, b)
    end
  in
    Env(insert_list(Env_inv(e), l, b))
  end
```

The `insert` function uses a helper function `insert_list` that finds the given link in the list and replaces its value. If the link is not found then it is added to the end of the list.

Initially the environment is empty. Every insert operation either adds a new link or updates an existing one. Here is an example of the execution path of an activity with outgoing links:

```
"[]:out l1 true a || out l2 ? b || (l1 and l2) => c"  - a ->
"[(l1, tt)]: out l2 ? b || (l1 and l2) => c" - b ->
"[(l1, tt),(l2, ff)]: (l1 and l2) => c" - t ->
"[(l1, tt),(l2, ff)]: nil"
```

Another interesting set of rules to look at are the rules for the join condition:

```
join_1
    defined(e, j), e: j = tt
-----------------------------------
    e: j => a - t -> e: a

join_2
    defined(e, j), e: j = ff
-----------------------------------
    e: j => a - t -> dpe(e, a): nil
```

These rules contain a number of user-defined functions. The function `defined` checks if all of the links used in the join condition are defined (have a value either *true* or *false*). Expression `e:j = tt` (or `ff`) invokes the `eval_join` function. This function returns *true* if the join condition `j` evaluates to the `new_bool` value specified. The function `dpe` assigns *false* to all the outgoing links of the activity. In order to accomplish this, it uses a helper function `mk_outgoing` that scans the given activity in order to find all its outgoing links. Then the value of each outgoing link is set to false using the `insert` function. Here is a Standard ML code snippet for the `dpe` function:

```
fun dpe (e, a) =
  let fun dpe_helper (e, []) = e
  | dpe_helper (e, hd::tl) =
    dpe_helper(insert(e, hd, False()), tl)
  in
    dpe_helper(e, mk_outgoing(a))
  end
```

To illustrate how the `dpe` function works consider a sample execution path for this small process:

```
"[]: out l1 false a || l1 =>(out l2 true (out l3 true b))"-a->
"[(l1, ff)]: l1 => (out l2 true (out l3 true b))"-t->
"[(l1, ff), (l2, ff), (l3, ff)]: nil"
```

When the activity `a` is executed its outgoing link `l1` is set to false because the transition condition for this link is `false`. The join condition for activity `b` evaluates to false. Therefore, activity `b` is discarded. The `dpe` function is called. It collects all the outgoing links of the discarded activity, assigns their value to false and inserts them into the environment.

4.3 Conclusion

After we have expressed the syntax and the semantics of the BPE-calculus in the required format, we can run the PAC. It produces a number of files. We add user-defined routines and recompile the CWB. That provides us with a version of the CWB that is capable of analyzing the BPE-calculus processes.

Chapter 5

Concurrency Workbench

The Concurrency Workbench (CWB) [CPS93] is a verification tool originally developed at North Carolina State University. It provides various analysis techniques for systems specified in a process algebra. We have extended the CWB to support the BPE-calculus. We use it to verify BPE-calculus processes.

The CWB provides three basic verification methods: equivalence checking, preorder checking and model checking. Equivalence checking allows to verify whether the system implementation satisfies the system specification. The specification is given in terms of the same process algebra as the implementation. The preorder method verifies that a process meets its minimum requirement, which is also specified as a high-level process. The process can satisfy the minimum requirement while possibly doing additional work. The third and final verification method is model checking. Using it we can verify that a system satisfies some desired property formulated in a logic. An example of such property is deadlock-freedom.

Once we have extended the CWB to support the BPE-calculus, we can use any of the analysis techniques provided by the CWB to verify BPE-calculus processes.

5.1 Model Checking

We are going to use the model checker to verify deadlock-freedom and other interesting properties.

The logic used in the CWB is the μ-calculus [Koz83] extended with some computational tree logic (CTL) [CES86] operators. These operators are syntactic sugar and include:

- `A` – for all possible execution paths

- `E` – there exists an execution path
- `G` – always (for every state of the execution path)
- `F` – eventually (there exists a state in the execution path)

Therefore, the property `AG` ϕ means that for every state along every execution path ϕ should hold. For more details about the syntax of the logic used in the CWB we refer the reader to [CS98].

In order to verify deadlock-freedom we first specify the property in the μ-calculus. Then we run the model checker on the process to check if the property holds. A state is deadlocked when there are no transitions from the state. Therefore, the state is deadlocked if the following property holds:

```
prop deadlock = not <->tt
```

where `-` denotes "any transition". For a process to be deadlock-free all of its possible states should be free of deadlock. This is expressed by the following property:

```
prop deadlock_free = AG(not deadlock)
```

The model checker can be invoked using the following command:

`chk` *process property*

Consider our travel agent example presented earlier:

```
[]: receiveRequest;
(out ca true reserveAC ++ out us ? reserveAA ++ reserveBritish
||getWeatherForecast
|| (us or ca => out r2a true rentCar)
|| (r2a =>t));
replyRequest
```

Execution of the model checker to verify deadlock-freedom for this example produces the following output:

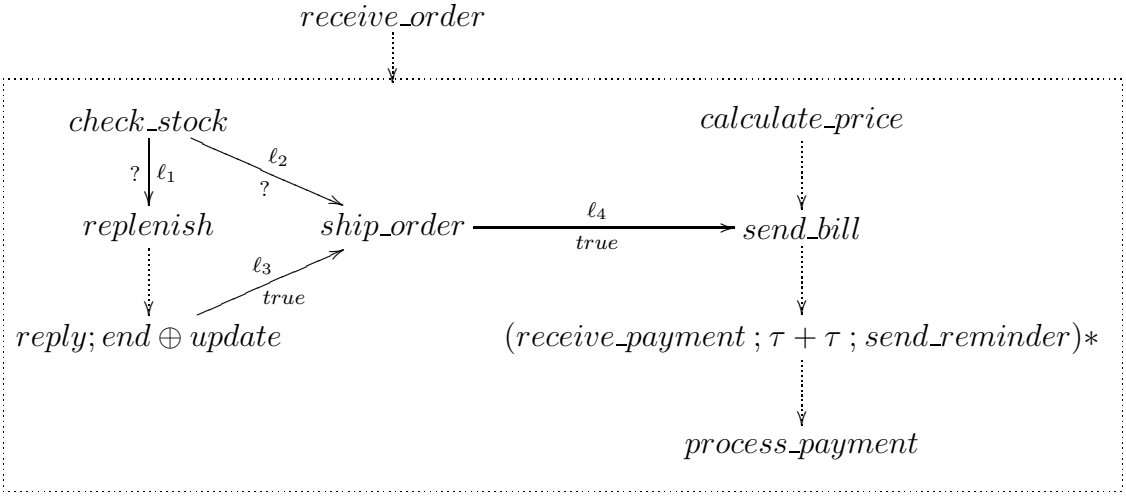
```
Invoking alternation-free model checker.
Building automaton...
States: 44
Transitions: 69
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.060,0.000,0.000,0.053)
```

We can verify other interesting properties for this example as well:

```
prop always_reply = AF <replyRequest>tt

prop reply_on_receive = AG [receiveRequest] always_reply

prop always_reserve = AF (<reserveAC>tt \/ <reserveAA>tt
   \/ <reserveBA>tt)

prop can_reply = EF <replyRequest>tt

prop reply_once = AG [replyRequest](not can_reply)
```

The `always_reply` property states that in every execution the process eventually sends a reply. We use this property to express the `reply_on_receive` condition. It states that whenever the request is received, the process eventually replies to it. We can also formulate the condition that a ticket is always reserved. Since there are three possible ways to reserve a ticket, the `always_reserve` property states that in every execution at least one reservation should be made. The `reply_once` property states that once the reply was sent the process should be incapable of replying again.

Let us look at a more advanced example adapted from an example in [Aal03]. Consider a book ordering process.



In the diagram the solid arrows represent links and the dotted arrows represent the flow of control in a sequence. The dotted frame denotes a flow and incorporates all of the activities in the flow.

The process starts when the order is received. The order is processed by checking if the book is in stock and by calculating the price of the order concurrently. If the book is in stock then the order is shipped. The activity `send_bill` waits until the order is shipped and then executes. The process then waits to receive a payment from the customer. If the payment is not received in a given period of time, the process sends the customer a reminder and starts waiting again. When the payment is received, it is processed by the `process_payment` activity. If the book is not in stock, then an attempt is made to re-stock by the `replenish` activity. If the attempt is successful the stock is updated and the order is shipped. If the attempt fails a reply is sent to the customer notifying them that the order cannot be completed. The process is then terminated using the `end` activity.

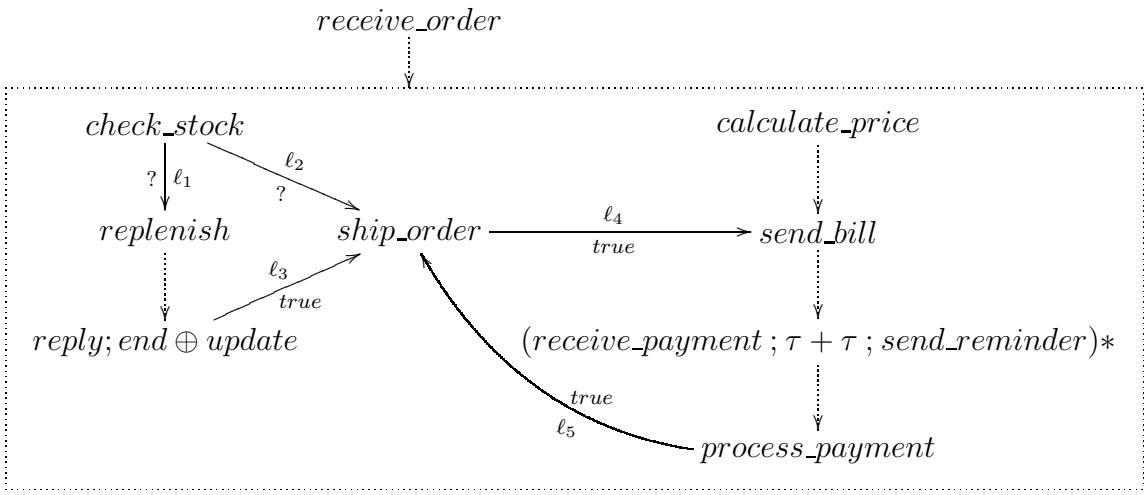
We can use the model checker to verify a number of interesting properties for this process, including deadlock freedom. Here are some of them:

```
prop can_ship = AG [replenish](EF <ship_order>tt)

prop can_complete_order = AG [receive_order]
                                    (EF <process_payment>tt)

prop will_end = AF <delta>tt

prop can_terminate = EF <gamma>tt
```

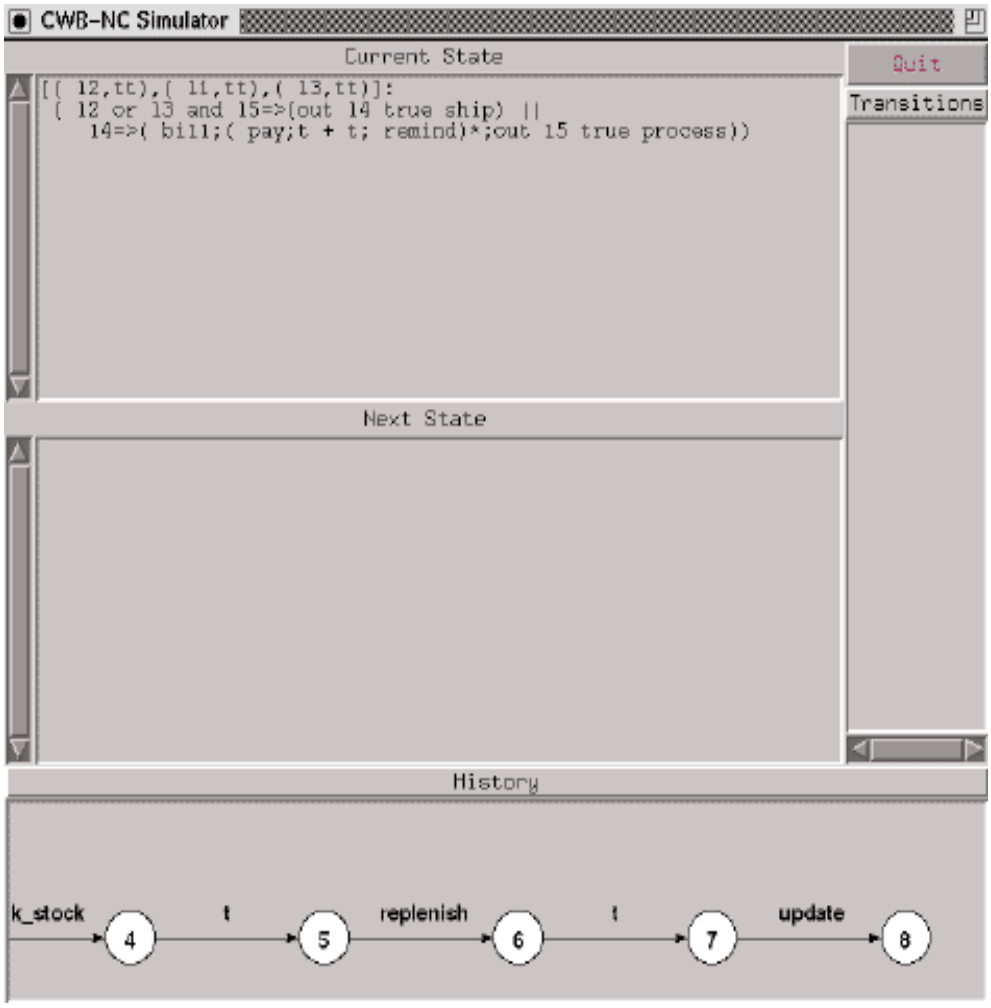
The first property states that it is possible to ship the order after replenishing stock. The second property states that it is possible to complete the order after the `receive_order` is executed. Note, however that it is not always possible to complete the order in this business process. Another property we can check is `will_end`. It checks whether the process will always terminate using the *end* activity. In order to specify this property we make use of the *delta* action. This action is executed when the process becomes an *end* activity. In the case of the book ordering process the property `will_end` does not hold, since in some executions of the process activity *end* will not be reached. The last property `can_terminate` is similar to `will_end`. It makes use of the special transition *gamma* that is used to signal successful termination. This property checks whether it is possible for the process to complete successfully becoming a nil process. The `can_terminate` property holds for the given process.

Consider now that we would like to change the process to ship the order only after the payment is received and processed. In order to do that we add another

link ℓ_5. The process now looks as follows:



The resulting process has a control cycle since the link ℓ_4 was not removed. We can use CWB's graphical user interface to verify this process. The graphical version of the CWB provides a built-in function `Find Deadlock`. The CWB detects the deadlock in our process. When the deadlock is found the program invokes the simulator to show the path that was taken before the deadlock state occurred. Therefore, the CWB provides the user not only with the information that there is a deadlock in the process, but also shows the exact place where it occurs. Here is a snapshot of the simulator screen:



The History window shows the path taken before the deadlock occurred. The last state is a deadlock state. The current state and the state after the transition is made are shown in the Current State and Next State windows respectively. These two windows display the state currently selected by the user. The user can select a state by putting mouse over the state number in the History window. The selected state in the snapshot is the deadlock state.

5.2 Preorder and Equivalence Checking

As was mentioned before the CWB also supports other verification methods. One of them is preorder checking. It is used to verify that a process fulfills

its minimum requirements. The minimum requirements are specified using a higher level process. The CWB offers two types of preorders: must preorder and may preorder (refer to, for example, [NH84] for more details). We are going to illustrate the use of preorder checking by verifying that the book ordering process satisfies its minimum requirements. We use the may preorder and we specify the requirements as follows:

```
receive_order;(check_stock || calculate_price);ship_order;
send_bill;receive_payment;process_payment
```

The CWB confirms that this process is in the may preorder relation with the book ordering process. If the CWB finds that two processes are not related by the preorder it returns some diagnostic information. This information can help the user to identify the problem in the process. In the case of the may preorder checking the CWB returns a trace of the more general process that cannot be matched by the more specific one. For example, consider the following minimum requirements for the second version of the book ordering process:

```
receive_order;(check_stock || calculate_price);
send_bill;receive_payment;process_payment;ship_order
```

The requirements are that the order is shipped only after the payment is received and processed. Recall, that this changed version of the book ordering process contains an error. It does not satisfy its minimum requirements. The CWB determines that and returns the following trace:

```
receive_order  check_stock  calculate_price  send_bill
```

This sequence of actions cannot be executed by the book ordering process since it has a deadlock.

Another way to check if the process works according to its specification is to use the equivalence checking method provided by the CWB. The equivalence checking method takes as its input two processes and determines if they are related by the behavioural equivalence relation. It can be used to check if a process is behaviourally equivalent to its specification. Several behavioural equivalences are supported by the CWB including bisimulation and observational equivalence. If the processes are not equivalent some diagnostic information is displayed. In order to help the user to determine the difference in the behaviour of the two processes the CWB returns a property that one of the processes satisfies and the other does not.

5.3 Conclusion

The CWB allows us to verify deadlock freedom along with other useful properties. It also provides preorder and equivalence checking. A very valuable feature of the CWB is that it provides diagnostic information to help the user determine the problem. Using the CWB we have successfully verified a number of small business processes. In the future, it would be interesting to apply it to verify large real life BPEL-processes.

Chapter 6

Conclusion

It is well known that it is not easy to get a concurrent program right. Concurrency is a key ingredient of the business process language BPEL. Therefore the need to develop verification tools for BPEL is pressing. In this thesis we presented such a tool. We have introduced a process algebra called BPE-calculus that contains the main control flow constructs of BPEL. We modelled our BPE-calculus by means of the labelled transition system. The grammar defining the syntax of the BPE-calculus and the rules defining the semantics of the BPE-calculus were used as input to the PAC. The PAC produced modules for the CWB so that the latter can be exploited for equivalence checking, preorder checking and model checking of BPE-processes.

In order to analyze a BPEL process it first has to be translated into the BPE-calculus. Since BPEL is an XML-based language this is a straightforward task. It can be done with the help of one of the publicly available XML parsers (like, for example, the Xerces parser [Xer]). The resulting process is then verified by the CWB using one of the available verification methods. We can check for deadlock using model checking. If deadlock can occur in the process, the CWB returns some diagnostic information. This can be related back to the BPEL process to assist the user in finding the source of the problem. Any other property that can be expressed in the μ-calculus can also be verified.

The BPE-calculus was designed specifically to model the control flow of BPEL processes. Two major benefits arise from that fact. First, the BPE-calculus models all of BPEL's control flow constructs including DPE, which cannot be captured by Petri nets [AH02]. Second, when the CWB finds a deadlock in the process it returns diagnostic information in terms of a trace of the BPE-process. This can be easily related to the original BPEL process (it would not be as easy with generic language representations of the process). Using the CWB as

the verification tool also proved to be very beneficial, since it allows not only to detect deadlock but also to verify other useful properties. It also provides preorder and equivalence checking that other tools like, for example, SPIN do not provide.

In the BPE-calculus we have abstracted from data, therefore making BPE-processes more generic than the corresponding BPEL processes. A BPE-process includes all of the execution paths in the BPEL process but possibly also some paths that are never executed in the BPEL process. For example, consider the following BPEL fragment:

```
<switch>
  <case condition = 'true'>
    activity_a
  </case>
  <case condition = 'false'>
    activity_b
  </case>
</switch>
```

In this example `activity_b` will never be executed. In the BPE-calculus the switch activity is modelled as a non-deterministic choice. Therefore, the above fragment will be translated into the following BPE-process:

$$activity\_a \oplus activity\_b$$

Hence, in the BPE-calculus version of the BPEL fragment `activity_b` can be executed. Now consider the situation where `activity_b` contains a deadlock and `activity_a` does not. Then, the CWB will indicate that this BPE-process may deadlock, even though the original BPEL process is deadlock free. Therefore, if the deadlock is found by the model checker it is not necessarily present in the BPEL process. But if the CWB indicates that the process is deadlock free it means that there is no possibility of deadlock on any possible execution path in the process. Therefore, the original BPEL process is deadlock free (provided that neither time nor fault and compensation handlers play an essential role in the BPEL specification).

In the BPE-calculus we have abstracted from fault handlers, compensation handlers and time. The CWB will not be able to verify processes that rely on these constructs in an essential way. In the future one of the possible improvements to the BPE-calculus would be to model fault handlers, compensation

handlers, and possibly even time. This extended calculus could form the basis for an even more accurate verification tool for BPEL.

BPEL processes often communicate with each other. In the future it would be interesting to use CWB to analyze the behavior of multiple communicating processes. In order to do that, a system of several processes can be expressed as a singe BPE-calculus process. This process can then be used as an input to CWB in the same way as a simple BPE-process.

Another interesting direction for future research would be to introduce some data into the model. Data approximation will allow us to check some of the interesting properties of BPE-processes, such as dead code and termination, with greater accuracy.

BIBLIOGRAPHY

[Aal00] W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, Berlin, 2000.

[Aal03] W.M.P. van der Aalst. Challenges in business process management: Verification of business processes using Petri nets. *Bulletin of the EATCS*, (80):174–199, 2003.

[AH02] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow patterns: on the expressive power of (Petri-net-based) workflow languages. In K. Jensen, editor, *Proceedings of the Fourth Workshop on the Practical Use of Coloured Petri Nets and CPN tools*, volume 560 of *DAIMI PB series*, pages 1–20, Aarhus, August 2002. University of Aarhus.

[BGE+02] T. Barclay, J. Gray, S. Ekblad, E. Strand, and J. Richter. TerraService.NET: An introduction to web services. Technical Report MS-TR-2002-53, Microsoft Research, Redmond, WA, June 2002.

[BPSMM00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (second edition). W3C Recommendation, available at http://www.w3.org/TR/REC-xml, October 2000.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, MA, 1998.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, V. Raman, F. Reiss, and M.A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 1st Semiannual Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programing Languages and Systems*, 8(2):244–263, April 1986.

[CGK+02] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, version 1.0. Available at http://www.ibm.com/developerworks/library/ws-bpel/, July 2002.

[CGMW02] R. Chinnici, M. Gudgin, J.-J. Moreau, and S. Weerawarana. Web Services Description Language (WSDL), version 1.2. W3C Working Draft, available at http://www.w3.org/TR/wsdl12, July 2002.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, NJ, July 1996. Springer-Verlag.

[CS98] R. Cleaveland and S. Sims. The concurrency workbench of North Carolina: User's manual. http://www.cs.sunysb.edu/∼cwb/, May 1998.

[CS02] R. Cleaveland and S.T. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, January 2002.

[DE00] J. Desel and T. Erwin. Modeling, simulation and analysis of business processes. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 129–141. Springer-Verlag, Berlin, 2000.

[DH01] M. Dumas and A.H.M. ter Hofstede. UML activity diagrams as a workflow specification language. In M. Gogolla and C. Kobryn, editors, *Proceedings of the 4th International Conference on the Unified Modeling Language*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90, Toronto, October 2001. Spring-Verlag.

[Fal01] D.C. Fallside, editor. XML Schema, Part 0: Primer. W3C Recommendation, http://www.w3.org/TR/xmlschema-0, May 2001.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[KGMW00] C. Karamanolis, D. Giannakopoulou, J. Magee, and S.M. Wheater. Model checking of workflow schemas. In *Proceedings of Enterprise Distributed Object Computing Conference*, pages 170–179, Makuhari, September 2000. IEEE Computer Society Press.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[Nak02] S. Nakajima. Verification of web services flows with model-checking techniques. In *Proceedings of the First International Symposium on Cyber Worlds*, pages 378–386, Tokyo, November 2002. IEEE Computer Society Press.

[NH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[NM02] S. Narayanan and S.A. Mcllraith. Simulation, verification and automated composition of web services. In *Proceedings of the Eleventh International World Wide Web Conference*, pages 77–88, Honolulu, May 2002. ACM.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PW03] G. Piccinelli and S.L. Williams. Workflow: A language for composing web services. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *Proceedings of the International Conference on Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12, Eindhoven, June 2003. Springer-Verlag.

[Rei85] W. Reisig. *Petri nets: an introduction*, volume 4 of *EATCS monographs on theoretical computer science*. Springer-Verlag, 1985.

[Rep99] J.H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Sch98] F.B. Schneider. On concurrent programming. *Communications of the ACM*, 41(4):128, April 1998.

[Sch99] M. Schroeder. Verification of business processes for a correspondence handling center using CCS. In A.I. Vermesan and F. Coenen, editors, *Proceedings of European Symposium on Validation and Verification of Knowledge Based Systems and Components*, pages 1–15, Oslo, June 1999. Kluwer.

[Sim99] S. Sims. The process algebra compiler: User's manual. http://www.reactive-systems.com/papers/pac-user.pdf, 1999.

[Xer] http://xml.apache.org/xerces-j/.