Verification of Business Processes for Web Services

Mariya Koshkina and Franck van Breugel

Technical Report CS-2003-11

October 2003

Department of Computer Science

4700 Keele Street, Toronto, Ontario M3J 1P3, Canada

Verification of Business Processes for Web Services*

Mariya Koshkina and Franck van Breugel

York University, Department of Computer Science
4700 Keele Street, Toronto, Canada M3J 1P3

mkosh@cs.yorku.ca and franck@cs.yorku.ca

October 2003

Abstract

A tool to verify properties, like for example deadlock freedom, of specifications in the business process execution language for web services (BPEL4WS) is presented. The four key steps in the development of this tool are highlighted. First of all, a process algebra named the BPE-calculus, that captures the flow of control of BPEL4WS but abstracts from many details, is introduced. Secondly, this BPE-calculus is modelled by means of a structural operational semantics. Thirdly, the grammar defining the syntax of the BPE-calculus and the rules defining the semantics of the BPE-calculus are used as the input to the process algebra compiler (PAC) to produce a front-end for the concurrency workbench (CWB). Finally, this front-end is used to adapt the CWB to the BPE-calculus, providing us with a tool to check properties of BPE-processes.

1 Introduction

Recently, IBM and Microsoft introduced the business process execution language for web services (BPEL4WS). This language has been designed to compose web services that interact over the Internet. For an introduction to web services, we refer the reader to, for example, [Mau03]. The lastest release of the BPEL4WS specification can be found in [].

2 BPEL4WS

BPEL4WS is XML based, that is, its syntax is defined in terms of an XML grammar. For example, the BPEL4WS snippet

```
<invoke partnerLink="shipping"
        portType="lns:shippingPT"
        operation="requestShipping"
        inputVariable="shippingRequest"
        outputVariable="shippingInfo">
</invoke>
```

invokes the web service operation `requestShipping` of the web service provided by the partner whose communication link is named `shipping` (The details of the request, including for example the destination, are stored in the variable `shippingRequest`. The acknowledgement will be stored in the container `acknowledgement`.

In BPEL4WS, the basic activities include assignments, invoking web service operations, receiving requests, replying to requests, waiting for a given amount of time, and terminating the whole process. These

*This research is supported by IBM.

basic activities are combined into structured activities using ordinary sequential control flow constructs like sequencing, switch constructs, and while loops.

Concurrency is provided by the flow construct. For example, in

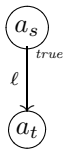
```
<flow>
   plane
   train
</flow>
```

the activities `plane` and `train`, whose behaviour has been left unspecified to simplify the example, are concurrent. The pick construct allows for selective communication. Consider, for example,

```
<pick>
   <onMessage partner="klm" operation="reply" container="klm-reply">
      fly-kml
   </onMessage>
   <onMessage partner="air-canada" operation="reply" container="air-canada-reply">
      fly-air-canada
   </onMessage>
</pick>
```

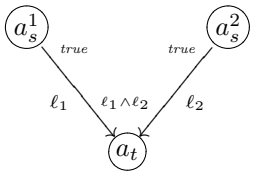
On the one hand, if a message from `klm` is received then the activity `fly-kml` is executed. In that case, the `fly-air-canada` activity will not be performed. On the other hand, the receipt of a message from `air-canada` triggers the execution of the `fly-air-canada` activity and discards the `fly-kml` activity. In the case that both messages are received almost simultaneously, the choice of activity to be executed depends on the implementation of BPEL4WS.

Synchronization between concurrent activities is provided by means of links. Each link has a source activity and a target activity. Furthermore, a transition condition is associated with each link. The latter is a Boolean expression that is evaluated when the source activity terminates. Its value is associated to the link. As long as the transition condition of a link has not been evaluated, the value of the link is undefined. In this paper, we will use, for example



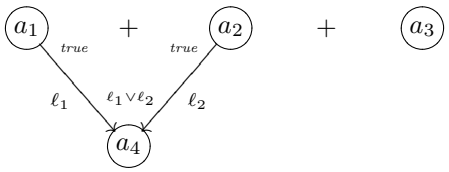
to depict that link ℓ has source a_s and target a_t and transition condition *true*.

Each activity has a join condition. This condition consists of incoming links of the activity combined by Boolean operators. Only when all the values of its incoming links are defined and its join condition evaluates to true, an activity can start. As a consequence, if its join condition evaluates to false then the activity never starts. We will use, for example,



to depict that the join condition of activity a_t is $\ell_1 \wedge \ell_2$. In the above example, activity a_t can only start after activities a_s^1 and a_s^2 have finished.

Let us consider the following activities and links.

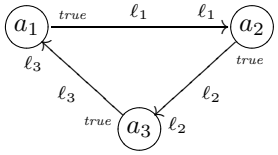


In the above picture, we use + to depict the pick construct. Note that the choice between the activities a_1, a_2 and a_3 determines whether activity a_4 is performed. For example, if a_3 is chosen then a_4 can never occur. As a consequence, the activity a_4 could be garbage collected. This can be achieved as follows.

- If a pick construct is executed, then we also assign false to all the outgoing links of those branches of the pick construct that are not chosen.
- If the join condition of an activity evaluates to false, then the activity is garbage collected after assigning false to its outgoing links.

This garbage collection scheme is named dead-path-elimination (DPE) in [CGK+02]. Let us briefly return to the above example. Assume that activity a_3 is chosen. Then, as a result of DPE, the value of the links ℓ_1 and ℓ_2 become false. At this point, the join condition of activity a_4 can be evaluated. Since its value is false, by DPE, activity a_4 is garbage collected.

According to the BPEL4WS definition [CGK+02], a link must not create a control cycle. That is, the initiation of the source activity must not semantically require the completion of the target activity. Obviously, a control cycle gives rise to deadlock. For example,



contains a control cycle. However, (the absence of) control cycles in BPEL4WS can be much more subtle. For example,

```
<sequence>
   <invoke operation="first">
      <target linkName="link"/>
   </invoke>
   <invoke operation="second">
      <source linkName="link"/>
   </invoke>
</sequence>
```

contains a control cycle as well, since `second`, the source of `link`, can only start once `first`, the target of `link`, has finished. The initial goal of our research was to develop a tool that can automatically check if a BPEL4WS specification is free of deadlocks.

Rather than considering BPEL4WS in its full complexity, we abstract from many details, in particular, from data. The majority of the data that is manipulated in a BPEL4WS specification is data received from web services. For this data, we only know its type but not its value. Considering all possible values would give rise to a huge number of different cases and, hence, would not allow us to consider any realistic BPEL4WS specification. Furthermore, we abstract from time. This simplifies matters considerably. As a drawback, we cannot verify a BPEL4WS specification the behaviour of which relies on time in an essential way. Finally, we ignore fault and compensation handlers. Abstracting from these details will simplify matters considerably and will make our objectives feasible (it is very difficult, if not almost impossible, to carry out such a study for BPEL4WS as a whole).

In this paper, we focus on a small language that includes all the concepts of BPEL4WS that we introduced above. The language captures the flow of control of BPEL4WS. As we already mentioned above, we abstract from many details. Furthermore, we leave unspecified some syntactic categories, like for example the basic activities. That is, we assume a set of basic activities but we do not specify their syntax. The so obtained language we call the BPE-calculus, as it is similar in flavour to calculi like, for example, CCS [Mil89]. A BPEL4WS specification can be mapped to a BPE-process in straightforward way. If the resulting BPE-process is free from deadlock then the original BPEL4WS specification is deadlock free as well (provided that neither time nor fault and compensation handlers play an essential role in the BPEL4WS specification).

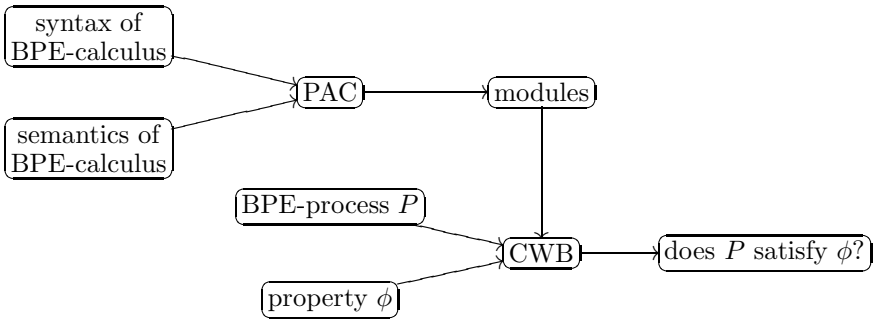
To verify that a BPE-process is deadlock free, we develop a model for the BPE-calculus. The predominant approach to model calculi like our BPE-calculus is to give a structural operational semantics as advocated by Plotkin [Plo81]. In this paper, we will follow this approach as well. We model the BPE-calculus by means of a labelled transition system. The transitions of the systems are defined by a collection of rules. Which rules apply to a BPE-process depends on its structure. An alternative approach to model business processes, namely by means of Petri nets [AH02a], is discussed in the concluding section of this paper.

To verify that BPE-processes are deadlock free, we will exploit the Concurrency Workbench (CWB) [CS96]. Originally, the CWB was designed for CCS-processes. The CWB supports three basic types of verification methods: equivalence checking, preorder checking and model checking. Equivalence checking allows us to verify whether processes are behaviourally equivalent. The CWB supports a number of different behavioural equivalences including trace equivalence and bisimilarity (see, for example, [Gla90] for an overview of different behavioural equivalences). Preorder checking is very similar to equivalence checking. In this case, one can check if processes are related according to some behavioural preorder, like the ones induced by may and must testing (see, for example, [NH84] for more details about these testing scenarios). In model checking, one verifies if a process satisfies a property. In the CWB, the property needs to be expressed in the μ-calculus [Koz83]. Deadlock freedom can be expressed in the μ-calculus and, hence, the CWB can check if a CCS-process is free of deadlock.

Due to its modular design, the CWB can be extended to support other calculi besides CCS. By adapting the CWB to the BPE-calculus, we obtain a tool that not only can verify if a BPE-process is deadlock free, our initial goal, but we can also check any other property expressible in the μ-calculus. Furthermore, we can apply equivalence checking and preorder checking to BPE-processes.

Manually implementing new modules for support of the BPE-calculus in 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 for the CWB. Given the syntax (in the form of a grammar) and the semantics (in the form of the rules defining the transitions of the labelled transition system) of the language, the PAC produces modules to support the language in the CWB. These modules include, for example, parsers and semantic functions. The modules can then be incorporated into the CWB. We use the PAC in order to generate a BPE-calculus front-end for the CWB.

The development of our tool can be summarized by means of the following picture.



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

The rest of this paper is organized as follows. In Section 3 we describe how we abstract from various details of BPEL4WS and we introduce the syntax of the resulting BPE-calculus. A structural operational

semantics of the BPE-calculus in terms of a labelled transition system is presented in Section 4. In Section 5, we describe how the syntax and semantics of the BPE-calculus are used as input for the PAC to generate modules for the CWB. In Section 6, we show how the CWB adapted to the BPE-calculus can be used to check that a BPE-process is free of deadlock. In the concluding section of this paper, we discuss related and future work.

3 The BPE-Calculus

Rather than studying BPEL4WS in its full complexity, we introduce the BPE-calculus. In the calculus, we focus on the flow of control. We abstract from data and time. Furthermore, we do not consider fault and compensation handlers.

3.1 Basic Activities

As we already mentioned in the introductory section of this paper, BPEL4WS contains various basic activities. We distinguish between external and internal basic activities.

The external basic activities involve interaction with the environment, that is, interaction with web services. Invoking web service operations, receiving requests from web services, and reply to requests of web services are all external basic activities. To simplify our calculus, we abstract from the particulars of each of these activities. All we want to express is that whenever one of these activities occurs, some interaction with the environment takes place. We do not distinguish between the different types of external basic activities. We introduce a set $\mathbb{A}_e$ of external basic activities. The syntax of these external basic activities is left unspecified. The concept of a external basic activity in the BPE-calculus is analogous to that of an action in calculi like, for example, CCS.

An internal basic activity does not interact with the environment. One internal basic activity deserves some special treatment. We will discuss that basic activity below. The other internal basic activities are the empty activity, assignments and waiting for a given amount of time. Since neither data nor time is considered in our calculus, the effect of these basic activities is not observable in our setting. All these internal basic activities are represented by τ in our calculus (again, our use of τ is very similar to its use in calculi like, for example, CCS).

In our calculus, we denote the terminate activity by †. Execution of this internal basic activity effects the whole process. As a result, the whole process stops as soon as possible.

3.2 Structured Activities

In the introduction we already discussed the structured activities of BPEL4WS. On the basis of simplified BPEL4WS examples, we will show how they are represented in the BPE-calculus.

We first consider the while loop.

```
<while condition="bpws:getContainerProperty(order, items) > 10">
   get-item
</while>
```

Since we abstract from data, the `condition` is not represented in the corresponding BPE-process. All we represent is that the body of the loop is executed a number of times. The choice to either exit the loop or continue for another iteration is nondeterministic in our setting. The above snippet is represented as $!P$, where the BPE-process P represents the BPEL4WS activity `get-item`.

In the BPE-calculus, we use $P_1 \,;\, P_2$ and $P_1 \parallel P_2$ to denote the sequential and concurrent composition of the processes P_1 and P_2. These correspond to the sequence and flow constructs of BPEL4WS.

BPEL4WS also contains the pick construct.

```
<pick>
   <onMessage partner="kml" operation="reply" container="klm-reply">
```

```
      fly-klm
    </onMessage>
    <onMessage partner="air-canada" operation="reply" container="air-canada-reply">
      fly-air-canada
    </onMessage>
    <onAlarm for="P3DT10H">
      time-out
    </onAlarm>
  </pick>
```

In a pick, the `onAlarm` is optional. In the above snippet, the `onAlarm` sets a timer that triggers the activity `time-out` if no message from either `kml` or `air-canada` is received within three days and ten hours. In the BPE-calculus, we represent the above snippet as

$$a_{\mathrm{kml}}\,;P_{\mathrm{klm}} + a_{\mathrm{air-canada}}\,;P_{\mathrm{air-canada}} + \tau;P_{\mathrm{time-out}}$$

where the external basic activities a_{kml} and $a_{\mathrm{air-canada}}$ correspond to the receipt of messages from `kml` and `air-canada`, respectively. The internal basic activity τ represents that the timer expires (recall that we abstract from time). The BPE-processes P_{klm}, $P_{\mathrm{air-canada}}$ and $P_{\mathrm{time-out}}$ correspond to the activities `fly-klm`, `fly-air-canada` and `time-out`, respectively.

Next, we look at an example of the switch construct.

```
  <switch>
    <case condition="bpws:getContainerProperty(trip, distance) > 500">
      plane
    </case>
    <case condition="bpws:getContainerProperty(trip, distance) <= 500">
      train
    </case>
  </switch>
```

Since we abstract from data, the `conditions` are not represented in the corresponding BPE-process. All we represent is that one of the cases is chosen. This choice is nondeterministic. The above snippet is represented as

$$P_{\mathrm{plane}} \oplus P_{\mathrm{train}}$$

where the processes P_{plane} and P_{train} correspond to the activities `plane` and `train`.

Both the pick and the switch construct involve a choice. For the former construct this choice depends on the environment, whereas for the latter it does not. The calculus CSP [Hoa85] also contains both choice constructs. The former is known as external choice and the latter as internal choice.

3.3 Links, Transition Conditions and Join Conditions

As we already explained in the introduction, links are used to provide synchronization between concurrent activities. We use $(\ell \uparrow b)P$ to denote that BPE-process P has link ℓ as an outgoing link. That is, P is the source of ℓ. In $(\ell \uparrow b)P$, we use b to represent the transition condition associated to ℓ. In BPEL4WS, the transition condition is an arbitrary Boolean expression. However, since we abstract from data in the BPE-calculus, we only consider three transition conditions. In the BPE-calculus, a transition condition is either *true* (which is the default in BPEL4WS), *false* or ?. The latter represents that the transition condition is either true or false. In this case, the choice (between *true* and *false*) is nondeterministic.

We use $c \Rightarrow P$ to express that BPE-process P has join condition c. In BPEL4WS, the join condition consists of incoming links combined by means of various Boolean operators. Without lack of generality, we restrict ourselves to those join conditions built from *true*, links, negation and conjunction. In BPEL4WS, the join condition and the incoming links of an activity are specified separately. The join condition of an activity

can only be evaluated once all incoming links of that activity are defined. To simplify the BPE-calculus, we omit the separate specification of the incoming links. Instead we rewrite the join condition c to contain all of the incoming links of process P. For example, if the incoming link ℓ of process P is not used in the join condition c of process $c \Rightarrow P$, then we change the join condition to be $c \wedge (\ell \vee \neg\ell)$.

3.4 Syntax

Before defining the syntax of the BPE-calculus, we first fix

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

Definition 1 The set $\mathbb{C}$ of join conditions is defined by

$$c ::= \mathit{true} \mid \ell \mid \neg c \mid c \wedge c$$

where $\ell \in \mathbb{L}$.

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

$$\begin{array}{lcl} P & ::= & \alpha \mid \dagger \mid (\ell \uparrow b)P \mid c \Rightarrow P \mid !P \mid P\,;P \mid P \parallel P \mid Q + Q \mid P \oplus P \\ Q & ::= & \alpha\,;P \mid Q + Q \end{array}$$

where $\alpha \in \mathbb{A}$, $\ell \in \mathbb{L}$, $c \in \mathbb{C}$ and $b \in \{\mathit{true}, \mathit{false}, ?\}$.

It is evident that the resulting process algebra is considerably simpler and hence more manageable than BPEL4WS. It captures all the features of the language related to the flow of control and abstracts from many details.

In BPEL4WS, each link should have a unique source and a unique target. Furthermore, links cannot cross boundaries of a while activity. That is, activities inside of the while loop cannot be linked to any activities outside of the while loop. We capture these restriction by means of the following very simple type system. Only if a process satisfies these restriction, it can be typed. The type of a process P is a pair: the set of links that are used as incoming in P, and the set of links that are used as outgoing in P.

Definition 2

(act) $\alpha : (\emptyset, \emptyset)$

(end) $\dagger : (\emptyset, \emptyset)$

(out) $$\frac{P : (I, O) \qquad \ell \notin O}{(\ell \uparrow b)P : (I, O \cup \{\ell\})}$$

(join) $$\frac{P : (I, O)}{c \Rightarrow P : (I \cup \mathit{links}(c), O)}$$

(while) $$\frac{P : (I, O) \qquad I = O}{!P : (I, O)}$$

(seq) $$\frac{P_1 : (I_1, O_1) \qquad P_2 : (I_2, O_2) \qquad I_1 \cap I_2 = \emptyset \qquad O_1 \cap O_2 = \emptyset}{P_1\,;P_2 : (I_1 \cup I_2, O_1 \cup O_2)}$$

$$(\textsc{flow})\ \frac{P_1 : (I_1, O_1) \qquad P_2 : (I_2, O_2) \qquad I_1 \cap I_2 = \emptyset \qquad O_1 \cap O_2 = \emptyset}{P_1 \parallel P_2 : (I_1 \cup I_2, O_1 \cup O_2)}$$

$$(\textsc{pick})\ \frac{P_1 : (I_1, O_1) \qquad P_2 : (I_2, O_2) \qquad I_1 \cap I_2 = \emptyset \qquad O_1 \cap O_2 = \emptyset}{P_1 + P_2 : (I_1 \cup I_2, O_1 \cup O_2)}$$

$$(\textsc{switch})\ \frac{P_1 : (I_1, O_1) \qquad P_2 : (I_2, O_2) \qquad I_1 \cap I_2 = \emptyset \qquad O_1 \cap O_2 = \emptyset}{P_1 \oplus P_2 : (I_1 \cup I_2, O_1 \cup O_2)}$$

In the above definition, we use $\mathit{links}(c)$ to denote the set of links that occur in the join condition c. Not every process can be typed. For example, the process $(\ell \uparrow \mathit{true})\dagger \parallel (\ell \uparrow \mathit{true})\dagger$ cannot be typed. However, if a process is well-typed then its type is unique. That is, if $P : (I_1, O_1)$ and $P : (I_2, O_2)$ then $I_1 = I_2$ and $O_1 = O_2$. Furthermore, each type is finite. That is, if $P : (I, O)$ then I and O are finite sets of links. A process P satisfies the above mentioned restrictions if $P : (I, O)$ for some I and O such that $I = O$.

4 Structural Operational Semantics

In the previous section, we introduced the syntax of the BPE-calculus. Next, we present its semantics. We model the BPE-calculus by means of a structural operational semantics [Plo81]. In this approach, a labelled transition system is defined by means of a collection of rules.

In our model, we need to keep track of the values of the links. The value of a link is either true, false, or undefined. The latter we denote by $\perp$. The status of the links is captured by an element of $\Sigma = \mathbb{L} \to \{\mathit{true}, \mathit{false}, \perp\}$. Initially, all the links are undefined.

A labelled transition system consists of a collection of states, a collection of labels and a collection of transitions. In this case, each state consists of a pair: a BPE-process or the nil process $\surd$, which we will use to model successful termination, and a link status. That is, a state is an element of $(\mathbb{P} \cup \{\surd\}) \times \Sigma$. As labels we use basic activities, that is, elements of $\mathbb{A}$. In the definition below 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 $\sigma[{}^{v}\!/_{\ell}]$. The transitions are defined by the following rules.

Definition 3

$$(\textsc{act})\ \langle \alpha, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma \rangle$$

$$(\textsc{out}_t)\ \frac{\langle P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma' \rangle}{\langle (\ell \uparrow \mathit{true})P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma'[{}^{\mathit{true}}\!/_{\ell}] \rangle}$$

$$(\textsc{out}_f)\ \frac{\langle P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma' \rangle}{\langle (\ell \uparrow \mathit{false})P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma'[{}^{\mathit{false}}\!/_{\ell}] \rangle}$$

$$(\textsc{out}_?)\ \frac{\langle P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma' \rangle}{\langle (\ell \uparrow ?)P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma'[{}^{\mathit{true}}\!/_{\ell}] \rangle} \qquad \frac{\langle P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma' \rangle}{\langle (\ell \uparrow ?)P, \sigma \rangle \xrightarrow{\alpha} \langle \surd, \sigma'[{}^{\mathit{false}}\!/_{\ell}] \rangle}$$

$$(\textsc{out})\ \frac{\langle P, \sigma \rangle \xrightarrow{\alpha} \langle P', \sigma' \rangle}{\langle (\ell \uparrow b)P, \sigma \rangle \xrightarrow{\alpha} \langle (\ell \uparrow b)P', \sigma' \rangle}$$

$$(\textsc{out}_\dagger)\ \langle (\ell \uparrow b)\dagger, \sigma \rangle \xrightarrow{\tau} \langle \dagger, \sigma \rangle$$

$$(\textsc{join}_t)\ \frac{\mathcal{C}(c)(\sigma) = \mathit{true}}{\langle c \Rightarrow P, \sigma \rangle \xrightarrow{\tau} \langle P, \sigma \rangle}$$

$$(\textsc{join}_f)\ \frac{\mathcal{C}(c)(\sigma) = \mathit{false} \qquad P : (I, O)}{\langle c \Rightarrow P, \sigma \rangle \xrightarrow{\tau} \langle \surd, \sigma[{}^{\mathit{false}}\!/_{O}] \rangle}$$

$$\text{(WHILE)}\ \langle !P,\sigma\rangle \xrightarrow{\tau} \langle \surd,\sigma\rangle \qquad \frac{P:(I,O)}{\langle !P,\sigma\rangle \xrightarrow{\tau} \langle P;!P,\sigma[\perp/O]\rangle}$$

$$\text{(WHILE}_\dagger\text{)}\ \langle !\dagger,\sigma\rangle \xrightarrow{\tau} \langle \dagger,\sigma\rangle$$

$$\text{(SEQ)}\ \frac{\langle P_1,\sigma\rangle \xrightarrow{\alpha} \langle \surd,\sigma'\rangle}{\langle P_1\,;P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_2,\sigma'\rangle} \qquad \frac{\langle P,\sigma\rangle \xrightarrow{\alpha} \langle P_1',\sigma'\rangle \qquad P_1' \neq \surd}{\langle P_1\,;P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_1'\,;P_2,\sigma'\rangle}$$

$$\text{(SEQ}_\dagger\text{)}\ \langle \dagger\,;P_2,\sigma\rangle \xrightarrow{\tau} \langle \dagger,\sigma\rangle$$

$$\text{(FLOW}_\ell\text{)}\ \frac{\langle P_1,\sigma\rangle \xrightarrow{\alpha} \langle \surd,\sigma'\rangle}{\langle P_1 \parallel P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_2,\sigma'\rangle} \qquad \frac{\langle P_1,\sigma\rangle \xrightarrow{\alpha} \langle P_1',\sigma'\rangle \qquad P_1' \neq \surd}{\langle P_1 \parallel P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_1' \parallel P_2,\sigma'\rangle}$$

$$\text{(FLOW}_r\text{)}\ \frac{\langle P_2,\sigma\rangle \xrightarrow{\alpha} \langle \surd,\sigma'\rangle}{\langle P_1 \parallel P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_1,\sigma'\rangle} \qquad \frac{\langle P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_2',\sigma'\rangle \qquad P_2' \neq \surd}{\langle P_1 \parallel P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_1 \parallel P_2',\sigma'\rangle}$$

$$\text{(FLOW}_\dagger\text{)}\ \langle \dagger \parallel P_2,\sigma\rangle \xrightarrow{\tau} \langle \dagger,\sigma\rangle \qquad \langle P_1 \parallel \dagger,\sigma\rangle \xrightarrow{\tau} \langle \dagger,\sigma\rangle$$

$$\text{(PICK)}\ \frac{\langle P_1,\sigma\rangle \xrightarrow{\alpha} \langle P_1',\sigma'\rangle \qquad P_2:(I_2,O_2)}{\langle P_1+P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_1',\sigma'[\mathit{false}/O_2]\rangle} \qquad \frac{\langle P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_2',\sigma'\rangle \qquad P_1:(I_1,O_1)}{\langle P_1+P_2,\sigma\rangle \xrightarrow{\alpha} \langle P_2',\sigma'[\mathit{false}/O_1]\rangle}$$

$$\text{(SWITCH)}\ \frac{P_2:(I_2,O_2)}{\langle P_1 \oplus P_2,\sigma\rangle \xrightarrow{\tau} \langle P_1,\sigma[\mathit{false}/O_2]\rangle} \qquad \frac{P_1:(I_1,O_1)}{\langle P_1 \oplus P_2,\sigma\rangle \xrightarrow{\tau} \langle P_2,\sigma[\mathit{false}/O_1]\rangle}$$

Let us briefly discuss the above rules.

(ACT) A process consisting of a single basic activity is capable of making just one transition. After the basic activity has been performed, the process can no longer make any transitions. The nil process $\surd$ cannot make any transitions.

(OUT) The outgoing link ℓ in the process $(\ell \uparrow b)P$ becomes activated only when the source activity P completes, that is, when P equals $\surd$. 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 nondeterministically. This reflects the fact that the transition condition depends on data values (from which we abstract) and can evaluate to true in some cases and to false in other cases.

(JOIN) To model the evaluation of join conditions, we introduce a function $\mathcal{C} : \mathbb{C} \to \Sigma \to \{\mathit{true}, \mathit{false}, \perp\}$ defined by

$$\begin{array}{rcl} \mathcal{C}(\mathit{true})(\sigma) & = & \mathit{true} \\ \mathcal{C}(\ell)(\sigma) & = & \sigma(\ell) \\ \mathcal{C}(\neg c)(\sigma) & = & \neg\mathcal{C}(c)(\sigma) \\ \mathcal{C}(c_1 \wedge c_2)(\sigma) & = & \mathcal{C}(c_1)(\sigma) \wedge \mathcal{C}(c_2)(\sigma) \end{array}$$

where

c	*true*	*false*	$\perp$
$\neg c$	*false*	*true*	$\perp$

and

$\wedge$	*true*	*false*	$\perp$
true	*true*	*false*	$\perp$
false	*false*	*false*	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

In case that the join condition c of process P evaluates to true, the process P will be executed. In case that the join condition c is false, the process P is skipped. In addition to skipping P, dead-path-elimination (DPE) is triggered in that case. It sets all the outgoing links of process P to false. In both

cases a τ-transition is used, since the evaluation of a join condition and the execution of DPE are both internal actions.

(WHILE) In the while loop $!P$, the process P can be repeated an arbitrary number of times. After every iteration there is a nondeterministic choice to either stop or continue for another iteration. This choice is internal and, therefore, the transition is labelled with τ. 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 that $I = O$ for the while loop to be well-typed).

(SEQ) The rules for sequencing are reminiscent to ones for sequential composition in ACP (see, for example, [Fok00]).

(FLOW) The rules for the flow construct are very similar to ones for composition in CCS.

(PICK) The pick construct performs a nondeterministic choice. The choice is external. This means that the choice of the process is dictated by the environment. The process that is capable of interacting with the environment is chosen. If both processes are capable of such interaction then the choice is made randomly.

In both the switch and the pick construct, the process that is not chosen is simply discarded. DPE sets all of the outgoing links of the discarded process to false. This is done in order to eliminate execution paths that will never be taken.

(SWITCH) Also the switch construct performs a nondeterministic choice. This time the environment has no influence on the choice. The choice is made internally. Therefore, we label this transition with τ.

The rules for pick and switch show similarities to the rules for external and internal choice in CSP [Plo82].

(END) The terminate activity, represented by † in the BPE-calculus, stops the whole process as soon as possible. When this activity is encountered all branches of the execution are abandoned. Additional rules for outgoing links, the while loop, sequencing and flows are introduced to model this behaviour. Note, that the activity † by itself is incapable of performing any transitions.

5 Process Algebra Compiler

Now that we have defined the BPE-calculus both syntactically and semantically, we are ready to use the process algebra compiler (PAC) [CS02]. As we already mentioned in the introduction, the PAC is used to adapt the concurrency workbench (CWB) to support a new language. It generates a front-end, written in Standard ML, for the CWB. The PAC takes as its input two files. One file contains the description of the syntax of the language, and the other file describes the semantics of the language. Below, we will describe these two files that the PAC will use to generate a front-end for the CWB to adapt it to the BPE-calculus.

5.1 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, the second describes the concrete syntax, and the third contains some additional syntactic constructs that are used to specify the rules defining the semantics.

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 include

```
sorts
  process, pick, activity, join, link, value, status
```

The sorts `process` and `pick` correspond to the nonterminals P and Q in Definition 1. The sorts `activity` and `link` correspond to the sets $\mathbb{A}$ and $\mathbb{L}$. We also declare the sort `value` that extends the Boolean type with $\perp$. This sort will be used to represent the value of a link. The sort `status` corresponds to the set Σ that keeps track of the values of the links.

In the `cons` section (of the first part), constructors for the sorts are introduced. For our calculus, these include

```
cons
  Nil:      unit -> process
  Activity: activity -> process
  Join:     join * process -> process
  Flow:     process * process -> process
  Pick:     pick * pick -> process
```

A `process` can be constructed in a number of different ways. For example, the constructor `Nil` takes no argument and returns a `process`, and the `Pick` constructor takes two `pick`s and produces a `process`. Based on above specification, the PAC will generate the actual Standard ML code for each constructor.

Constructors for some simple sorts are not defined in the `cons` section. Instead, they are listed in the `funs` section. The constructors in this section are not generated by the PAC, but have to be written by the user. This is done to give the user an opportunity to program simple types efficiently or to re-use existing code. The user should also provide some functions for the semantic description. We include such functions as join conditions evaluation and dead-path elimination. The `funs` section specifies the signatures for all the user-defined functions.

In the second section, the concrete syntax is specified. In order to specify the concrete syntax, first the tokens are defined.

```
  "nil"  => NIL
  "\|\|" => PAR
  "\=\>" => ARROW
  "+"    => PLUS
```

These tokens are subsequently used to capture the syntactic structure of the BPE-calculus.

```
grammar
  process : NIL                  (Nil())
          | activity             (Activity(activity))
          | join ARROW process   (Join(join, process))
          | process PAR process  (Flow(process1, process2))
          | pick PLUS pick       (Pick(pick1, pick2))
```

In the third and final section, we introduce some additional syntax to specify the rules defining the semantics. This `rules syntax` section has a format similar to the one described above.

5.2 Semantics Description File

The semantics is defined by means of a collection of rules. Their format is very similar to that of the rules in Definition 3. For example, the rules for the flow construct look as follows.

```
flow_1
       <P1, s> - a -> <nil, s'>
  ------------------------------
  <P1 || P2, s> - a -> <P2, s'>
```

```
flow_2
       <P1, s> - a -> <P1', s'>, not(P1' = nil)
  ------------------------------------------
  <P1 || P2, s> - a -> <P1' || P2, s'>

flow_3
       <P2, s> - a -> <nil, s'>
  ----------------------------
  <P1 || P2, s> - a -> <P1, s'>

flow_4
       <P2, s> - a -> <P2', s'>, not(P2' = nil)
  ------------------------------------------
  <P1 || P2, s> - a -> <P1 || P2', s'>
```

In the above rules, `P1`, `P2`, `P1'` and `P2'` are processes, `s` and `s'` contain the status of the links, and `a` is a basic activity.

The rules for the join condition are

```
join_1
       eval(c, s) = tt
  ------------------------
  <c => P, s> - t -> <P, s>

join_2
       eval(c, s) = ff
  ---------------------------------
  <c => P, s> - t -> <nil, dpe(s, P)>
```

In the above rules, `eval` and `dpe` are user-defined functions. The function `eval` evaluates the join condition given the status of the links. The function `dpe` updates the status of the links by assigning *false* to all outgoing links of the process P.

We have implemented `status` as a list. Each element of the list contains a link and its status. Whenever a new outgoing link is encountered, it is added to the end of the list. The list is scanned when we need to find the value of a link.

One of the properties we would like to check is deadlock freedom. A state is deadlocked if it is incapable of making any transitions. In order not to confuse a terminated state with a deadlocked state, we have introduced the following rules.

```
nil_rule

------------------------
<nil, s> - t -> <nil, s>

end_rule

------------------------
<end, s> - t -> <end, s>
```

This ensures that if a process terminates, then it is not considered deadlocked, since it is capable to making τ-transitions.

After we have expressed the syntax and semantics of the BPE-calculus in the required format, we can run the PAC. It produces a new front-end for the CWB. Recompiling the CWB provides us with the version that is capable of analyzing BPE-processes.

6 Concurrency Workbench

The concurrency workbench (CWB) [CS96] is a powerful verification tool. As we already discussed in the introduction, it allows us to do equivalence checking, preorder checking and model checking. Now that we have extended the CWB to the BPE-calculus, we can apply these verification techniques to BPE-processes.

For example, we can use the CWB to check if a BPE-process is deadlock free. This can be done in a number of ways. First of all, we can express the property we want to check, deadlock freedom, in terms of a logic. The logic used in the CWB is 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 an execution path)
- `F` – eventually (there exists a state in the execution path)

For more details about the syntax of the logic used in the CWB we refer the reader to the CWB user manual [CS98]. Recall that a state is deadlocked if it cannot make any transitions. Therefore, we can express that a state is deadlocked as:

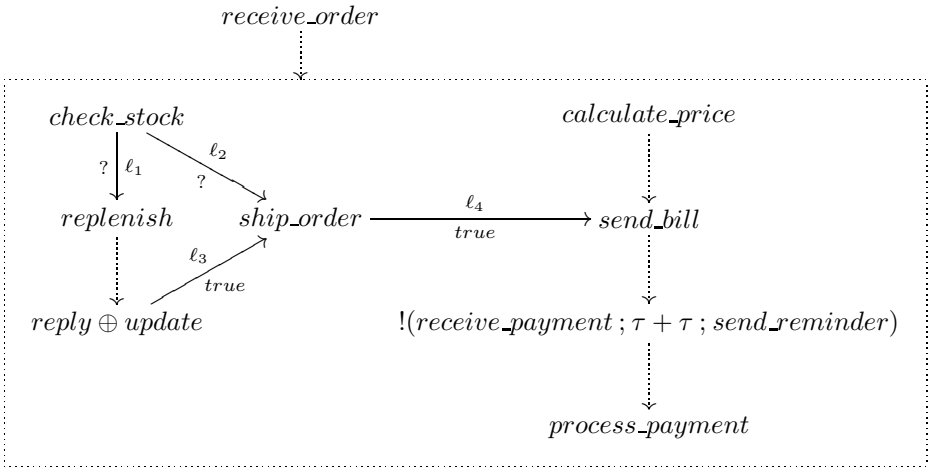
```
deadlock = not <->tt
```

For a process to be free of deadlock, all processes reachable from this process should be deadlock free. This can be expressed as follows.

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

Now we can use the model checker to verify if a process satisfies the property `deadlock_free`. Alternatively, we can also use the "find deadlock" button on the graphical user interface of the CWB. If a deadlock is found, a simulator is invoked. This simulator depicts a sequence of transitions from the initial process to the one that may cause the deadlock.

Besides checking for deadlocks, we can verify other interesting properties as well. For example, consider the book ordering process adapted from an example in [Aal03]



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 an 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 within 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 the reply is sent to the customer notifying them that the order cannot be completed.

We can use the CWB to verify properties like

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

prop always_check_stock = AG [receive_order] (AF <check_stock>tt)
```

The first property states that it is possible to ship the order after replenishing stock. The second property states that at some point after the order is received the stock is checked.

As we mentioned before the CWB also supports such verification methods as equivalence checking and preorder checking. Due to space limitation, we do not provide any examples of these verification methods here. We refer the reader to [Kos03].

Using the CWB we have successfully verified a number of business processes.

7 Conclusion

Let us first summarize our contributions. We introduced a new calculus, named the BPE-calculus, that contains the main control flow constructs of BPEL4WS. We modelled our BPE-calculus by means of a structural operational semantics. The grammar defining the syntax of the BPE-calculus and the rules defining the semantics of the BPE-calculus were used as input of 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.

Also in [PW03], Piccinelli and Williams 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.

In our study, we used a labelled transition system to model the BPE-calculus. Petri nets provide an alternative approach to model business processes. For an overview, we refer the reader to, for example, [AH02a]. These Petri nets can also be used as a basis to develop verification tools. For an example, we refer the reader to [Aal97, NM02]. We believe that labelled transition systems are superior to Petri nets when it comes to modelling BPEL4WS, since, as also pointed out in [AH02b], advanced synchronization patterns like DPE cannot easily be captured by means of Petri nets.

In [Sch99], Schroeder presents a translation of business processes into CCS. Subsequently, the CWB can be used for verification. The business process language that is studied in that paper is considerably simpler than BPEL4WS. It is not clear to us if this approach is also applicable to BPEL4WS. In particular, 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.

Nakajima [Nak02] describes how to use the SPIN model checker to verify web services flows. The language used to specify the flows is WSFL (Web Services Flow Language) which is one of BPEL4WS's predecessors. In order to do the verification using SPIN, processes are first translated into the Promela specification language provided by SPIN. 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 the case of the BPE-calculus, the trace returned by the CWB is closely related to the trace in the original BPEL4WS process. Another advantage of the CWB over the SPIN is that the former provides other verification methods in addition to model checking. The approach in [KGMW00] that uses the LTSA toolkit and the FSP process algebra suffers from the same weaknesses.

In the future, we would like to investigate if our BPE-calculus can be extended to include fault and compensation handlers and possibly even time.

Acknowledgements

References

[Aal97] W.M.P. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Proceedings of the 18th International Conference on Applications and Theory in Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, Toulouse, June 1997. Springer-Verlag.

[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.

[AH02a] W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2002.

[AH02b] 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.

[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.

[CS96] R. Cleaveland and S.T. 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*, 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.

[Fok00] W.J. Fokkink. *Introduction to Process Algebra*. Springer-Verlag, 2000.

[Gla90] R.J. van Glabbeek. The linear time-branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of the 1st International Conference on Concurrency Theory*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, August 1990. Springer-Verlag.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[KGMW00] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. M. Wheater. Model checking of workflow schemas. In *Proceedings of the 4th International Enterprise Distributed Object Computing Conference*, pages 170–179, Makuhari, Japan, September 2000. IEEE Computer Society Press.

[Kos03] M. Koshkina. Verification of business processes for web services. Master's thesis, York University, 2003.

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

[Mau03] J. Maurer, editor. *Queue*, March 2003.

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

[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. Report DAIMI FN-19, Aarhus University, Aarhus, September 1981.

[Plo82] G.D. Plotkin. An operational semantics for CSP. In D. Bjorner, editor, *Proceedings of IFIP Working Conference on Formal Description of Programming Concepts - II*, pages 199–223, Garmisch-Partenkirchen, June 1982. North-Holland.

[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.

[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.