Stepwise Design of BPEL Web Services Compositions: An Event_B Refinement Based Approach

Idir Ait-Sadoune and Yamine Ait-Ameur

Abstract. Several web services compositions languages and standards are used to describe different applications available over the web. These languages are essentially syntactic ones, their descriptions remain informal and are based on graphical notations. They do not offer any guarantee that the described services achieve the goals they have been designed for. The objective of this paper is twofold. First, it focusses on the formal description, modelling and validation of web services compositions using the Event_B method. Second, it suggest a refinement based method that encodes the BPEL models decompositions. Finally, we show that relevant properties formalized as Event_B properties can be proved. A tool encoding this approach is also available.

1 Introduction

With the development of the web, a huge number of services available on the web have been published. These web services operate in several application domains like concurrent engineering, semantic web or electronic commerce. Moreover, due to the ease of use of the web, the idea of composing these web services to build composite ones defining complex workflows arose.

Nowadays, even if several industrial standards providing specification and/or design XML-oriented languages for web services compositions description, like BPEL [16], CDL [9], BPMN [14] or XPDL [24] have been proposed, the activity of composing web services remains a syntactically based approach. Some of these languages support a partial static semantics checking provided by languages like XML-schema. If these languages are different from the description point of view, they share several concepts in particular the service composition. Among the shared

Idir Ait-Sadoune and Yamine Ait-Ameur
Laboratory of Applied Computer Science (LISI-ENSMA), Téléport 2-1, avenue Clément Ader, BP 40109, 86961, Futuroscope, France
e-mail: {idir.aitsadoune,yamine}@ensma.fr

concepts, we find the notions of *activity* for producing and consuming messages, *attributes* for instance correlation, message decomposition, service location, compensation in case of failure, events and faults handling. These elements are essential to describe services compositions and their corresponding behavior.

Due to the lack of formal semantics of these languages, ambiguous interpretations remain possible and the validation of the compositions is left to the testing and deployment phases. From the business point of view, customers do not trust these services nor rely on them. As a consequence, building correct, safe and trustable web services compositions becomes a major challenge.

Our work proposes to provide a formal semantics for the BPEL language (Business Process Execution Language) and ensure the correctness of web services compositions during the early design phase. We suggest to use formal methods, precisely the Event_B method [2], in order to produce formal models describing web services compositions.

The contribution of this paper is twofold: methodological and technical. From the *methodological point of view*, our objective is not to change the design process mastered by the web services compositions designers, but it is to enrich this process with a formal modelling and verification activity by providing formal models beside each web services composition description. More precisely, our approach consists in deriving an Event_B model from each BPEL web service compositions and encode the decomposition relationship, offered by BPEL, by an Event_B refinement relationship. Therefore, it becomes possible to enrich the Event_B models by the relevant properties in order to check not only the correctness of the web services compositions described in the BPEL designs but also to check the correctness of the decomposition encoded by the sequence of BPEL models linked by the decomposition relationship. Regarding the formal modelling activity, the complexity of the proofs is reduced thanks to refinement.

From a *technical point of view*, we are aware that several work addressed the problem of formal web services verifications [7]. These approaches use model checking as a verification procedure facing the state number explosion problem. Abstraction techniques are used in order to reduce the complexity of the verification. The consequence of this abstraction is the loss of relevant properties modelling and verification like data transformation properties or message exchanges. We claim that, thanks to the Event_B method, we are capable to overcome this limitation by supplying a technique supporting data transformations.

This paper is structured as follows. Next section presents an overview of the state of the art in formal verification of web services compositions. Then sections 3 and 4, describe Event_B method and our approach for transforming any BPEL design into an Event_B model. Static and dynamic BPEL parts are distinguished in this transformation process. Then, we show in section 5 how the refinement offered by Event_B can be used to encode the BPEL decomposition operation. As a result, we obtain an incremental development of both the BPEL design and the Event_B models. Section 6 discusses the properties verification capabilities that result from this approach. Finally, a conclusion and some perspectives are outlined.

2 Formal Validation of Services Compositions: Related Work

Various approaches have been proposed to model and to analyze web services and services compositions, especially formal modelling and verification of BPEL processes. In this section, we overview the formal approaches that have been set up for formal validation of services composition descriptions.

Petri nets were often used to model web service compositions. They have been set up by *Hinz et. al* [15] to encode BPEL processes and to check standard Petri nets properties, and some other properties formalized in CTL logic. *van der Aalst et. al* [1, 25] have defined specific Petri nets called workflow nets to check some properties like termination of a workflow net, and detection of unreachable nodes. Recently, *Lohmann* [17] has completed the BPEL semantics with Petri nets by encoding the new elements appeared in the version 2.0 of BPEL.

Classical transitions systems were also set up to specify web service compositions, especially BPEL processes. We quote the work of *Nakajima* [20, 21] who mapped a BPEL activity part to a finite automaton encoded in Promela with the model checker SPIN to analyze behavioral aspects and to detect potential deadlocks that may occur in the Promela description. In [18, 19], *Marconi et. al* present the approach that translates a BPEL process to a set of state transition systems. These systems are composed in parallel and the resulting parallel composition is annotated with specific web services requirements and is given as input to a set of tools in charge of synthesizing web service compositions expressed in BPEL. FSP (Finite State Process) and the associated tool (LTSA) are used by *Foster et. al* [12, 13] to check if a given BPEL Web Service compositions behaves like a Message Sequence Charts (MSC).

Some work used process algebra for processes and activities formalization. Indeed, *Salaun et. al* [23] show how BPEL processes are mapped to processes expressed by the LOTOS process algebra operations. The same authors in [22] applied their approach to CCS descriptions to model the activities part of a BPEL specification.

Abstract State Machines (ASM) have been used by *Farahbod et. al* [11] to model BPEL workflow descriptions. They take into account exchanged messages and some BPEL real time properties like timeouts. This work has been extended by *Fahland* [10] to model dead-path-elimination.

Other approaches proposed formal models for web service compositions and the BPEL processes. We did not mention them in the above summary due to space limitations. An overview of these approaches can be found in [7].

The whole related work outlined above has two major drawbacks. First, it does not address the description of the static part of BPEL available in the WSDL descriptions. This is due to the abstraction made by the majority of the applied formal methods. Indeed, most of these methods abstract parameters, exchanged messages, preconditions, postconditions and join conditions by Booleans. This abstraction is useful for model checking since it reduces the space explored during verification but it decreases the accuracy of the obtained formal BPEL model. Second, all these proposals translate a BPEL process to another formal model, without offering the

CONTEXT	**MACHINE**
$c1$	$m1$
SETS	**REFINES**
...	$m2$
CONSTANTS	**SEES**
...	$c1$
AXIOMS	**VARIABLES**
...	...
THEOREMS	**INVARIANTS**
...	...
	THEOREMS
	...
	EVENTS
	...
	END

Fig. 1 The structure of an Event_B development

capability to decompose BPEL precess. So, the decomposition operator offered by BPEL is never addressed by the used formal techniques. The resulting model is often complex to analyze. Thus, the efficiency of the model checking technique often used by existing approaches for checking BPEL processes properties is reduced.

Finally, these methods suffer from the absence of error reporting. Indeed, when errors are detected in the formal model, these approaches do not localize the source of the error on the original BPEL model.

Our work is proof oriented and translates the whole BPEL language and all its constructs into an Event_B model. All the Event_B models presented in this paper have been checked within the RODIN platform[1].

3 Event_B Method

An Event_B model [2] encodes a state transition system where the variables represent the state and the events represent the transitions from one state to another. The refinement capability [3] offered by Event_B allows to decompose a model (thus a transition system) into another transition system with more and more design decisions while moving from an abstract level to a less abstract one. The refinement technique preserves the proved properties and therefore it is not necessary to prove them again in the refined transition system (which is usually more complex). The structure of an Event_B model is given in figure 1.

A **MACHINE** *m1* is defined by a set of clauses. It may refine another **MACHINE** *m2*. Briefly:

[1] The Rodin Platform is an Eclipse-based IDE for Event-B : http://www.event-b.org/platform.html

- **VARIABLES** represents the state variables of the model of the specification.
- **INVARIANTS** describes by first order logic expressions, the properties of the attributes defined in the **VARIABLES** clause. Typing information, functional and safety properties are described in this clause. These properties shall remain true in the whole model.Invariants need to be preserved by events clauses.
- **THEOREMS** defines a set of logical expressions that can be deduced from the invariants. They do not need to be proved for each event like for the invariant.
- **EVENTS** defines all the events that occur in a given model. Each event is characterized by its guard (i.e. a first order logic expression involving variables). An event is fired when its guard evaluates to true. If several guards evaluate to true, only one is fired with a non deterministic choice. The events occurring in an Event_B model affect the state described in **VARIABLES** clause.

An Event_B model may refer to a **CONTEXT** describing a static part where all the relevant properties and hypotheses are defined. A **CONTEXT** *c1* consists of the following clauses:

- **SETS** describes a set of abstract and enumerated types.
- **CONSTANTS** represents the constants used by the model.
- **AXIOMS** describes with first order logic expressions, the properties of the attributes defined in the **CONSTANTS** clause. Types and constraints are described in this clause.
- **THEOREMS** are logical expressions that can be deduced from the axioms.

4 From BPEL Designs to Event_B Models

The formal approach we develop in this paper uses BPEL, one of the most popular web services compositions description language, as a web services design language and Event_B as a formal description technique. We will note indifferently *BPEL design* or *BPEL model* to define the web services composition described in the BPEL language.

4.1 The BPEL Web Services Composition Language

BPEL (Business Process Execution Language [16]) is a standardized language for specifying the behavior of a business process based on interactions between a process and its partners. The interaction with each between partner occurs through Web Service interfaces (*partnerLink*). It defines how multiple service interactions, between these partners, are coordinated to achieve a given goal. Each service offered by a partner is described in a WSDL [8] document through a set of *operations* and handled *messages*. These messages are built from *parts* typed in an XML schema.

A BPEL process uses a set of *variables* to represent the messages exchanged between partners. They also represent the state of the business process. The content of these *messages* is amended by a set of *activities* which represent the process flow. This flow specifies the *operations* to be performed, their ordering, activation conditions, reactive rules, etc. The BPEL language offers two categories of activities: 1) *basic activities* representing the primitive operations performed by the process. They are defined by *invoke*, *receive*, *reply*, *assign*, *terminate*, *wait* and *empty* activities. 2) *structured activities* obtained by composing primitive activities and/or other structured activities using the *sequence*, *if*, *while* and *repeat Until* composition operators that model traditional control constructs. Two other composition operators are defined by the *pick* operator defining a nondeterministic choice based on external events (e.g. message reception or timeout) and the *flow* operator defining the concurrent execution of nested activities.

In the remained of this paper, we refer to the decomposition of a structured activity by the activities it contains by *the decomposition operation*.

The *invoke*, *receive*, *reply* and *assign* activities manipulate and modify the contents of the messages and of the data exchanged between partners. The *if*, *while* and *repeat Until* activities evaluate the content of these messages and data to determine the behavior of the BPEL process. BPEL also introduces systematic mechanisms for exceptions and faults handling and also supports the definition of how individual or composite activities, within a unit of work, are compensated in cases of an exception occurrence or a reversal partner request.

BPEL provides the ability to describe two types of processes: executable and abstract. An executable process can be implemented in a programming language and executed by an orchestrator. It describes the desired behavior of the service orchestration. An abstract process represents a view of a BPEL process. It abstracts parts or whole of the process and hides implementation details. An abstract process is not executable.

Finally, an XML representation is associated to the description of every BPEL process. A graphical representation is available and several graphical editors[2,3] offer the capability to design BPEL processes. For the rest of the paper, both representations will be used.

4.2 BPEL and the Event_B Semantics

BPEL defines a language for describing the behavior of a business process based on interactions between the process and its partners. In order to achieve a business goal, a BPEL process defines how multiple service interactions with these partners are coordinated together with the logic of this coordination.

An Event_B model encodes a state transitions system where variables represent the state and events the transitions from one state to another. The refinement capability offered by Event_B makes it possible to introduce more and more design decisions while moving from an abstract level to a less abstract one. The events of a model are atomic events and are interleaved defining event traces. The semantics of an Event_B model is a trace based semantics with interleaving. A system is characterized by the set of licit traces corresponding to the fired events of the model which

[2] BPEL Designer : http://www.eclipse.org/bpel/index.php

[3] Orchestra : http://orchestra.ow2.org/xwiki/bin/view/Main/WebHome

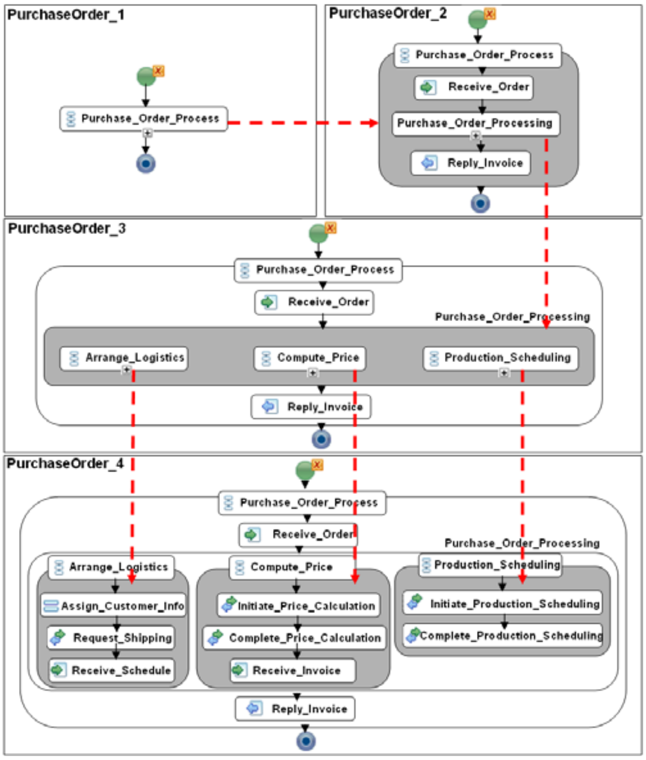


Fig. 2 Graphical editor BPEL decomposition

respects the described properties. The traces define a sequence of states that may be observed by properties.

Our approach for formal modelling of BPEL with Event_B is based on the observation that a BPEL definition is interpreted as a transition system interpreting the processes coordination. A state is represented in both languages by a *variables* element in BPEL and by the VARIABLES clause in Event_B. The various activities of BPEL represent the transitions. They are represented by the events of the EVENTS clause of Event_B.

4.3 A Case Study

The examples used in the following sections are based on the well known case study of the *PurchaseOrder* BPEL specification presented in [16]. This example, issued from electronic commerce, describes a service that processes a purchase order. On receiving the purchase order from a customer, the process initiates three paths concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. When some processes are concurrent, they induce control and data dependencies between the three paths. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three concurrent paths are completed, invoice processing can proceed and the invoice is sent to the customer.

The BPEL model *PurchaseOrder_4* at bottom of Figure 2 shows how the *Purchase_Order_ Process* process is described as a sequence of *Receive_Order*, *Purchase_Order_Processing* and *Reply_Invoice* activities. *Purchase_Order_Processing* is itself decomposed as a flow of *Production_ Scheduling*, *Compute_Price* and *Arrange_Logistics* activities. *Production_Scheduling* is a sequence of *Initiate_ Production_Scheduling* and *Complete_Production_Scheduling* activities. *Compute_Price* is a sequence of *Initiate_Price_Calculation*, *Complete_Price_Calculation* and finally *Receive_Invoice* activities. *Arrange_ Logistics* is a sequence of *Assign_Customer_Info*, *Request_Shipping* and *Receive_Schedule* activities.

When designing the web services composition corresponding to this case study graphical editors tools are set up by the designers. These tools offer two design capabilities.

1. **A one shot web services composition description.** Here the designer produces the whole services composition in a single graphical diagram. The BPEL model named *PurchaseOrder_4* at bottom of figure 2 shows the graphical representation of the case study with a sequence of three services with one flow process.
2. **A stepwise web services composition design.** In this case, the designer uses the decomposition operator offered by the graphical BPEL editor. This operator makes it possible to incrementally introduce more concrete services composing more abstract ones. Figure 2 shows a sequence of three decomposed BPEL models namely *PurchaseOrder_1*, *PurchaseOrder_2*, *PurchaseOrder_3* and *PurchaseOrder_4*.

Nowadays, it is well accepted that formal methods are useful to establish the relevant properties at each step. The presence of refinement during decomposition would help to ensure the correctness of this decomposition. We claim that the Event_B method is a good candidate to bypass these insufficiencies.

4.4 Transforming BPEL Designs to Event_B Models: Horizontal Transformation

This section gives an overview of the horizontal transformation process depicted on figure 3 describing the transformation of any BPEL design to an Event_B model. The proposed approach of [5] is based on the transformation of a BPEL process into an Event_B model in order to check the relevant properties defined in the Event_B models by the services designers. The details of the description of this transformation process and of the rules associated to this transformation process can be found in [5]. For a better understanding of this paper, these rules are briefly recalled below.

This translation process consists of two parts: *static and dynamic*. The BPEL2B tool [6] supports this translation process.

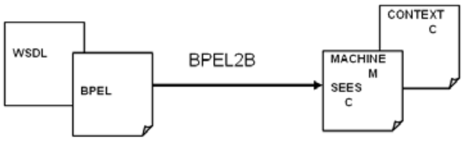
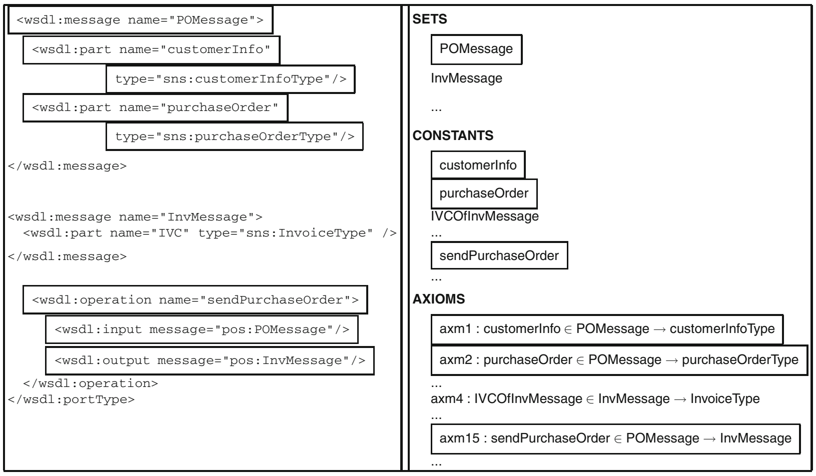


Fig. 3 From BPEL to Event_B: the horizontal transformation



```
<wsdl:message name="POMessage">
  <wsdl:part name="customerInfo"
             type="sns:customerInfoType"/>
  <wsdl:part name="purchaseOrder"
             type="sns:purchaseOrderType"/>
</wsdl:message>


<wsdl:message name="InvMessage">
  <wsdl:part name="IVC" type="sns:InvoiceType" />
</wsdl:message>

  <wsdl:operation name="sendPurchaseOrder">
     <wsdl:input message="pos:POMessage"/>
     <wsdl:output message="pos:InvMessage"/>
  </wsdl:operation>
</wsdl:portType>
```

```
SETS
  POMessage
  InvMessage
  ...
CONSTANTS
  customerInfo
  purchaseOrder
  IVCOfInvMessage
  ...
  sendPurchaseOrder
  ...
AXIOMS
  axm1 : customerInfo ∈ POMessage → customerInfoType
  axm2 : purchaseOrder ∈ POMessage → purchaseOrderType
  ...
  axm4 : IVCOfInvMessage ∈ InvMessage → InvoiceType
  ...
  axm15 : sendPurchaseOrder ∈ POMessage → InvMessage
  ...
```

Fig. 4 A message and a port type modelled by an Event_B CONTEXT

4.4.1 Static Part

The first part translates the WSDL definitions that describe the various web services, their data types, messages and port types (the profile of supported operations) into the different data types and functions offered by Event_B. This part is encoded in the **CONTEXT** part of an Event_B model.

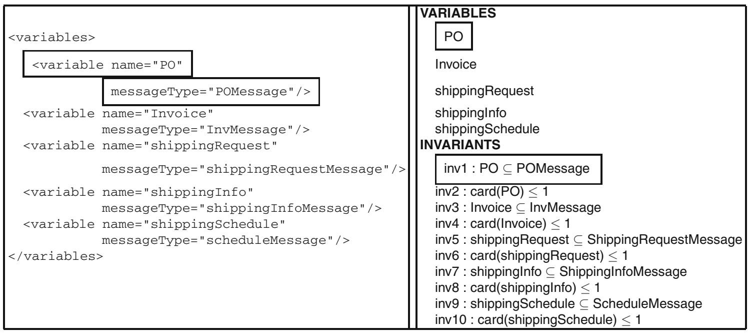
A BPEL process references data types, messages and operations of the port types declared in the WSDL document. In the following, the rules translating these elements into a B CONTEXT are inductively defined.

1- The WSDL *message* element is formalized by an abstract set. Each *part* attribute of a *message* is represented by a functional relation corresponding to the template $part \in message \rightarrow type\_part$ from the *message* type to the part *type*. On the case study of figure 4, the application of this rule corresponds to the functions and axioms $axm1$ and $axm2$ declarations.

2- Each *operation* of a *portType* is represented by a functional relation corresponding to the template $operation \in input \rightarrow output$ from the message type of the *input* attribute to the message type of the *output* attribute. On the case study of figure 4, the application of this rule corresponds to the *SendPurchaseOrder* function and axiom $axm15$ declarations.

4.4.2 Dynamic Part

The second part concerns the description of the orchestration process of the activities appearing in a BPEL description. These processes are formalized as B events, each simple activity becomes an event of the B model and each structured or composed activity is translated to a specific events construction. This part is encoded in a **MACHINE** of an Event_B model.



```
<variables>
  <variable name="PO"
            messageType="POMessage"/>
  <variable name="Invoice"
            messageType="InvMessage"/>
  <variable name="shippingRequest"
            messageType="shippingRequestMessage"/>
  <variable name="shippingInfo"
            messageType="shippingInfoMessage"/>
  <variable name="shippingSchedule"
            messageType="scheduleMessage"/>
</variables>
```

```
VARIABLES
  PO
  Invoice
  shippingRequest
  shippingInfo
  shippingSchedule
INVARIANTS
  inv1 : PO ⊆ POMessage
  inv2 : card(PO) ≤ 1
  inv3 : Invoice ⊆ InvMessage
  inv4 : card(Invoice) ≤ 1
  inv5 : shippingRequest ⊆ ShippingRequestMessage
  inv6 : card(shippingRequest) ≤ 1
  inv7 : shippingInfo ⊆ ShippingInfoMessage
  inv8 : card(shippingInfo) ≤ 1
  inv9 : shippingSchedule ⊆ ScheduleMessage
  inv10 : card(shippingSchedule) ≤ 1
```

Fig. 5 BPEL variables modelled in the VARIABLES clause.

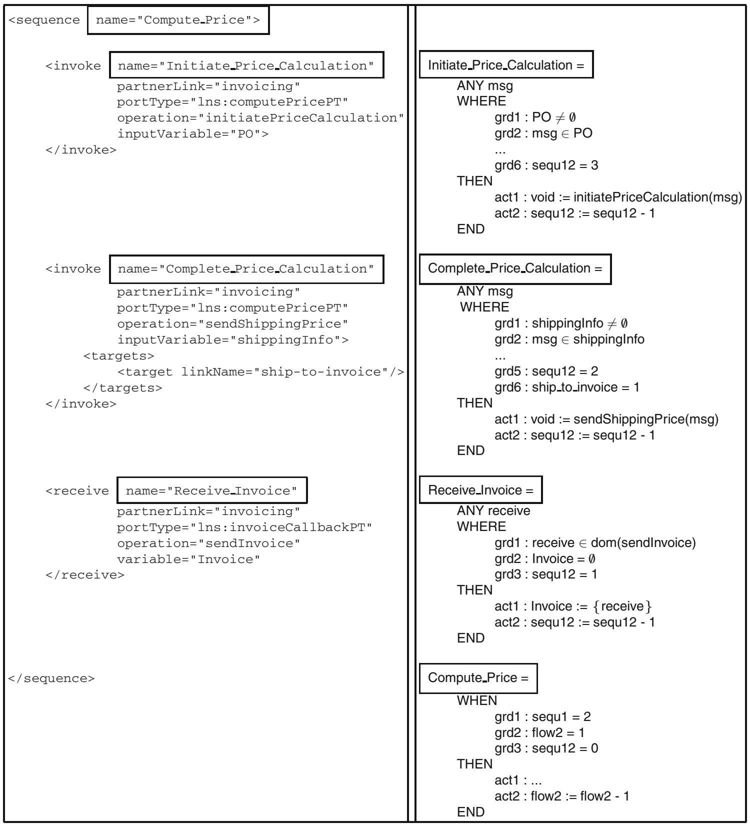
A BPEL process is composed of a set of variables and a set of activities. Each BPEL variable corresponds to a state variable in the VARIABLES clause, and the activities are encoded by events. This transformation process is inductively defined on the structure of a BPEL process according to the following rules.

3- The BPEL *variable element* is represented by a variable in the VARIABLES clause in an Event_B **MACHINE**. This variable is typed in the INVARIANTS clause using *messageType* BPEL attribute. The variables and invariants corresponding to the case study are given on figure 5. For example the BPEL variable *PO* is declared and typed.

4- Each BPEL *simple activity* is represented by a single event in the EVENTS clause of the Event_B **MACHINE**. For example, on figure 6 the *ReceiveInvoice* BPEL atomic activity is encoded by the *ReceiveInvoice* B event. It shows that the operation *SendInvoice* declared in the CONTEXT is referred.

5- Each BPEL *structured activity* is modelled by an Event_B description which encodes the carried composition operator. Modelling composition operations in Event_B follows the modelling rules formally defined in [4]. Again, on figure 6 the structured activity *Compute_Price* is encoded by a sequence of three atomic activities. A variant $sequ12$ is generated, it decreases from value 3 to the final value 0 in the guard of the *Compute_Price* event.

Notice that all the variables, invariants, constants, typing expressions, events and guards are automatically generated from the information available in BPEL designs by the the BPEL2B tool [6].



```
<sequence name="Compute_Price">

    <invoke name="Initiate_Price_Calculation"
            partnerLink="invoicing"
            portType="lns:computePricePT"
            operation="initiatePriceCalculation"
            inputVariable="PO">
    </invoke>

    <invoke name="Complete_Price_Calculation"
            partnerLink="invoicing"
            portType="lns:computePricePT"
            operation="sendShippingPrice"
            inputVariable="shippingInfo">
        <targets>
            <target linkName="ship-to-invoice"/>
        </targets>
    </invoke>

    <receive name="Receive_Invoice"
            partnerLink="invoicing"
            portType="lns:invoiceCallbackPT"
            operation="sendInvoice"
            variable="Invoice"
    </receive>

</sequence>
```

```
Initiate_Price_Calculation =
    ANY msg
    WHERE
        grd1 : PO ≠ ∅
        grd2 : msg ∈ PO
        ...
        grd6 : sequ12 = 3
    THEN
        act1 : void := initiatePriceCalculation(msg)
        act2 : sequ12 := sequ12 - 1
    END

Complete_Price_Calculation =
    ANY msg
    WHERE
        grd1 : shippingInfo ≠ ∅
        grd2 : msg ∈ shippingInfo
        ...
        grd5 : sequ12 = 2
        grd6 : ship_to_invoice = 1
    THEN
        act1 : void := sendShippingPrice(msg)
        act2 : sequ12 := sequ12 - 1
    END

Receive_Invoice =
    ANY receive
    WHERE
        grd1 : receive ∈ dom(sendInvoice)
        grd2 : Invoice = ∅
        grd3 : sequ12 = 1
    THEN
        act1 : Invoice := {receive}
        act2 : sequ12 := sequ12 - 1
    END

Compute_Price =
    WHEN
        grd1 : sequ1 = 2
        grd2 : flow2 = 1
        grd3 : sequ12 = 0
    THEN
        act1 : ...
        act2 : flow2 := flow2 - 1
    END
```

Fig. 6 Atomic and composite (sequence) BPEL activities and their corresponding Event_B representation.

5 From BPEL to Event_B: A Refinement Based Methodology

The previous section addressed the transformation of BPEL models to Event_B models. This transformation process, named *horizontal transformation* produces an Event_B model from any BPEL model whatever is the reached level of the design on the BPEL side. As outlined before, this approach may lead to complex Event_B models with complex proof obligations requiring interactive proofs. It neither takes into account the stepwise decomposition, shown on figure 2, supported by the BPEL descriptions and encoded by the decomposition operator, nor the refinement capability offered by the Event_B method. This section presents a methodology, depicted on figure 7, that exploits Event_B refinement. This section represents the main contribution of this paper.

5.1 The Methodology: Vertical Decomposition

An incremental construction of the BPEL specification using process decomposition operations will help to build reusable BPEL specifications and less complex Event_B models. Our claim is to encode each BPEL decomposition operation (adding new activities) by the refinement development operation of Event_B (adding new events). As depicted on figure 7, we get on the one hand a sequence of abstract BPEL specifications, one being the decomposition of the other, and on the other hand a sequence of proved Event_B refinements corresponding to the abstract BPEL specification. Then a formal stepwise refinement of BPEL specification is obtained. From the methodological point of view, the last obtained BPEL process description is the one that is deployed for execution by the orchestrator in charge of execution.

5.2 Simultaneous BPEL Models Decomposition and Event_B Models Refinement

As shown on figures 2 and 7, when an abstract BPEL process $bpel_i$ contains one or more structured activities, these structured activities are decomposed other sub-activities. We get a new BPEL process $bpel_{i+1}$ with more details. Each abstract BPEL process $bpel_i$ interacts with a set of services that are described in a WSDL document $wsdl_i$.

When the transformation rules proposed in sections 4.4.1 and 4.4.2 are applied to each abstract BPEL process and WSDL document, a set of Event_B machines and a set of Event_B contexts are produced. Each $bpel_i$ process is associated to an Event_B MACHINE m_i and each $wsdl_i$ document is associated to an Event_B CONTEXT c_i. When the $bpel_{i+1}$ process is obtained after decomposing the $bpel_i$ process, the m_{i+1} MACHINE refines the m_i MACHINE.

Each MACHINE m_i refers to its CONTEXT c_i. More precisely, when a MACHINE m_{i+1} refines a MACHINE m_i, the CONTEXT c_i referred by the MACHINE m_i is extended by a CONTEXT c_{i+1} by adding definitions of the services introduced by the MACHINE m_{i+1}. Figure 7 illustrates this approach.

5.3 Application to the Case Study

When applied to our case study, the proposed methodology leads to a development of a sequence of 4 Event_B machines, each one refining the previous one. As depicted on the right hand side of figure 7, the Event_B machines *PurchaseOrder_1*, *PurchaseOrder_2*, *PurchaseOrder_3* and *PurchaseOrder_4* define the development of our case study. In order to show the benefits of this approach, we have chosen to comment below the machines obtained after the third refinement.

5.3.1 The Second Refinement

Figure 8 shows part of a machine obtained after refinement. The event *Compute_Price* computes the total amount of the invoice to be produced at the end

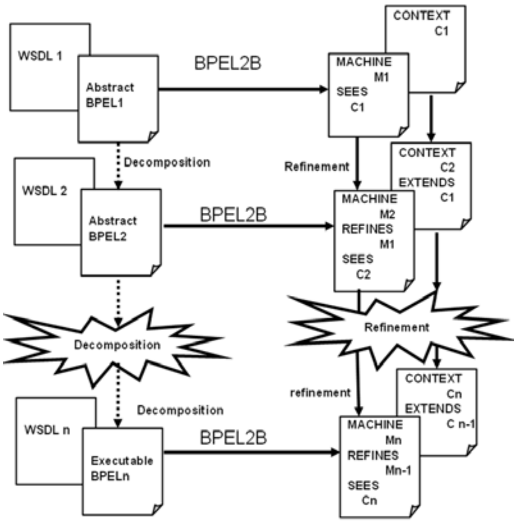
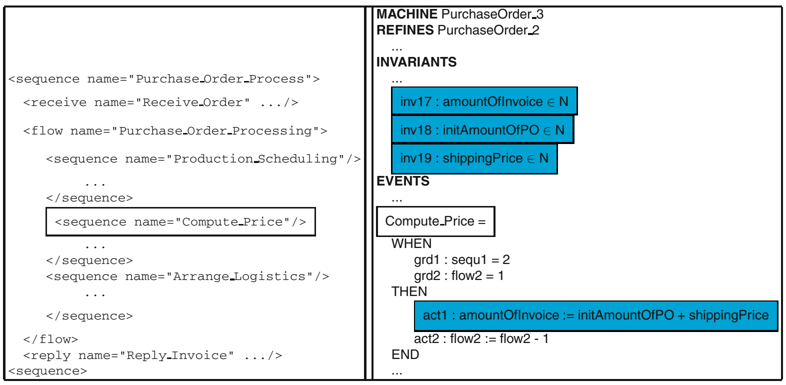


Fig. 7 A refinement based methodology: vertical decomposition.

of shipping. This machine is completed by the relevant resources, represented in grey square boxes, needed to ensure the correct computation of the total amount of the invoice. The invariants *inv*17, *inv*18 and *inv*19 and the action *act*1 are interactively added by the developer. There is no way to generate them from the BPEL code since they are not available.

5.3.2 The Third Refinement

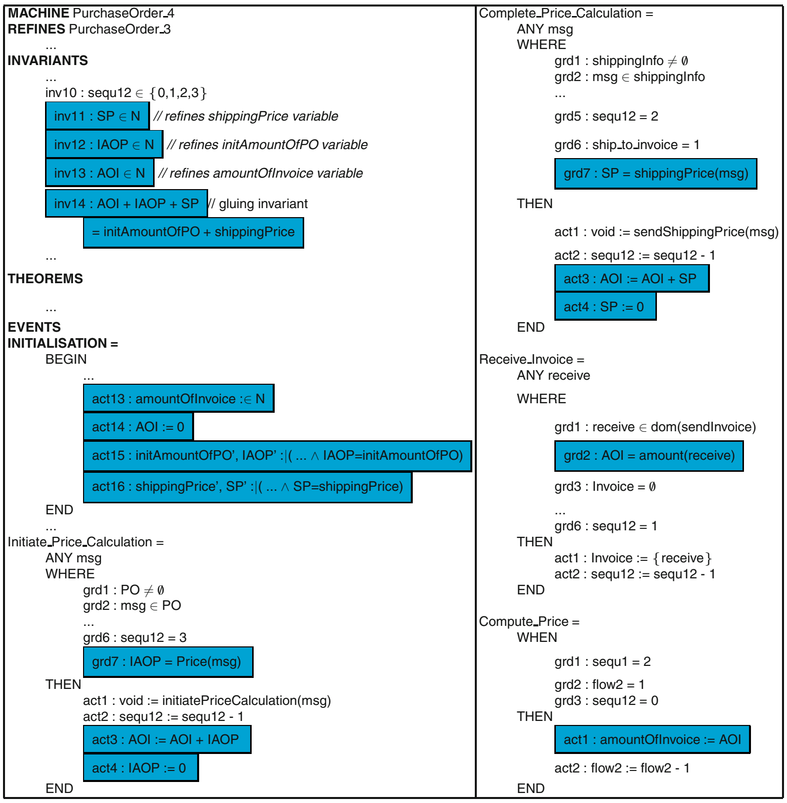
The machine *PurchaseOrder_4* refines the machine *PurchaseOrder_3* as shown on figure 9. Again, the grey square boxes show the information completed by the



```
<sequence name="Purchase_Order_Process">
  <receive name="Receive_Order" .../>
  <flow name="Purchase_Order_Processing">
     <sequence name="Production_Scheduling"/>
          ...
     </sequence>
     <sequence name="Compute_Price"/>
          ...
     </sequence>
     <sequence name="Arrange_Logistics"/>
          ...
     </sequence>
  </flow>
  <reply name="Reply_Invoice" .../>
<sequence>
```

```
MACHINE PurchaseOrder_3
REFINES PurchaseOrder_2
  ...
INVARIANTS
  ...
  inv17 : amountOfInvoice ∈ N
  inv18 : initAmountOfPO ∈ N
  inv19 : shippingPrice ∈ N
EVENTS
  ...
Compute_Price =
  WHEN
     grd1 : sequ1 = 2
     grd2 : flow2 = 1
  THEN
     act1 : amountOfInvoice := initAmountOfPO + shippingPrice
     act2 : flow2 := flow2 - 1
  END
  ...
```

Fig. 8 The second refinement focussing on the Compute_Price event.



```
MACHINE PurchaseOrder_4
REFINES PurchaseOrder_3
        ...
INVARIANTS
        ...
        inv10 : sequ12 ∈ {0,1,2,3}
        inv11 : SP ∈ N          // refines shippingPrice variable
        inv12 : IAOP ∈ N        // refines initAmountOfPO variable
        inv13 : AOI ∈ N         // refines amountOfInvoice variable
        inv14 : AOI + IAOP + SP // gluing invariant
                = initAmountOfPO + shippingPrice
        ...
THEOREMS
        ...
EVENTS
INITIALISATION =
        BEGIN
                ...
                act13 : amountOfInvoice :∈ N
                act14 : AOI := 0
                act15 : initAmountOfPO', IAOP' :|( ... ∧ IAOP=initAmountOfPO)
                act16 : shippingPrice', SP' :|( ... ∧ SP=shippingPrice)
        END
        ...
Initiate_Price_Calculation =
        ANY msg
        WHERE
                grd1 : PO ≠ ∅
                grd2 : msg ∈ PO
                ...
                grd6 : sequ12 = 3
                grd7 : IAOP = Price(msg)
        THEN
                act1 : void := initiatePriceCalculation(msg)
                act2 : sequ12 := sequ12 - 1
                act3 : AOI := AOI + IAOP
                act4 : IAOP := 0
        END

Complete_Price_Calculation =
        ANY msg
        WHERE
                grd1 : shippingInfo ≠ ∅
                grd2 : msg ∈ shippingInfo
                ...
                grd5 : sequ12 = 2
                grd6 : ship_to_invoice = 1
                grd7 : SP = shippingPrice(msg)
        THEN
                act1 : void := sendShippingPrice(msg)
                act2 : sequ12 := sequ12 - 1
                act3 : AOI := AOI + SP
                act4 : SP := 0
        END
Receive_Invoice =
        ANY receive
        WHERE
                grd1 : receive ∈ dom(sendInvoice)
                grd2 : AOI = amount(receive)
                grd3 : Invoice = ∅
                ...
                grd6 : sequ12 = 1
        THEN
                act1 : Invoice := {receive}
                act2 : sequ12 := sequ12 - 1
        END
Compute_Price =
        WHEN
                grd1 : sequ1 = 2
                grd2 : flow2 = 1
                grd3 : sequ12 = 0
        THEN
                act1 : amountOfInvoice := AOI
                act2 : flow2 := flow2 - 1
        END
```

Fig. 9 The third refinement showing the sequence decomposing the Compute_Price event and the gluing invariant.

developer in order to ensure the correctness of the development. The concrete variables *SP*, *IAOP* and *AOI* of the invariants *inv*11, *inv*12 and *inv*13 together with the gluing invariant *inv*14 and the guards *grd*7 and *grd*2 are introduced. The introduction of the *grd*7 and *grd*2 guards of the *Complete_Price_Calculation* and *Receive_Invoice* events, ensures that the concrete variables introduced for defining the gluing invariant are correctly related to the exchanged messages produced by the functions carried by the specific services.

The actions modifying these variables are also defined. They show how the variables evolve. The sequencing of the events *Initiate_Price_Calculation*, *Complete-_Price_Calculation* and *Receive_ Invoice* supplying the result *amountOfInvoice* to the *Compute_Price* refined event remains automatically produced.

The refinement of figure 9 represents the last refinement. The corresponding BPEL document (a part of this document is given on figure 6) is the one deployed by the orchestrator in charge of running it.

6 Validation of Services Properties

When the Event_B models formalizing a BPEL description are obtained, we have seen in the previous section that they may be enriched by the relevant properties that formalize the user requirements and the soundness of the BPEL defined process. In Event_B, these properties are defined in the AXIOMS, INVARIANTS and THEOREMS clauses. Preconditions and guards are added to define the correct event triggering. Our work considers arbitrary sets of values for parameters defining their types. There is no abstraction related to the parameters, preconditions, postconditions nor join operations. The expressions are represented as they appear in BPEL. The proof based approach we propose does not suffer from the growing number of explored states. More precisely, regarding the formal verification of properties, our contribution is summarized in the following points.

- *BPEL type control.* Static properties are described in the CONTEXT of the Event_B model. They concern the description of services, messages and their corresponding types (WSDL part). Event_B ensures a correct description of the types and function composition.
- *Orchestration and services composition.* Dynamic properties are described in the MACHINE of an Event_B model and concern the variables (messages exchanged between services) and the BPEL process behavior (BPEL part). The introduction of variants guarantee the correct services triggering order and message passing.
- *Deadlock freeness.* It is always possible to trigger at least one event. This property is ensured by asserting (in the THEOREMS clause) that the disjunction of all the abstract events guards implies the disjunction of all the concrete events guards. It ensures that if at least one guard of the abstract Event_B model is true, then at least one guard of the refined model is true.
- *No LiveLock.* A decreasing variant is introduced for the purpose of the definition of each refinement corresponding to the encoding of a composition operator. When this variant reaches the value 0, another event may be triggered.
- *Pre-condition for calling a service operation: input message is not empty.* In the orchestration tools, the condition for triggering a BPEL activity is the correct reception of the message used by this activity. Our representation of the call of a service operation takes into account this condition in the events guards corresponding to this activity and an invariant guarantees the existence of this state.
- *Data transformation.* Data properties are expressed in the in the INVARIANTS and AXIOMS clauses. They are checked for each triggered event. This ensures a correct manipulation and transformation of data and messages exchanged between all the participants (partners).
- *Transactional properties.* When modelling fault and compensation handlers by a set of events [5], it becomes possible to model and check properties related to transactional web services.

But, one of the major interests of the proposed methodology is error reporting. Indeed, when an Event_B model cannot be proved due to an erroneous BPEL design and/or decomposition, it is possible to report, on the BPEL code, the occurred error in the concerned activity. This functionality is very helpful for designers that are usually non specialists of formal methods.

Since each BPEL concept is attached to a single Event_B element preserving the same name as in the original BPEL design, it becomes possible to identify, localize and visualize the BPEL concept that corresponds to the B element whose proof obligations cannot be discharged. Notice, that this reporting is only possible for the Event_B parts generated from the BPEL design.

7 Conclusion and Perspectives

This paper presented a fully formalized method for designing correct web services compositions expressed in the BPEL description language. Our contribution addresses both technical and methodological points of view.

From the technical point of view, this approach consists in encoding each BPEL services composition description by an Event_B model in which relevant properties related to deadlock or livelock, data transformation, messages consistence or transactions are modelled. Establishing these properties requires an enrichment of the Event_B models by the relevant information that are not available in the original BPEL description.

From the methodological point of view, our approach suggests to encode the decomposition relationship available on the BPEL modelling side. As a result, the refinement chain of Event_B models follows the decomposition process offered by the BPEL description language. The interest of this approach is double. On the BPEL side it offers a stepwise design approach while it eases the proof activity on the Event_B side since the proof obligations become simpler thanks to the presence of gluing invariants.

Moreover, regarding the approaches developed in the literature, our work covers the whole characteristics of the formal verification of web services compositions. Indeed, the Event_B models we generate support the validation of the properties related to both data (transformation of data) and services (services orchestration).

The BPEL2B tool, presented as an Eclipse PlugIn, encodes the transformation process described in this paper and contributes to the dissemination of formal methods. The details of the formal modelling activities are hidden to the BPEL designer.

Finally, this work opens several perspectives. We consider that two of them need to be quoted. The first one relates to the properties formalization. It is important to formalize more relevant properties. We plan to work more on the transaction based properties. The second one is related to the semantics carried by the services. Indeed, composing in sequence a service that produces distances expressed in centimeters with another one composing services expressed in inches should not be a valid composition. Up to now, our approach is not capable to detect such inconsistencies. Formal knowledge models, ontologies for example, expressed beside Event_B models could be a path to investigate.

References

1. van-der Aalst, W., Mooij, A., Stahl, C., Wolf, K.: Service interaction: Patterns, formalization, and analysis. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 42–88. Springer, Heidelberg (2009)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, cambridge edn. Cambridge University Press, Cambridge (2010)
3. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundamenta Informaticae 77, 1–28 (2007)
4. Aït-Ameur, Y., Baron, M., Kamel, N., Mota, J.-M.: Encoding a process algebra using the Event B method. Software Tools and Technology Transfer 11(3), 239–253 (2009)
5. Ait-Sadoune, I., Ait-Ameur, Y.: A Proof Based Approach for Modelling and Veryfing Web Services Compositions. In: 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, Potsdam, Germany, pp. 1–10 (2009)
6. Ait-Sadoune, I., Ait-Ameur, Y.: From BPEL to Event_B. In: Integration of Model-based Formal Methods and Tools Workshop (IM_FMT 2009), Dusseldorf, Germany (February 2009)
7. van Breugel, F., Koshkina, M.: Models and Verification of BPEL. Draft (2006)
8. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL). W3C Recommendation Ver. 1.1, W3C (2001)
9. Consortium, W.W.W.: Web Services Choreography Description Language (WS-CDL). W3C Recommendation Version 1.0, W3C (2005)
10. Fahland, D.: Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. Tech. rep., Humboldt-Universitat zu Berlin, Institut fur Informatik, Germany (2005)
11. Farahbod, R., Glsser, U., Vajihollahi, M.: A formal semantics for the Business Process Execution Language for Web Services. In: Web Services and Model-Driven Enterprise Information Services (2005)
12. Foster, H.: A Rigorous Approach To Engineering Web Service Compositions. Ph.D. thesis, Imperial College London, University of London (2006)
13. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based Verification of Web Service Composition. In: IEEE International Conference on Automated Software Engineering (2003)
14. Group, O.M.: Business Process Model and Notation (BPMN). OMG Document Number: dtc/2009-08-14 FTF Beta 1 for Version 2.0, Object Manager Group (2009)
15. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri-Nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 220–235. Springer, Heidelberg (2005)
16. Jordan, D., Evdemon, J.: Web Services Business Process Execution Language (WS-BPEL). Standard Version 2.0, OASIS (2007)
17. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: Web Services and Formal Methods International Workshop, WSFM 2007 (2007)
18. Marconi, A., Pistore, M.: Synthesis and Composition of Web Services. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 89–157. Springer, Heidelberg (2009)
19. Marconi, A., Pistore, M., Traverso, P.: Specifying Data-Flow Requirements for the Automated Composition of Web Services. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), Pune, India (September 2006)
20. Nakajima, S.: Lightweight Formal Analysis of Web Service Flows. Progress in Informatics 2 (2005)

21. Nakajima, S.: Model-Checking Behavioral Specifications of BPEL Applications. In: WLFM 2005 (2005)
22. Salaun, G., Bordeaux, L., Schaerf, M.: Describing and Reasoning on Web Services using Process Algebra. In: IEEE International Conference on Web Service, ICWS 2004 (2004)
23. Salaun, G., Ferrara, A., Chirichiello, A.: Negotiation among web services using LOTOS/CADP. In: Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 198–212. Springer, Heidelberg (2004)
24. Specification, T.W.M.C.: Process Definition Interface – XML Process Definition Language (XPDL). Document Number WFMC-TC-1025 Version 2.1a, The Workflow Management Coalition (2008)
25. Verbeek, H., van-der Aalst, W.: Analyzing BPEL processes using Petri-Nets. In: Second International Workshop on Application of Petri-Nets to Coordination, Workflow and Business Process Management (2005)