

# Formal Modelling and Verification of Transactional Web Service Composition: A Refinement and Proof Approach with Event-B

Idir Ait-Sadoune and Yamine Ait-Ameur

**Abstract** Several languages for describing Web service compositions, like BPEL (Business Process Execution Language), make use of fault and compensation constructs to handle internal and/or external runtime errors of the composed service. This situation particularly occurs for transactional services. However, the absence of a rigorous definition of these BPEL constructors makes it difficult to correctly define the transactional behaviour of a BPEL process. The definitions of such constructs are usually given by their informal descriptions available in the standards. Our contribution proposes an approach to formally define the semantics of these operators. Thus, this paper presents a new Event-B semantics for formal modelling of Web service compositions that covers the scope, the fault and the compensation handlers introduced by the BPEL language specification. It also proposes a methodology showing how we can use Event-B method to design transactional BPEL processes. The proposed approach is illustrated by a case study.

## 1 Introduction

Service-oriented architectures (SOA) are increasingly used in various application domains. Indeed, a wide range of services operate on the Web and access different distributed and shared resources like databases, data warehouses and scientific calculation repositories. Moreover, the compositions of such services define complex systems not only in the area of computing but also in various businesses like manufacturing or scheduling. Some of these services performing transactional activities are called transactional Web services. This kind of services must satisfy the relevant properties related to transactional systems (atomicity, consistency, isolation and durability (ACID) properties) traditionally required by

---

I. Ait-Sadoune (✉)

LRI - CentraleSupélec, 3, rue Joliot-Curie, 91190 Gif-Sur-Yvette, France

e-mail: [idir.aitsadoune@supelec.fr](mailto:idir.aitsadoune@supelec.fr)

Y. Ait-Ameur

IRIT - ENSEEIHT, Toulouse, France

e-mail: [yamine@n7.fr](mailto:yamine@n7.fr)

© Springer International Publishing Switzerland 2015

B. Thalheim et al. (eds.), *Correct Software in Web Applications and Web Services*,

Texts & Monographs in Symbolic Computation,

DOI 10.1007/978-3-319-17112-8\_1

database management systems. Although the notion of transactions is well mastered by traditional shared and distributed databases, SOA suffers from a lack of formal semantics of transactional Web services. Indeed, in the current SOA tools, overall business transactions may fail or be cancelled when many ACID transactions are committed.

BPEL (Business Process Execution Language [22]) is considered as a standard of Web service composition specification. It offers a compensation mechanism by providing resources for flexible control of reversal and/or resume activities. To reach this goal, BPEL offers the possibility to define fault handling and compensation in an application-specific manner. BPEL has an XML-based textual representation, and its dynamic semantic description is still informally defined in the standards. Due to the lack of formal semantics of BPEL, ambiguous interpretations remain possible, especially the use of different mechanisms (fault handlers and compensation handlers) to handle transactional behaviour. Such languages do not offer the capability to formally handle the transactional Web service aspects and to ensure their correct behaviour.

Several approaches have proposed formal semantics to the BPEL language using different formal descriptions. Petri nets, transition systems, pi-calculus and process algebra have been set up to model BPEL processes; they are summarised in Sect. 8. In our previous work ([5, 6]), we have defined an Event-B-based [2] semantics for the various elements of data and service descriptions and for simple and structured BPEL activities. In this paper, we propose to extend this work to cover the scope, the fault and the compensation handlers introduced by the BPEL language specification. We also propose a methodology to design a transactional BPEL process by assisting a designer for detecting and verifying a transactional behaviour in a BPEL process.

This paper is structured as follows. The next section presents the Event-B method, and Sect. 3 gives an overview of the BPEL language and the case study used in this paper to illustrate the proposed approach. Sections 4, 5 and 6 present, respectively, the proposed approach for analysing transactional Web services and how Event-B method is used to encode these scope, fault and compensation constructs. Section 7 presents an application of the proposed approach to a case study. Section 8 discusses our approach compared to the state of the art in formal verification of BPEL processes and transactional Web services. Finally, a conclusion and some perspectives are outlined.

## 2 The Event-B Method

The Event-B method [2] is a recent evolution of the B method [1]. This method is based on the notions of preconditions and post-conditions [16], the weakest precondition [9] and the calculus of substitution [1]. It is a formal method based on first-order logic and set theory.

## 2.1 Event-B Model

An Event-B model is defined by a set of variables, defined in the VARIABLES clause that evolves thanks to events defined in the EVENTS clause. It encodes a state transition system where the variables represent the state and the events represent the transitions from one state to another.

An Event-B model is made of several components of two kinds: Machines and Contexts. The Machines contain the dynamic parts (states and transitions) of a model, whereas the Contexts contain the static parts (axiomatisation and theories) of a model. A Machine can be refined by another one, and a Context can be extended by another one. Moreover, a Machine can see one or several Contexts (Fig. 1).

The refinement operation [3] offered by Event-B encodes model decomposition. A transition system is decomposed into another transition system with more and more design decisions while moving from an abstract level to a less abstract one. A refined Machine is defined by adding new events, new state variables and a gluing invariant. Each event of the abstract model is refined in the concrete model by adding new information expressing how the new set of variables and the new events evolve.

A Context is defined by a set of clauses (Fig. 2) as follows:

- CONTEXT represents the name of the component that should be unique in a model.
- EXTENDS declares the Context extended by the described Context.

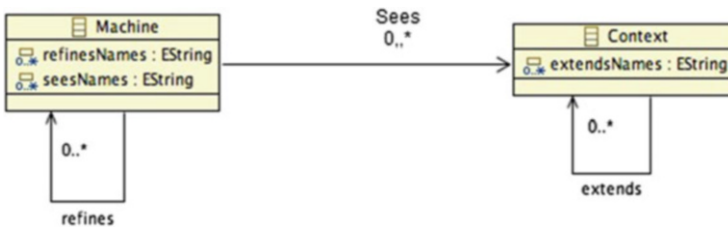


Fig. 1 MACHINE and CONTEXT relationships

CONTEXT	<i>context_identifier<sub>1</sub></i>	MACHINE	<i>machine_identifier<sub>1</sub></i>
EXTENDS	<i>context_identifier<sub>2</sub></i>	REFINES	<i>machine_identifier<sub>2</sub></i>
SETS		SEES	
<i>s</i>		<i>context_identifier<sub>1</sub></i>	
CONSTANTS		VARIABLES	
<i>c</i>		INVARIANTS	
AXIOMS		<i>inv</i> : <i>I(s, c, v)</i>	
<i>axm</i> :	<i>A(s, c)</i>	THEOREMS	
THEOREMS		<i>thm</i> :	<i>T(s, c, v)</i>
<i>thm</i> :	<i>T(s, c)</i>	VARIANT	
END		<i>V(s, c, v)</i>	
		EVENTS	
		< <i>event_list</i> >	
		END	

Fig. 2 The structure of an Event-B development

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

Similarly to Contexts, a Machine is defined by a set of clauses (Fig. 2). Briefly, the clauses mean:

- MACHINE represents the name of the component that should be unique in a model.
- REFINES declares the Machine refined by the described Machine.
- SEES declares the list of Contexts imported by the described Machine.
- VARIABLES represents the state variables of the model of the specification. Refinement may introduce new variables in order to enrich the described system.
- INVARIANTS describes, by first-order logic expressions, the properties of the variables defined in the VARIABLES clause. Typing information, functional and safety properties are usually described in this clause. These properties shall remain true in the whole model. Invariants need to be preserved by events. It also expresses the gluing invariant required by each refinement for property preservation.
- 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.
- VARIANT introduces a decreasing natural number which states that the “convergent” events terminate.
- EVENTS defines all the events (transitions) that occur in a given model. Each event is characterised by its guard and by the actions performed when the guard is true. Each Machine must contain an “Initialisation” event. The events occurring in an Event-B model affect the state described in VARIABLES clause.

An event consists of the following clauses (Fig. 3):

- *status* can be “ordinary”, “convergent” (the event has to decrease the variant) or “anticipated” (the event must not increase the variant).
- *refines* declares the list of events refined by the described event.
- *any* lists the parameters of the event.

Fig. 3 Event structure

Event	$evt \hat{=}$
Status	convergent
any	
	$x$
where	
	grd : $G(s, c, v, x)$
then	
	act : $v :  BA(s, c, v, x, v')$
end	

$\langle \text{variable\_identifier} \rangle$	$:= \langle \text{expression} \rangle$	(1)
$\langle \text{variable\_identifier\_list} \rangle$	$:  \langle \text{before\_after\_predicate} \rangle$	(2)
$\langle \text{variable\_identifier} \rangle$	$:\in \langle \text{set\_expression} \rangle$	(3)

**Fig. 4** The kinds of actions of an event

- *where* expresses the guard of the event. 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.
- *then* contains the actions of the event that are used to modify the state variables.

Event-B offers three kinds of actions that can be deterministic or not (Fig. 4). For the first case, the deterministic action is represented by the “assignment” operator that modifies the value of a variable. This operator is illustrated by the action (1). For the case of the non-deterministic actions, the action (2) represents the “before-after” operator acting on a set of variables whose effect is represented by a predicate, expressing the relationship between the contents of variables before and after the triggering of the action. Finally, the action (3) represents the non-deterministic choice operator, acting on a variable, by modifying its content with an undetermined value in a set of values.

## 2.2 Proof Obligation Rules

Proof obligations (POs) are associated to any Event-B model. They are automatically generated. The *proof obligation generator plug-in* in the Rodin platform [24] is in charge of generating them. These POs need to be proved in order to ensure the correctness of developments and refinements. The obtained PO can be proved automatically or interactively by the *prover plug-in* in the Rodin platform.

The rules for generating proof obligations follow the substitution calculus [1] close to the weakest precondition calculus [9]. In order to define some proof obligation rules, we use the notations defined in Figs. 2 and 3 where  $s$  denotes the seen sets,  $c$  the seen constants and  $v$  the variables of the Machine. Seen axioms are denoted by  $A(s, c)$  and theorems by  $T(s, c)$ , whereas invariants are denoted by  $I(s, c, v)$  and local theorems by  $T(s, c, v)$ . For an event  $evt$ , the guard is denoted by  $G(s, c, v, x)$ , and the action is denoted by the before-after predicate  $BA(s, c, v, x, v')$  (the action (2) of Fig. 4).

- *The theorem proof obligation rule*: this rule ensures that a proposed context or machine theorem is indeed provable:

$$A(s, c) \Rightarrow T(s, c)$$

$$A(s, c) \wedge I(s, c, v) \Rightarrow T(s, c, v)$$

- *Invariant preservation proof obligation rule*: this rule ensures that each invariant in a machine is preserved by each event:

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \Rightarrow I(s, c, v')$$

- *Feasibility proof obligation rule*: the purpose of this proof obligation is to ensure that a non-deterministic action is feasible:

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \Rightarrow \exists v'. BA(s, c, v, x, v')$$

- *The numeric variant proof obligation rule*: this rule ensures that under the guards of each convergent or anticipated event, a proposed numeric variant is indeed a natural number:

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \Rightarrow V(s, c, v) \in \mathbb{N}$$

- *The variant proof obligation rule*: this rule ensures that each convergent event decreases the proposed numeric variant. It also ensures that each anticipated event does not increase the proposed numeric variant. The rule in the case of a convergent event is

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \Rightarrow V(s, c, v') < V(s, c, v)$$

The rule in the case of an anticipated event is

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \Rightarrow V(s, c, v') \leq V(s, c, v)$$

There are other rules for generating proof obligations to prove the correctness of refinement. These rules are given in [2].

### 2.3 Semantics of Event-B Models

The new aspect of the Event-B method [2], in comparison with classical B [1], is related to the semantics. Indeed, the events of a model are atomic events of state transition systems. The semantics of an Event-B model is trace-based semantics with interleaving. A system is characterised by the set of licit traces corresponding to the fired events of the model which respects the described properties. The traces define a sequence of states that may be observed by properties. All the properties will be expressed on these traces.

This approach proved the capability to represent event-based systems like railway systems, embedded systems or Web services. Moreover, decomposition (thanks to refinement) supports building of complex systems gradually in an incremental manner by preserving the initial properties, thanks to the preservation of a gluing invariant.

### 3 Service Composition Description Languages

The Web service composition description languages are XML-based languages. The most popular languages are BPEL [22], CDL [30], OWL-S [29], BPMN [23] and XPD [31]. If these languages are different from the description point of view, they share several concepts, in particular the service composition. Among the shared 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 event handling. These elements are essential to describe service compositions and their behaviour.

However, due to their XML-based definition, these languages suffer from a lack of semantics. It is usually informally expressed in the standards that describe these languages using natural language or semiformal notations. Hence, the need of a formal semantics expressing this semantics emerged. Formal description techniques and the corresponding verification procedures are very good candidates for expressing the semantics and for verifying the relevant properties.

The formal approach we develop in this paper uses BPEL as a service composition description language and Event-B as a formal description technique. The proposed approach can be extended to be used with other service composition description language.

#### 3.1 Overview of BPEL

BPEL (Business Process Execution Language [22]) is a standardised language for specifying the behaviour of a business process based on interactions between a process and its service partners (*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 document through a set of *operations* and of handled *messages*.

WSDL (Web Service Description Language [28]) is a standardised language for describing the published interface (input and output parameter types) of the Web service. It provides with the address of the described service, its identity, the operations that can be invoked and the operation parameters and their types. Thus, WSDL describes the function provided by Web service operations. It defines the exchanged messages that envelop the exchanged data and parameters. The composition of such Web services is described in languages, like BPEL, supporting such composition operators.

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. Figures 5 and 6 show the XML structure of a WSDL description and a BPEL process.

```

<message name="InfosMessage">
  <part name="DebitInfosPart" type="DebitInfos"/>
  <part name="CreditInfosPart" type="CreditInfos"/>
</message>
<message name="ResponseMessage">
  <part name="ResponsePart" type="int"/>
</message>
<message name="DebitMessage">
  <part name="DebitInfosPart" type="DebitInfos"/>
</message>
<message name="CreditMessage">
  <part name="CreditInfosPart" type="CreditInfos"/>
</message>
<portType name="BankTransferPortType">
  <operation name="BankTransferOperation">
    <input message="InfosMessage"/>
    <output message="InfosOut"/>
  </operation>
  <operation name="DebitOperation">
    <input message="DebitMessage"/>
  </operation>
  <operation name="CreditOperation">
    <input message="CreditMessage"/>
  </operation>
</portType>

```

**Fig. 5** The XML WSDL (Web Service Description Language) description of *BankTransfer* services

```

<process name="BankTransfer" ...>
  <variables>
    <variable name="TransactionIn" messageType="InfosMessage"/>
    <variable name="TransactionOut" messageType="InfosOut"/>
    <variable name="DebitInfo" messageType="DebitInfosMessage"/>
    <variable name="CreditInfo" messageType="CreditInfosMessage"/>
  </variables>
  <sequence name="BankTransferProcess">
    <receive name="ReceiveTransferInfos" variable="TransactionIn" ... operation="BankTransferOperation"/>
    <assign name="AssignTransferInfos">...</assign>
    <invoke name="InvokeDebit" ... operation="DebitOperation" inputVariable="DebitInfo"/>
    <invoke name="InvokeCredit" ... operation="CreditOperation" inputVariable="CreditInfo"/>
    <assign name="AssignResponse">...</assign>
    <reply name="Reply" variable="TransactionOut" ... operation="BankTransferOperation" />
  </sequence>
</process/>

```

**Fig. 6** The XML BPEL description of *BankTransfer* process

BPEL offers two categories of activities: (1) atomic activities representing the primitive operations performed by the process (they are defined by the *invoke*, *receive*, *reply*, *assign*, *terminate*, *wait* and *empty* activities and correspond to basic Web services) and (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 sequential control constructs. Three other composition operators are defined by the *pick* operator defining a



non-deterministic choice, the *flow* operator defining the concurrent execution and the *scope* operator defining subprocess execution. BPEL also introduces systematic mechanisms for fault handling by defining a set of activities to be executed for handling possible errors anticipated by the Web service composition designer. A compensation handler can be associated to the fault handler; it starts from the erroneous process itself to undo some steps that have already been completed and return the control back at the identified checkpoints.

### 3.2 A Case Study

We have chosen to illustrate our approach on the pedagogical case study of the *BankTransfer* commonly used to describe transactional Web services (Figs. 5, 6 and 7). This example describes a service processing a bank transfer between two

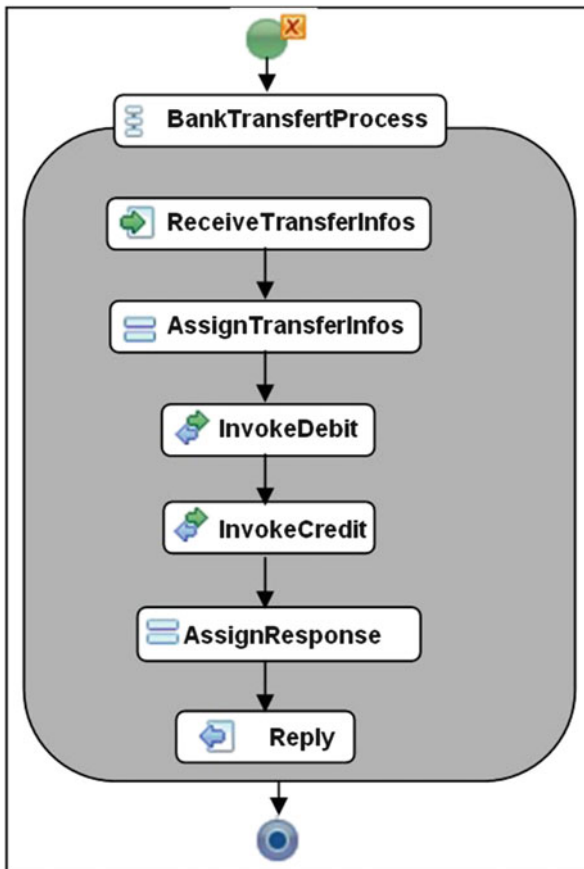


Fig. 7 The graphical BPEL description of *BankTransfer* process

bank accounts from two different banks. On receiving the transfer order from a customer, the process makes, in sequence, a debit from the source bank account and a credit to the target bank account. The BPEL description of Fig. 6 shows how the *BankTransfer* process is decomposed into a sequence of *ReceiveTransferInfos*, *AssignTransferInfos*, *InvokeDebit*, *InvokeCredit*, *AssignResponse* and *Reply* activities. The transaction order information, sent by the customer, is stored in the *TransactionIn* message, and the transaction receipt is sent to the customer in the *TransactionOut* message. The *DebitInfo* and *CreditInfo* messages contain the information sent to the two bank accounts to make the transfer.

## 4 Event-B for Analysing Transactional Web Services

Our idea is to provide assistance to BPEL developers using the Event-B method [2]. The choice of this formal method is guided by the fact that proof-based methods do not suffer from the state number explosion problem and proofs are eased by the refinement thanks to the decomposition it provides. Our motivation for the use of this formal method is detailed in the previous work [5, 6]. More precisely, our objective is to provide a methodology for detecting the BPEL process parts that handle critical resources. Traditionally, the developer builds its BPEL process and describes the process behaviour without using fault and compensation handlers to guarantee transactional constraints. He/she adds these constraints by observing the behaviour of the obtained process.

In the approach we promote, the designed BPEL process is translated into an Event-B model according to the rules defined in [5] and [6]. Then, the transactional properties and the properties related to the consistencies of resources used by the BPEL process are expressed in the form of consistency invariants (*consistency property*). These invariants are defined by the designer because there is no BPEL resource supporting their expression nor a technique for their checking. Once this enrichment is performed, proof obligations (POs) are generated. Some of these POs related to invariants involving the transactional properties are unprovable because triggering some events separately violates the consistency invariants. Then, BPEL activities related to the event source of these unprovable POs are detected and isolated in a BPEL scope element. This element allows the designer to define a particular BPEL part on which specific mechanisms only apply to this isolated part in an atomic manner. In our case, for transactional BPEL parts, we recommend to apply the mechanisms for fault and compensation handling to the scope element (*compensational atomicity property: guarantees that a transaction consisting of a Web service request will run in such a way that either all of its requests are completed successfully or all requests that occurred during the processing of this transaction will be compensated*) and the “isolated” attribute of the corresponding scope is set to “true”. As a consequence, the execution of this part is isolated by the orchestration tools (*isolation property*), and at the same time, consistency of the resources used by these activities is guaranteed. Transactional properties may

require a redesign of the defined BPEL process. From a methodological point of view, our approach relies on the following steps.

Step 1 Translate the BPEL model into an Event-B model.

Step 2 Introduce, in the Event-B model, the relevant invariants related to the suited transactional behaviour.

Step 3 Isolate the events of the Event-B model whose POs, associated to the introduced invariant of step 2, are not provable.

Step 4 Redesign the BPEL model of step 1 by introducing a BPEL scope embedding the events identified at step 3 and compensation/fault handlers.

This step-based approach is applied until the associated Event-B model is free of unproven POs.

In the following sections, we present the Event-B models for scope, fault handler and compensation handler BPEL concepts. These models extend the proposed ones of the general approach of [5] and [6] to support transactional BPEL processes modelling or checking the behaviour of composed Web services in the case of an internal or external runtime error of a BPEL process.

## 5 Modelling Scope, Fault and Compensation Handlers

Before addressing the formal modelling of the transactional behaviour, we reviewed the proposed Event-B formal semantics of BPEL ([5, 6]).

### 5.1 Formal Modelling of BPEL ([5, 6])

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 process 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 encoded by events of the Event-B EVENTS clause. For a better understanding of this paper, the transformation rules from BPEL to Event-B models are briefly recalled below. This translation process consists of two parts: *static and dynamic*.

#### 5.1.1 Static Part

The first part translates the WSDL definitions that describe the various Web services and 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.

```

CONTEXT      BankTransferContext
SETS
  Void InfosMessage ResponseMessage DebitMessage CreditMessage DebitInfosType CreditInfosType
CONSTANTS
  DebitInfosPart CreditInfosPart DebitPart CreditPart ResponsePart BankTransferOperation
  DebitOperation CreditOperation
AXIOMS
a12 : DebitInfosPart ∈ InfosMessage → DebitInfosType
a13 : CreditInfosPart ∈ InfosMessage → CreditInfosType
a14 : (DebitInfosPart ⊗ CreditInfosPart) ∈ InfosMessage → (DebitInfosType × CreditInfosType)
a15 : ResponsePart ∈ ResponseMessage → {0, 1}
a16 : DebitPart ∈ DebitMessage → DebitInfosType
a17 : CreditPart ∈ CreditMessage → CreditInfosType
a18 : BankTransferOperation ∈ InfosMessage → ResponseMessage
a19 : DebitOperation ∈ DebitMessage → Void
a20 : CreditOperation ∈ CreditMessage → Void
END

```

**Fig. 8** The Event-B CONTEXT of *BankTransfer* process

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 an Event-B Context are inductively defined:

- 1 The WSDL message element is formalised 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 in Fig. 5, a set named *InfosMessage* is defined in the SETS clause, and the application of this rule corresponds to the axiom *a12* and *a13* declarations in the *BankTransferContext* in Fig. 8.
- 2 Each operation of a WSDL portType is represented by a functional relation corresponding to the template  $operation \in input \rightarrow output$  mapping the input message type on the output message type. On the case study in Fig. 5, the *BankTransferOperation* operation is encoded by the functional relation described by axiom *a18* in Fig. 8.

### 5.1.2 Dynamic Part

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

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

<b>MACHINE</b>	BankTransferMachine
<b>SEES</b>	BankTransferContext
<b>VARIABLES</b>	<i>TransactionIn DebitInfo CreditInfo Response varSeq_I</i>
<b>INVARIANTS</b>	<i>i1: TransactionIn <math>\subseteq</math> InfosMessage <math>\wedge</math> card(TransactionIn) <math>\leq</math> 1</i> <i>i2: DebitInfo <math>\subseteq</math> DebitMessage <math>\wedge</math> card(DebitInfo) <math>\leq</math> 1</i> <i>i3: CreditInfo <math>\subseteq</math> CreditMessage <math>\wedge</math> card(CreditInfo) <math>\leq</math> 1</i> <i>i4: Response <math>\subseteq</math> ResponseMessage <math>\wedge</math> card(Response) <math>\leq</math> 1</i> <i>i5: varSeq-I <math>\in</math> {0,1,2,3,4,5,6}</i>

**Fig. 9** The Event-B MACHINE of *BankTransfer* process (part 1)

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 in Fig. 9. For example, the BPEL variable *DebitInfo* 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, in Fig. 10, the *ReceiveTransferInfos* BPEL atomic activity is encoded by the *ReceiveTransferInfos* Event-B event.
- 5 Each BPEL *structured activity* is modelled by an Event-B description which encodes the carried composition operator. Modelling composition operations in Event-B follow the modelling rules formally defined in [4]. Again, on the same example (Fig. 6), the structured activity *BankTransferProcess* is encoded by a sequence of six events controlled by the *varSeq\_I* variable initialised to value 6 (Fig. 10).

When the Event-B models formalising a BPEL description are obtained, they may be enriched by the relevant properties that formalise the user requirements and the soundness of the BPEL-defined process like BPEL type control, orchestration and service composition, deadlock freeness, no live-lock, precondition for calling a service operation and data transformation ([5, 6]).

## 5.2 An Event-B Model for Scope

The BPEL language provides a particular mechanism for subprocess description thanks to the scope construct. Scope encapsulates subprocess behaviours and includes a context used by the execution of the activities that describe its behaviour. This context contains a state composed by a set of variables. Each scope has a required primary activity that describes its behaviour (Fig. 11).

<pre> Initialisation begin   init1:  varSeq.I := 6   init2:  TransactionIn := ∅   init3:  DebitInfo := ∅   init4:  CreditInfo := ∅   init5:  Response := ∅ end Event   BankTransferProcess ≐ when   grd1:  varSeq.I = 0 then   skip end Event   ReceiveTransferInfos ≐ any   receive   where     grd1:  receive ∈ InfosMessage     grd2:  TransactionIn = ∅     grd3:  varSeq.I = 6   then     act1:  TransactionIn := {receive}     act2:  varSeq.I := varSeq.I - 1   end Event   AssignTransferInfos ≐ any   from to.I to 2   where     grd1:  to.I ∈ DebitMessage     grd2:  to.2 ∈ CreditMessage     grd3:  from ∈ TransactionIn     grd4:  TransactionIn ≠ ∅     grd5:  DebitInfo = ∅     grd6:  CreditInfo = ∅     grd7:  DebitInfosPart(from) = DebitPart(to.I)     grd8:  CreditInfosPart(from) = CreditPart(to.2)     grd9:  varSeq.I = 5   then     act1:  DebitInfo := {to.I}     act2:  CreditInfo := {to.2}     act3:  varSeq.I := varSeq.I - 1   end </pre>	<pre> Event   InvokeDebit ≐ any   msg   where     grd1:  msg ∈ DebitInfo     grd2:  DebitInfo ≠ ∅     grd3:  msg ∈ dom(DebitOperation)     grd4:  varSeq.I = 4   then     act1:  varSeq.I := varSeq.I - 1   end Event   InvokeCredit ≐ any   msg   where     grd1:  msg ∈ CreditInfo     grd2:  CreditInfo ≠ ∅     grd3:  msg ∈ dom(CreditOperation)     grd4:  varSeq.I = 3   then     act1:  varSeq.I := varSeq.I - 1   end Event   AssignResponse ≐ any   to   where     grd1:  to ∈ ResponseMessage     grd2:  Response = ∅     grd3:  ResponsePart(to) = 1     grd4:  varSeq.I = 2   then     act1:  Response := {to}     act2:  varSeq.I := varSeq.I - 1   end Event   Reply ≐ any   reply   where     grd1:  reply ∈ Response     grd2:  Response ≠ ∅     grd3:  varSeq.I = 1   then     act1:  Response := ∅     act2:  varSeq.I := varSeq.I - 1   end </pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

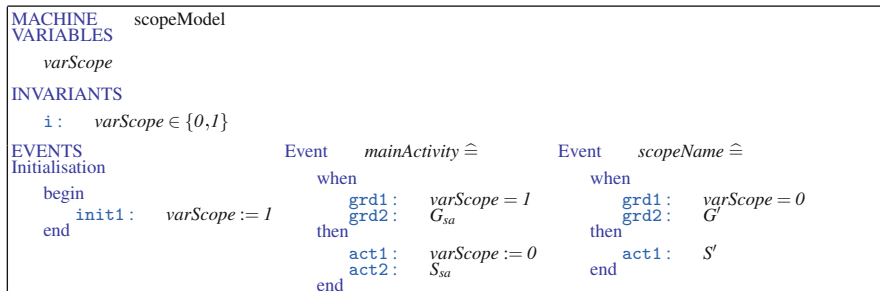
Fig. 10 The Event-B MACHINE of *BankTransfer* process (part 2)

```

<scope name="scopeName" isolated="yes—no"? ... >
  <variables>?...</variables>
  <faultHandlers>?
    <catch faultName="fault.1"?... >*>
      <activity name="catchFault.1">...</activity>
    </catch>
    <catchAll>?
      <activity name="catchAllFaults">...</activity>
    </catchAll>
  </faultHandlers>
  <compensationHandler>
    <activity ... >...</activity>
  </compensationHandler>
  <activity name="mainActivity"...>...</activity>
</scope>

```

Fig. 11 A BPEL *scope* element containing fault and compensation handlers



**Fig. 12** An Event-B model for a BPEL scope element

The *scopeModel* MACHINE in Fig. 12 formalises the scope behaviour. This MACHINE introduces a *mainActivity* event that models the main activity of a scope and a *scopeName* event that models the end of a scope execution. The variable *varScope* is initialised to 1. This variable triggers the *scopeName* event at the end of the *mainActivity* event. If the scope’s main activity is of structured type, the transformation rules defined for structured activities in [5] are applied. Similarly, the scope variables are modelled according to the variable transformation rules defined in [5].

$G_{sa}$ ,  $G'$ ,  $S_{sa}$  and  $S'$  represent the guards and the actions of the *mainActivity* and *scopeName* events. They result from variable and activity modelling and are related to the data manipulation and to the process behaviour. The same reasoning applies for fault and compensation handler models presented below.

### 5.3 An Event-B Model for Fault Handler

A BPEL fault handler is a process that is triggered when an error rose from a partner Web service or an internal BPEL process. It provides a mechanism to define a set of customised “fault-handling” activities. A *catch* element is defined to intercept (to catch) a specific kind of fault, defined by a “*faultName*” attribute. If no predefined name is associated to the intercepted fault, this fault will be processed by a *catchAll* element (see Fig. 11).

The *faultHandlerModel* MACHINE in Fig. 13 formalises a fault Handler as described in Fig. 11. The *mainActivity* event models the normal behaviour of the scope. Different types of errors are taken into account by the defined fault handler. Error types are defined by an enumerated set called *faultType* in the SETS clause. It contains all fault types identified by the designer and used in the BPEL process description. The model in Fig. 13 formalises the case of errors called *fault\_1* among other faults. A *varFault* variable is introduced to determine whether the current behaviour of the process is catching errors ( $varFault=1$ ) or describes a normal execution ( $varFault=0$ ).

CONTEXT	faultHandlerContext		
SETS	<i>faultType</i>		
CONSTANTS	<i>fault_1</i> <i>otherFault</i>		
AXIOMS	<b>a1</b> : <i>partition</i> ( <i>faultType</i> , { <i>fault_1</i> }, { <i>otherFault</i> })		
END			
MACHINE	faultHandlerModel		
SEES	faultHandlerContext		
VARIABLES	<i>currentFault</i> <i>varFault</i>		
INVARIANTS	<b>i1</b> : <i>currentFault</i> ∈ <i>faultType</i> <b>i2</b> : <i>varFault</i> ∈ {0,1}		
EVENTS	Event	<i>mainActivity</i> ≐	
Initialisation	when	end	
begin	grd1:	<i>varFault</i> = 0	
<b>init1</b> : <i>varFault</i> := 0	grd2:	<i>G</i>	
<b>init2</b> : <i>currentFault</i> := ∈	then	<b>act1</b> : <i>S</i>	
<i>faultType</i>	end	end	
end			
Event	Event	Event	
<i>faultOccurs</i> ≐	<i>catchFault_1</i> ≐	<i>catchAllFaults</i> ≐	
any	when	where	
<i>ff</i>	grd1:	<i>varFault</i> = 1	
where	grd2:	<i>currentFault</i> =	
grd1: <i>varFault</i> = 0	<i>fault_1</i>	<i>currentFault</i> =	
grd2: <i>ff</i> ∈ <i>faultType</i>	grd3: <i>G<sub>1</sub></i>	<i>otherFault</i>	
then	then	grd3: <i>G<sub>n</sub></i>	
<b>act1</b> : <i>varFault</i> := 1	<b>act1</b> : <i>S<sub>1</sub></i>	then	
<b>act2</b> : <i>currentFault</i> := <i>ff</i>	end	<b>act1</b> : <i>S<sub>n</sub></i>	
end		end	

Fig. 13 An Event-B model for a BPEL fault handler

The occurrence of an error is formalised by the *faultOccurs* event that is arbitrarily triggered (non-deterministic occurrence of the event). When this event is triggered, it ceases the normal behaviour described by the *mainActivity* event thanks to the action *varFault:=1*. The fault processing, corresponding to the error defined by the *currentFault* variable content, starts. If this variable contains the *fault\_1* value, the *catchFault\_1* event, formalising the processing of the *fault\_1* error, is triggered. Otherwise (*currentFault=otherFault*), the *catchAllFaults* event, formalising the *catchAll* element processing, is triggered.

Similarly, this model can be used to formalise a fault handler associated to a BPEL process.

#### 5.4 An Event-B Model for Compensation Handler

A compensation handler is a wrapper for a process that performs compensation. A compensation handler for an activity is available for invocation only when the activity completes successfully. Generally, the invocation of a compensation handler



```

<process ... >
  ...
  <faultHandlers>
    <catch faultName="scopeFault"... >
      <compensate name="catchScope"/>
    </catch>
  </faultHandlers>
  <sequence name="mainActivity">
    ...
    <scope name="scopeName" ...>
      <compensationHandler>
        <activity name="compensationActivity">...</activity>
      </compensationHandler>
    </scope>
  </sequence>
</process>

```

**Fig. 14** BPEL fault and compensation handlers

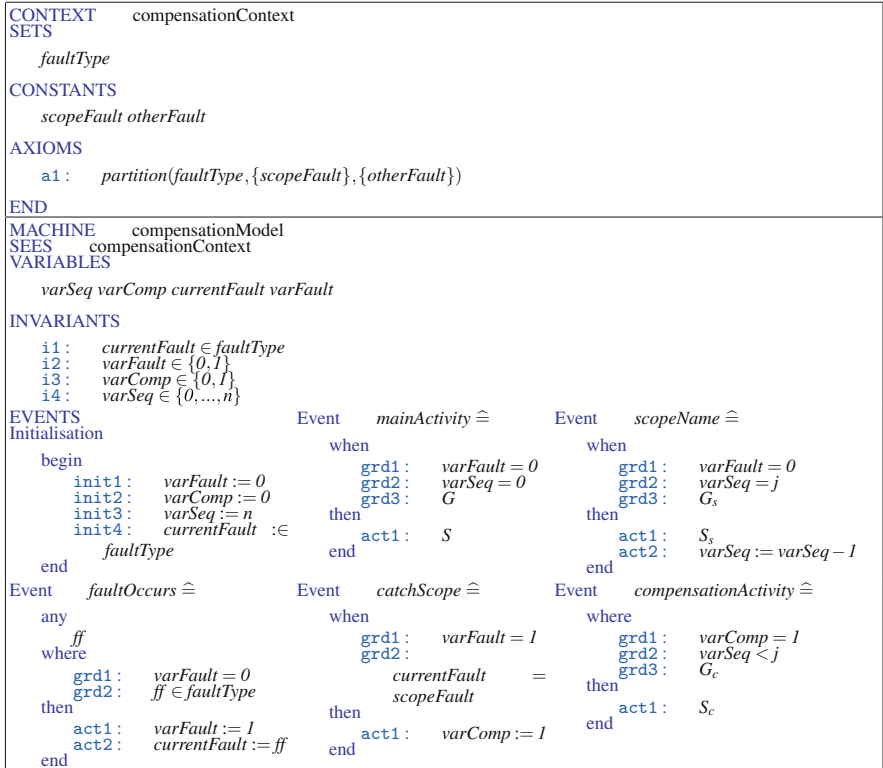
is achieved by a fault handler to undo the effects of already completed activities in the case of a runtime error.

The BPEL process described in Fig. 14 contains two behaviours: the normal behaviour described by the *mainActivity* sequence element and the error catching behaviour described by the fault handler. The normal behaviour consists of a sequence of activities, and one of these activities is required to be a scope. This scope contains a compensation handler to cancel its effect in the case of a runtime error. Then, the fault handling process consists of triggering the *compensate* activity.

The *compensationModel* MACHINE described in Fig. 15 formalises a BPEL process conforming to Fig. 14. The normal behaviour described by a sequence activity is modelled by a *mainActivity* event according to the defined rules in [5]. Being the  $j$ th activity of this sequence, the scope is modelled by the *scopeName* event which represents part of the scope model obtained using the rules defined in Sect. 5.2. The *scopeName* event is triggered when  $varSeq=j$ .

The fault handler part is modelled by the *catchScope* event. It formalises a *compensate* activity named *catchScope*. This part describes the *scopeFault* handling process. The *catchScope* event triggers the compensation handler contained in the *scopeName* scope, in the case that *currentFault* is “scopeFault”. This action is done by assigning value 1 to the *varComp* variable.

The compensation process is described by the *compensationActivity* event. It can be triggered if the *scopeName* scope has completed its execution ( $varSeq < j$ ) and the *catchScope* event is triggered ( $varComp=1$ ). The *compensationActivity* specifies the process of compensation. It can be described by one or a set of activities. In this case, the transformation rules of BPEL activities defined in [5] are applied to detail this description.



**Fig. 15** An Event-B model for BPEL fault and compensation handlers

## 6 Use of the Tools

The proposed approach is implemented by integrating various plug-ins in the Rodin platform which is based on the Eclipse core. We have used existing plug-ins of Eclipse platform (*BPEL editor*<sup>1</sup>) and of Rodin platform (*Event-B editor, prover* [24] and *ProB model checker* [18]) and developed plug-ins (*BPEL2B plug-in* [5]). These plug-ins are used at different steps of our approach:

- Designing BPEL process with the *BPEL editor* and translating the obtained model into an Event-B model using the BPEL2B plug-in (*step 1*).
- Introducing the relevant invariants related to the suited transactional behaviour with the *Event-B editor* (*step 2*). Isolating the events of the model whose POs, associated to the introduced invariant of step 2, cannot be proved within the *Rodin prover*.

<sup>1</sup>BPEL Designer Project: <http://eclipse.org/bpel/>

- The *ProB model checker* assisting the developer to confirm the diagnostic and to get a counterexample associated to the isolated events (*step 3*).
- Redesigning the BPEL model of step 1 by introducing a BPEL *scope* embedding the events identified at step 3 and a compensation/fault handler (*step 4*). The graphical view of the *BPEL editor* can be used at this step to facilitate the insertion of the new elements.

## 7 Application to the Case Study

The application of the approach outlined in Sect. 4 on the example in Fig. 6 is given in the following steps. The reader can find all Event-B CONTEXTs and MACHINES source codes on this website.<sup>2</sup>

*Step 1* This step on the *BankTransfer* BPEL process in Fig. 6 leads to the Event-B model in Figs. 8, 9 and 10 described in Sect. 5.1.

*Step 2* The transactional properties are expressed in the form of invariant in the Event-B model of step 1. Starting from the model in Figs. 8 and 10, we obtain the model in Fig. 16 by defining a default fault handler with the *defaultFaultHandler* event. *BankAccount1* and *BankAccount2* variables formalise the contents of two bank accounts and the *amount* variable contains the value to be transferred. The *InvokeDebit* and *InvokeCredit* events invoke Web services that make the transaction. The *consistency* property is expressed by invariant *i12*. The sum of *BankAccount1* and *BankAccount2* variables shall be constant to ensure consistency of the contents of two bank accounts before triggering *InvokeDebit* event and after triggering *InvokeCredit* event. Invariant *i13* expresses the state after triggering *InvokeDebit* event and before triggering the *InvokeCredit* event.

*Step 3* The POs are generated by the Rodin platform, and those associated to the invariant *i12* and to the action *varSeq\_1:=0* of *defaultFaultHandler* event, which aborts the BPEL process, cannot be proved (Fig. 17). An inconsistency state results from this abortion.

The invariant violation is usually caused by *defaultFaultHandler* event triggering when the *faultOccurs* event is triggered after *InvokeDebit* event. This diagnostic is confirmed by the ProB model checker [18] that gets a counterexample corresponding to this scenario (Fig. 18). The inconsistency state is *varSeq=0* (*defaultFaultHandler* event) and  $BankAccount1 + BankAccount2 = ConsistencyState - amount$ .

*Step 4* If an error occurs during the transaction, a fault handler should restore a consistent state of the process before aborting it. The *InvokeDebit* and *InvokeCredit* activities are isolated in the BPEL scope, a fault handler is associated

---

<sup>2</sup><http://idir.aitsadoune.free.fr>

<pre> CONTEXT      BankTransferContext SETS   ...faultType  CONSTANTS   ...ConsistencyState otherFault  AXIOMS   ...: ...   a12: ... partition(faultType, {otherFault})   a13: ... ConsistencyState ∈ ℕ<sub>I</sub>  END </pre>	
<pre> MACHINE      BankTransferWithoutScope SEES        BankTransferContext VARIABLES   ...varFault currentFault BankAccount1 BankAccount2 amount  INVARIANTS   ...: ...   i7: ... varFault ∈ ℕ ∧ varFault ∈ {0,1}   i8: ... currentFault ∈ faultType   i9: ... BankAccount1 ∈ ℕ   i10: ... BankAccount2 ∈ ℕ   i11: ... amount ∈ ℕ   i12: ... (varSeq-1 ≠ 3) ⇒ (BankAccount1 + BankAccount2 = ConsistencyState)   i13: ... (varSeq-1 = 3) ⇒ (BankAccount1 + BankAccount2 = ConsistencyState - amount)  EVENTS Initialisation   begin     ...: ...     init3: ... BankAccount1, BankAccount2 :       [... ∧ BankAccount1' + BankAccount2' =         Consistency)     init4: ... amount : ∈ ℕ<sub>I</sub>     init5: ... currentFault : ∈ faultType     init6: ... varFault := 0   end  Event  faultOccurs ≐   when     grd2: ... varFault = 0     grd3: ... varSeq-1 ≠ 0   then     sub1: ... currentFault := otherFault     sub2: ... varFault := 1   end  Event  defaultFaultHandler ≐   when     grd1: ... varFault = 1     grd2: ... currentFault = otherFault   then     sub1: ... varSeq-1 := 0   end </pre>	
	<pre> Event  ReceiveTransferInfos ≐ Event  AssignTransferInfos ≐ Event  InvokeDebit ≐    where     ...: ...     grd4: ... varSeq-1 = 4     grd5: ... amount ≤ BankAccount1     grd6: ... varFault = 0   then     act1: ... varSeq-1 := varSeq-1 - 1     act2: ... BankAccount1 := BankAccount1 -       amount   end  Event  InvokeCredit ≐   where     ...: ...     grd4: ... varSeq-1 = 3     grd5: ... varFault = 0   then     act1: ... varSeq-1 := varSeq-1 - 1     act2: ... BankAccount2 := BankAccount2 +       amount   end  Event  AssignResponse ≐ Event  Reply ≐ Event  BankTransferProcess ≐ </pre>

Fig. 16 The consistency property expression of *BankTransfer* process

to this scope, and the mechanism for compensation handling is applied to the *InvokeDebit* and *InvokeCredit* activities (*compensational atomicity property*). The “isolated” attribute of this scope is also set to “true” (*isolation property*). The BPEL process obtained by this step is given in Figs. 19 and 20.

When applying again *step 1* on the obtained BPEL process, the Event-B model in Fig. 21 with *BankTransferContext* and *BankTransferMachine* components is obtained. Only elements that differ from the Event-B model in Figs. 8 and 10 are shown in Fig. 21. The *BankTransferProcess* is encoded by a sequence of five events. One of them is *BankTransferScope* which formalises the *BankTransferScope*

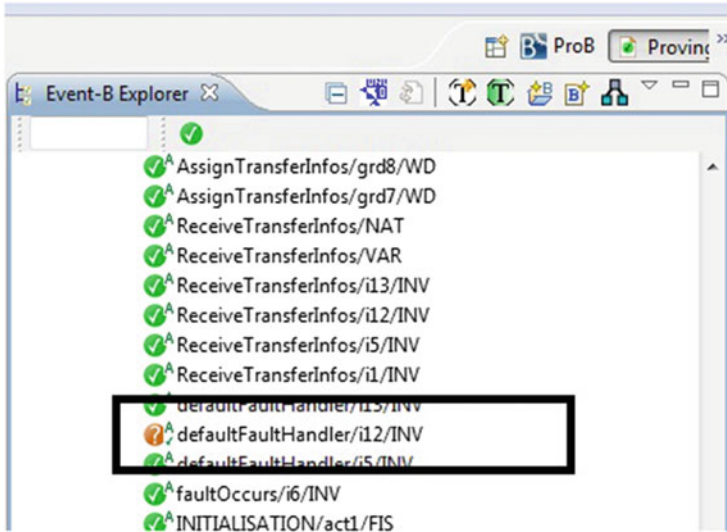


Fig. 17 The Rodin diagnostic

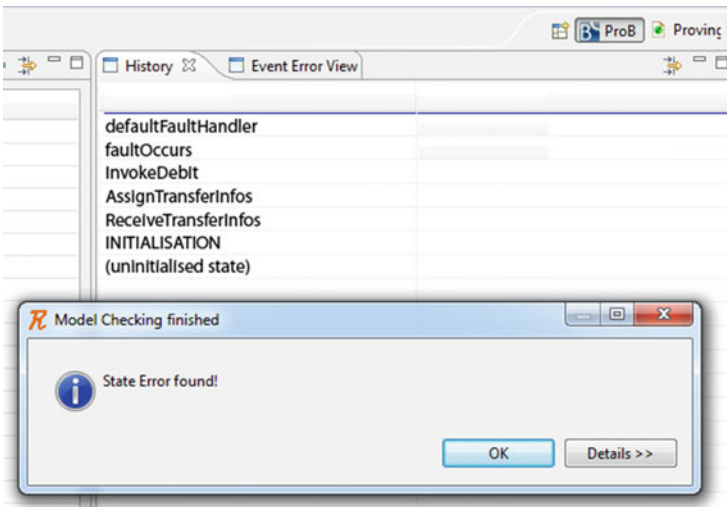


Fig. 18 The ProB diagnostic

scope. The main activity of this scope is a *BankTransfer* activity which is decomposed to a sequence of *InvokeDebit* and *InvokeCredit* activities controlled by the *varSeq\_2* variable initialised to value 2. The *scopeFaultOccurs* event formalises the triggering of a runtime error inside the scope. This event triggers the scope fault handler by assigning the *currentFault* variable to “scopeFault”. The fault handler process is described by the *compensate*, *rethrow* and *scopeFaultHandler*

```

<process name="BankTransfer" ...>
...
<sequence name="BankTransferProcess">
  <receive name="ReceiveTransferInfos" ... />
  <assign name="AssignTransferInfos"> ... </assign>
  <scope name="BankTransferscope" isolated="true">
    <faultHandlers>
      <catchAll>
        <sequence>
          <compensate/>
          <rethrow/>
        </sequence>
      </catchAll>
    </faultHandlers>
    <sequence name="BankTransfer">
      <invoke name="InvokeDebit" ...>
        <compensationHandler>
          <invoke name="InvokeCancelDebit" .../>
        </compensationHandler>
      </invoke>
      <invoke name="InvokeCredit" ...>
        <compensationHandler>
          <invoke name="InvokeCancelCredit" .../>
        </compensationHandler>
      </invoke>
    </sequence>
  </scope>
  <assign name="AssignResponse">...</assign>
  <reply name="Reply" .../>
</sequence>
</process/>

```

**Fig. 19** The BPEL description of a redesigned *BankTransfer* process

events. The *compensate* event triggers different compensation handlers that consist of invoking *InvokeCancelDebit* and *InvokeCancelCredit* events to cancel the effect of the *InvokeDebit* and *InvokeCredit* events.

In this case, the invariant *i12* is changed and adapted to the modifications made by introducing a scope. All POs are proved, and unlike the case in Fig. 16, the invariant *i12* is not violated by the fault handler process if *scopeFaultOccurs* event is triggered after *InvokeDebit* event (Fig. 22). The *scopeFaultHandler* event triggers the compensation process (*compensate* event) before aborting the BPEL process.

## 8 Related Work

Various approaches have been proposed to model and to analyse BPEL processes. Most of the work in the literature shows that the proposed approaches use transition systems for representing the business processes, activities and workflows and model checking as the underlying formal verification technique for property validation. *Hinz et al.* [15] and *van der Aalst et al.* [27] have used Petri nets to encode

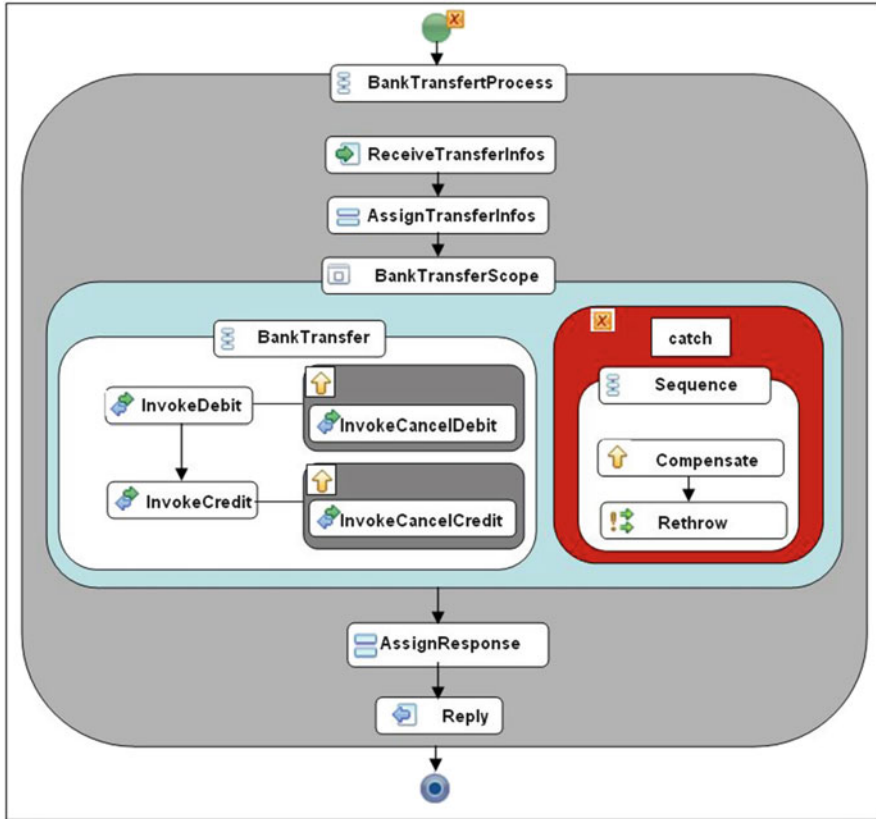


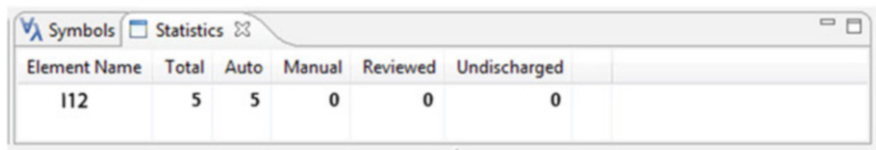
Fig. 20 The graphical BPEL description of a redesigned *BankTransfer* process

BPEL processes. *Nakajima* [21] has mapped a BPEL activity part to a finite automaton encoded in Promela. FSP (finite state process) and the associated tool (LTSA) are used by *Foster et al.* [12] to check if a BPEL Web service composition behaves like an MSC Web service composition specification captured by message sequence charts (MSCs). *Marconi et al.* [20] present the approach that translates a BPEL process to a set of state transition systems. These systems are composed of parallel and the resulting parallel composition is annotated with specific Web service requirements. *Salaun et al.* [25] show how BPEL processes are mapped to processes expressed by LOTOS and CCS process algebra operations. Abstract state machines (ASM) have been developed by *Farahbod et al.* [11] and *Fahland* [10] to model BPEL composition process descriptions. *Borger et al.* [7] have used ASM for modelling workflow, specifically BPMN process. The B method and StaAC were used by *Butler et al.* [8] to model business transactions and their compensation with an application to a BPEL process. Other approaches proposed formal models for Web service compositions. An overview of these approaches can be found in [26].

<pre> CONTEXT      BankTransferContext SETS   ...faultType CONSTANTS   ...otherFault scopeFault CancelDebitOperation CancelCreditOperation AXIOMS   ...   op1: ... CancelDebitOperation ∈ DebitMessage → Void   op2: ... CancelCreditOperation ∈ CreditMessage → Void   a13: partition(faultType, {otherFault}, {scopeFault}) END MACHINE      BankTransferMachine SETS        BankTransferContext VARIABLES   ...varSeq_1 varSeq_2 varSeq_3 varScope varComp... INVARIANTS   ...   i5: ... varSeq_1 ∈ {0, 1, 2, 3, 4, 5}   i6: ... varSeq_2 ∈ {0, 1, 2}   i8: ... varScope ∈ {0, 1}   i9: ... varComp ∈ {0, 1}   i10: ... varSeq_3 ∈ {0, 1, 2}   ...   i12: ... (varSeq_2 ≠ 1) ⇒ (BankAccount1 + BankAccount2 = ConsistencyState)   i13: ... (varSeq_2 = 1) ⇒ (BankAccount1 + BankAccount2 = ConsistencyState - amount) EVENTS   Initialisation   begin     init1: varSeq_1, varSeq_2 := 5, 2     init2: varSeq_3, varScope := 2, 1     init3: varFault, varComp := 0, 0   end   ... Event otherFaultOccurs ≐   when     grd2: varFault = 0     grd3: varSeq_1 ≠ 3   then     sub1: currentFault := otherFault     sub2: varFault := 1   end Event defaultFaultHandler ≐   when     grd1: varFault = 1     grd2: currentFault = otherFault   then     sub1: varSeq_1 := 0     sub2: varFault := 0   end Event scopeFaultOccurs ≐   when     grd2: varFault = 0     grd3: varSeq_1 = 3   then     sub1: currentFault := scopeFault     sub2: varFault := 1   end Event compensate ≐   when     grd1: currentFault = scopeFault     grd2: varFault = 1     g3: varSeq_2 = 2   then     sub1: varComp := 1     a1: varSeq_3 := varSeq_3 - 1   end Event rethrow ≐   when     g1: varFault = 1     g2: varSeq_3 = 1     g3: varSeq_2 = 2   then     a1: currentFault := otherFault     a2: varSeq_3 := varSeq_3 - 1   end Event ScopeFaultHandler ≐   when     g1: varSeq_3 = 0   then     skip   end </pre>		<pre> Event invokeCancelDebit ≐   any   msg   where     grd1: varComp = 1     grd2: varSeq_2 = 1     grd3: DebitInfo ≠ ∅     grd4: msg ∈ DebitInfo     grd5: msg ∈ dom(CancelDebitOperation)   then     sub1: BankAccount1 := BankAccount1 + amount     sub2: varSeq_2 := 2     sub3: varScope := 1     sub4: varComp := 0   end Event invokeCancelCredit ≐   any   msg   where     grd1: varComp = 1     grd2: varSeq_2 &lt; 1     grd3: CreditInfo ≠ ∅     grd4: msg ∈ CreditInfo     grd5: msg ∈ dom(CancelCreditOperation)     grd6: amount ≤ BankAccount2   then     sub1: BankAccount2 := BankAccount2 - amount     sub2: varSeq_2 := 1   end Event ReceiveTransferInfos ≐ Event AssignTransferInfos ≐ Event InvokeDebit ≐ Event InvokeCredit ≐ Event BankTransfer ≐   when     grd1: varScope = 1     grd2: varSeq_1 = 3     grd3: varSeq_2 = 0     nf: varFault = 0   then     sub1: varScope := 0   end Event BankTransferScope ≐   when     grd1: varSeq_1 = 3     grd2: varScope = 0     nf: varFault = 0   then     act1: varSeq_1 := varSeq_1 - 1   end Event AssignResponse ≐ Event Reply ≐ Event BankTransferProcess ≐ END </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Fig. 21** The Event-B model of the redesigned *BankTransfer* process





Element Name	Total	Auto	Manual	Reviewed	Undischarged
i12	5	5	0	0	0

Fig. 22 The Rodin statistic view about i12 invariant

In all these proposals, a BPEL description is transformed to a formal model to be checked. However, some of this work did not take into account the fault and compensation mechanisms that enable transactional behaviours of BPEL processes. No formal generic approach handling the full transactional Web service design process is available. Some approaches have given formal semantics for the fault and compensation handlers and encoded them in other formal techniques like Petri nets [14, 19], pi-calculus [13], ASM [11], StaAC and B [8] or SAL [17]. In these approaches, the designer describes by himself/herself the parts that are handled by the fault and compensation handlers, and they check properties related to the behaviour like deadlock freeness. There is no systematic formal modelling approach for handling transactions. Moreover, there is no way to detect the transactional part to be isolated in order to guarantee a correct behaviour of the transactional Web service. Furthermore, existing approaches do not provide the possibility to check if the consistency of the execution context is guaranteed. This is due to the abstraction of data in the model checking techniques applied to Web service validation for reducing the state space exploration. As a consequence, these works don't describe nor check properties related to the consistency of data produced by the BPEL processes.

Our work is proof oriented; it translates the BPEL language and its constructs into an Event-B model. We encode manipulated data and transactional behaviour, check traditional properties like deadlock freeness and transactional properties [5, 6] and offer to the developer assistance to improve his/her BPEL design by isolating transactional parts to ensure a correct process behaviour. Moreover, it is tool supported.

Notice that this work applies for all transactional-based process compositions.

## 9 Conclusion

In this paper, we propose an extension of the BPEL Event-B semantics proposed in [5] and [6] by covering the *scope*, the fault and the compensation handlers. We have also sketched a methodology showing how the obtained Event-B model can be used to handle a transactional behaviour in Web services. Transactional services that access and manage critical resources are isolated in a *scope* elements with compensation and fault handlers. When modelling fault and compensation

handlers by a set of events, it becomes possible to model and check the properties related to transactional Web services. Moreover, the obtained results are not specific to Web service compositions. These results can be reused for the definition of transactional service compositions that occur for areas like telecommunication and network, manufacturing or scheduling. Indeed, the fact that services are Web based is not specific to the proposed approach. BPEL is used as a language for service composition description whatever is the nature or the application domain of the manipulated services.

This work opens several perspectives. One of them is related to the explicit semantics carried by the services. For example, composing in sequence a service that produces distances expressed in centimetres with another one consuming distances expressed in inches should not be a valid composition. Up to now, our approach handles implicit semantics only; it does not handle such a composition. Formal knowledge models carried out by ontologies expressed besides the Event-B models should be investigated.

## References

1. Abrial, J.R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York (1996)
2. Abrial, J.R.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
3. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. *Fundam. Inform.* **77**, 1–28 (2007)
4. Ait-Ameur, Y., Baron, M., Kamel, N., Mota, J.M.: Encoding a process algebra using the Event B method. *Int. J. Softw. Tools Technol. Transfer* **11**(3), 239–253 (2009)
5. Ait-Sadoune, I., Ait-Ameur, Y.: A proof based approach for modelling and verifying web services compositions. In: *14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 1–10. IEEE Computer Society, Potsdam (2009)
6. Ait-Sadoune, I., Ait-Ameur, Y.: Stepwise design of BPEL web services compositions, an Event B refinement based approach. In: *8th ACIS International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 51–68, Montreal (2010)
7. Borger, E., Thalheim, B.: Modeling workflows, interaction patterns, web services and business processes: the ASM-based approach. In: *Abstract State Machines, B and Z (ABZ 2008)*. Lecture Notes in Computer Science, vol. 5238. Springer, Heidelberg (2008)
8. Butler, M., Ferreira, C., Ng, M.Y.: Precise modelling of compensating business transactions and its application to BPEL. *J. Univers. Comput. Sci.* **11**(5), 712–743 (2005)
9. Dijkstra, E.W.: *A Discipline of Programming*, 1st edn. Prentice Hall PTR, Upper Saddle River (1977)
10. Fahland, D., Reisig, W.: ASM-based semantics for BPEL: the negative Control Flow. In: *12th International Workshop on Abstract State Machines*, pp. 131–151 (2005)
11. Farahbod, R., Glasser, U., Vajihollahi, M.: An abstract machine architecture for web service based business process management. In: *Business Process Management Workshops*. Lecture Notes in Computer Science, vol. 3812, pp. 144–157. Springer, Heidelberg (2005)
12. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based verification of web service compositions. In: *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pp. 152–163 (2003)

13. Guidi, C., Lucchi, R., Mazzara, M.: A formal framework for web services coordination. *Electron. Notes Theor. Comput. Sci.* **180**, 55–70 (2007)
14. He, Y., Zhao, L., Wu, Z., Li, F.: Formal modeling of transaction behavior in WS-BPEL. In: *International Conference on Computer Science and Software Engineering (CSSE 2008)* (2008)
15. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to petri nets. In: *Springer-Verlag* (ed.) *3rd International Conference on Business Process Management. Lecture Notes in Computer Science*, vol. 2649. Springer, Heidelberg (2005)
16. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**, 576–580 (1969)
17. Kovacs, M., Varro, D., Gonczy, L.: Formal analysis of BPEL workflows with compensation by model checking. *Int. J. Comput. Syst. Sci. Eng.* **23**(5), 35–49 (2008)
18. Leuschel, M., Butler, M.: ProB: a model checker for B. In: *Formal Methods, International Symposium of Formal Methods Europe (FME'03). Lecture Notes in Computer Science*, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
19. Lohmann, N.: A feature-complete petri net semantics for WS-BPEL 2.0. In: *Web Services and Formal Methods International Workshop WSFM 2007* (2007)
20. Marconi, A., Pistore, M.: Synthesis and composition of web services. In: *Formal Methods for Web Services - 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services. Lecture Notes in Computer Science*, vol. 5569. Springer, Heidelberg (2009)
21. Nakajima, S.: Model-checking behavioral specification of BPEL applications. *Electron. Notes Theor. Comput. Sci.* **151**, 89–105 (2006)
22. OASIS: Web Services Business Process Execution Language Version 2.0. <http://bpel.xml.org/> (April 2007)
23. OMG: Business Process Model and Notation (BPMN) Version 2.0. <http://www.omg.org/spec/BPMN/2.0> (June 2010)
24. Rodin: User Manual of the RODIN Platform. <http://deploy-eprints.ecs.soton.ac.uk/11/1/manual-2.3.pdf> (October 2007)
25. Salaun, G., Bordeaux, L., Schaerf, M.: Describing and reasoning on web services using process algebra. In: *IEEE International Conference on Web Services (ICWS'04)*, pp. 43–51 (2004)
26. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal methods for service composition. *Ann. Math. Comput. Teleinformatics* **1**(5), 1–14 (2007)
27. van der Aalst, W.M., Mooil, A.J., Stahl, C., Wolf, K.: Service interaction: patterns, formalization, and analysis. In: *Formal Methods for Web Services - 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services. Lecture Notes in Computer Science*, vol. 5569. Springer, Heidelberg (2009)
28. W3C: Web Service Definition Language (WSDL 1.1). <http://www.w3.org/TR/wsdl> (February 2004)
29. W3C: OWL-S: Semantic Markup for Web Services. <http://www.w3.org/Submission/OWL-S/> (November 2004)
30. W3C: Web Services Choreography Description Language Version 1.0. <http://www.w3.org/TR/ws-cdl-10/> (November 2005)
31. WMC-WS: Process Definition Interface - XML Process Definition Language. <http://www.wfmc.org/xpdl.html> (October 2008)