From BPEL to SRML: A Formal Transformational Approach*

Laura Bocchi[1], Yi Hong[1], Antónia Lopes[2], and José Luiz Fiadeiro[1]

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{bocchi,yh37,jose}@mcs.le.ac.uk
[2] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, Portugal
mal@di.fc.ul.pt

Abstract. The SENSORIA Reference Modelling Language (SRML) provides primitives for modelling business processes in a technology agnostic way. At the core of SRML is the notion of module as a composition of tightly coupled components and loosely coupled, dynamically discovered services. This paper presents an encoding of BPEL processes into SRML modules using model transformation techniques. The encoding provides the means to create high-level declarative descriptions of BPEL processes that can be used for building more complex modules, possibly including components implemented in other languages. The composition can be modelled and analysed as an ensemble, relying on the rich formal framework that is being developed within SENSORIA.

1 Introduction

The SENSORIA Reference Modelling Language (SRML) is a high-level modelling language for Service Oriented Architectures (SOAs) developed in the context of SENSORIA, an IST-FET Integrated Project on *Software Engineering for Service-Oriented Overlay Computers*. The goal of SRML is to provide a set of primitives that is expressive enough to model applications in the service-oriented paradigm and simple enough to be formalized. Through the notion of module, SRML provides primitives for modelling business processes as assemblies of (1) tightly coupled components that may be implemented using different technologies (including wrapped-up legacy systems) and (2) loosely coupled, dynamically discovered services.

The structure of a SRML module is illustrated in Fig. 1. Both the service provided and the external services required by the module are represented through what we call external interfaces, which are rich descriptions of the behaviour that can be observed of the interactions with these parties. The language primitives used for description and specification of service and component behaviour have been presented in [5].

* This work was partially supported through the IST-2005-16004 Integrated Project *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*, and the Marie-Curie TOK-IAP MTK1-CT-2004-003169 *Leg2Net: From Legacy Systems to Services in the Net*.

M. Dumas and R. Heckel (Eds.): WS-FM 2007, LNCS 4937, pp. 92–107, 2008.

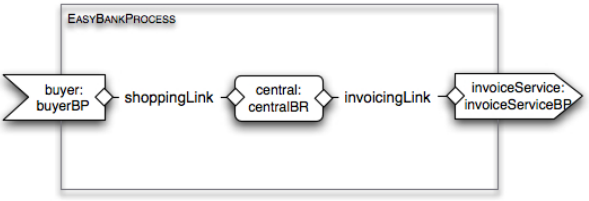


Fig. 1. The module *EasyBankProcess* has one component – *central* of type *centralBR* – which orchestrates the interactions with the external parties. The node *invoiceService* (of type *invoiceServiceBP*) is an external-interface for a service required by the module. The node *buyer* (of type *buyerBP*) is the external interface for the provided service. The edges *shoppingLink* and *invoicingLink* provide the protocols that coordinate the interactions between the involved parties.

The interconnections between different parties are represented as wires labelled with interaction protocols [1].

SRML modules were inspired by the Service Component Architecture (SCA) [13]. SCA is a set of specifications that describe a middleware-independent model for building applications over SOAs. It provides a convenient framework to manage the deployment and configuration of service-oriented systems. In SCA, applications are built as assemblies of heterogeneous components and external services. SCA offers specific support for a variety of component implementations, namely for WS-BPEL [14]. More concretely, a BPEL client and implementation specification is defined that allows a component to be written is BPEL and deployed and assembled with other components written in any SCA implementation language.

In this paper, we present an encoding of WS-BPEL processes, including the WSDL interfaces with which they are associated, into SRML modules. As in SCA, the synthesis of high-level declarative descriptions allows existing BPEL processes to be used together with other components when defining models for composite services as SRML modules. The models consist of the assembly of a number of SRML modules that can be derived from existing components (implemented in BPEL or any other language for which an encoding into SRML has been provided), or for which an implementation still has to be provided. As a consequence, a given BPEL process can be used in the implementation of different composite services.

A basic difference between our encoding and the one provided through SCA results from the fact that, whereas SCA abstracts from the business logic provided by components, SRML provides a high-level declarative description of that business logic. Therefore, our encoding also addresses the orchestrations performed by BPEL processes within the assembly structures.

As a consequence, our encoding provides the means for WS-BPEL processes to be analysed, both individually and within composite services, by relying on the rich formal framework that is being developed within SENSORIA [15]. However, it should be clear that our aim is not to provide BPEL with yet another formal semantics and associated verification techniques, but only to the extent in which BPEL

processes can be used in SRML to define composite services, possibly in conjunction with other service components implemented also in BPEL or in other languages.

The encoding of BPEL processes into SRML is formalized by means of model transformation rules based on triple graph grammars (TGG) [8]. The definition of model transformations with TGGs relies on: (1) *a source meta-model* representing the abstract syntax of the source language (e.g. BPEL) as a typed graph, (2) a *target meta-model* representing the abstract syntax of the target language (e.g. SRML) as a typed graph, (3) a third graph grammar — *the correspondence meta-model* — that connects related elements of (1) and (2) and is used to control the transformation process which, in general, is bidirectional. In our case, we provide directional transformation rules that specify only one direction – from BPEL to SRML.

The structure of the paper is as follows. Section 2 discusses the strategy of the encoding in more detail and presents an example. Section 3 presents the transformation rules for the module structure. Section 4 presents the encoding of the control flow. Finally, Section 5 presents final conclusions and discusses future work.

2 The Strategy of the Encoding

As described in [5], SRML provides mechanisms for assembling two modules via an external wire that establishes how the provides-interface *EX-P* of one module matches a requires-interface *EX-R* of the other module (Fig. 2). Assembly can be performed at design-time in order to define composite services (orchestrated systems), or dynamically, at run-time, through the discovery and binding mechanisms of the underlying SOA platform. In this paper, we do not address how SRML supports the dynamic aspects. The algebraic semantics of assembly is discussed in [6].

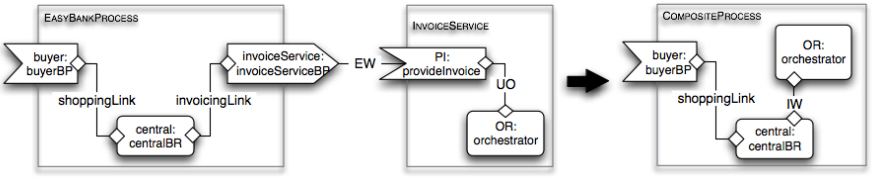


Fig. 2. The operation that assembles two modules into a module internalises the external wire EW that establishes a matching between the external interfaces (specifications) *invoiceService* (requires) and *PI* (provides)

The strategy of our encoding is precisely to abstract modules from BPEL processes, i.e. to identify the external interfaces (provides and requires) and the internal component that orchestrates the interactions involved, so that these BPEL-modules can then be used together with all other sorts of modules to define more complex services. In the resulting system, the original BPEL process will be connected with other components (possibly implemented in other languages like Java) through internal wires that establish the interaction protocols through which they communicate.

The encoding we propose involves both the module structure and the control flow involved in BPEL processes. A tool has been developed at the University of Leicester that provides semi-automated support [10] for this process. More precisely, it considers a subset of the BPEL constructs that concern service structure, produces a skeleton of a SRML module, and supports the manual definition of the missing aspects. The tool parses the XML tree representing the BPEL process with DOM and implements the transformation with a number of Java classes.

Our paper presents a more encompassing encoding than the one implemented by the tool. We encode structured activities (excluding scopes), control links and correlation sets. Table 1 shows which aspects of the BPEL control flow are considered in the encoding and which are supported by the tool. The encoding of the missing aspects is still work in progress. The main reason is that, as already explained, our aim is not to define yet another semantics for BPEL but to encode BPEL processes in a way that they can be used for defining SRML modules; as such, we have to take into account how the constructs provided by BPEL can be used within SRML. For instance, fault handling is a feature that, in SRML, is not handled at the same level of abstraction as the orchestration primitives (ditto for correlation sets). Therefore, we will consider the encoding of the throw primitive once we have extended SRML itself. The same applies to the constructs that relate to session handling, including correlation, which in SRML are treated as part of configuration management and treated in a fragment of the language that is still under construction.

Table 1. BPEL tags for control flow encoded in the tool and in this paper. BA stands for basic activity and SA for structured activity.

BPEL Tag/Construct	Tool	Encoding
Invoke , Receive, Reply, Assign (BA)	✓	✓
Wait, Empty, Exit (BA)	✗	✓
Throw (BA)	✗	✗
Sequence, Switch (SA)	✓	✓
Flow, While (SA)	✗	✓
Control Links, Scopes, Correlation Sets	✗	✗

The encoding of the SRML module structure from a BPEL process is in line with the one defined in [14] for embedding BPEL processes into SCA. The encoding of the control flow is inspired by the formal semantics of WS-BPEL in Petri Nets presented in [12]. The resulting approach is compositional because it describes any activity as a black box that is activated by the enclosing structured activity. This makes it easier to extend the encoding to the other types of BPEL activity.

To illustrate our approach we use a simple BPEL process — *easyBankProcess* – that receives an order from a buyer, uses an external service to create an invoice, and returns the invoice to the buyer. The following fragment defines the participants of the process and the links among them.

```
<process name="easyBankProcess"...
   <portType name="ShopPortType"> ...
   <portType name="InvoicingPortType"> ...
```

```
<portType name="BuyerPortType"> ...
<partnerLinkType name="invoicingLinkType">
   <role   name="invoiceService">   <portType   name="ns:InvoicePortType"/>
   </role>
</partnerLinkType>
<partnerLinkType name="shoppingLinkType">
   <role name="buyer"><portType name="ns:BuyerPortType"/></role>
   <role name="shop"><portType name="ns:ShopPortType"/></role>
</partnerLinkType>
<partnerLinks>
   <partnerLink name="invoicingLink" partnerLinkType="ns:invoicingLinkType"
      partnerRole="ns:invoiceService"/>
   <partnerLink name="shoppingLink" partnerLinkType="ns:shoppingLinkType"
      myRole="ns:shop" partnerRole="ns:buyer"/>
</partnerLinks>
```

The *partnerLinkType* elements define a link type between pairs of roles, each of which is associated with a certain *portType* element. A *portType* is a set of operations supported by a service. For example, *shoppingLinkType* defines a link type between two roles: *buyer*, of type *BuyerPortType*, and *shop*, of type *ShopPortType*. When only one role is specified, the other role can be associated with any *portType*, as in the case of *invoicingLinkType*. The *partnerLink* elements define an instance of *partnerLinkType* that specifies which of the roles belongs to the process (*myRole*) and to the partners (*partnerRole*).

The structure of the corresponding SRML module is illustrated in Fig. 1. Every partner role is represented as an external interface, either an EX-P (provides) or an EX-R (requires) interface, as discussed in Section 3. The central component represents all the roles assumed by the BPEL process (i.e., *myRole*). In the example, the SRML module has two external interfaces, *buyer* and *invoiceService*. The central component assumes the role of *shop* (i.e., port type *ShopPortType*) in *shoppingLink* and the generic role in *invoicingLink*.

Every role in the BPEL process is associated with a *portType* that declares a set of interactions:

```
<portType name="ShopPortType">
 <operation name="placeOrder">
  <input message="ns:placeOrderInput"/>
 </operation>
</portType>

<portType name="BuyerPortType">
 <operation name="receiveBill">
  <input message="ns:receiveBillInput"/>
 </operation>
</portType>
```

```
<portType name="InvoicingPortType">
 <operation name="doInvoice">
  <input message="ns:doInvoiceInput"/>
  <output message="ns:doInvoiceOutput"/>
 </operation>
</portType>
```

In SRML, each external interface and component, to which we refer as nodes, is an instance of a business protocol or business role, respectively. The business role and protocols of the SRML *easyBankProcess* module are given below. The business role *centralBR* supports (1) the operation *PlaceOrder* of *shopPortType* and (2) the complementary interactions for *invoicePortType* and *buyerPortType*.

```
BUSINESS ROLE centralBR is
  INTERACTIONS
   rcv shopPortType.placeOrder
    🔔 placeOrderInput.product:Product
   s&r invoicePortType.doInvoice
    🔔 doInvoiceInput.product:Product
    ✉ doInvoiceOutput.bill:Bill
   snd buyerPortType.receiveBill
    🔔 receiveBillInput.bill:Bill
```

```
BUSINESS PROTOCOL buyerBP is
INTERACTIONS
snd shopPortType.placeOrder
    🔔 placeOrderInput.product:Product
 rcv buyerPortType.receiveBill
    🔔 receiveBillInput.bill:Bill ...

BUSINESS PROTOCOL invoiceServiceBP is
INTERACTIONS
  r&s invoicePortType.doInvoice
    🔔 doInvoiceInput.product:Product
    ✉ doInvoiceOutput.bill:Bill ...
```

Business roles and protocols declare a number of interactions in a way that is similar to BPEL port types. The specification of a node *n* defined in the encoding supports the interactions that: (1) correspond to an operation supported by the *portType* associated with *n*, (2) the complementary interactions (i.e., a send is complementary to a receive) of the operations supported by the node to which *n* is wired. The fragment of the BPEL process that models the control flow declares two variables, *order* and *bill*, and defines the orchestration as a sequence of one receive and two invocations:

```
<variables>
  <variable name="order" messageType="ns:orderData"/>
  <variable name="bill" messageType="ns:invoiceData"/>
</variables>
<sequence>
  <receive name="rcvOrder" partnerLink="ns:shoppingLink" operation="ns:placeOrder"
     portType="ns:ShopPortType" variable="order" createInstance="yes"/>
  <invoke name="askInvoice" partnerLink="ns:invoicingLink" operation="ns:doInvoice"
      portType="ns:InvoicePortType" inputVariable="order" outputVariable="bill"/>
  <invoke name="sndBill" partnerLink="ns:shoppingLink" operation="ns:receiveBill"
      portType="ns:BuyerPortType" variable="bill"/>
</sequence>
```

Both business roles and business protocols define causal relationships among the events that occur as part of the supported interactions. Business roles express this causality in terms of an orchestration, i.e. state-transition based description of the process through which a component reacts to and initiates such events. Business protocols provide specifications of provided or required behaviour in terms of properties (expressed in a temporal logic) that abstract such causal relationships from the processes that run in the co-parties. In the case of provides-interfaces, we provide a specification of the protocol offered to the co-party and, in the case of requires-interfaces, that of the protocol that the co-party is required to adhere to.

Because BPEL does not support such semantically reach external interfaces, we focus exclusively on how to extract the orchestration of the business role *centralBR* from a BPEL specification. As an example, we present the orchestration of the central business role of the SRML module that is derived from the activities of the BPEL process *easyBankProcess*. In the next sections, we will discuss in detail how the different elements of the module are synthesised.

```
BUSINESS ROLE centralBR is
INTERACTIONS
      ...
ORCHESTRATION
local                    order.product:Product,
    start,exit,end,ra,rb,rc,rd,fa,fb,fc,fd:Boolean,
    na,nb,nc,cd:Natural
initialisation
    start=end=exit=false
    ra=rb=rc=rd=fa=fb=fc=fd=false
    na=nb=nc=nd=0
transition harness
    triggeredBy true
    guardedBy ¬start ∨ fa
    effects (¬start ⊃ start'∧ra')
      ∧ (fa ⊃ ¬fa'∧end')
transition transition_A (sequence)
    triggeredBy true
    guardedBy (ra ∨ fb ∨ fc ∨ fd) ∧ ¬exit
    effects (ra ⊃ rb'∧¬ra')
      ∧ (fb ⊃ rc'∧¬fb')
      ∧ (fc ⊃ rd'∧¬fc')
      ∧ (fd ⊃ fa'∧¬fd')
transition transition_B (receive)
    triggeredBy shopPortType.placeOrder🔔?
    guardedBy rb ∧ ¬exit
    effects ¬rb' ∧ fb'
      ∧ order.product'=shopPortType.placeOrder🔔.placeorderInput.product
transition transition_C (first invoke)
    triggeredBy true
    guardedBy rc ∧ ¬exit
    effects ¬rc'
    sends invoicePortType.doInvoice🔔!
      ∧ invoicePortType.doInvoice🔔.doInvoiceInput.product=order.product
transition transition_C' (first invoke 2nd part)
    triggeredBy invoicePortType.doInvoice✉?
    guardedBy
    effects fc' ∧ bill.bill'=invoicePortType.doInvoice✉.doInvoiceOutput.bill
transition transition_D (second invoke)
    triggeredBy true
    guardedBy rd
    effects ¬rd' ∧ fd'
    sends buyerPortType.receiveBill🔔!
      ∧ buyerPortType.receiveBill🔔!.receiveBillInput.bill=bill.bill
```

The local variables describe the state of the component: *order.product* and *bill.bill* are variables from the BPEL process; the others model control flow.

The orchestration is described by transition rules.

triggeredBy is a condition, typically a receive-event as in *transition_B*. When the condition is true the transition is triggered once the guard becomes true.

guardedBy is a condition that identifies the states in which the transition can take place

The sentence **sends** describes the interaction events that are sent and the values taken by their parameters

🔔 and ✉ identify request and reply events that may occur during conversational interactions.

3 Definition of the Module Structure

A BPEL process provides contextual information involving the external participants interacting with the process (i.e., the roles and port types wired to the business process through the partner links). In contrast, a SRML business role provides no information on the context in which it is used: the interface defines a set of interactions that is not partitioned according to the number of expected interacting parties. This is why, in

order to preserve the contextual information in the encoding, we map BPEL processes not to business roles, but to modules that represent contextualized business roles.

The transformation rules, in line with the QVT standard for Model Transformation [11], are represented with the following syntax: the source and target (fragments of) meta-models are represented by UML class diagrams and the correspondence is represented by meta-relations. Following [9], meta-relations are represented as dashed wires with a diamond enclosing the constraints of the relation instance. The shadowed classes on the right hand side are the classes added to the model by the rule. The diamond for a relation instance created by the rule is shadowed as well.

Fig. 3 illustrates the transformation rule of the root element of a WSDL/BPEL (left hand side) that generates a SRML module (right hand side) having the same name of the BPEL process, the module central component and corresponding business role.

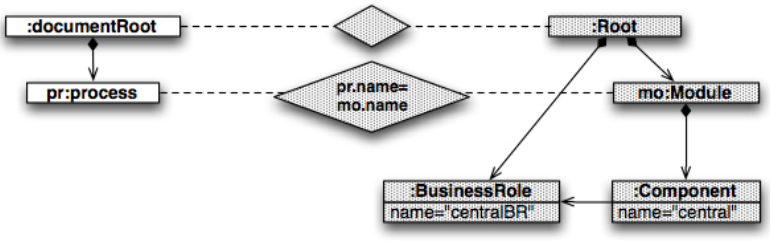


Fig. 3. Transformation rule generating the module

The application of the transformation rules in Fig. 3 to *myBankProcess* generates a module with name *myBankProcess* and one component named *central* of type *centralBP*. The set of other nodes that are wired to the central component is defined considering any *partnerLink* in the BPEL process. For any *partnerLink* element (representing a participant interacting with the business process) is created an external interface connected to the central component and the corresponding business protocol.

We discriminate between what must be encoded into an *EX-P* and into an *EX-R* by looking for the presence or absence of a *receive* operation having the *createInstance* attribute set to "*yes*", which is the mechanism used in BPEL to represent the invocation of a business process. The *partnerLink* associated with such operation is the one that, if it exists, invokes the service modelled by the BPEL process. Hence, an EX-P is created for such *partnerLink*. All the other *partnerLink* elements create an *EX-R* through the rule described in Fig. 4, which requires the absence of a *receive* operation having *createInstance* attribute set to "*yes*".

In the *myBankProcess* example, the rule described in Fig. 4 creates an *EX-R* named *invoiceService* of type *invoiceServiceBP*. The *EX-R* is connected to the central component with the wire *invoicingLink*. The rule for the *EX-P* interfaces creates an *EX-P* named *buyer* of type *buyerBP*, connected with the wire *shoppingLink*. The structure of the resulting module is the one presented in Fig. 1.

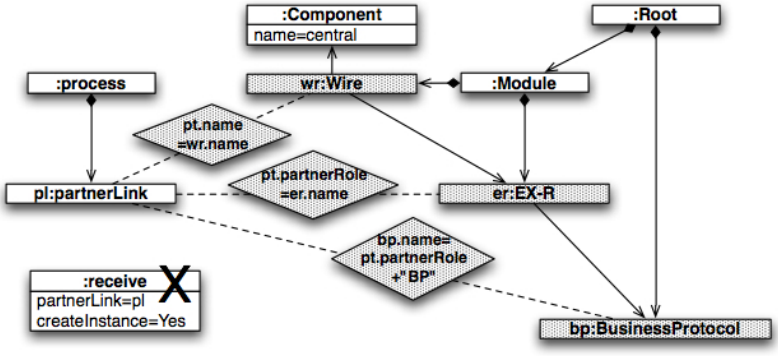


Fig. 4. Transformation rule generating a requires-interface and the corresponding wire that connects it to the central component. The rule that generates the external provides-interface is similar but, instead, requires the absence of a receive operation having *createInstance* attribute set to "*yes*".

Using the same method, we define the transformation rules for the set of interactions supported by all the nodes. Every node corresponds to a BPEL *portType* element. A node represents one or more port types and must support, for any operation in the *portType,* a corresponding SRML interaction. In addition, it must support the complementary interactions of the *portType* elements connected to the node through a *partnerLink*. We omit the details of the rules for the interactions and their parameters. Table 2 shows, for each WSDL operation type, the corresponding SRML interaction types/parameters and their complements. For example, the request-response operation is encoded as a *r&s* (i.e., receive and send) interaction, whose complementary element is an *s&r* interaction. The input parameters are encoded as 🔔-parameters, which are the parameters for transmitting data when the interaction is initiated.

The current OASIS draft for WS-BPEL [2] specifies that some of the WSDL operations must not be supported by BPEL processors (i.e., *notification* and *solicit-response*); hence, we consider the supported operations only.

Table 2. WSDL operations and SRML interactions

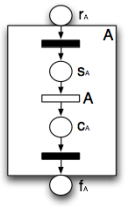
WSDL	SRML	SRML (complementary)
one-way	*rcv*	*snd*
> input parameter	> 🔔 parameter	> 🔔 parameter
request-response	*r&s*	*s&r*
> input parameter	> 🔔 parameter	> 🔔 parameter
> output parameter	> ✉ parameter	> ✉ parameter

We encode the operation with name '*op*' of the *portType* '*pt*' as the interaction with name '*pt.op*'. Because any interaction event in SRML may occur at most once during a session, we have to define a family of interactions for each operation. This family defines an arbitrary number of interactions, each identified by an index

(e.g., *pt.op[i]*). For each interaction we define a variable *pt.opB* (and also *pt.opE* for *r&s* and *s&r* interactions) of type *Natural* that is initially *0* and is incremented at each occurrence of the 🔔- (and ✉-) event of the interaction *pt.op* and stores the index that must identify the next occurrence. In the example presented in Section 2, we omitted these indexes for readability, as each operation is invoked only once.

4 Transformation of Control Flow

The encoding of control flow into SRML has been inspired by the Petri Net-based semantics of WS-BPEL presented in [12]. Therein, a generic activity *A* is represented as a Petri net having: (1) an initial state r_A, in which the transition is ready to be executed, (2) a final state f_A, (3) the state s_A/c_A in which the activity starts/completes.




```
transition A
  triggeredBy …
  guardedBy ra ∧ …
  effects                                  ¬ra'
     ∧ fa'
     ∧ …
  sends …
```

Fig. 5. A simplified version the Petri Net representation of a generic BPEL activity *A* [12], and the corresponding SRML transition. Additional statements have to be added to model the execution of the activity.

In the SRML encoding, the execution of the orchestration begins with a "harness" transition that uses a special boolean variable *start,* initially set to false. The harness triggers the activity *A*, corresponding to the root activity of the BPEL process, by setting *ra* to *true*:

```
transition harness
triggeredBy true
guardedBy ¬start ∨ fa
effects (¬start ⊃ start'∧ra') ∧ (fa ⊃ ¬fa'∧end')
```

4.1 Encoding Basic Activities

In a BPEL process, a parameter sent by an operation consists of a *Variable* element that has already been declared and assumed a meaningful value by means of an *Assign* activity. Fig.6. presents the transformation rule for creating a local variable in the orchestration of the SRML module from the corresponding variable in the BPEL process. The *DataType* object, defining the type of the variable, does not need to be created if an object already exists for the same type.

Some additional local variables have to be defined for handling the control flow:

- *start*/*end*, of type Boolean, is true when the process instance starts/ends.

- *exit*, of type Boolean, disables, when true, the execution of any transition. It is initially false. The value may be changed by the *Exit* activity.
- *ra* and *fa* for any activity *A*.

To improve readability, we present the transformation rules for activities by using a textual notation, showing the correspondence between the two languages. The generated transitions belong to the orchestration of the business role *centralBR*.

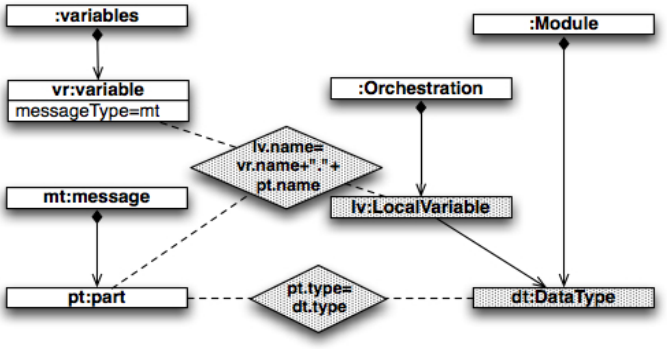


Fig. 6. Transformation rule for variables declaration. A BPEL variable refers to a message composed by parts. A local variable is created, for each part of each BPEL variable, in the orchestration of the business role typing the central component that is the only orchestration in the module.

Assign. The *assign* activity refers to a couple of variables and to a specific part of the message that types each variable. The *assign* activity in the BPEL code fragment below (on the left) is encoded into the SRML transition *transition_A* also shown below (on the right). The effects of the transition include the assignment of the part *formP* of the variable *fromV* to the part *toP* of *toV*.

```
<assign>
 <copy>
  <from variable="fromV" part="fromP"/>
  <to variable="toV" part="toP"/>
 </copy>
</assign>
```

```
transition transition_A
triggeredBy true
guardedBy ra ∧ ¬exit
effects ¬ra' ∧ fa'
    ∧ toV.toP'=fromV.fromP
```

Invoke. The *invoke* activity is used to invoke a service and may refer to either a one-way or request-response WSDL operation. The invocation, in BPEL, is modelled from the perspective of the invoked party and in SRML corresponds to either a *snd* or a *s&r* interaction of the central component. The BPEL code fragment below invokes a request-response operation.

```
<invoke      partnerLink="pl"      portType="pt"      operation="op"
      inputVariable="iv" outputVariable="ov"/>
```

The invoke statement above is transformed into the two SRML transitions below.

```
transition transition_A
triggeredBy true
guardedBy ra ∧ ¬exit
effects ¬ra' ∧ pt.opB'=pt.opB+1
sends pt.op[pt.opB]🔔!
 ∧ pt.op[pt.opB]🔔.iv.p1=iv.p1 ∧ …
 ∧ pt.op[pt.opB]🔔.iv.pn=iv.pn
```

```
transition transition_A'
triggeredBy pt.op[pt.opE]✉?
guardedBy ¬exit
effects fa'
∧ ov.p1'=pt.op[pt.opE]🔔.ov.p1 ∧ …
∧ ov.pm'=pt.op[pt.opE]🔔.ov.pm
∧ pt.opE'=pt.opE+1
```

The first part of the request-response is modelled by *transition_A* that sends the interaction event *pt.op[pt.opB]🔔!* where *pt* is the name of the *portType* and *op* is the operation. The parameters are assigned to the corresponding parts of the input variable *iv*. We assume, with no loss of generality, that the message type of *iv* consists of the parts *p1,…,pn*. The second part of the request-response is represented by *transition_A'* that receives the interaction event *pt.op[pt.opE]✉?* and assigns the value of the output parameters (in all the *m* parts) to the output variable. We do not need to add a guard to enable *transition_Ab* after *transition_A* as SRML ensures that, *pt.op[i]✉?* is always enabled after (and only after) *pt.op[i]🔔!*.

The invoke of a one-way operation, where the output variable is not specified, is transformed in *transition_A* where the effects include the statement *fa'*.

Receive and Reply. The *receive* activity refers to either a one-way or a request-response operation and it is encoded according to the transformation rule that follows.

```
<receive partnerLink="pl"
   portType="pt"
   operation="op"
    variable="v"
     createInstance=…/>
```

```
transition transition_A
triggeredBy pt.op[pt.opB]🔔?
guardedBy ra ∧ ¬exit
effects ¬ra' ∧ fa' ∧ pt.opB'=pt.opB+1
  ∧ v.p1'=pt.op[pt.opB]🔔.v.p1 ∧ …
  ∧ v.pn'=pt.op[pt.opB]🔔.v.pn
```

The *reply* activity refers to either a one-way or (the second part of) a request-response operation. In case of a one-way operation it is encoded according to the transformation rule that follows. The rule for the request-response is similar but it sends the interaction event *pt.op[pt.opE]✉!*.

```
<reply  partnerLink="pl"
    portType="pt"
   operation="op"
  variable="v"…/>
```

```
transition transition_A
triggeredBy true
guardedBy ra ∧ ¬exit
effects ¬ra' ∧ fa' ∧ pt.opB'=pt.opB+1
sends  n.pt.op[pt.opB]🔔! ∧ n.pt.op[pt.opB]🔔.v.p1=v.p1
  ∧ … ∧ n.pt.op[pt.opB]🔔.v.pm=v.pm
```

Wait. The *wait* activity specifies a deadline (time interval or future instant of time).

```
<wait> <for>t</for>(<until>t</until>)</wait>
```

SRML provides a number of primitives for handling time [5], including the function *now* which returns the present time from a global clock. The *wait* activity is transformed into the transitions above:

```
transition transition_A (for)
triggeredBy true
guardedBy ra ∧ ¬exit
effects timeA=now ∧ ¬ra'

transition transition_A' (for)
triggeredBy now=timeA+t
guardedBy ¬exit
effects fa'
```

```
transition transition_A (until)
triggeredBy  now≥t
guardedBy ra ∧ ¬exit
effects fa' ∧ ¬ra'
```

Exit and Empty. The *exit* activity terminates the execution of the process. It is encoded into *transition_A* below that gives the value true to the Boolean local variable *exit*. This disables any further transition. The *empty* activity performs no action. It is encoded into *transition_B* below.

```
transition transition_A (exit)
triggeredBy true
guardedBy      ra      ∧      ¬exit
effects ¬ra' ∧ fa' ∧ exit'
```

```
transition transition_B (empty)
triggeredBy true
guardedBy      rb      ∧      ¬exit
effects ¬rb' ∧ fb'
```

4.2 Encoding Structured Activities

Sequence. The *sequence* activity is used to execute two activities in sequence, in the specified order. Let us suppose we have two activities *A* and *B* represented by two transitions *transition_A* and *transition_B*. We model the sequence activity *X* as the transition *transition_X*. We denote with *rx*, *ra* and *rb* the boolean variables that trigger the execution of *X, A* and *B*, respectively. The variables *fx*, *fa* and *fb* denote the end of the corresponding activity. The transition *transition_X* is executed three times: when the parent activity triggers *X* (by setting *rx* to true), when the enclosed activity *A* terminates (and sets *fa* to true) and, analogously, when *B* terminates.

```
<sequence
   name="X">
   activity A
   activity B
</sequence>
```

```
transition transition_X
triggeredBy true
guardedBy (rx ∨ fa ∨ fb)  ∧ ¬exit
effects (rx ⊃ ra'∧¬rx) ∧ (fa ⊃ rb'∧¬fa') ∧ (fb ⊃ fx'∧¬fb')
```

Flow. The *flow* activity executes the enclosed activities in parallel. We consider the case of two parallel activities with no loss of generality. The transition *transition_X* models the flow activity for *A* and *B*. The flow activity on the left side is transformed into *transition_X*. The transition is executed two times: when the parent activity triggers *X* (by setting *rx* to true) and when the enclosed activities *A* and *B* both terminate (i.e., synchronization).

```
<flow name="X">
   activity A
   activity B
</flow>
```

```
transition transition_X
triggeredBy true
guardedBy (rx ∨ (fa ∧ fb)) ∧ ¬exit
effects (rx ⊃ ra'∧ rb'∧¬rx') ∧ (fa ∧ fb ⊃ fx'∧¬fa'∧¬fb')
```

Switch. The *switch* activity executes one of two activities, depending on a condition. If all the conditions are false no activity is executed. The conditions are evaluated in the specified order. We consider a *switch* statement involving two conditions with no loss of generality. The switch activity on the left side is transformed into *transition_X*.

The transition is executed two times: when the enclosing activity triggers *X* (by setting *rx* to *true*), when one the enclosed activities terminates.

```
<switch name="X">
 <case>
  <condition>      z1      </condition>
  activity A
 </case>
 <case>
  <condition>      z2      </condition>
  activity B
 </case>
</switch>
```

```
transition transition_X
triggeredBy true
guardedBy (rx ∨ fa ∨ fb) ∧ ¬exit
effects (rx∧z1 ⊃ ra'∧¬rx')
  ∧ (rx∧¬z1∧z2 ⊃ rb'∧¬rx')
  ∧ (rx∧¬z1∧¬z2 ⊃ fx'∧¬rx')
  ∧ (fa∨fb ⊃ fx'∧¬fa'∧¬fb')
```

Pick. The *pick* activity waits for a set of events, each associated to an activity, and executes (only) the activity associated to the first event that occurs. The events can be triggered by an external message or by an alarm. We consider, without loss of generality, a set of two events: one triggered by a message and one triggered by the alarm. The *Pick* activity on the left side is transformed into *transition_X* and *transition_X'*. First *transition_X* is executed: *rx* becomes *false* and the the present time is stored in the variable *tX*. Then *transition_X'* is executed when either *e1🔔?* occurs or when the deadline expires and triggers the corresponding activity, *A* or *B*. Notice that, because of its guards, *transition_X'* is executed only once: before either *A* or *B* is triggered. ***P**_e1[i]🔔?* is true if the event *e1[i]🔔?* occurred in the past. The transition *transition_X* is executed again when either *fa* or *fb* is *true; fx* is then set to *true*.

```
<pick name="X">
 <onMessage      e1>
    A
 </onMessage>
 <onAlarm>
  <for>           t
  </for>
    B
 </onalarm>
</pick>
```

```
transition transition_X
triggeredBy true
guardedBy (rx ∨ fa ∨ fb) ∧ ¬exit
effects (rx ⊃ ¬rx'∧tX'=now+t)
    ∧ (fa ⊃ ¬fa'∧fx') ∧ (fb ⊃ ¬fb'∧fx')

transition transition_X'
triggeredBy e1[i]🔔? XOR now=tX
guardedBy ¬ra ∧ ¬rb ∧ ¬exit
effects ra'=P_e1🔔? ∧ rb'=now=tX
```

While. The *while* activity iterates an activity *A* until a condition is true. The *while* activity on the left side is transformed into *transition_X*. The transition is executed the first time when *rx* is true and then when the execution of any enclosed activity terminates. During the first iteration *rx* is set to false. The condition determines either the execution of *A* or the end of *transition_X*. The end of a previous iteration (*fa=true*) determines either the end of *the while* activity or execution of the next iteration (*ra=true*), depending on the condition *z*.

```
<while name="X">
 <condition>z
 </condition>
  activity A
</while>
```

```
transition transition_X
guardedBy (rx ∨ fa) ∧ ¬exit
effects rx ⊃ (¬rx'∧ ra'≡z ∧ fx'≡¬z)
     ∧ fa ⊃ (¬fa' ∧ ra'≡z ∧ fx'≡¬z)
```

5 Conclusions and Future Work

In this paper, we have discussed an encoding of WS-BPEL processes into SRML – the modelling language that is being developed within the SENSORIA project for

supporting the engineering of complex services. In SRML, modules provide abstractions of composite services as provided through assemblies of components and externally services procured and bound at run-time.

SRML is inspired on SCA [13]. Like in SCA, the components used inside a module need not be homogeneous: they can be Java programs, BPEL process, wrapped-up legacy systems, and so on. SRML provides a language in which these components can be modelled as transition-based systems obtained through abstraction mappings. The purpose of this paper was precisely to illustrate the abstract encoding that we defined for BPEL-processes, which is richer than the one provided for SCA because we are able to encode the business logic operated by the component. We should stress that the purpose of the encoding is not to provide a new semantics for BPEL but to abstract from BPEL processes the SRML modules that allow them to be combined with other modules to define more complex services, avoiding having to develop required orchestrations from scratch. This is why the proposed encoding does not consider aspects that, like fault handling and correlation sets, are not directly relevant for the fragment of SRML that is concerned with composition. We are working on an extension of the encoding that considers some of the missing aspects in the context of the fragment of SRML that handles configuration management.

Our encoding is performed through model-transformation rules based on triple graph grammars. We present the design of some of the transformation rules that we developed so far. Their implementation is straightforward if using a tool for modeling graph transformations: we are using the Tiger tool environment [3], based on Eclipse Modelling Framework (EMF). The source WSDL/BPEL meta-model derives from the combination of the meta-model obtained from the XSD specifications of WSDL and the meta-model of WS-BPEL defined in the context of the Eclipse BPEL project[1]. The target SRML meta-model has been produced using the Eclipse Graphical Modelling Framework (GMF). The meta-models are modelled as EMF trees. In this way, it will be possible to easily implement the transformation rules using tools s.a. Tiger [3], which transform meta-models expressed in EMF.

Another strong point of our approach is that it will make it possible for the rich analysis framework being developed in SENSORIA [15] to be used for analysing and verifying properties of BPEL processes. The goal is to reason about the properties of modules assembled from possibly heterogeneous components, e.g. through model checking [7].

Acknowledgments

We would like to thank our colleagues in SENSORIA for many useful discussions, and our Leicester colleagues Reiko Heckel and Karsten Ehrig in particular for guidance in graph transformations and Tiger. Finally, we would like to thank the reviewers for extensive and profound comments and suggestions that have definitely improved the quality of this version.

[1] http://www.eclipse.org/bpel/

References

1. Abreu, J., Bocchi, L., Fiadeiro, J.L., Lopes, A.: Specifying and composing interaction protocols for service-oriented system modelling. In: Formal Methods for Networked and Distributed Systems. LNCS, Springer, Heidelberg (to appear, 2007)
2. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Technical report, TC OASIS (2007), available from http://www.oasis-open.org/
3. Biermann, E., Ehrig, K., Koehler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the Eclipse Modeling Framework. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 425–439. Springer, Heidelberg (2006)
4. Bisztray, D., Heckel, R.: Rule-level verification of business process transformation using CSP. In: Graph Transformation and Visual Modeling Techniques. Electronic Communications of the EASST (2007)
5. Fiadeiro, J.L., Lopes, A., Bocchi, L.: A formal approach to service-oriented architecture. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 193–213. Springer, Heidelberg (2006)
6. Fiadeiro, J.L., Lopes, A., Bocchi, L.: Algebraic semantics of service component modules. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 37–55. Springer, Heidelberg (2007)
7. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating UML state machines. In: ACIS International Conference on Software Engineering Research, Management and Applications, pp. 331–338 (2004)
8. Grunske, L., Geiger, L., Lawley, M.: A graphical specification of model transformations with triple graph grammars. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 284–298. Springer, Heidelberg (2005)
9. Hausmann, J.H.: Dynamic Meta Modelling: a semantics description technique for visual modelling languages. PhD Thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, Germany (2005)
10. Hong, Y.: WSDL and BPEL to SRML-P Language Transformation. MSc Dissertation, University of Leicester (2006)
11. Object Management Group, MOF QVT Final Adopted Specification (2007), available from: http://www.omg.org/docs/ptc/05-11-01.pdf
12. Ouyang, C., Verbeek, E., van del Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL (revised version). BPM Center Report BPM-05-15, BPMcenter.org (2005)
13. SCA Consortium Building Systems using a Service Oriented Architecture. Whitepaper version 0.9 (2005), available from: http://www.oracle.com/technology/tech/webservices/standards/sca/pdf/SCA_White_Paper1_09.pdf
14. SCA Consortium SCA Client and Implementation Model Specification for WS-BPEL. Version 1.00 (2007), available from: http://www.osoa.org/download/attachments/35/SCA_ClientAndImplementationModelforBPEL_V100.pdf?version=1
15. Wirsing, M., Bocchi, L., Clark, A., Fiadeiro, J., Gilmore, S., Hölzl, M., Koch, N., Pugliese, R.: SENSORIA: Engineering for Service-Oriented Overlay Computers (submitted, 2007)