Formal Verification of BPEL4WS Business Collaborations

Jesús Arias Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos

Telematic Engineering Department
Carlos III University of Madrid
Avda. Universidad, 30
28911 Leganés, Madrid, Spain
{jaf,luis,cdk}@it.uc3m.es

Abstract. Web services are a very appropriate communication mechanism to perform distributed business processes among several organisations. These processes should be reliable, because a failure in them can cause high economic losses. To increase their reliability at design time, we have developed VERBUS, a framework for the formal verification of business processes. VERBUS can automatically translate business process definitions to specifications verifiable in several available tools. It is based on a modular and extensible architecture: new process definition languages and verification tools can be added easily to the framework. The prototype of VERBUS presented in this work can verify BPEL4WS process specifications, by translating them to Promela. The Promela specifications are verified with the well known model checker Spin. In this paper we describe the general architecture of VERBUS and how BPEL4WS specifications are translated and verified. The explanation is completed by describing what types of properties can be verified and providing an overview of the implementation.

1 Introduction

Inter–organisational business processes are a key technology for business to business collaborations. Nowadays many enterprises have automated their internal business processes with workflow technologies. They have now a new challenge: the automation of their collaborations with partner enterprises, in open and very dynamic environments, to accelerate their business in a cost–effective manner. Web services are a promising technology to support these type of collaborations [1, 2]. It is an XML–based middleware technology that provides RPC–like remote communication, using in most cases SOAP over HTTP.

Web services provide a state–less communication mechanism: WSDL can specify remote operations and their input and output parameters, but not the relations between several operations. Business processes have a state. Therefore, new languages are necessary to execute business processes on top of Web services. Several languages have been used to model these business processes [2]. They are often called *choreography languages*, because they specify the order in

K. Bauknecht, M. Bichler, and B. Pröll (Eds.): EC-Web 2004, LNCS 3182, pp. 76–85, 2004.

which the activities of the process must be executed. The most important are BPEL4WS [3], BPML [4] and ebXML BPSS [5]. Among them, only BPEL4WS is specific for Web services. The Web Services Choreography Working Group of the World Wide Web Consortium (W3C) is also currently developing a new Web services choreography language.

Complex business collaborations require the specification of complex business processes. Specificating complex processes is error prone, due to concurrency in the execution of activities, the possibility of communication errors, faults in remote systems, etc. Enterprises can only trust in this technology if the correctness of the processes can be ensured, because a failure in them can cause high economic losses. In this work we present VERBUS (*VERification for BUSiness processes*), a system for automatic verification of business processes using model–checking. Its main objective is to help process designers to ensure the correctness of the defined processes. The current prototype receives an input BPEL4WS process specification and a set of properties that the designer wants to verify. Then the system automatically translates the specification to a formal specification language and verifies it using a model–checker. If a property is found to be false, the system gives a counter–example to the designer. VERBUS is modular and extensible: new process definition languages and verification tools can be easily added to the framework.

Several works have been done previously on business processes verification. Woflan [6] is a Petri–Net based verification tool. It can perform verifications on workflow definitions, and was integrated with several commercial workflow management systems. In [7] formal semantics are defined for UML activity diagrams to allow the verification of workflow processes defined with these formalisms. It uses the SMV model checker. A framework for the verification of Web services is proposed in [8]. It can perform analysis on a Web service described with DAML-S by translating the description to a Petri–Net based model.

None of these works can be applied to BPEL4WS processes. The results of these works are specific, both in terms of process modelling language and verification tool. However, VERBUS proposes a framework in which several process definitions languages and verification tools can be integrated, based on a common intermediate formal model. This formal model is very simple, but can represent complex semantics like the fault handling mechanism of BPEL4WS in a straightforward way, as showed in the next sections.

This paper is organised as follows. Section 2 makes a brief introduction to BPEL4WS. Section 3 describes the main architecture of VERBUS. Section 4 explains how VERBUS translates BPEL4WS processes to its formal model. Section 5 explains the possibilities that VERBUS offers for performing verifications. Finally, the main conclusions of this work are summarised.

2 Modelling Business Processes with BPEL4WS

Business Process Execution Language for Web Services (BPEL4WS) [3] is an XML notation for specifying business process behaviour based on Web services.

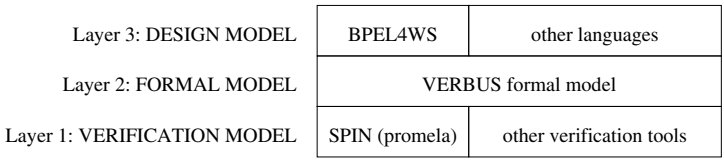

Fig. 1. Architecture of the VERBUS framework

It was developed by Microsoft, IBM, BEA, Siebel Systems and SAP. It allows the specification of executable processes and business protocols. An executable process defines the behaviour of a participant in a collaboration. A business protocol defines the message exchange of all the participants involved in a collaboration.

BPEL4WS provides basic activity structuring (sequential and parallel composition, conditional execution and loops), variables, hierarchical activity composition with scopes, Web services based communication (invocation of remote Web services operations and providing Web services operations to remote systems), event handling and fault and compensation handling mechanisms.

Activities are executed in the context provided by a *scope*. Each scope contains activities and data. All the activities contained in a scope share the same context. Scopes can be hierarchically nested. The root of the hierarchy is the process, that can be viewed as a special scope. It can contain any number of children scopes. Each scope can contain also children scopes, and so on. Scopes store data in variables, that can be accessed by any activity contained by this scope, or any scope nested in it.

Scopes are also important for fault and compensation handling. The fault handling mechanism is very similar to the Java exception handling mechanism. An activity can throw *faults* to notify an error in its execution. Faults can be handled by a scope or, if not, they are re–thrown to the next enclosing scope. Section 4.4 explains this in detail. The compensation mechanism allows the specification of transactional behaviour for business processes. Each scope can define a compensation handler to make the rollback of the actions that were executed by the scope. This handler is executed when the scope has successfully completed but it must be compensated due to faults that occurred in other scopes.

3 The VERBUS Framework

VERBUS is a modular and extensible framework for the verification of business processes. It proposes an architecture with three layers, as showed in Fig. 1. The *design model* (layer 3) deals with the design of business processes, using specification languages like BPEL4WS or BPML, for example. The *formal model* (layer 2) deals with the specification of processes using a formal model. VERBUS defines its own formal model for this layer, based on Finite State Machines (FSMs). The *verification model* (layer 1) deals with the verification of business processes. Several general–purpose verification tools can be placed in this layer, such as Spin [9] or SMV [10].

Layer 3 specifications are translated to formal (layer 2) specifications using automatic translation tools. There is one tool for each layer 3 language. Layer 2 specifications are also translated to verification languages using automatic translation tools. There is one tool for each verification language, because each verification tool has normally a specific input language. Layer 2 is an intermediate layer that increases the modularity and extensibility of the system, by disconnecting the design and verification layers. Thus, only one translation tool is needed when introducing a new verification tool in the framework, and it will be available to specifications defined in any language in the design layer. The same applies to the introduction of new layer 3 languages.

The current prototype of VERBUS implements two translation tools. One of them translates a BPEL4WS process specification to a formal specification. The other translates a layer 2 specification to a Promela [9] specification, that can be verified with the model–checker Spin.

3.1 The VERBUS Formal Model

The formal model used in the layer 2 of VERBUS is based on FSMs. It is briefly presented here. Its formal definition is given in [11].

A process is composed by a set of attributes and a set of transitions. At a given moment, each attribute has a value within a set of possible values. The value of all the attributes of the process at a given moment establishes its state. The process progresses from one state to another by means of transitions. A transition is a pair of states (origin and destination) that defines a possible progress of the process. The process starts at an initial state. Then it fires transitions, until it reaches a state that is not in the origin of any transition. This state implies the completion of the process and is called a *final state*.

The FSM of a business process has normally many transitions. In order to avoid defining them explicitly, VERBUS represents them with *functional transitions*. A functional transition is defined by two predicates: *domain* and *action*. The domain defines a set of states that are origin of transitions. The action defines how these origin states change to obtain the final states. Therefore a functional transition represents a set of transitions that share a similar behaviour.

The concepts of entity, entity type and activity are introduced as notational elements, to make specifications more readable. However, they do not affect the basic formalism. An entity type is a group of typed fields (boolean, enumerated or integer types). An entity is an instance of an entity type. Each field of an entity type generates as many attributes as times its data type is instantiated. An activity is a logical unit for grouping related functional transitions.

4 Translating BPEL4WS to the VERBUS Formal Model

The translation of BPEL4WS specifications to the formal model is the most complex functionality of VERBUS. This section summarises how it is done. The translation of *sequences* and the *fault–handling* mechanism were selected

```
<xsd:complexType name="Order">
  <xsd:sequence>
    <xsd:element name="productId"
                 type="xsd:string" />
    <xsd:element name="colour">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="white"/>
          <xsd:enumeration value="red"/>
          <xsd:enumeration value="blue"/>
          <xsd:enumeration value="black"/>
  (...)
</xsd:complexType>
<message name="OrderMessage">
  <part name="urgent" type="xsd:boolean"/>
  <part name="order" type="tns:Order"/>
</message>
<variable name="order"
          messageType="tns:OrderMessage"/>
```

```
enttype OrderMessage {
  urgent: boolean;
  order__productId: abstract;
  order__colour: enum (white, red, blue,
                       black);
}
entity order: OrderMessage;
```

Fig. 2. Mapping between BPEL4WS variables and VERBUS entities

as representative examples of how the translation is performed. The current prototype of VERBUS can translate also any of the other activities. In the web page of the VERBUS project (`http://www.it.uc3m.es/jaf/verbus`) there are several examples that show how VERBUS translates these other activities.

4.1 Variables

BPEL4WS variables are mapped to VERBUS entities. Each variable is an instance of a data type defined by a WSDL message, an XML element or an XML Schema type. First, the data type is transformed to an entity type. Then it is instantiated as an entity. Simple data types are transformed to VERBUS data types if possible (`boolean`, `enumerated` and `integer`), or declared as *abstract* otherwise. Complex data types are transformed by recursively transforming their components. Fig. 2 shows an example. The message type `OrderMessage` has two parts: `urgent` and `order`. The part `urgent` is a simple type and so it is translated to a boolean field in the VERBUS entity type. The part `order` is a complex type: the sequence of the elements `productId` (`string`) and `colour` (enumerated data type). It is translated to two fields, one for each element.

4.2 Activities

The execution of each BPEL4WS activity instance is controlled by a life–cycle. Depending on the type of activity two different life–cycle types were identified in this work. The *general life–cycle* is used for activities that can have handlers (`process`, `scope` and `invoke`). The *simple life–cycle* is used for the other activities. Both life–cycle types are represented in Fig. 3.

Each activity is mapped to an entity and several functional transitions. The entity represents the state of the activity in its life–cycle. The functional transitions represent the way the activity can progress through its life–cycle and how

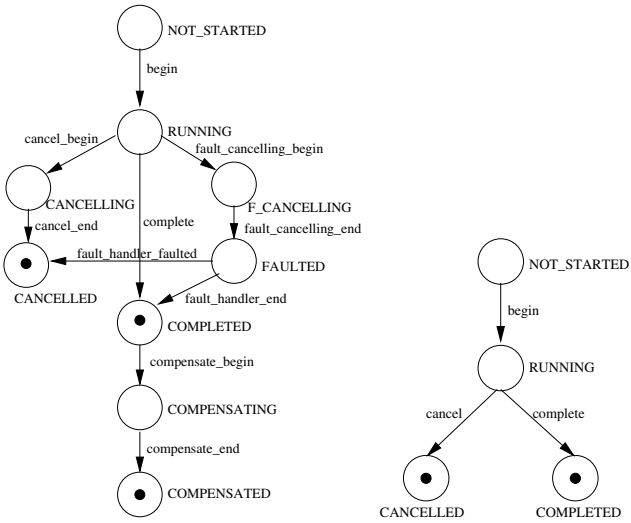

Fig. 3. Life–cycle for activities. The general life–cycle is on the left and the simple life–cycle is on the right. States are labelled with uppercase letters and transitions with lowercase letters. States containing a black dot can be final states of the life–cycle

it affects to the attributes of the process. The concrete functional transitions of each activity depend on its activity type. However, there are several rules that are common to almost all activities.

Activity instances have always a *begin* transition, that represents the beginning of their execution. Its domain represents the preconditions of the activity. Normally it is a condition that checks the state of other activities (depending on the type of its parent activity), and its own state (it must be `not_started`). Its action changes the state of the activity to `running`.

Activity instances have normally a `complete` transition also, that represents the end of their execution. Its domain checks that the activity is in `running` state. Depending on the activity type, it may include other conditions. Its action puts the activity in `complete` state, and can change also attributes of the process to represent the effects of the activity execution.

If an activity has a non–deterministic behaviour then it is normally modelled with several mutually–exclusive functional transitions. Each of them represents a different behaviour of the activity. Examples of non–deterministic behaviour are `pick` activities, activities that can throw faults, `receive` or `invoke` activities that can receive messages with different data, etc.

4.3 Sequence Activity

The BPEL4WS sequence activity can contain one or more inner activities, that must be executed in sequential order. Given an activity in the sequence, it can begin its execution only if its preceding activity has been completed. The sequence itself is completed when its last inner activity has been completed.

```
<sequence name="main">
 <receive name="init" .../>
 <switch name="switch">...</switch>
 <scope name="end">...</scope>
</sequence>
```

```
activity main_act_2 {
 transition begin {
   domain: {main_act_2_lc__.state=not_started}
   action: {main_act_2_lc__.state=running}}
 transition complete {
   domain: {(main_act_2_lc__.state=running &
             end_act_11_lc__.state=completed)}
   action: {main_act_2_lc__.state=completed}}
}
activity init_act_3 {
 transition begin {
   domain: {(main_act_2_lc__.state=running &
             init_act_3_lc__.state=not_started)}
   action: {init_act_3_lc__.state=running}}
 transition complete {...}
}
activity switch_act_4 {
 transition begin {
   domain: {(init_act_3_lc__.state=completed & ...)}
   action: {(switch_act_4_lc__.state=running & ...)}}
 ... }
activity end_act_11 {
 transition begin {
   domain: {(switch_act_4_lc__.state=completed & ...)}
   action: {end_act_11_lc__.state=running}}
... }
```

Fig. 4. Mapping a BPEL4WS sequence. The examples is abbreviated to highlight the most important conditions and transitions

This behaviour is modelled by adding a condition to the domain of the `begin` transition of each inner activity and a condition to the domain of the `complete` transition of the sequence activity. The condition added to the first inner activity states that the sequence activity must be in `running` state. The condition added to the other inner activities states that the previous activity must be in `completed` state. The condition added to the `complete` transition of the sequence activity states that the last inner activity must be in `completed` state. Fig. 4 shows an example.

4.4 Fault Handling

BPEL4WS has a powerful fault–handling mechanism. The `process`, `scope` and `invoke` activities can contain fault handlers. When a fault is thrown in a given activity, a handler in the immediately enclosing `scope`, `process` or `invoke` is selected, based on the fault name and variable type. Before the handler is executed, all the running inner activities of this scope are cancelled. If no handler is appropriate, then the whole scope is cancelled, and the fault is re–thrown to the next enclosing scope. A fault that reaches the `process` level causes the cancellation of the whole process.

To manage the fault–handling mechanism, several attributes are added to the general life–cycle entity type. One of them is boolean and its value is `true` when a fault has occurred in the activity. The other is enumerated, and its value

specifies which is the selected handler when a fault has occurred. The activity contained in each fault handler checks these variables as a precondition for its execution.

The cancellation mechanism is needed to implement fault–handling. It is implemented in VERBUS in this way: a scope that must be cancelled puts itself in `fault-cancelling` state. Its inner activity has a transition that cancels itself if the scope is in `fault-cancelling` state. In a similar way, if this activity has inner activities, they detect this cancellation and cancel themselves, and so on. The scope has a transition that puts itself in `faulted` state when none of its inner activities is running. At this moment the activity of the fault handler is allowed to start its execution. When this activity completes its execution, the scope puts itself in `completed` state.

4.5 Prototype Implementation

The current prototype of VERBUS is mainly composed by a BPEL4WS to VERBUS translator and a VERBUS to Promela translator. It is based on the BPEL4WS specification version 1.1 [3]. It was developed in Java and uses the open source libraries Xerces, Xalan and WSDL4J. The prototype works in command line, but a graphical user interface is currently under development. It will incorporate a graphical editor of BPEL4WS processes.

The main lack of the current prototype is the *compensation* mechanism, because of the complexity associated with it: a copy of all the variables must be saved for each completed activity instance, because they must be compensated using the value that variables had when they were completed. `While` loops even make this more complex, because multiple instances of each inner activity can be created. This feature will be handled in future versions of VERBUS, by storing a copy of attributes for each completed activity. This will increase the complexity of the verification, and therefore a configurable parameter will be added to limit the maximum number of activity instances.

5 Verification of Processes

The main goal of VERBUS is the verification of business process specifications. VERBUS allows the modeller to state properties that must be true for a given process specification, and checks whether these properties are true or false for it. If some property is found to be false, VERBUS gives a counterexample. From the point of view of the formal layer, properties are expressed with boolean predicates about the value of the attributes of the process. The current prototype of VERBUS can verify several types of *safety* and *liveness* properties:

- *Invariants*: an invariant is a predicate that must be true in every reachable state. From the point of view of the BPEL4WS process, invariants look like *for every state if the activity named "init" is running, the part "urgent" of the global variable "order" must have a false value*. The counterpart property in the formal model layer is: `!(init_act_3.state=running & order)`.

- *Goals*: a goal is a predicate that must be true in every reachable final state. I.e. the predicate must be true whenever the process stops its execution. VERBUS adds automatically one goal to ensure that the process and all the activities are in a valid final state of their life–cycle (`not_started`, `completed`, `cancelled` or `compensated`) when the process reaches a final state. Thus any dead–lock or process block is detected. Goals like *when the process completes its execution the part "urgent" of the global variable "order" must have a false value* can detect functional errors in specifications.
- *Transition pre and post–conditions*: given a transition, a pre–condition (post–condition) is a predicate that must be true always immediately before (after) the execution of the transition. An example is: *immediately before the activity named "init" completes its execution the part "urgent" of the global variable "order" must have a true value.*
- *Activity reachability analysis*: VERBUS can detect transitions that can not be executed in any trace of the process. Thus activities that can never be started are detected, for example.
- *Properties defined with LTL*: VERBUS can check properties expressed in LTL (*Linear Temporal Logic*). Using LTL the modeller can specify temporal causalities like *if the part "urgent" of the global variable "order" has a true value, then sometime in the future it must have a false value.*

Formal layer specifications can be translated to Promela in a very straightforward way. The generated Promela specifications have a main `do` loop, in which all the transitions of the process are defined. The domain of each transition acts as a guard, and appears before the action. There is an `else` statement that breaks the loop when no transition can be selected (process completion). After the loop, there is an assertion for each goal property. Assertions for invariants are placed in a concurrent Promela process. Assertions for pre and post–conditions are placed before or after the action of each transition. In [11] this translation is explained in more detail.

6 Conclusions

This work presents VERBUS, a modular and extensible framework for automatic business process verification. It proposes an architecture with three layers: the design layer, the formal layer and the verification layer. The formal layer is a business process specification model based on the FSMs formalism. It disconnects process description languages and verification languages. Process definitions (design layer) can be automatically translated to specifications in the formal layer. These specifications can be automatically translated to specifications in the verification layer and verified using verification tools.

Works had been done previously on business processes verification, but they can not be applied directly to BPEL4WS compositions. They use specific process description languages and verification tools. On the contrary, VERBUS provides an open framework in which several process description languages and verification tools can be integrated.

The implementation of a prototype of VERBUS has demonstrated the feasibility of the framework. The prototype is mainly composed by two translation tools. The first one translates BPEL4WS specifications to the formal model. The second one translates formal model specifications to Promela specifications, that can be verified using Spin. The VERBUS formal layer can model the flow control primitives commonly used in business processes. It is even expressive enough to model the complex fault handling and cancellation mechanisms of BPEL4WS.

As future work, this first prototype will be completed by implementing the BPEL4WS compensation mechanism. Support for new process specification languages like BPML and verification tools like SMV will be added to VERBUS.

Acknowledgements

This work is partially supported by the Spanish Science and Technology Ministry, in the project TIC2003-07208 “Infoflex”.

References

1. Jae-yoon Jung, W.H., Kang, S.H.: Business Process Choreography for B2B Collaboration. IEEE Internet Computing **8** (2004) 37–45
2. Aissi, S., Malu, P., Srinivasan, K.: E-business process modeling: the next big step. IEEE Computer **35** (2002) 55–62
3. Andrews, T., Curbera, F., Dholakia, H., et al.: Business Process Execution Language for Web Services. Version 1.1 Specification. (2003) Available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel.
4. Arkin, A.: Business Process Modelling Language. Business Process Management Initiative. (2002)
5. ebXML Business Process Team: ebXML Business Process Specification Schema. Version 1.01. (2001) Available at http://www.ebxml.org/specs/ebBPSS.pdf.
6. Aalst, W.M.P.: Woflan: A petri-net-based workflow analyzer. Systems Analysis – Modelling – Simulation **35** (1999) 345–357
7. Eshuis, R.: Semantics and Verification of UML Activity Diagrams for Workflow Modelling. PhD thesis, University of Twente (2002)
8. Narayanan, S., McIlraith, S.: Simulation, Verification and Automated Composition of Web Services. In: Proceedings of the Eleventh International World Wide Web Conference, Budapest, Hungary (2002)
9. Holzmann, G.J.: The Spin model checker. Addison-Wesley (2003)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
11. Fisteus, J.A., Marin, A., Delgado, C.: VERBUS: A Formal Model for Business Process Verification. In: Proceedings of the 2004 IRMA International Conference, New Orleans, USA (2004)