Transforming BPEL into Intermediate Format Language For Web Services Composition Testing *

Mounir Lallali [1], Fatiha Zaidi [2,3], Ana Cavalli [1]
[1] TELECOM SudParis - CNRS SAMOVAR
9 rue Charles Fourrier, F-91011 Evry, France
Email: {Mounir.Lallali, Ana.Cavalli}@it-sudparis.eu
[2] Univ Paris-Sud, LRI, UMR 8623, Orsay F-91405;
[3] CNRS, Orsay, F-91405
Email: Fatiha.Zaidi@lri.fr

Abstract

BPEL is a standard language for Web services composition. To test a composite Web service, the design of a formal model is very useful, because it facilitates the application and the automatization of test generation methods. In this paper, we propose a transformation procedure of the BPEL specification into an Intermediate Format (IF) model that is based on timed automata. This IF format is well adapted to model BPEL (timed) constructs and to handle faults, events, termination, message correlation and activities synchronization. The proposed transformation was implemented in the BPEL2IF tool, which is also presented in this paper.

1. Introduction

Web services provide standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks [14]. Business Process Execution Language for Web Service (WS-BPEL) [10] is emerging as the standard composition language for specifying business process behavior based on Web services.

A BPEL business process implements a new composite Web service by specifying its interactions with existing Web services (called partners). It provides constructs to describe complex business processes that can interact synchronously or asynchronously with their partners. A basic process in BPEL is defined as one root element consisting of one or more child elements describing *partners*, a set of *variables*, *correlation sets*, *fault* and *compensation* handlers and activities. These latter define the interaction logic of a process and its partners. The BPEL activities that can be performed by a business process instance are categorized into basic (e.g., *receive*, *exit*) and structured activities (e.g., *sequence*, *flow*).

In our work, we focus on unit testing of Web services composition. The BPEL description of Web services composition is considered as the specification of what the system is expected to do. To facilitate the application and automatization of test generation algorithms, a formal model of BPEL is required. Inspired in our previous transformation of BPEL into Timed Extended Finite State Machine for Web Services (WS-TEFSM) [6] we propose a BPEL transformation into an Intermediate Format (IF) language. This language is based on communicating timed automata extended with variables and is associated to an efficient open-source simulator [13]. Using a TestGen-IF tool [2], which is based on this simulator, we can explore by an exhaustive simulation the state space of the model and generate test cases. In this paper, we present the IF timed automaton and we detail how to transform the BPEL process and its constructs into IF model. We define also how to handle faults, events, activities synchronization, termination and message correlation.

This paper is organized as follows. In Section 2, we discuss related work. Section 3 introduces the IF model which is used to model BPEL composition. The transformation of BPEL into IF model is detailed in Section 4. Section 5 describes the tool BPEL2IF that implements the BPEL transformation procedure. Finally, Section 6 concludes this paper.

2. Related-work

In the last years, several formal models of BPEL description and Web services composition have been proposed

*This Research is supported in part by the French National Agency of Research within the WebMov Project http://webmov.lri.fr

(e.g., process algebras, Petri nets and automata) [8, 12]. In our work, we do not consider process algebras nor Petri nets formalisms because we wanted to use our experience on formal methods (as in [3]) and tools in the area of testing. In addition, the IF timed automaton is powerful enough to specify the temporal behaviors of web services and are suitable to automate testing.

In [4], a BPEL process is transformed into PROMELA (the input language of SPIN) and used by the model checker SPIN to generate a test suite for BPEL specifications. The authors of [15] proposed a BPEL transformation into annotated deterministic finite state automaton for service discovery. All these works do not cover either compensation, fault, event nor termination handling.

In [16], the BPEL semantics were modeled by Web Service Automata (WSA) as an intermediate formal model and then the SPIN model checker and the NuSMV model checker were used to generate test cases. This WSA formalism does not allow to cover either message correlation handling nor capturing the timing aspects of some BPEL constructs (e.g., *wait*, *onAlarm*).

In [9], they propose another formalism that deals with data variables, the extended finite state automata, but no timing constraints are considered. In [5], a formalism taking into account timing constraints is proposed, the WSTTS. Nevertheless, this formalism uses only clocks but no data variables. Our work improves the state of art, because the obtained IF formal model includes all relevant aspects of BPEL constructs: time constraints, message correlation, fault and event handling, termination, fault propagation and activities synchronization.

3. IF Overview

A communicating system described using IF language [1] is composed of active processes instances running in parallel and interacting asynchronously through shared variables and signals via *signalroutes* or by direct addressing. A process instance can be created and destroyed dynamically during the system execution. It has local data and a private FIFO buffer. Each IF process is described as a timed automaton extended with discrete data variables, communication primitives and urgency attributes on transitions, i.e., IF-TA.

Definition 1 (IF Timed Automaton) *The IF Timed Automaton is a tuple* $TA = (Q, Act, X, T, q_0)$ *where:* Q *is a finite set of states,* q_0 *is the initial state,* Act *is a finite set of actions,* X *is a set of typed variables (including data variables and clocks) and* $T \subseteq Q \times G(X) \times 2^{Act} \times U \times Q$ *is a set of transitions such that* $G(X)$ *is a set of boolean guard conditions on data variables and clocks, and* $U = \{eager, lazy, delayable\}$ *is the urgency set.*

Each transition $t = q \xrightarrow{g\ a}_{u} q' \in T$ is annotated with a set of guards g, a set of actions $a \in Act$ and an urgency attribute $u \in U$. The actions in Act represent observable (i.e., signal input and output) or internal actions (e.g., assignment action, dynamic process creation and destruction). The clocks values are real numbers. They can be set and reset. Time progresses in states and transitions take zero time to be executed. The transition urgency [7] is used to control the time progress:

(i) *lazy transition*: is never urgent and never blocks time progress;

(ii) *delayable transition*: allows waiting as long as time progress does not disable it;

(iii) *eager transition*: is urgent as soon as it is enabled, and blocks time progress.

In the IF-TA semantics [7], we can distinguish *discrete* and *timed* transition. The *timed* transition indicates that the IF timed automaton does not execute any action (does not change state), but increments the current value of the clocks. A *timed* transition does not block any transition.

The *discrete* transition $q \xrightarrow{g\ a}_{u} q' \in T$ indicates that if the guard g is *true*, then the automaton follows the transition by executing the action a, changing the current values of the data variables by executing all the assignments, changing the current values of the clocks by executing all the time setting/resetting, updating the buffers content of the system by consuming the first signals required by input actions and by appending all signals required by output actions and, finally, moving in the next state q'.

To obtain an IF system, IF timed automata can be composed by using an associative and commutative parallel operator [7].

4. Transforming BPEL into IF

In this BPEL transformation procedure, we consider data, predicates, messages and partner links handling, basic and structured activities, fault propagation, termination and synchronization of activities, scopes, message correlations, event and fault handling, BPEL process element, WSDL interface of BPEL process clients and partners. The compensation handler transformation is not considered in this work. Each BPEL activity is described as an IF process. In particular, a non-basic activity is transformed into an IF process which can dynamically create its sub-activities processes. Note that each IF process can be the parent of other IF processes or a child of another one. The BPEL process is described by an IF system which its IF processes are executed in parallel and interact asynchronously through signals (IF messages used to communicate between IF processes) via *signalroutes* (communication buffers) [13].

In this section, we present the transformation of the main constructs and functionalities of BPEL which are summarized in Table 1. Note that in this section, we use the *Purchasing* example given in [10].

BPEL	IF Language
Message Type	Complex data type
Condition	Boolean variables & logical constraints
Partner Link Type	Enumeration = {*name,porttype,operation*}
Correlation Set	Complex type = {*name,status,properties*}
Message	Signal(PL type,message type)
Partner Link	Signalroutes
Fault handling	Propagation & handling of *fault* message
Termination Handling	Propagation and handling of *terminate* message & use of *exit* variable
Correlation Handling	Modification of the messaging construct processes by adding guard on correlation status & updating of properties values
Basic Activity	IF process
Structured Activity	IF process with sub-processes
Activities Synchronization	*LinksManager* process & exchanges with *source* and *target* activities
Fault Handlers	IF process with *catch* & *catchAll* sub-processes
Event Handlers	IF process with *onMessage* & *onAlarm* sub-processes
Scope & Process Element	IF process with *event handlers* & *fault handlers* sub-processes
Client & Partners	IF environment

Table 1. BPEL Transformation into IF

4.1. Data, Message and Partner Links

A BPEL variable can be declared as a WSDL message type, XML schema element or XML schema type [10]. These types (defined in the WSDL interfaces) are transformed in simple or complex IF types by using the IF type constructors (e.g., enumeration and range). In the IF simulator, all the possible input parameters are given during simulation. As we cannot control those values, in general it is common to face the state explosion problem. In order to reduce the problem size, we limit the values of some input parameters. Note that choosing those parameters should be done carefully by an expert who has better knowledge on Web services.

Because the BPEL partner links are bidirectional, each WSDL partner link type is transformed in one or more IF enumeration types which contains the partner link name, and its associated *portType* and *operation*. For example, the *Buyer* partner link type is described by two IF types: *toBuyer* = ***enum*** *Buyer.BuyerPT.PurchaseResponse* and *fromBuyer* = ***enum*** *Buyer.PurchasingPT.PurchaseRequest*.

Each WSDL message type is described as an IF *signal*. For instance, the message ***<message*** *name="POMessage">* *<part name="PO" type="PurchaseOrder" />* ***</message>*** is described as ***signal*** *POMessage(fromBuyer,POMessageType)* where *POMessageType* is a complex type with *PO* as member, and *fromBuyer* is a partner link type. Each BPEL partner link can be associated to one or more IF *signalroutes*. However, the BPEL partner links are bidirectional when IF *signalroutes* are unidirectional. For this reason, we associate for each partner link *PL* at least two *signalroutes*: *fromPL* and *toPL*. The first one is used to transport the input messages while the second one is used to transport the output messages. Let *POMessage* and *POResponse* be two messages. For instance, the partner link *Buyer* used in *receive* and *invoke* activities (of the *Purchasing* service) is associated to the two following *fromBuyer* and *toBuyer* *signalroutes* where *env* is the IF environment [13] and *intermediateEnv* is an intermediate environment process (see Section 4.10):

signalroute *fromBuyer()*	***signalroute*** *toBuyer()*
from *IntermediateEnv* ***to*** *ReceiveProc* ***with*** *POMessage;*	***from*** *invokeProc* ***to*** *env* ***with*** *POResponse;*

4.2. Fault Propagation and Termination

An IF process describing a BPEL activity forwards a *fault* message to its parent process when it receives a *fault* message from its sub-process (describing its sub-activity). The *fault* message is propagated until the fault handler of one of the enclosed scope handles this fault. We model this propagation by allowing each IF process (describing an enclosing activity) to receive a *fault* message in each state and to send it to its parent.

In BPEL, the termination is activated by an *exit* activity (first case) or when a fault is thrown by an *invoke* activity or a *throw* activity (second case). We use *terminate* and *done* messages to handle the termination according to the following cases:

1. When an *exit* activity is reached, the *exit* process assigns *true* to the *exit* variable of the IF process describing the BPEL process element. This latter initiates the termination of its children by propagating the *terminate* message. Before terminating an IF process, all of its children must be terminated. For this end, the parent process propagates the *terminate* message to its children and waits to their termination, i.e., receiving *done* message from all its children. If a terminated IF process has no children or all its children have terminated (normally or abnormally), it stops immediately its execution and sends a *done* message to its parent.
2. Each IF process finishes its control flow when it receives a *fault* message and sends this message to its parent. If one of the enclosing scopes can handle this fault, the fault propagation is stopped.

The *terminate* and *done* messages must have a higher priority than a *fault* message which has a higher priority than a normal incoming message (generated from BPEL). The *empty*, *throw* and *rethrow* activities may be allowed to complete, and the started *exit* activity must not be terminated [10]. We detail the termination of each activity below when we describe the IF process of each activity.

4.3. Synchronization of Activities

Flow activity provides concurrency and synchronization dependencies between its sub-activities [10]. This synchronization is expressed by a *link* construct. Each BPEL activity has optional nested standard elements *source* and *target* [10] that define a link which connects two activities and can change the sequential order of activities. An activity may declare itself to be the target (respectively the source) of one or more links by including one or more target (respectively source) elements. An activity can be the source of multiple links, thus allowing multiple branches to be executed in parallel. Each link can have an associated *transition condition* attribute. The source sets the link guard to the logic value of *transition condition* or to *true* if this attribute is not specified. The target activity may have a *join condition* attribute specified. It is executed only if its attribute is evaluated to *true*. If the *join condition* is not specified, it is interpreted as an *or* logical operator between the incoming links.

In order to not complicate the task of the *source*, *target* and *flow* activities, we used a specific IF process, called *linksManager*, to handle the links and the synchronization of the *flow* sub-activities. This *linksManager* uses *source* (respectively *target*) messages to communicate with source (respectively target) activities. Each target or source activity declares itself to *linksManager*. For each link, when a source activity finishes, it evaluates the guard and sends a *source* message to *linksManager* with the guard value. *LinksManager* sends a *target* message with this guard value to the target activity which must wait until the source activities finish. When a target activity receives a *target* message of all incoming links, it evaluates its guard (i.e., *join condition* or logical formula on incoming links). When this guard is not satisfied, a target activity propagates the *join failure* fault to its parent.

4.4. Basic Activities

Basic activities [10] describe elemental steps of the BPEL process behavior. Basic activities are: *invoke*, *receive*, *reply*, *assign*, *wait*, *empty*, *exit*, *throw* and *rethrow*. Each basic activity is described by a simple IF process that executes one step and sometimes handles its forced termination. For instance, the IF process of the *invoke*, *receive* or *reply* activity uses the IF communicating actions (i.e., *input* and/or *output*). We note that all the input actions of this messaging process are not urgent and never block time progress. They have a *lazy* urgency (see Section 3). The IF process for the *receive* activity ***<receive*** *partnerLink="Buyer" portType="PurchasingPT" operation="PurchaseRequest" variable="PO"* /> is illustrated in Figure 1.

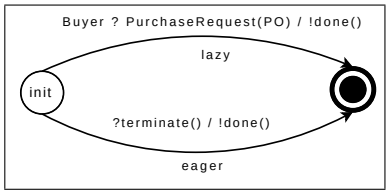

Figure 1. The Receive Process

4.5. Structured Activities

Structured activities describe the order in which its sub-activities are executed [10]. *Sequence*, *if*, *while*, *repeat until* and serial *for each* activities provide a sequential control. *Flow* activity provides concurrency and synchronization between activities. *Pick* activity provides a choice controlled by events. We have defined the transformation of all the structured activities as indicated in Table 1.

4.5.1. Sequence

A *sequence* activity is described as an IF process that performs sequentially its sub-activities [10]. This *sequence* process creates, in the appearing order, a sub-activity process and waits for its termination (a reception of *done* message) before creating the next sub-activity process. The *sequence* process terminates normally when the last sub-activity process terminates. It is interrupted when it receives a *terminate* or a *fault* message. In this case, it terminates its behavior and applies a forced termination to its active sub-activity process.

4.5.2. If

The *if* activity consists of an ordered list of conditional branches [10]. If no branch is taken, an optional *else* branch is taken, if present. The *if* activity is described as an IF process that selects, in the appearing order, one of the *if* branches. The *if* process creates a selected branch sub-activity process and waits for its termination, corresponding to the branch process termination. When the *if* activity process is interrupted by receiving a *terminate* or *fault* message, it terminates immediately when no branch is selected, or the forced termination is applied to the IF process of the selected branch sub-activity.

4.5.3. While and Repeat Until

The *while* and *repeat until* activities provide repeated execution of its sub-activities [10]. They are described as an IF process. The *while* process creates a sub-activity process as long as the *while* condition evaluates to *true* in each iteration. The *repeat until* process creates a sub-activity process

until the *repeat* condition becomes *true*. The *while* and *repeat until* processes wait for the sub-activity termination before each iteration. They terminate when their condition is evaluated to *false*. When they are interrupted, by receiving a *terminate* or *fault* message, their iteration is interrupted and the forced termination is applied to their sub-activity process.

4.5.4. Flow

Flow activity allows to specify one or more activities to be performed concurrently [10]. The links defined in the *flow* activity permit to enforce precedence between sub-activities, i.e., synchronization. The *flow* activity is described as an IF process that creates simultaneously the IF processes of all the enclosed sub-activities. The synchronization of the *flow* sub-activities is handled by the *linksManager* process described in section 4.3. The *flow* process completes when all sub-activities are completed. It is interrupted when it receives a *terminate* or a *fault* message. In this case, it terminates its behavior and applies forced termination to its active sub-activities processes.

4.5.5. Pick

The *pick* activity waits for one event occurrence, then executes the associated activity [10]. It is described as an IF process that waits the occurrence of one event, creates the IF process of the associated activity of each event and waits for its termination. The event of the *pick* activity has two forms: *OnMessage* considered as an IF input action and *onAlarm* considered as an IF waiting action. The *pick* process termination is similar to the *if* process termination. All the input actions (modeling the *onMessage* elements) of the *pick* process are not urgent and do not block time progress. These input actions have a *lazy* urgency. In the contrary, the time-out action (modeling the *onAlarm*) is always urgent and it has a *eager* urgency.

4.6. Scope

A *scope* provides the context that influences the execution of its enclosed activities [10]. The *scope* context includes variables, partner links, correlation sets, event handlers, fault handlers, a compensation handler and a termination handler. Each *scope* has a primary activity which defines the normal behavior of the scope. It is described as an IF process which creates two sub-processes, respectively, one of its primary activity and another of its event handlers. The *scope* process can have defined variables considered as private variables. When the *scope* process is interrupted, by receiving a *terminate* or *fault* message, it applies a forced termination to the IF processes of its primary activity and event handlers by propagating a *terminate* message, and waits for their termination. Finally, in the case of *fault* message reception, the *scope* process creates the IF process of its fault handlers to handle the fault and waits for its termination. If the scope fault handler do not handle the occurred fault, the *scope* process propagates (to its parent) the fault message. The transformation of correlation sets, fault and event handlers are presented in the next sections.

4.7. Message Correlation

A correlation set is a set of properties shared by all messages in the correlated group [10]. Correlation can be declared in a BPEL *process* or a *scope*. Correlation set names are used in the BPEL messaging constructs (e.g, *invoke*, *receive* and *onMessage*). We limit each messaging activity to one correlation set (except the synchronous *invoke* which is limited to two correlation sets). We handle the message correlation as follows:

- The IF process of a BPEL process or *scope* is extended with a complex type declaration of the correlation set construct: {name; status; properties}. The *Status* variable indicates if the correlation set is being initiated;
- The IF process of each messaging activity that carries the correlation set *cs_name* is extended by two variable declarations: *initiate* and *cs_name*;
- When the *initiate* variable is set to *yes*, the IF process of the messaging activity initiates the correlation set (by setting the *status* variable to *true*) and defines the correlation set properties according to the correlation values of the exchanged message;
- If a correlation set *cs_name* is already initialized, i.e., *status* variable is set to *no*, then the correlation values of the IF process of each messaging activity that carries the correlation set *cs_name* must be identical to the values of the properties of this correlation set;
- A *fault* (i.e., correlation violation) message is propagated (by the IF process of the messaging activity) when a correlation set is already initiated (respectively has not been initiated) and the *initiate* attribute is set to *yes* (respectively to *no*). This fault message is also propagated if the values of the correlation are different from the values of the correlation set in the message;
- The *propertyAlias* elements (that permit to retrieve correlation values from a message [10]) are transformed into IF procedures which have a property name and a WSDL message type as parameters, and return a WDSL message part.

Figure 2 illustrates the IF process for the following *receive* activity with correlation: *<**receive** partnerLink="Buyer" portType="PurchasingPT" operation= "PurchaseRequest" variable="PO"> <**correlations**> <correlation set ="PurchaseOrder" initiate="yes"/> </**correlations**> </**receive**>*.

This activity initiates the correlation set *PurchaseOrder*. In Figure 2, the *transition 1* checks if the correlation is already initiated (*status = false*). In this case, the *receive* process sets the *status* variable to *true*, defines the properties of the *PurchaseOrder* correlation and sends to the parent process a *done* message. Else (*status = true*), the *receive* process propagates (by the *transition 2*) the correlation violation fault. If the *receive* process receives a *terminate* message, the *transition 3* and the *transition 4* terminate immediately this process and send a *done* message to its parent.

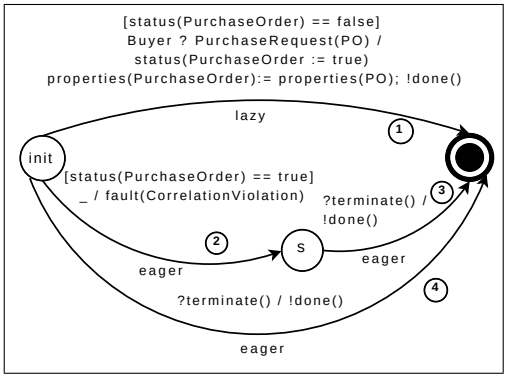

Figure 2. The Receive Process with Correlation Handling

4.8. Fault and Event Handlers

The *fault handler* of a *scope* or a *BPEL process* is a set of catch clauses defining how the *scope* should respond to different types of faults [10]. The *fault handlers* are described as an IF process which combines an *if* activity applied to various sequences of *catch* or a *catchAll* activities (conditional branches) and the creation of its IF sub-activities processes. Each *catch* branch is considered as a comparison between the propagated fault and its handled fault. The *catchAll* branch is used to catch all the faults that are not handled by the defined *catch* branch.

The BPEL *process* element and each *scope* can be associated with a set of event handlers that are invoked concurrently when the corresponding event occurs [10]. There are two event types: input message (*onEvent* element) and alarm (*onAlarm* element). Event handlers are described, in the same way as a *pick* activity, as an IF process where each *onEvent* is considered as an IF input action and *onAlarm* considered as an IF waiting action. The event handlers process termination is similar to the *if* process termination.

4.9. BPEL Process Element

A BPEL process always starts with the *process* element, i.e., the root of the BPEL document. It is composed of the following optional children: *partner links*, *variables*, *correlation sets*, *fault handlers*, *compensation handlers* and *event handlers*. Note that *compensation handlers* are not considered in this work.

The *process* element contains one main activity declaration representing the process workflow definition. This BPEL *process* element is described as an IF process (schematized in Figure 3) which creates two sub-processes, respectively, of its primary activity and its event handlers. The transformation of its optional children is detailed in the previous sections.

When the IF process of the BPEL *process* element receives a *fault* (the case of the *transition 3′*) or when its *exit* variable is assigned to *true* by the *exit* process (the case of the *transition 3″*), it applies a forced termination to the IF processes of its primary activity and event handlers by propagating a *terminate* message and waits for their termination. Afterwards, in the case of *fault* message reception (the case of the *transition 4*), the IF process of the BPEL *process* element creates the IF process of its fault handlers to handle the fault and waits for its termination.

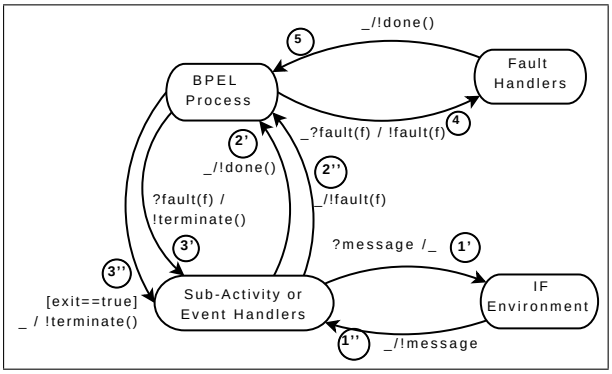

Figure 3. The BPEL Process Machine

4.10. BPEL Client and Partners

The client and the partners of a BPEL process are considered as the environment of the IF system. In IF, the communication between two IF processes and the communication between an IF process and the environment are handled in a different way. Each IF process has its own FIFO queue and the messages from other processes are stored in this queue. As the communication is asynchronous it may take time to consume the messages in the FIFO queue. The messages

from the environment are, however, consumed as soon as the environment sends a message, i.e., the communication between the IF environment and IF process is synchronous.

In the proposed transformation, the IF process of BPEL messaging constructs can receive the messages (e.g., *done* signal) from the environment as well as from other processes. In this case the order of consumption of the messages cannot be guaranteed to be the same order of their reception. The messages in the queue, which is already received from other IF processes, may need to wait for the processing of newly incoming messages from the environment.

In order to solve this problem, we introduce an intermediate environment process, called *IntermediateEnv*. Every message from the environment is sent to this *IntermediateEnv* process and then it passes each message to the appropriate destination. As each signal is defined to have only one destination in the proposed model, the Intermediate process can distribute the incoming messages from the environment to the proper destinations. By introducing this intermediate process, we can guarantee the order of consumption of messages as all the messages coming from the environment are passed to the FIFO queue of the IF process.

5. BPEL2IF Tool

We have developed BPEL2IF which is used to transform BPEL (BPELWS 1.1 and WS-BPEL 2.0) into IF. It implements our BPEL transformation procedure defined in section 4. The BPEL document is processed as a tree where each node corresponds to a BPEL construct. A depth-first walk of the BPEL tree is performed and associated XSL transformation rules are applied to each node in order to produce an IF specification document.

This tool was used to transform the Loan Web service given in [11] which contains *receive*, *invoke* and *pick* activities. The obtained IF specification was obtained by the transformation of the BPEL specification of the Loan service and the WSDL description of its partner (i.e., the Credit Rating service). This IF specification contains the declaration of five signals describing the BPEL messages, three internal signals (i.e., *done*, *fault* and *terminate*), and four *signalroutes* associated to the BPEL partner links. The User and the Credit Rating services are described by the IF environment, i.e., *env*.

6. Conclusion

This paper proposes a transformation procedure of BPEL specification into Intermediate Format (IF) model. This later can model the BPEL constructs and can handle faults, events, activities synchronization, message correlation, fault propagation and termination of the BPEL process and its sub-activities. The proposed procedure has been implemented in the BPEL2IF tool. We actually use this transformation procedure to test the BPEL specification of Web services composition and to generate test cases from IF specifications.

Our future work is to handle compensation and to investigate the extension of the proposed transformation procedure to the choreography of Web services.

References

[1] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF toolset. In *SFM-04*, volume 3185 of *LNCS*, pages 237–267. Springer-Verlag, june 2004.

[2] A. Cavalli, E. Montes De Oca, W. Mallouli, and M. Lallali. Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints. In *Proceedings of DS-RT'08*, Vancouver, Canada, October 2008.

[3] M. Dumas and R. Heckel, editors. *Web Services and Formal Methods, WS-FM'07 Workshop, Brisbane, Australia, Sep., 2007. Proceedings*, volume 4937 of *LNCS*. Springer, 2008.

[4] J. G. Fanjul, J. Tuya, and C. de la Riva. Generating Test Cases Specifications for Compositions of Web Services using SPIN. In *Proceedings of WS-MaTe'06 Workshop*, pages 83–94, 2006.

[5] R. Kazhamiakin, P. Pandya, and M. Pistore. Timed modelling and analysis in web service compositions. *ares*, Volume 0:840–846, 2006.

[6] M. Lallali, F. Zaidi, and A. Cavalli. Timed Modeling of Web Services Composition for Automatic Testing. In *Proceedings of SITIS'07*, Shangai, China, December 2007.

[7] M. Bozga and J.-C. Fernandez and L. Ghirvu and S. Graf and J.P. Krimm and L. Mounier and J. Sifakis. IF: An Intermediate Representation for SDL and its applications. In *SDL Forum*, pages 423–440, 1999.

[8] Maurice, A. Bucciaroneand, and S. Gnesi. A Survey on Service Composition Approaches: From Industrial Standards to Formal Methods. Technical Report 2006-TR-15, Consiglio Nazionale delle Ricerche, 2006.

[9] S. NAKAJIMA. Lightweight formal analysis of web service flows. *Progress in Informatics*, Volume 2:57–76, 2005.

[10] OASIS Standard. WSBPEL Ver. 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[11] ORACLE. Service Oriented Architecture. http://www.oracle.com/technologies/soa/index.html.

[12] F. van Breugel1 and M. Koshkina. Models and Verification of BPEL. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf.

[13] Verimag/IMAG. IF Toolset. www-if.imag.fr/.

[14] W3C. http://www.w3.org/.

[15] A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming BPEL into Annotated Deterministic Finite State Automata for Service Discovery. In *Proceedings of ICWS'04*, page 316, 2004.

[16] Y. Zheng, J. Zhou, and P. Krause. An Automatic Test Case Generation Framework for Web Services. *Journal of Software*, 2(3):64–77, September 2007.