A Methodology and a Tool for Model-based Verification and Simulation of Web Services Compositions

AL-GAHTANI Ali, AL-MUHAISEN Badr and DEKDOUK Abdelkader

Department of Information and Computer Science
King Fahd University of Petroleum and Minerals,
dekdouk@ccse.kfupm.edu.sa

Abstract. In this paper we present a methodology and a tool developed in .NET framework, to design and verify business processes defined as compositions of distributed web services. The tool takes a business specification described in BPEL4WS, performs a syntactic validation and schema validation using XML validator, and then translates it into a PROMELA specification. After that we carry out model-based verifications and simulations on the corresponding PROMELA description using SPIN tool.

1) INTRODUCTION

Web services have been introduced to facilitate universal interoperability between applications by exploiting web standards. Essentially, they are functionalities exchanged in a variety of domains including business-to-business, business-to-customer, and enterprise applications [1]. In a nutshell, web services are XML (EXtensible Markup Language)-based messages that uses SOAP (Simple Object Access Protocol) in order to carry out invocations or results between systems regardless of their platforms. For composing web services, different languages were defined such as XLANG by Microsoft and WSFL (Web Service Flow Language) by IBM. Recently, a language called Business Process Execution Language for Web Services (BPEL4WS) [1] has been defined as a standard for specifying and composing web services. It was released by BEA, IBM, Microsoft and others to replace existing web service composition languages.

The purpose of this paper is to address a methodology and a prototype of an integrated development environment (developed in .NET framework) to design and verify composition of distributed web services described in BPEL4WS. The BPEL4WS specification is first validated with an XML validator using particularly Internet Explorer-6 to check the well-formedness of the BPEL4WS specification. It is also validated on the Business Process Schema defined by Word Wide Web Consortium (W3C) and then translated semantically to a PROMELA specification [2]. This semantically equivalent PROMELA description is then fed to XSpin model checker to carry out, on it, model-based verification of some properties such as safety and liveness properties. In addition to this, our system also benefits from the integration of XSpin and its powerful verification by illustrating some simulations of BPEL4WS models.

In this paper, we present an integrated development environment for verifying and simulating business processes. It is organized as follows: Section 2 explores the high-level architecture of our IDE by visiting each component and specifying its design facets. Section 3, discusses the design specification which shows the scope of this work. Within the design specification, we provide an overview of BPEL4WS activities that will be considered in this environment. Moreover, the modules of the IDE are expressed in terms of their features and structure. In Section 4, we provide a case study for one BPEL4WS example and explain the way they are translated to PROMELA to help in clearing the picture of the translation methodology. Then, we present a snapshot of our IDE along with the simulation of a BPEL4WS specification and the properties verified on this specification. Finally we conclude by some remarks and future works.

2) High-Level Architecture

As shown by Figure 1, in UML [7] notation, our IDE is composed of three modules. First module is the XML-Validator which uses Internet Explorer-6 for syntactically validating BPEL4WS specification and Schema validation. Also, the module parses BPEL4WS specification and puts the activities (main constructs of BPEL4WS) into an appropriate data structure namely a Document Object Model (DOM). Second module is the translator which takes each entry from the created data structure and translates it into PROMELA specification accordingly. The translator module is responsible for generating a syntactically and semantically correct mapping of BPEL4WS to PROMELA in order for the model checker to simulate and verify the specification. The third module is the XSpin model checker. This module is used to verify the model PROMELA on certain Linear-time Temporal Logic formulae (LTL) properties. Since BPEL4WS is a composition of web services, the specification may exhibit anomalies such as loss of exchanged messages, non-respect of message orders or deadlocks [2]. Additionally, this module is used to simulate the behavior of the specified model. Simulation is one of the formidable features of XSpin which allows visually observing the behavior of interacting processes.

3) Design Specification

3.1. The language BPEL4WS

Our goal is to devise a mechanism for the verification of BPEL4WS specification that insures the correctness of the specification and the faultless interaction between composed web services [5]. BPEL4WS has a rich set of constructs and extensions in order to provide developers with a large number of features for web services composition. For the sake of optimization and efficiency, we are restricting BPEL4WS [1] to a sub-language including *simple* and *structured* activities which are the major

Top

players in web services interaction and have the most influential effect on the specification behavior.

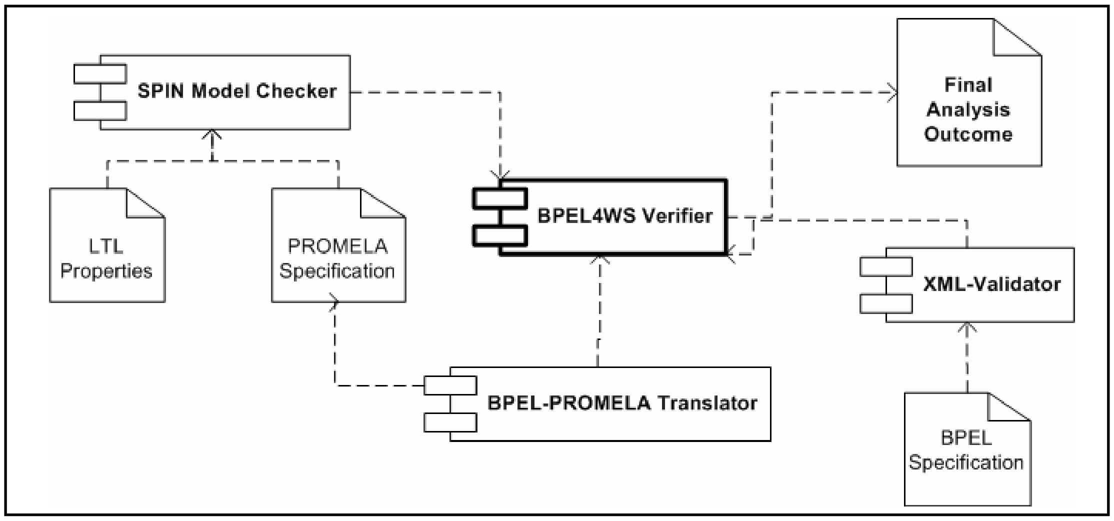


Figure 1: UML component diagram of the IDE

3.1.1. *Simple Activities*

Simple activities include the following activities as shown in Figure 2:

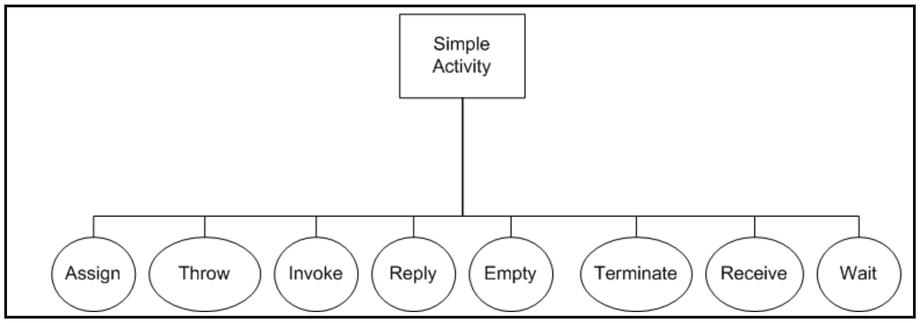


Figure 2: Simple Activities

- The **<receive>…</receive>** activity enables the business process to perform a blocking wait until the reception of the message.
- The **<reply>…</reply>** activity enables the process to send a message in reply to a received message through the **<receive>…</receive>** construct. This activity is helpful only in synchronous interactions. This means that only one request and its corresponding reply can be outstanding at a time.
- The **<invoke>…</invoke>** activity enables the process to invoke a synchronous request-response or asynchronous one-way operation. Note that in the synchronous invocation we specify the input and the output variables. In the asynchronous invocation we specify the input variable only.

Top

- The **<assign>…</assign>** activity is used to handle data assignments within variables.
- The **<throw>…</throw>** activity is used to throw the exceptions generated within the business process.
- The **<wait>…</wait>** activity enables the process to wait for a certain period of time.
- The **<empty>…</empty>** activity inserts "no-op" statement into a business process which helps in synchronization of concurrent activities, which permits to wait for unspecified period of time.
- The **<terminate>…</terminate>** activity terminates the current running processes without exception handling [1].

3.1.2. *Structured Activities*

Structured activities include the following activities as shown in Figure 3:

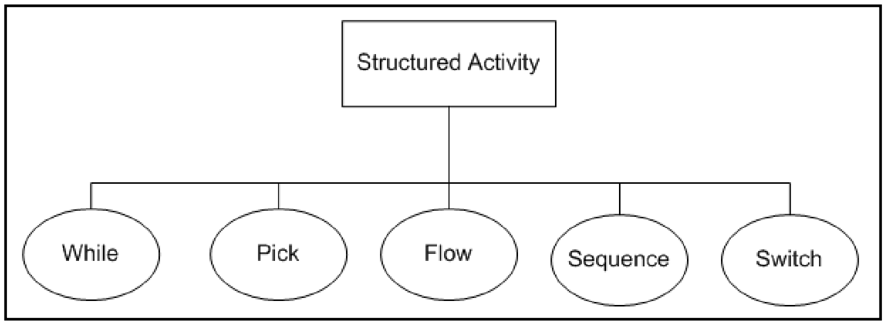


Figure 3: Structured Activities

- The **<while>… </while>** activity enables repeating an activity as long as a condition is satisfied.
- The **<pick>… </pick>** activity enables triggering an activity whenever an event occurs.
- The **<flow>… </flow>** activity is used to concurrently perform more than one activity.
- The **<sequence>… </sequence>** activity allows executing activities sequentially.
- The **<switch>… </switch>** activity enables branching one activity from a set of choices [1].

3.2. Main Components

In this part of the paper, we describe the design features of each module and its functionality provided to the system.

Top

3.2.1. *The XML-Validator Module*

This module reads the BPEL4WS specification that has either been entered interactively or uploaded. After that, the process of parsing starts by reading each BPEL4WS tag and uploads it to a DOM Xml Document. Since BPEL4WS is an XML based language, the DOM representation is considered to be the optimal data structure for processing BPEL4WS.

Microsoft .NET Class Library provides the essential classes for parsing and processing XML documents. In the development of our prototype we mainly use *System.Xml* namespace and its classes. As indicated in Figure 4, the major target classes that we are using are *XmlReader* [6] and its derived classes: *XmlTextReader* and *XmlValidatingReader*, *XmlNode, XmlNodeList,* and *XmlDocument*.

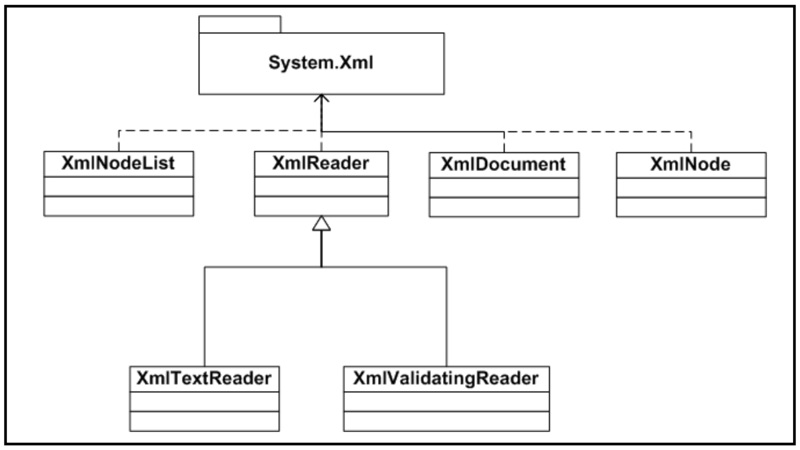


Figure 4: Class Diagram of System.Xml Namespace with XmlReader Class and its Derived Classes

XmlReader represents a reader that provides fast, non-cached, forward-only access to XML data. The current node refers to the node on which the reader is positioned. The reader is advanced in using any of the read methods and properties that reflect the value of the current node. *XmlTextReader* is the fastest implementation of XmlReader. It checks for well-formed XML, but does not support data validation. This reader cannot expand general entities and does not support default attributes *XmlValidatingReader* is an implementation of *XmlReader* that can validate data using DTDs or Schemas. This reader can also expand general entities and supports default attributes. *XmlNodeList* is used to represent an ordered collection of nodes. *XmlNode* is used to represent a single node in the XML document. Finally, we make use of *XmlDocument* class to implement the W3C Document Object Model (DOM) Level 1 Core and the Core DOM Level 2. The DOM is an in-memory (cache) tree representation of an XML document and enables the navigation and editing of this document [6].

3.2.2 *The BPEL4WS to PROMELA Translator Module*

This module is responsible for traversing the tree produced by the XML-Validator module and converting each node to its corresponding PROMELA code fragment. In this

Top

module, we are using a *translator* class to help the translator in matching the tree node that contains a BPEL4WS activity to the proper PROMELA specification. This is done by traversing several cases to find the corresponding PROMELA specification for that node. Deciding whether the activity is simple or structured is done while translating. In the sequel, we present an example where we show our followed approach to tackle the BPEL4WS-PROMELA translation.

Suppose that we have an online trade settling facility through which customers (buyers) and sellers can have their deal settled. The required web services for doing so are obtaining the buyer information, obtaining the seller information, settling the trade, confirming the trade back to the buyer and the seller. BPEL4WS is used to compose these web services into one business process in an expressive behavior of when each web service is processed and how web services interact with each other.

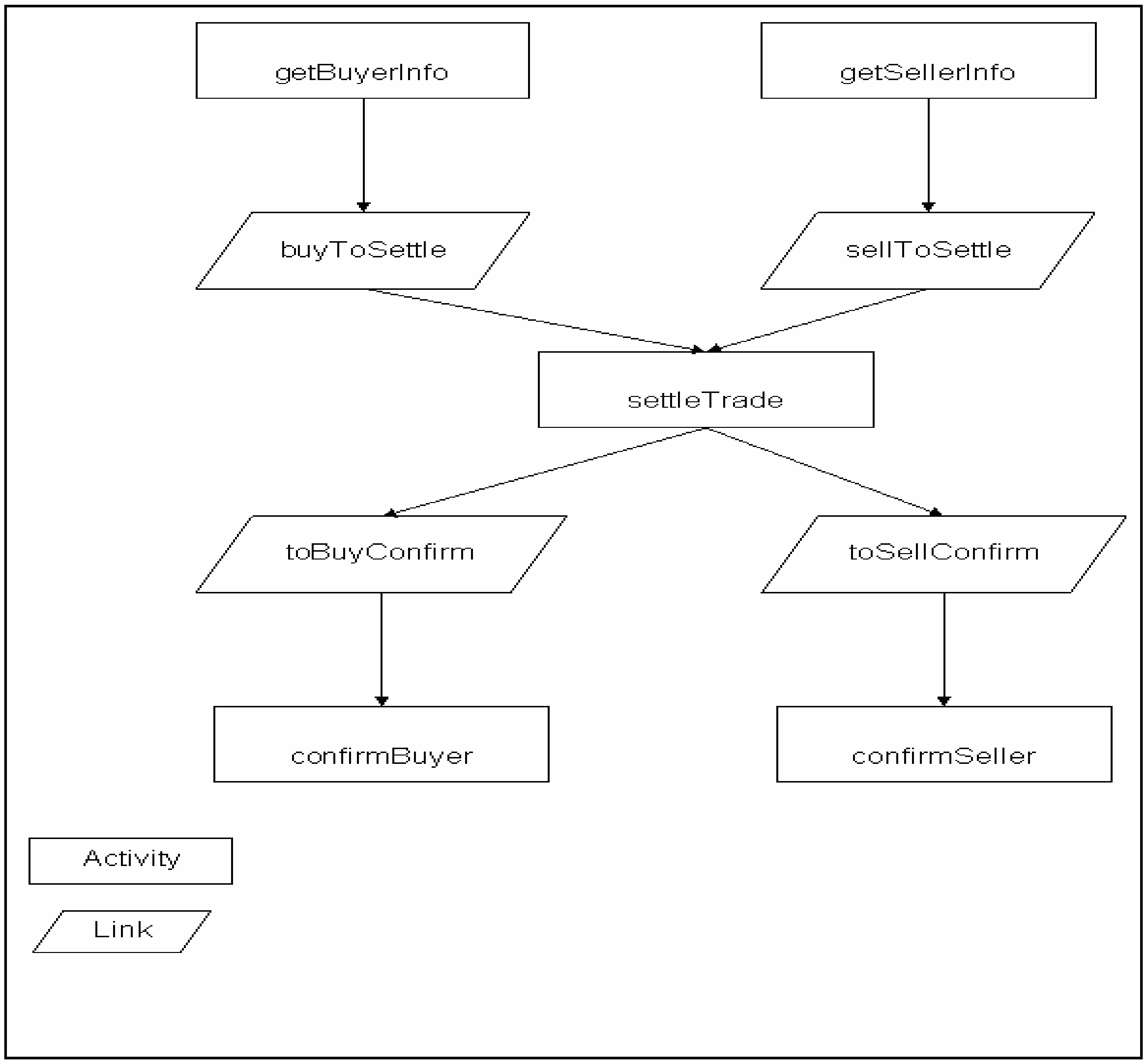


Figure 5: Flow Diagram of the Trade Settling business process

As shown in Figure 5, the activities with the names getBuyerInfo, getSellerInfo**,** settleTrade, confirmBuyer, and confirmSeller are nodes of a graph defined within the

Top

<flow>…</flow> activity. The following *links* (links are used to maintain control dependencies between concurrent activities) are defined as follows:

- The link named buyToSettle starts at getBuyerInfo (specified through the corresponding *source* element nested in getBuyerInfo) and ends at settleTrade (specified through the corresponding *target* element nested in settleTrade).

- The link named sellToSettle starts at getSellerInfo and ends at settleTrade.

- The link named toBuyConfirm starts at settleTrade and ends at confirmBuyer.

- The link named toSellConfirm starts at settleTrade and ends at confirmSeller.

Based on the graph structure defined by the flow, the activities getBuyerInfo and getSellerInfo can run concurrently. The settleTrade activity is not performed before both of these activities are completed. After settleTrade completes the two activities, confirmBuyer and confirmSeller are performed concurrently again.

Our approach in translating BPEL4WS to PROMELA is summarized in the following Table 1. The table contains the main constructs on both languages which emphasizes the translation scheme that we follow in our translation. Note that T(P), T(P1) and T(P2) are the PROMELA transforms of BPEL4WS processes P, P1 and P2 respectively.

BPEL4WS	PROMELA
<**link name** = *queue*>….</**link**>	**chan** queue
<switch>	**if**
Case1..	:: (condition 1)
.	:: ….
.	:: ….
Case *n*..	:: (condition *n*)
</**switch**>	**fi**
<while condition = "cond1" = "true">	**do**
P	T(P)
</while>	(!cond1) → break
	od
<**sequence**>	chan sync[0] of {bit}
P1	bit x;
P2	T(P1) **synchronizes** with
</**sequence**>	T(P2) on the channel sync.
<flow>	**init{**
P1	**run**(T(P1));
P2	**run**(T(P2));
</flow>	}
A simple activity, for example	**Proctype receive**( ) { … }
<receive>…**<receive>**	

Table 1: Translation Approach.

The following table (Table 2) illustrates the PROMELA specification that is mapped from the BPEL4WS example above. *Links* in BPEL4WS are translated to *channels (chan)* in PROMELA. Channels required for activities to communicate, are

Top

declared globally in PROMELA. An mtype (constant) enumeration of messages to be exchanged by process is defined globally. Each *activity* in BPEL4WS is mapped to a *process (proctype)*. The *invoke* activity has a *join condition* that the two channels buyToSettle and sellToSettle must contain data in order to store their contents in two other channels toBuyConfirm and toSellConfirm respectively which in turn are received in both replyConfirmSeller and replyConfirmBuyer respectively. This *join condition* is taken care of the moment processes are instantiated within the *initial (init)* process as shown below. Finally, the five processes corresponding to each activity in the BPEL4WS specification are instantiated within *init* block [2]. The *flow* activity (which implies performing activities inside it concurrently) is mapped to the *init* process which exactly models the behavior of *flow* by instantiating concurrently processes contained within its block.

BPEL4WS spec. of the example	Corresponding **PROMELA** spec.
<?xml version="1.0" encoding="utf-8" ?>	chan buyToSettle ;
<process xmlns="**http://schemas.xmlsoap.org/ws/2002/07/business-**	chan sellToSettle ;
process/">	chan toBuyConfirm ;
	chan toSellConfirm ;
<flow suppressJoinFailure="**yes**">	mtype {msg1, msg2, msg3, msg4, msg5, msg6};
<links>	proctype receiveBuyer( ) {
<link name="**buyToSettle**" />	
<link name="**sellToSettle**" />	buyToSettle ! msg1
<link name="**toBuyConfirm**" />	}
<link name="**toSellConfirm**" />	
</links>	proctype receiveSeller( ) {
<receive name="**getBuyerInfo**">	sellToSettle ! msg2
<source linkName="**buyToSettle**" />	
</receive>	}
<receive name="**getSellerInfo**">	proctype invokeSettleTrade( ) {
<source linkName="**sellToSettle**" />	
</receive>	buyToSettle ? msg3;
	sellToSettle ? msg4;
<invoke name="**settleTrade**"	toBuyConfirm ! msg3 ;
joinCondition="**bpws:getLinkStatus('buyToSettle') and**	toSellConfrim ! msg4
bpws:getLinkStatus('sellToSettle')">	
<target linkName="**getBuyerInfo**" />	}
<target linkName="**getSellerInfo**" />	
<source linkName="**toBuyConfirm**" />	proctype replyConfirmBuyer( ) {
<source linkName="**toSellConfirm**" />	
</invoke>	toBuyConfirm ? msg5
	}
<reply name="**confirmBuyer**">	
<target linkName="**toBuyConfirm**" />	proctype replyConfirmSeller ( ) {
</reply>	
	toSellConfrim ? msg6
<reply name="**confirmSeller**">	}
<target linkName="**toSellConfirm**" />	
</reply>	init {
	run receiveBuyer( ) ;
</flow>	run receiveSeller( ) ;
</ process>	run invokeSettleTrade( );
	run replyConfirmBuyer( ) ;
	run replyConfirmSeller( )
	}

Table 2: BPEL4WS to PROMELA Mapping.

Top

3.2.3. *The Verification & Simulation Module (XSpin Model Checker)*

This module integrates the tool XSpin into our environment. XSpin is a generic verification tool that supports the design and verification of asynchronous process systems described in PROMELA formalism [2]. When XSpin is invoked from the main IDE menu, it checks for syntax errors on PROMELA specification. If there are no syntax errors, the specification is verified on the required correctness LTL properties.

Another feature of XSpin is the visual simulation of the interactions and states of processes within the model. The simulation functionality in XSpin provides a Message Sequence Chart (MSC) that facilitates observing the behavior of processes and the messages exchanged between them. In the following case study, we explore by providing an example of a loan process [4] the roadmap of using our IDE for verifying BPEL4WS specification.

4) Case Study

This is an example of a loan approval business process combined of several web services. First, customer information is received and his/her account is examined. If the amount of money in the customer's account is less than 10,000 then this loan request is transferred to a loan assessment web service where the risk is assessed. Otherwise the loan request is approved and replied back to the customer. For the assessment, if the risk is low the request is approved. Otherwise the loan request is not accepted. This example is illustrated in the flow chart below (Figure 6):

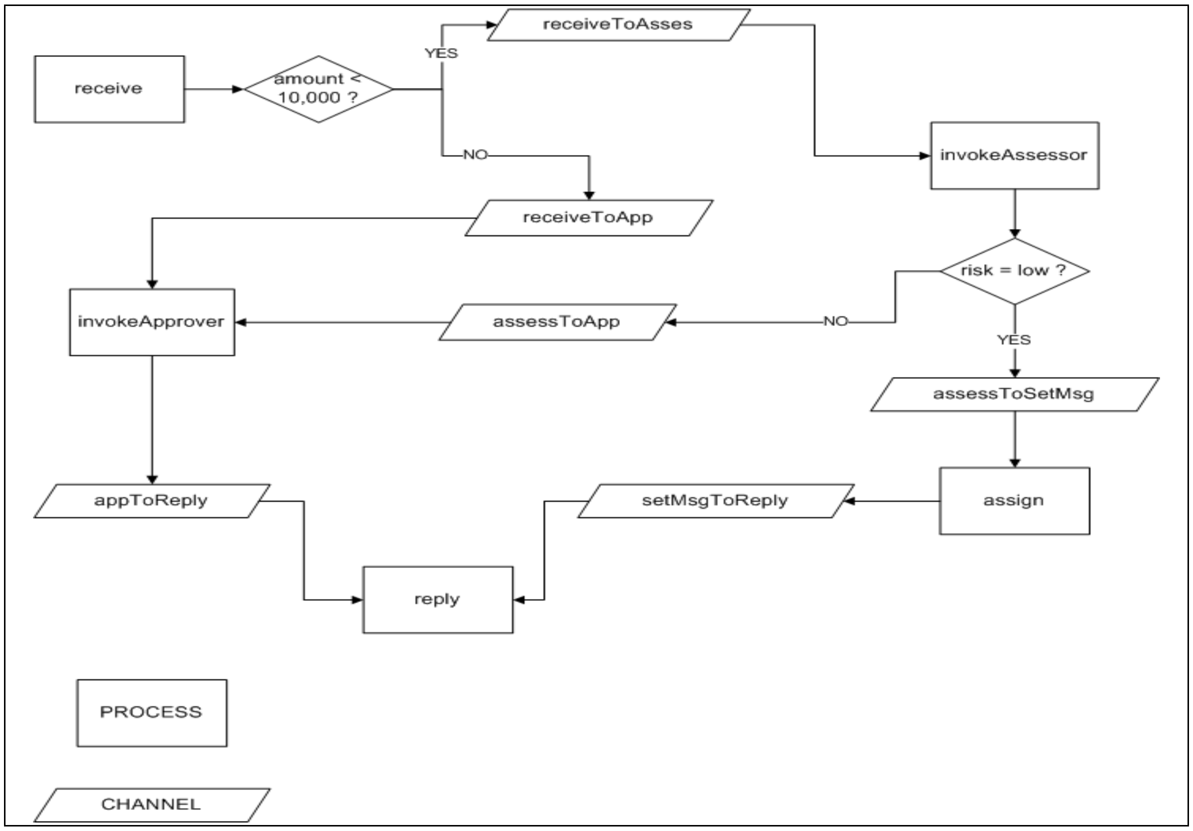


Figure 6: Flow Chart Representation of the Loan Approval Process

The following is the BPEL4WS code of this example:

```
<?xml version="1.0" encoding="utf-8" ?>
 <process name="loanApprovalProcess"
   targetNamespace="http://acme.com/loanprocessing"
   suppressJoinFailure="yes"
   xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
   xmlns:lns="http://loans.org/wsdl/loan-approval"
   xmlns:loandef="http://tempuri.org/services/loandefinitions"
   xmlns:asns="http://tempuri.org/services/loanassessor"
   xmlns:apns="http://tempuri.org/services/loanapprover">

<containers>
  <container name="request"
     messageType="loandef:creditInformationMessage"/>
  <container name="riskAssessment"
   messageType="asns:riskAssessmentMessage" />
  <container name="approvalInfo" messageType="apns:approvalMessage" />
   <container name="error" messageType="loandef:loanRequestErrorMessage"
   />
</containers>

<partners>
  <partner name="customer" serviceLinkType="lns:loanApprovalLinkType"
   myRole="approver" />
  <partner name="approver" serviceLinkType="lns:loanApprovalLinkType"
   partnerRole="approver" />
  <partner name="assessor" serviceLinkType="lns:riskAssessmentLinkType"
   partnerRole="assessor" />
</partners>

<faultHandlers>
   <catch faultName="lns:loanProcessFault" faultContainer="error">
  <reply partner="customer" portType="apns:loanApprovalPT"
   operation="approve" container="error" faultName="invalidRequest" />
  </catch>
</faultHandlers>

<flow>
<links>
     <link name="receive-to-assess" />
     <link name="receive-to-approval" />
     <link name="approval-to-reply" />
     <link name="assess-to-setMessage" />
     <link name="setMessage-to-reply" />
     <link name="assess-to-approval" />
</links>

<receive name="receive1" partner="customer"
   portType="apns:loanApprovalPT" operation="approve" container="request"
   createInstance="yes">
  <source linkName="receive-to-assess"
   transitionCondition="bpws:getContainerData('request', 'amount')<10000"
   />
  <source linkName="receive-to-approval"
   transitionCondition="bpws:getContainerData('request', 'amount')>=10000"
   />
```

```
</receive>

<invoke name="invokeAssessor" partner="assessor"
   portType="asns:riskAssessmentPT" operation="check"
   inputContainer="request" outputContainer="riskAssessment">
  <target linkName="receive-to-assess" />
  <source linkName="assess-to-setMessage"
   transitionCondition="bpws:getContainerData('riskAssessment', 'risk')='low'"
   />
  <source linkName="assess-to-approval"
   transitionCondition="bpws:getContainerData('riskAssessment',
   'risk')!='low'" />
</invoke>

<assign name="assign">
   <target linkName="assess-to-setMessage" />
  <source linkName="setMessage-to-reply" />
  <copy>
  <from expression="'yes'" />
  <to container="approvalInfo" part="accept" />
  </copy>
</assign>

<invoke name="invokeapprover" partner="approver"
   portType="apns:loanApprovalPT" operation="approve"
   inputContainer="request" outputContainer="approvalInfo">
  <target linkName="receive-to-approval" />
  <target linkName="assess-to-approval" />
  <source linkName="approval-to-reply" />
</invoke>

<reply name="reply" partner="customer" portType="apns:loanApprovalPT"
   operation="approve" container="approvalInfo">
  <target linkName="setMessage-to-reply" />
  <target linkName="approval-to-reply" />
</reply>

</flow>
</process>
```

4.1. The Translation to PROMELA

In mapping this process to PROMELA, transition conditions in BPEL4WS are taken care of in PROMELA as *if statements*. We also introduced a Boolean type variable *risk* which indicates whether the loan assessment is risky or not. Here is the PROMELA code for this example:

```
mtype = { cus_info, risky, not_risky, accepted};
show mtype holder;
byte amount=0;
bool risk = false;
chan rec_to_assess = [1] of { mtype };
chan rec_to_app = [1] of { mtype };

chan app_to_rep = [1] of {mtype};
chan assess_to_setmsg=[1] of{mtype};
```

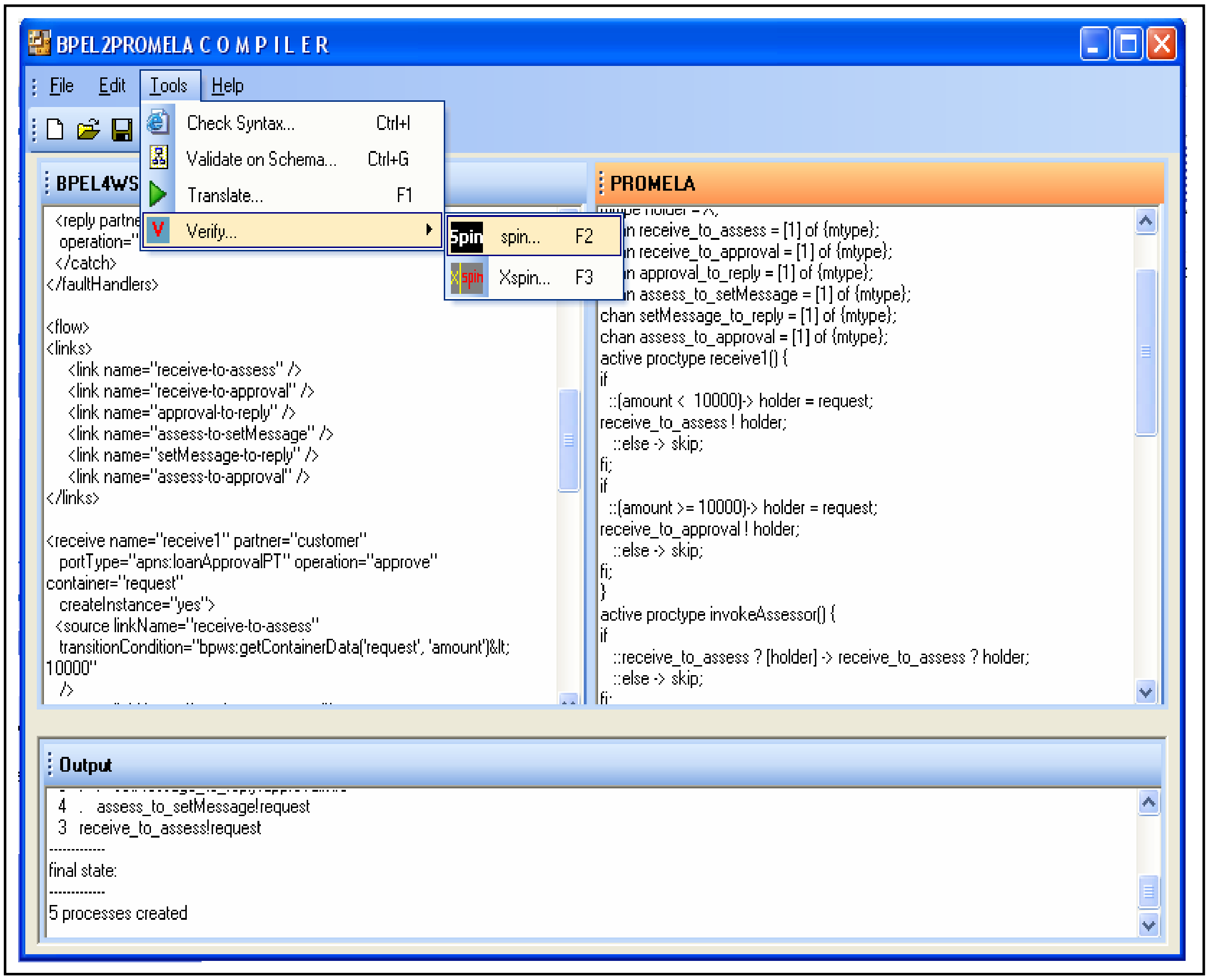
Top

```
chan setmsg_to_rep = [1] of{mtype};
chan assess_to_app=[1] of {mtype};

proctype Receive1(){

holder = cus_info;
do
            ::(amount < 10000) -> rec_to_assess ! holder;
            ::else-> rec_to_app !holder;
od
}

proctype InvokeAssessor() {
do
            ::rec_to_assess ? [holder]->rec_to_assess ? holder;
            if
            ::(!risk) -> holder= not_risky; assess_to_setmsg! holder;
            ::else -> holder= risky; assess_to_app ! holder;
            fi;
            ::else -> skip

od
}

proctype Assign() {
do
            ::assess_to_setmsg ? [holder] -> assess_to_setmsg?holder;holder =
                    accepted; setmsg_to_rep! holder;
            ::else -> skip
od
}

proctype InvokeApprover() {
do
            ::assess_to_app ? [holder] -> assess_to_app ? holder; app_to_rep ! holder;
            ::rec_to_app ? [holder]->rec_to_app ? holder;app_to_rep ! holder;
            ::else -> skip;
od
}

proctype Reply() {
do
            ::app_to_rep ? [holder] ->app_to_rep ? holder;
            :: else -> if
                        ::setmsg_to_rep ? [holder] -> setmsg_to_rep ? holder;
                        ::else -> skip
                        fi
od

}
init {

run Receive1();
run InvokeAssessor();
run Assign();
run InvokeApprover();
run Reply();
}
```

Top

```
/* LTL definitions */
#define p (holder == cus_info )
#define q (holder == accepted)
```

Shown below (Figure 7) is the IDE snapshot of the translation from BPEL4WS specification of this example to PROMELA description.



```
<reply partne
 operation="
 </catch>
</faultHandlers>

<flow>
<links>
    <link name="receive-to-assess" />
    <link name="receive-to-approval" />
    <link name="approval-to-reply" />
    <link name="assess-to-setMessage" />
    <link name="setMessage-to-reply" />
    <link name="assess-to-approval" />
</links>

<receive name="receive1" partner="customer"
  portType="apns:loanApprovalPT" operation="approve"
container="request"
  createInstance="yes">
 <source linkName="receive-to-assess"
  transitionCondition="bpws:getContainerData('request', 'amount')&lt;
10000"
  />
```

```
n receive_to_assess = [1] of {mtype};
n receive_to_approval = [1] of {mtype};
n approval_to_reply = [1] of {mtype};
n assess_to_setMessage = [1] of {mtype};
chan setMessage_to_reply = [1] of {mtype};
chan assess_to_approval = [1] of {mtype};
active proctype receive1() {
if
  ::(amount <  10000)-> holder = request;
receive_to_assess ! holder;
   ::else -> skip;
fi;
if
  ::(amount >= 10000)-> holder = request;
receive_to_approval ! holder;
   ::else -> skip;
fi;
}
active proctype invokeAssessor() {
if
   ::receive_to_assess ? [holder] -> receive_to_assess ? holder;
   ::else -> skip;
```

```
 4  .  assess_to_setMessage!request
 3   receive_to_assess!request
-------------
final state:
-------------
5 processes created
```

Figure 7: IDE Interface (A BPEL4WS code translated to PROMELA)

4.2. The Verification and Simulation Methodology with XSpin:

In this part, we feed the PROMELA code corresponding to the loan approval business process to XSpin and specify LTL properties to validate on our model. An LTL formula *f* may contain any proposition *p*, combined with unary or binary, Boolean and/or temporal operators with respect to the following grammar [2].

Top

```
f ::=          p
             | true
             | false
             | ( f )
             | f binop  f
             | unop  f

unop  ::=      [ ]          (always)
             | <>           (eventually)
             | !            (logical negation)

binop  ::=     U            (strong until)
             | &&           (logical and)
             | ||           (logical or)
             | ->           (implication)
             | <->          (equivalence)
```

Figure 8: LTL Grammar

Consider the loan approval example, and the following basic properties:

- **p = (holder == cus_info)**
- **q = (holder = = accepted)**

Using these propositions, we can check for instance the safety property that is the liveness property: [] (p - > <>q) which means that: always, if we have a message cus_info sent through the channel rec_to_assess and contained in a variable named *holder*, then eventually we receive a message *accepted* through the channel setmsg_to_rep again contained in *holder*. This is due to the transition conditions of the specification. Checking this logical formula with XSpin on the PROMELA model ensures a liveness property of our BPEL4WS model. The property is negated by XSpin so it can search for any occurrence at some state where the property is true. Below is output of the verification of the mentioned property on the loan approval model:

```
(Spin Version 4.1.3 -- 24 April 2004)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 72 byte, depth reached 512, errors: 1 ———— the property is found to be true
     277 states, stored (278 visited)
     250 states, matched
     528 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
```

Top

Running the verification generated by XSpin with the option *–DSAFETY* to check for cycles in the model which eventually leads to *deadlocks* guarantees the safety of the model. Below is the output of the safety verification:

```
(Spin Version 4.1.3 -- 24 April 2004)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 68 byte, depth reached 7013, errors: 0 ———— No deadlocks
   32460 states, stored
   89745 states, matched
  122205 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 2352 (resolved)
(max size 2^18 states)
```

The simulation of the PROMELA model of the *Loan Approval* process is presented graphically in Figure 9a and in text-based simulation in Figure 9b.

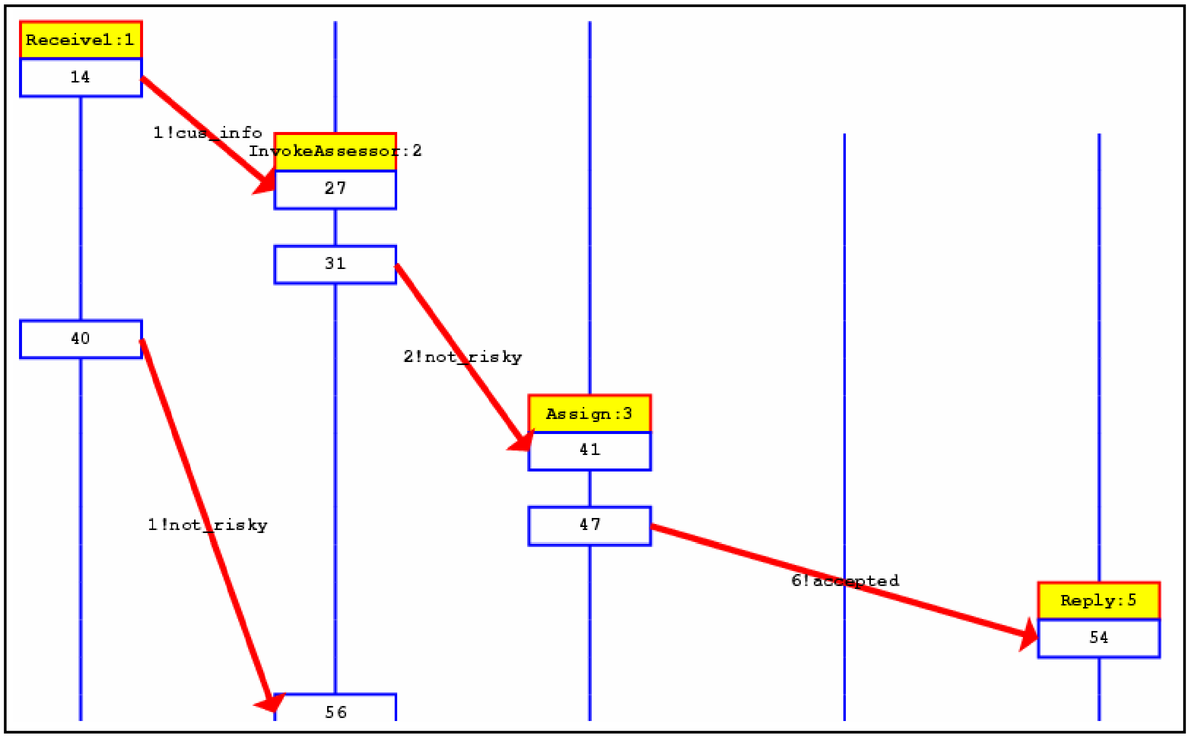
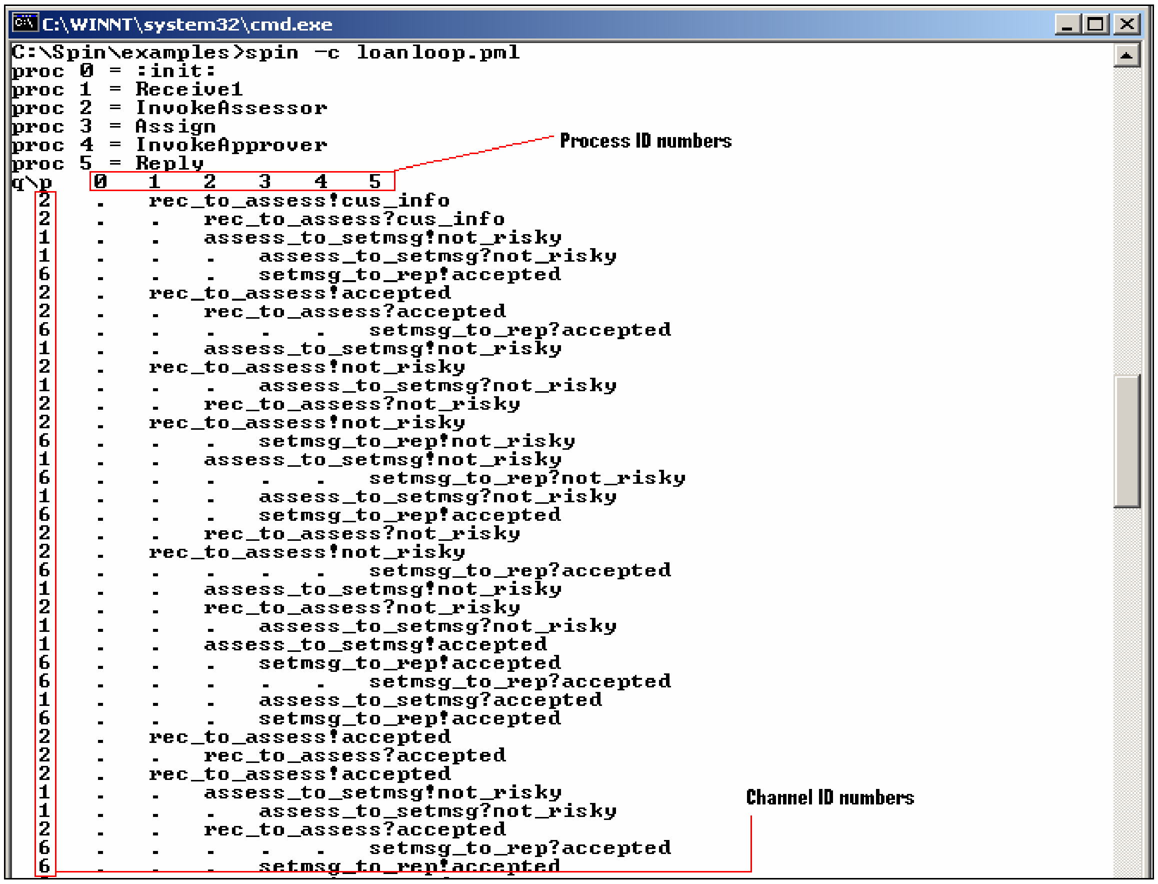


Figure 9 a: Simulation Chart of Processes of the loan approval model

Top



```
C:\Spin\examples>spin -c loanloop.pml
proc 0 = :init:
proc 1 = Receive1
proc 2 = InvokeAssessor
proc 3 = Assign
proc 4 = InvokeApprover
proc 5 = Reply
q\p   0   1   2   3   4   5
  2   .   rec_to_assess!cus_info
  2   .   .   rec_to_assess?cus_info
  1   .   .   assess_to_setmsg!not_risky
  1   .   .   .   assess_to_setmsg?not_risky
  6   .   .   .   setmsg_to_rep!accepted
  2   .   rec_to_assess!accepted
  2   .   .   rec_to_assess?accepted
  6   .   .   .   .   .   setmsg_to_rep?accepted
  1   .   .   assess_to_setmsg!not_risky
  2   .   rec_to_assess!not_risky
  1   .   .   .   assess_to_setmsg?not_risky
  2   .   .   rec_to_assess?not_risky
  2   .   rec_to_assess!not_risky
  6   .   .   .   setmsg_to_rep!not_risky
  1   .   .   assess_to_setmsg!not_risky
  6   .   .   .   .   .   setmsg_to_rep?not_risky
  1   .   .   .   assess_to_setmsg?not_risky
  6   .   .   .   setmsg_to_rep!accepted
  2   .   .   rec_to_assess?not_risky
  2   .   rec_to_assess!not_risky
  6   .   .   .   .   .   setmsg_to_rep?accepted
  1   .   .   assess_to_setmsg!not_risky
  2   .   .   rec_to_assess?not_risky
  1   .   .   .   assess_to_setmsg?not_risky
  1   .   .   assess_to_setmsg!accepted
  6   .   .   .   setmsg_to_rep!accepted
  6   .   .   .   .   .   setmsg_to_rep?accepted
  1   .   .   .   assess_to_setmsg?accepted
  6   .   .   .   setmsg_to_rep!accepted
  2   .   rec_to_assess!accepted
  2   .   .   rec_to_assess?accepted
  2   .   rec_to_assess!accepted
  1   .   .   assess_to_setmsg!not_risky
  1   .   .   .   assess_to_setmsg?not_risky
  2   .   .   rec_to_assess?accepted
  6   .   .   .   .   .   setmsg_to_rep?accepted
  6   .   .   .   setmsg_to_rep!accepted
```

Figure 9 b: Simulation of Promela model of the loan approval example

5) Conclusions & Future Work

We have presented a methodology and a prototype IDE to design, verify and simulate business processes described in BPEL4WS language. This methodology rests on PROMELA/Spin technology and highly benefits from its power to verify distributed processes. It consists firstly in translating BPEL4WS model to a semantically equivalent PROMELA specification. Then taking the PROMELA model and carry out model-checking of safety LTL properties and simulations.

Since our translator is implemented to translate a subset of BPEL4WS, it can be extended in the future to encompass the whole language specification. A design feature of our system is its structure. More enhancements can be added to a module without affecting the other modules. For example, the XML-Validator module can be enhanced to detect syntax errors of the BPEL4WS on the fly. More functionality can be added to this system like generating an UML activity diagram of the input BPEL4WS specification. At last but not least, verification of business processes involves other issues such as dynamic process instances handling [3] which can be investigated and analyzed to even progress on this interesting field of web services integration.

Top

7) References

[1] Business Process Execution Language for Web Services (BPEL), Version 1.1. http://www.ibm.com/developerworks/library/ws-bpel.

[2] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.

[3] H. Foster, S. Uchitel, J. Kramer, and J. Magee. *Model-Based Verification of Web Services Compositions*. Presented at Automated Software Engineering (ASE) Conference 2003, Montreal, Canada, October 2003.

[4] LOANAPPROVAL Example. *BPEL4WS Samples*http://www.doc.ic.ac.uk/ ~hf1/phd/bpel4wssamples.htm

[5] Roozbeh Farahbod, Uwe Glasser and Mona Vajihollahi. *Specification and Validation of the Business Process Execution Language for Web Services*. Technical Report, School of Computer Science, Simon Fraser University., 2003.

[6] System.Xml namespace. *.Net Framework Class Library.* http://msdn.microsoft.com/library/

default.asp?url=/library/en-us/cpref/html/frlrfsystemxmlxmlreaderclasstopic.asp

[7] *Introduction to OMG UML*. http://www.omg.org/gettingstarted/what_is_uml.htm

Top