A method of Verifying Web Service Composition

Thang Huynh Quyet
Hanoi University of Technology
Number 1, Dai Co Viet Street
Hanoi, Vietnam
84-4-38692463
thanghq@it-hut.edu.vn

Quynh Pham Thi
Hanoi National University of Education
Number 136, Xuan Thuy Street
Hanoi, Vietnam
84-904000240
ptquynh@hnue.edu.vn

Duc Bui Hoang
Hanoi University of Technology
Number 1, Dai Co Viet Street
Hanoi, Vietnam
84-972347051
ducbuihoang@gmail.com

ABSTRACT

Service composition is one of the primary tasks in developing service-oriented systems. However, there are currently some challenges to check its correction. In this paper, we propose a visual methodology and a tool for verifying business processes written in BPEL by using the SPIN model checker. We present algorithms to translate BPEL processes into PROMELA programs via labeled control flow graphs. The use of label control graphs in the tool will help regular users understand BPEL business processes and the verification process with a model checker more easily. Finally, the Spin model checker will verify important properties of the PROMELA program that represents a BPEL business process..

Categories and Subject Descriptors

D.2.4 **[Software/Program Verification]**: Model checking.

General Terms

Algorithms, Languages, Verification.

Keywords

BPEL, SPIN, PROMELA, software verification.

1. INTRODUCTION

Building software systems based on web services has brought huge benefits such as decreasing cost, avoiding risk and easy maintenance. In those, the composition of web services which are executed following a business process is a significant requirement. BPEL (Business Process Execution Language) [1] was created in order to do that task.
One significant requirement when creating BPEL processes is the verification of their correctness. Today, many researches have been conducted to solve this problem [3]. These researches

SPIN [2] is a popular model checker that verifies programs written in PROMELA. Although it's possible to transform directly from BPEL processes into PROMELA programs, that method does not help users understand the specifications of the BPEL processes and verified models.

In this paper, we propose another method that verifies BPEL processes using SPIN model checker. In our method, a BPEL process is transformed from an XML document to a graph visually. Key information of the process is preserved and arranged neatly on graph. After that, the graph form is transformed into a PROMELA program which is verified by SPIN. In this paper, we focus on the problem related to synchronization dependencies in a BPEL process.

The rest of this paper is organized as follows: Section 2 gives a transformation method from a BPEL process to a graph. Section3 presents a transformation method from a graph to a PROMELA program. The implementation is presented in Section 4. We also provide a case study that illustrates the whole method in Section 5. Section 6 gives related work and discussion. The final section is about conclusion and future works.

2. TRANSFORMATION FROM BPEL TO LCFG

A BPEL process specifies a business process like a flowchart [1]. Every element in the process is called an activity. Each activity is either basic or structured. A structured activity encloses one or many other activities. Thus, formally, we propose Labeled Control Flow Graph (LCFG) which is a form of graph for BPEL. The LCFG is derived from traditional CFG and added label for each nodes in graph.

It is defined as follow: LCFG (N, E) is a directed graph, in which set N is a set of nodes and set E is a set of edges that represent exchanges between nodes. In set N, there is only one Start and Stop nodes. Other nodes represent activities in BPEL process. These nodes are labeled to describe important information of activities. Edges in set E represent the order of vertices.

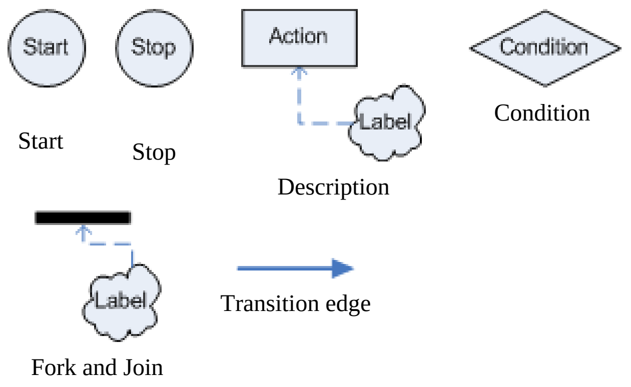
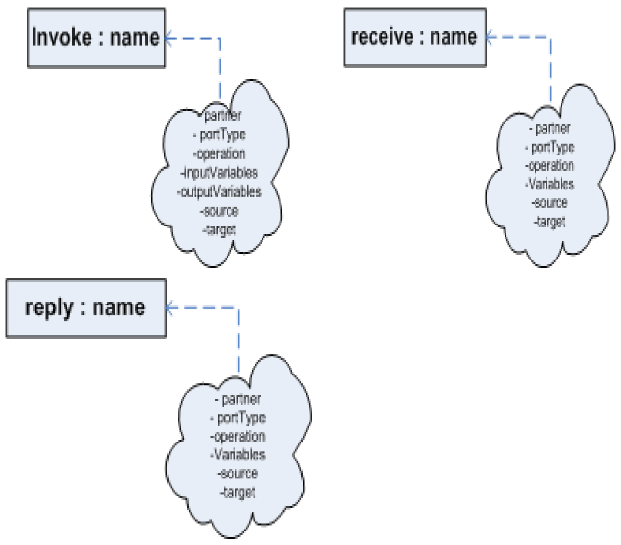


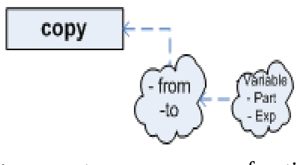
Figure 1. Elements in LCFG

Activities in BPEL are categorized into 2 types: basic activities and structured activities. Every activity has a name attribute. So that the name of a vertex is: ***activity type: activity name.*** The assignment of label for every vertex type depends on its corresponding activity.

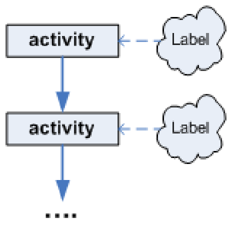
The activities that relate to the operations of services such as **invoke, receive** and **reply** activities provide service-related information via the partner attributes that describe the called *portType* and *operation*. However, the role of each activity will determine the meaning of arguments that are required. For example, invoke activities need both *inputVariable* and *outputVariable*. In receive activity, the *Variable* attribute represents the message that activity receives. On the other hand, the *Variable* attribute represents the message that is sent by the Reply activity.



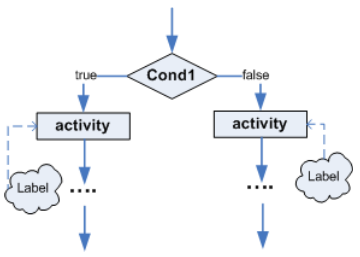
Assignment activity assigns data from element *from* to element *to*. These elements may be a variable, an attribute or an expression. So we have to assign the labels at two levels. Labels for assignment activity are from and to. After that, we continue to label the element from and to which are *Variables, Part* and *Expression*.



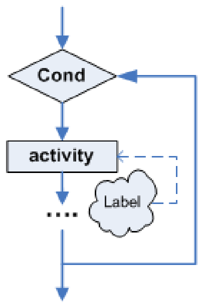
Sequence activity executes a sequence of activities. Thus, this activity is described by a set of vertices that represent a sequence of enclosed activities.



If activity is a branching activity. A branch is executed when a condition is satisfied. So we need to add condition vertices. Each branch consists of one or a set of activities.



While activity represents the repetition of activities when a condition is true. It's necessary to add a condition vertex. When the execution of activities inside the loop completes, the condition is checked again.

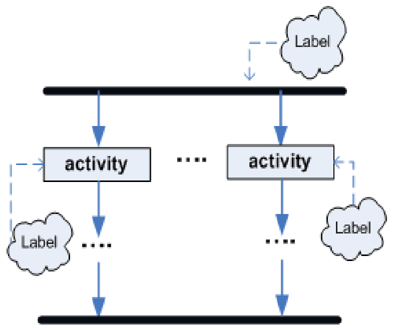


Flow activity describes concurrent activities. Each branch of the flow activity is executed separately. Each branch is either a basic or structured activity. However, there are synchronization dependencies which are described in the *link* attribute. The activities with <source> or <target> tags can be considered as the start and the end of links. The activity which is labeled with <target> is executed only if the joinCondition is evaluated to true. If the joinCondition is omitted, the joinCondition is the logical OR of all incoming links' status so the activity will execute when one of the link status is true. A link's status can be either 'true', 'false' or 'unset'. If the suppressJoinFailure attribute is set to 'false', then the is no fault to be thrown and the out coming links' status is set to false, otherwise, the bpel:joinFailure fault will be thrown and be caught by a fault handler. Each activity has one or many <source> or <target>

element. The <source> element contains the <transitionCondition> element which specifies the condition that determines when the <target> element is executed.

We simply consider the activities in each branch of the flow to be sequential if they are source and target of a link. The activity with the source label will execute first and the activity with the target label will execute later. The number of the outgoing edges of an activity equals to the number of its <source> labels. The names of these edges are the names of links. If there is the <transitionCondition> in the source label then we add a condition vertex to this link.

A flow activity is represented by a fork and a join node. In those, the label of the fork node stores information about the links in the flow activity.



3. TRANSFORMATION FROM LCFG TO PROMELA

To verify the process, the LCFG structure needs to be transformed into a PROMELA program. Each PROMELA program contains 3 main components: process, message channels and variables. The graph's main structure will be transformed into main process; the labels of nodes are the necessary information for the message channel definitions and variables.

Variables in the PROMELA program are determined from the labels with variable tag. If the data type of a variable exists in PROMELA then we create a variable with the same data type. If there is no corresponding data type of a variable, we will define them as *mtype*. For the complex variable with parts, we will create structured data type in PROMELA using *typedef*.

Each label *portType* corresponds with 2 message channels which are in channel and out channel. The in channel is used for receiving messages and out channel is used for sending messages. The type of channel is the data type of messages that transferred through the portType.

Each graph corresponds to a main *proc* of the program. The transformation principles from LCFG to PROMELA instructions are based on the similarity of semantics.

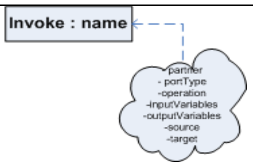
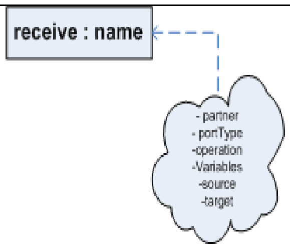
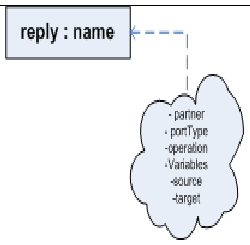
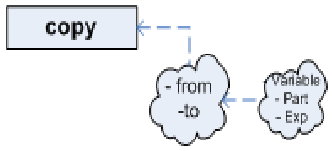
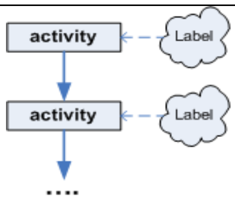
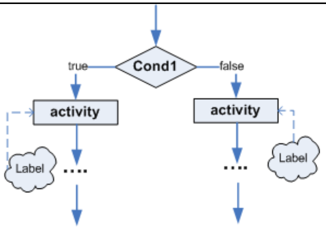
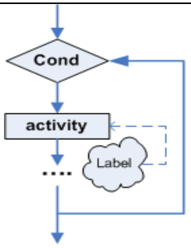
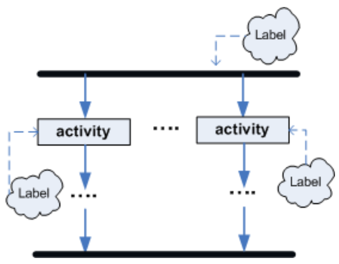
Node	Instruction
 Invoke : name - partner - portType -operation -inputVariables -outputVariables -source -target	***portType_OUT ! output_var*** ***portType_IN ? input_var*** Receive and send information between channels.
 receive : name - partner - portType -operation -Variables -source -target	***portType_IN ? variable*** Receive data from a channel into a variable
 reply : name - partner - portType -operation -Variables -source -target	***portType_OUT ! variable*** Send data from a variable to a channel
	***to=from*** When assigning, we take into account the data parts of variables or XPath expressions.
	Sequential instructions.
 true Cond1 false activity activity Label .... Label	If :: Cond1 -> :: … :: else -> fi
	Do :: Cond -> … Od
 Label activity .... activity Label .... Label	- Each branch corresponds to a proc. - Define each proc according to the rules in this table. - In the main proc, create **run** instructions that invoke the above processes.

Table 1. From LCFG to PROMELA

The following algorithm describes the graph traversal process and transformation to PROMELA language.

Step 1: Create declaration and main process.

Step 2: Traversing LCFG

Step 2.1: Visit one vertex in graph

Step 2.2: If it is Stop vertex, go to Step 3; else go to 2.3

Step 2.3: Determine the type of the vertex and use the transformation rules in Table

Step 2.4: If it a vertex corresponding to a structured activity, go to step 2.1; else go to step 2.5

Step 2.5: Consider the label of the vertex, add new variable, channel and data type into the declaration.

Step 3: Synthesize the declaration

Step 3.1: Determine the data types: *typedef* and *mtype*.

Step 3.2: Declare variables.

Step 3.3: Declare channels.

//The traversal process is sequential

This is a recursive algorithm which supports in visiting every vertex in the LCFG. Step 2 is primary. In step 2, we have to identify whether considered vertex is basic or structured. If it is structured node, all its nested nodes will compose to a sub-LCFG. The traversal processes of sub-LCFG and LCFG are similar. This algorithm finishes when Stop node is visited.

4. IMPLEMENTATION

In this section, we will describe the overview architecture of a tool built to transform and verify BPEL processes. The main architecture of the tool is described in Figure 2.

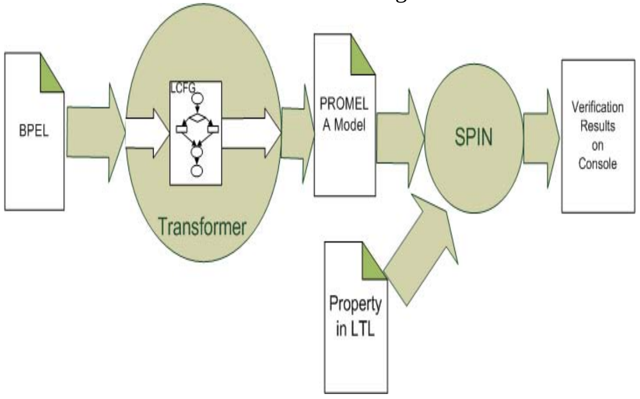


Figure 2. Elements in tool

The main element of this tool is Transformer. This is the element responsible for the transformation between forms: from BPEL to LCFG and to PROMELA. The input of this element is a BPEL process and the output is a PROMELA program corresponding to the process. With that role, this element contains packages corresponding to the transformation between models.

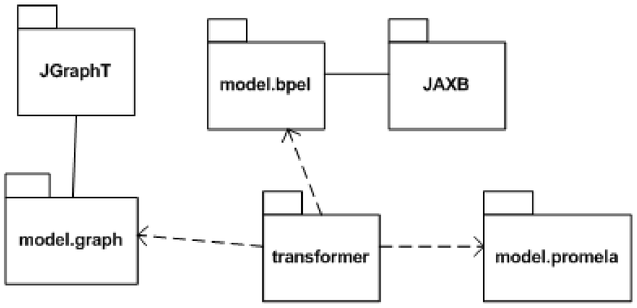


Figure 3. Packages in Transformer

In BPEL model, the package model.bpel.abs and model.bpel.exe contain hundreds of classes generated from BPEL2.0 XML schema for abstract and executable processes by using JAXB library [8]. TProcess in JAXB is an important class. Every element in BPEL process can be found in the class. We use this information to transform BPEL document into LCFG.

In package model.graph, LCFG class represents a LCFG. It contains attributes and methods that are specific for elements of LCFG as Section 2 shows. LCFG class inherits from GraphT class – describes graph's internal structure for storage and Graph5 class – represents graph's external structure for display.

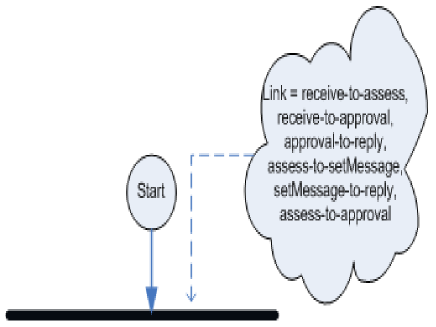
In PROMELA model, there are 3 packages which specify structure of PROMELA programs. The model.promela package contains model classes of PROMELA language. The model.promela.literal package contains model classes of literal characters in PROMELA language. The model.promela.literal.op package contains model classes of operations in PROMELA language. Besides, we also implement algorithm to translate from LCFG to PROMELA program in this model.

5. A CASE STUDY

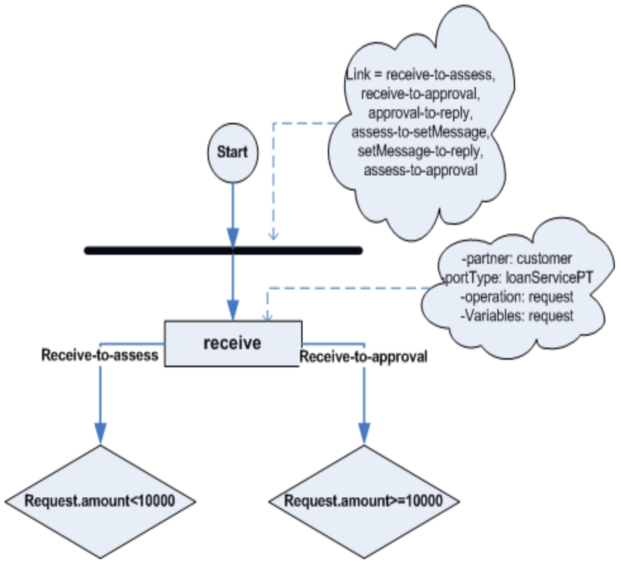
To illustrate the transformation process over different forms and checking attributes of BPEL process, we will use the process Loan Approval [7]. Roughly observation may make readers falsely think that the process only contains the concurrent activities. However, after more detailed analysis, we can see these activities have synchronization dependencies. In the following part, we will describe in detail each step of the transformation process into different representations and checking the required properties of this process.

The first step is to represent the BPEL process in the LCFG form. The transformation principles were described in Section 2. The whole process will be represented as a LCFG graph.

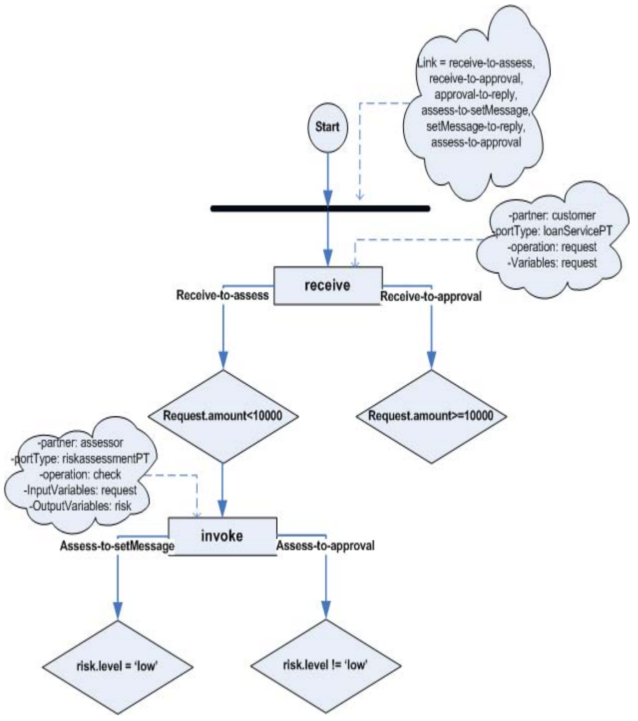
The first activity – flow activity is represented and labeled as the following:


Link = receive-to-assess, receive-to-approval, approval-to-reply, assess-to-setMessage, setMessage-to-reply, assess-to-approval
Start

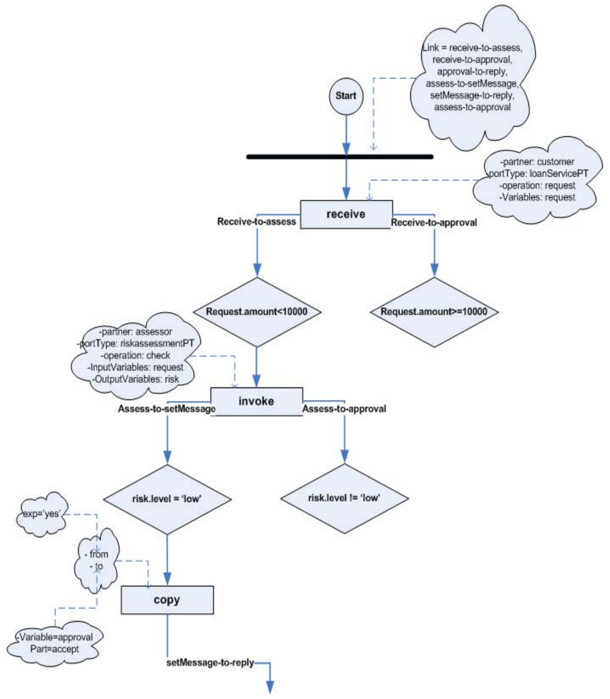
Then, receive activity – beside the common information of this activity, it has 2 labels <source> and <transitionCondition> elements. This activity has not <target> label so it is executed immediately the fork vertex of the flow activity. This vertex also is a start of 2 edges and has conditions that correspond to the <transitionCondition>.



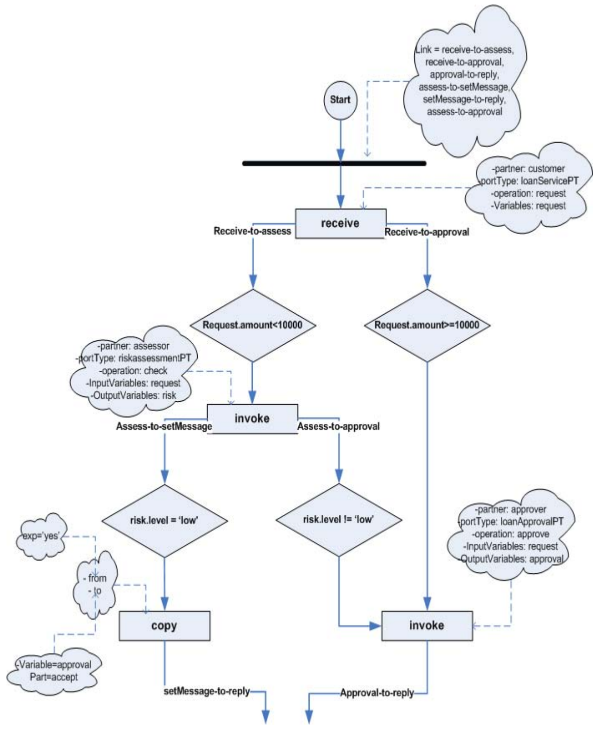
The following invoke activity has a <target> label and 2 <source> labels. Thus, the vertex representing this activity is the target of the receive-to-assess edge and is the starting point of 2 other edges. These 2 edges have additional vertices that represent the <transitionCondition>.



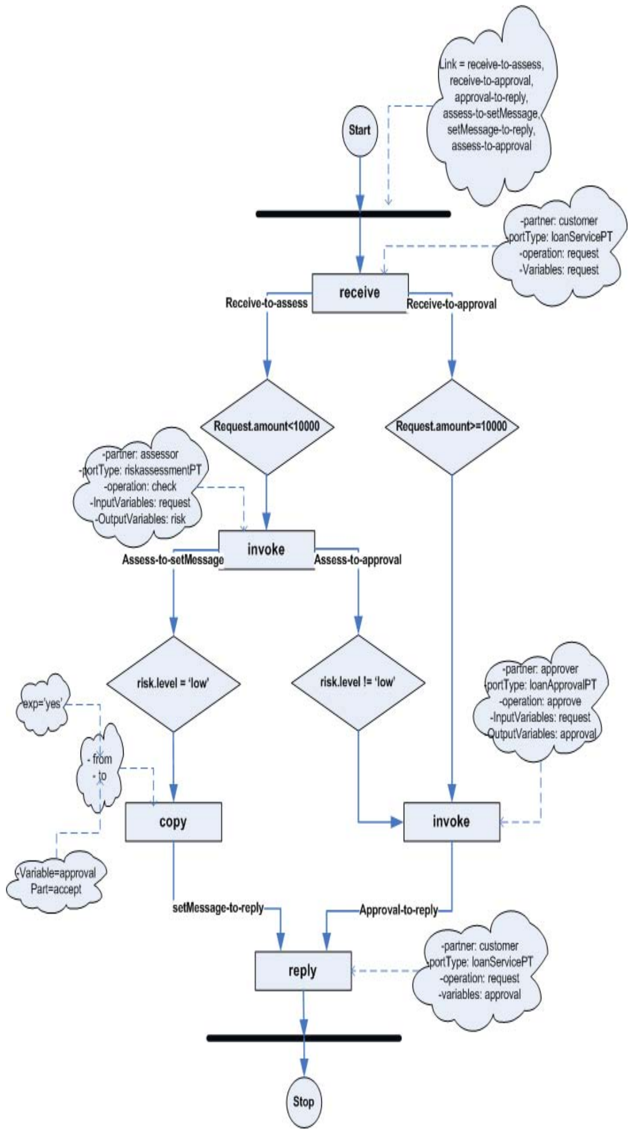
After that, the assign activity whose target is the assess-to-setMessage edge and is the starting vertex of the setMessage-to-reply. The <from> element is an expression and the <to> element contains variables and parts.



The following invoke activity is represented by vertex which is the target of receive-to-approval edge and assess-to-approval edge. This vertex is also the starting point of the approval-to-reply edge.



Finally, the reply activity has only 2 <target> elements. Therefore, the vertex representing this activity is the target of 2 edges setMessage-to-reply and approval-to-reply. The next vertex is the join vertex that represents the end of concurrent activities. The final stop vertex indicates the end of whole process.


Link = receive-to-assess, receive-to-approval, approval-to-reply, assess-to-setMessage, setMessage-to-reply, assess-to-approval
Start
-partner: customer -portType: loanServicePT -operation: request -Variables: request
receive
Receive-to-assess
Receive-to-approval
Request.amount<10000
Request.amount>=10000
-partner: assessor -portType: riskassessmentPT -operation: check -InputVariables: request -OutputVariables: risk
invoke
Assess-to-setMessage
Assess-to-approval
risk.level = 'low'
risk.level != 'low'
-partner: approver -portType: loanApprovalPT -operation: approve -InputVariables: request -OutputVariables: approval
exp='yes'
- from - to
copy
invoke
-Variable=approval Part=accept
setMessage-to-reply
Approval-to-reply
reply
-partner: customer -portType: loanServicePT -operation: request -variables: approval
Stop

After creating LCFG graph of BPEL process, the next step is to transform it into a PROMELA program.

The following PROMELA program is the outcome of the process:

```
typedef creditInformationMessage{
    short amount
};

typedef approvalMessage{
    bool accept
};

typedef riskAssessmentMessage{
    bit level
};

chan loanServicePT_IN = [0] of {creditInformationMessage};
chan loanServicePT_OUT = [0] of {approvalMessage};

chan riskAssessmentPT_IN = [0] of
{riskAssessmentMessage};
chan riskAssessmentPT_OUT = [0] of
{creditInformationMessage};

chan loanApprovalPT_IN = [0] of {approvalMessage};
chan loanApprovalPT_OUT = [0] of
{creditInformationMessage};

creditInformationMessage request;
riskAssessmentMessage risk;
approvalMessage approval;

byte result=0;

proctype loanApproval(){
    loanServicePT_IN ? request;
    if
    :: request.amount<10000 ->
            /*invoke*/
            riskAssessmentPT_OUT ! request;
            riskAssessmentPT_IN ? risk;
            if
            ::risk.level== 1 ->
            /* assign */
            approval.accept = true;
            ::risk.level== 0 ->
            /* invoke */
            loanApprovalPT_OUT ! request;
            loanApprovalPT_IN ? approval;
            fi;
            :: request.amount>=10000 ->
                        /* invoke */
            loanApprovalPT_OUT ! request;
            loanApprovalPT_IN ? approval;
            fi;
            /* reply */
            loanServicePT_OUT!approval;
    }

    proctype customer(){
            approvalMessage customer_Receive;
  creditInformationMessage customer_Send;
    /*randomize the amount*/
    short r;
    do
    :: r++;/* randomly increment */
    :: r--;/* or decrement      */
    :: break;           /* or stop        */
    od;
    customer_Send.amount=r;
    loanServicePT_IN ! customer_Send;
    loanServicePT_OUT ? customer_Receive;
}
```

```
proctype assessor(){
creditInformationMessage assessor_Receive;
riskAssessmentMessage assessor_Send;
end:riskAssessmentPT_OUT? assessor_Receive;
    /*simulate the risk level*/
    bit r;
    if
    ::      true->r=1;
    ::      true->r=0;
    fi;
    assessor_Send.level=r;
    riskAssessmentPT_IN!assessor_Send;
}

proctype approver(){
creditInformationMessage approver_Receive;
approvalMessage approver_Send;
end:loanApprovalPT_OUT?approver_Receive;
    /*simulate the accept*/
    bool r;
    if
    ::      true->r=true;
    ::      true->r=false;
    fi;
    approver_Send.accept=r;
    loanApprovalPT_IN!approver_Send;
}


init{
    run loanApproval();
    run assessor();
    run approver();
    run customer();
}
```

Table 2. PROMELA program for Loan Approval process

Finally, we use SPIN model checker to verify common properties of the process Loan Approval. And the result shows that the program ends at valid states and there is no error yet.

```
 (Spin Version 5.2.4 -- 2 December 2009)
            + Partial Order Reduction
Full statespace search for:
            never claim          - (none specified)
            assertion violations +
            cycle checks         - (disabled by -DSAFETY)
            invalid end states   +
State-vector 96 byte, depth reached 65561, ••• errors: 0 •••
  1667332 states, stored
    65537 states, matched
  1732869 transitions (= stored+matched)
        0 atomic steps
hash conflicts:    959894 (resolved)
  218.113 memory usage (Mbyte)
unreached in proctype loanApproval
            (0 of 18 states)
unreached in proctype customer
            (0 of 10 states)
unreached in proctype assessor
            (0 of 10 states)
unreached in proctype approver
            (0 of 10 states)
unreached in proctype :init:
            (0 of 5 states)
pan: elapsed time 2.69 seconds
pan: rate 620518.05 states/second
```

Table 3. Result of checking common properties

Firstly, 4 processes: loanApproval, assessor, approver and customer are created. Then, the customer creates a random amount for request variable and sends it to loanApproval process. The loanApproval process will continue according to the value of the amount. It may interact with assessor and approver when it executes. The process calls assessor to get the risk level of the request when needed. If the risk level is not low or the amount is greater than 10000 then loanApproval will call approver. If the amount is less than 10000 and the risk level is low then loanApproval will accept the request automatically. Finally, loanApproval will continue to its end.

Beside the above properties, users can create queries in the form of LTL formulas to check whether the process satisfies the queries. For example, in the process Loan Approval, there is a query: For requests of the same amount, is there any case in which one of the requests is approved but the other is processed differently?

To answer this question, we declare a variable named **result** of type int. When the process starts, **result** is initialized to 0. If the request is approved (after the activity assign), the value of **result** is 1. If the request is checked further (after the invoke activity of approver) then the value of result is 2. Beside the declaration instructions and alternate the value of result, the query will be written as the following:

```
#define accepted (approval.accept==true)

#define rejected (approval.accept==false)

!(<>(accepted && rejected))
```

The result of checking shows that the Loan Approval process satisfies this LTL formula.

6. DISCUSSION AND RELATED WORK

There are currently verification techniques and tools using SPIN for BPEL process. Fu, Bultan and Su [4] present a framework to verify properties of a web service composition. Each BPEL process is translated to a guarded automaton.

In [5], Nakajima presents a translation from BPEL to PROMELA. This translation includes two parts. First, a BPEL activity is mapped to an extended finite automaton. This provides a formal model for BPEL activities. Second, the automaton is represented in PROMELA.

In our methodology, the use of LCFG as a intermediate format is a main idea. LCFG is very useful in translating from BPEL to PROMELA, especially it can solve problem related to synchronous dependencies.

In fact that, we can translate from BPEL to PROMELA code directly. But, LCFG will bring following advantages:

- This is visual representation. It is easy to understand.
- We can extract information that unnecessary for verification. The useful information is stored in each node's label.
- This is a complete solve for concurrent activities. Depending on attributes of the <flow> activity, we will restructure all its nested activities by a sequence of activities or branch activities.

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a method of verifying a BPEL process visually. The transformation from a BPEL process into LCFG graph helps developers easily grasp the process and removes the information which is unnecessary for the verification. Besides, we have proposed solution for the synchronization activity.

Then, the LCFG graph is transformed into a PROMELA program. The label of the vertex stores necessary information for this transformation process. The elements from the BPEL process are mapped into PROMELA elements according to the equivalence of semantics.

Finally, the verification of properties of the BPEL process is performed by SPIN model checker. Beside the default properties such as liveness and safeness, users can adds other checking conditions via LTL formulas.

As such, this method not only takes advantage of the SPIN model checker's power but also open an approach of applying SPIN easily and users need not to have too much theoretical knowledge of model checking. Besides, we have given a method of solving problems related to synchronization relationship between concurrent activities.

In the future, we will continue improve the transformation activities related to error catching in BPEL processes. Furthermore, we will support users to create LTL formulas more easily.

8. REFERENCES

1. http://www.w3.org/DOM/
2. http://spinroot.com/
3. Franck van Breugel and Maria Koshkina. Models and verification of BPEL, September 2006. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf
4. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In S.I. Feldman, M. Uretsky, M. Najork, and C.E. Wills, editors, Proceedings of the 13th International World Wide Web Conference, pages 621-630, New York, NY, USA, May 2004. ACM.
5. S. Nakajima. Lightweight formal analysis of web service ows. Progress in Informatics, 1(2):57-76, November 2005.
6. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, Proceedings of the 3rd International Conference on Business Process Management, volume 2649 of Lecture Notes in Computer Science, pages 220{235, Nancy, France, September 2005. Springer-Verlag.
7. Business Process Execution Language for Web Services (BPEL), Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
8. https://jaxb.dev.java.net/