

# Monitoring Web Service Networks in a Model-based Approach

Yuhong Yan

National Research Council  
yuhong.yan@nrc.gc.ca

Yannick Pencolé

The Australian National University  
Yannick.Pencole@anu.edu.au

Marie-Odile Cordier

IRISA, Rennes, France  
cordier@irisa.fr

Alban Grastien

IRISA, Rennes, France  
agrastie@irisa.fr

Nov. 14 - 16, 2005, ECOWS05, Växjö, Sweden

---

# The Problem of Monitoring Web Service Networks

- Individual Web services belong to different organizations
- No central control, nor central model
- Need mechanism to monitor Web service network and find the causes of faults

---

## Our Solution

- Transform Web service process descriptions (e.g. in BPEL) into Discrete Event System model.
- Apply model-based reasoning approach to find the cause of the faults.

---

# What are the faults in Web services

- Web service invocation faults
  - e.g. parameter number or type mismatch, timeout ...
  - throw intrinsic Web service exceptions.
  - switch to exception processing.
- Business logic faults
  - e.g. received order with wrong items, wrong prices
  - these faults can propagate among several Web services before an invocation faults thrown, or they remain as semantic errors which can't be detected by Web service exception handling mechanism.

---

# The tasks of fault management in Web service network

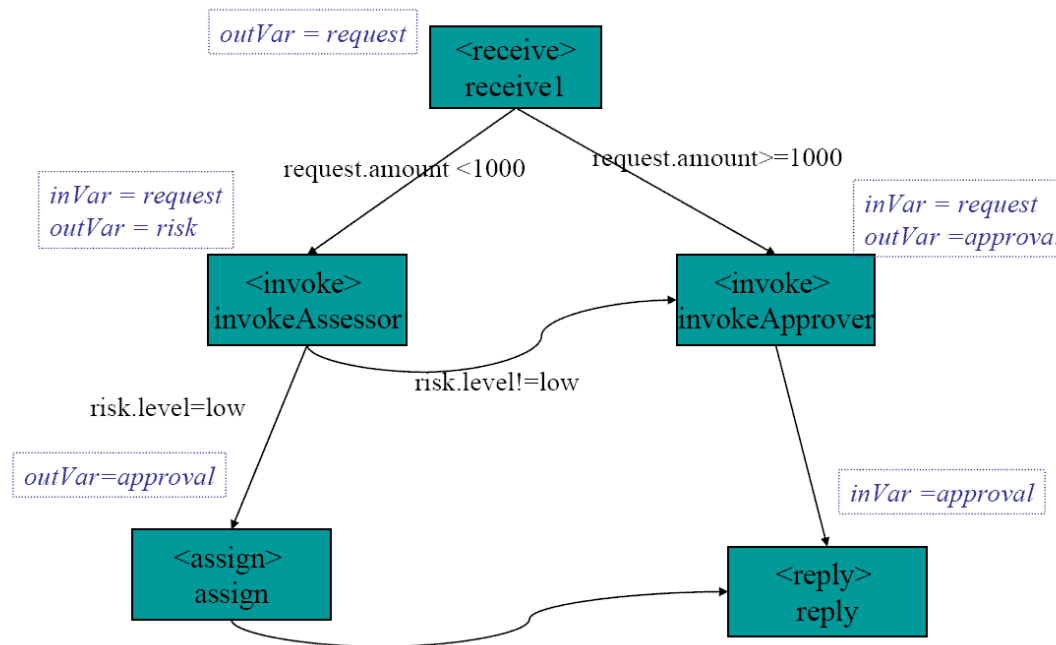
- Monitor the status of execution and record necessary and sufficient information for online/offline diagnosis
- Detect the faults: by detecting the thrown information or comparing observations to the predictions of the behavior model
- Diagnose the causes of faults: exceptions are the symptoms of faults
- Fault recovery: eliminate the effects of faults and reach the goals of the process

---

# **BPEL4WS: behavior model for Web service network**

- Basic activities: ⟨receive⟩ , ⟨reply⟩ ,⟨invoke⟩
- Structural activities: ⟨sequence⟩ , ⟨flow⟩
- Synchronization links: ⟨source⟩ , ⟨target⟩
- Link joint conditions: not discussed

# A BPEL4WS example: a loan approval process in block diagram



---

# A BPEL process is a composition of activities

- $V$ : is a finite set of variables;
- $\mathcal{D}$  is the finite domain for the variables  $V$ ;
- $R$  is a finite set of rules defined as follows:  $(pre(V)) \xrightarrow{event} (post(V))$  where  $pre(V)$  is a precondition (or requirement) (boolean expression on the variables  $V$ ) and  $post(V)$  is the postcondition (or effect).

---

# Discrete-event system

**Definition 1.** *A discrete-event system  $\Gamma$  is a tuple  $\Gamma = (X, \Sigma, T, I, F)$  where:*

- *$X$  is a finite set of states;*
- *$\Sigma$  is a finite set of events;*
- *$T \subseteq X \times \Sigma \times X$  is a finite set of transitions;*
- *$I \subseteq X$  is a finite set of initial states;*
- *$F \subseteq X$  is a finite set of final states.*

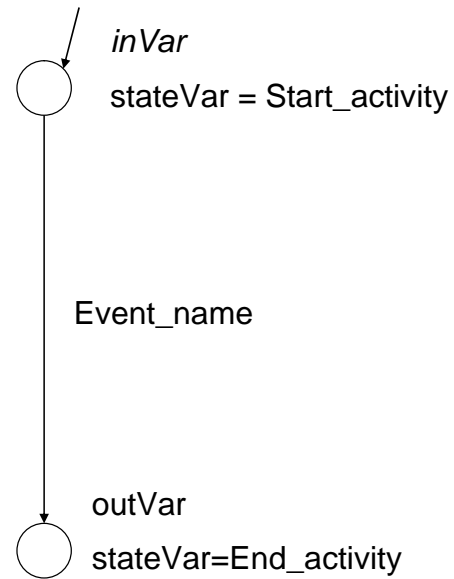
---

## Model BPEL activity in DES

- $Activity(\{Start, End, Event_i\}, inVar, outVar, stateVar)$
- state variable:  $inVar \in V, outVar \in V, stateVar \in \{Start\_activity, End\_activity\} \subset V$
- event:  $(Start\_activity) \xrightarrow{Event\_name} (End\_activity)$
- transition rule:  $(pre(inVar) \wedge stateVar = Start\_activity) \xrightarrow{Event\_name} (post(outVar) \wedge stateVar = End\_activity)$

---

# Model BPEL activity in DES (2)



---

## From BPEL to DES model

- Model basic activities
- Model structure activitie
- Model synchronization links

---

## ⟨Receive⟩

State variables:  $soapMsg$ ,  $received$ ,  
 $stateVar \in \{Start\_receive, End\_receive\}$

Internal variable:  $msgType$  is a predefined message type

Events:  $Receive$

Rules:

- $(stateVar = Start\_receive \wedge soapMsg.type = msgType) \xrightarrow{Receive}$   
 $(received = soapMsg \wedge stateVar = End\_receive)$

---

## ⟨Reply⟩

State variables:  $rep$ ,  $soapMsg$ ,  
 $stateVar \in \{Start\_reply, End\_reply\}$

Events:  $Reply$

Rules:

- $(stateVar = Start\_reply \wedge exists(rep)) \xrightarrow{Reply} (soapMsg = rep \wedge stateVar = End\_reply)$

---

## ⟨Invoke⟩ : Synchronous

State variables:  $inVar$ ,  $outVar$ ,

$stateVar \in \{Start\_invoke, End\_invoke, Wait\}$

Events:  $Invoke$ ,  $Receive$

Rules: Synchronous invocation

- $(stateVar = Start\_invoke \wedge exists(inVar)) \xrightarrow{Invoke} (stateVar = Wait)$
- $(stateVar = Wait) \xrightarrow{Receive} (stateVar = End\_invoke \wedge exist(outVar))$

---

## ⟨Invoke⟩ : **Asynchronous**

State variables:  $inVar$ ,  $outVar$ ,

$stateVar \in \{Start\_invoke, End\_invoke, Wait\}$

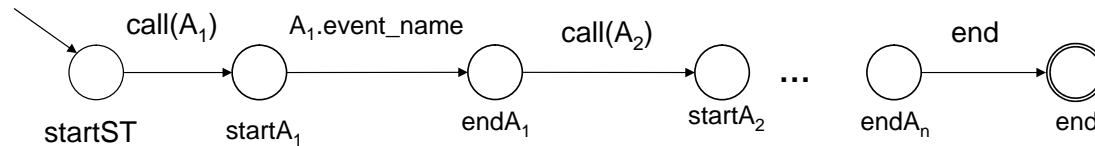
Events:  $Invoke$ ,  $Receive$

Rules: Asynchronous invocation

- $(stateVar = Start\_invoke \wedge exists(inVar)) \xrightarrow{Invoke} (stateVar = End\_invoke)$

---

## ⟨Sequence⟩



State variables:  $stateVar \in \{State\_sequence, End\_sequence, StartA_i, EndA_i, i \in \{1, \dots, n\}\}$

Events:  $\{Call(A_i), End, A_i.event\_name, i \in \{1, \dots, n\}\}$

---

Transitions:

$$(Start\_sequence) \xrightarrow{Call(A_1)} (StartA_1)$$

$$(State\_A_1) \xrightarrow{A_1.event\_name} (EndA_1)$$

$$(EndA_1) \xrightarrow{Call(A_2)} (StartA_2)$$

...

$$(EndA_i) \xrightarrow{Call(A_{i+1})} (StartA_{i+1})$$

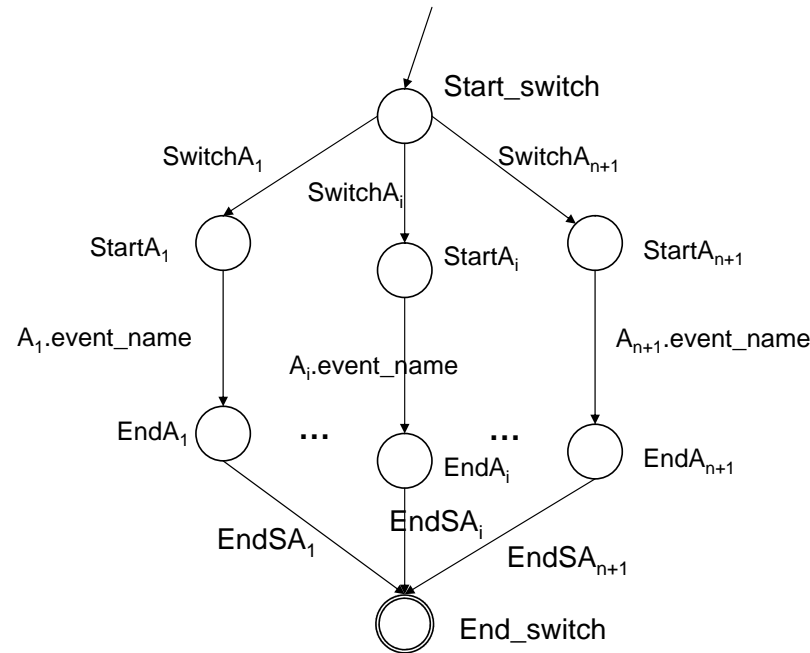
...

$$(EndA_n) \xrightarrow{Call(A_{i+1})} (StartA_{i+1})$$

Rules for transitions:

- $(stateVar = Start\_sequence) \xrightarrow{Call(A_1)} (stateVar = StartA_1)$
- $(stateVar = EndA_i) \xrightarrow{Call(A_{i+1})} (stateVar = StartA_{i+1})$
- $(stateVar = EndA_n) \xrightarrow{End} (stateVar = End\_sequence)$

# ⟨Switch⟩



State variables:  $V_1, \dots, V_n$  are variable sets on  $n$  ⟨case⟩ branches,  
 $stateVar \in \{Start\_switch, End\_switch, StartA_i, EndA_i, i \in \{1, \dots, n + 1\}\}$

---

Events:  $\{SwitchA_i, EndSA_i, A_i.event\_name, i \in \{1, \dots, n + 1\}\}$

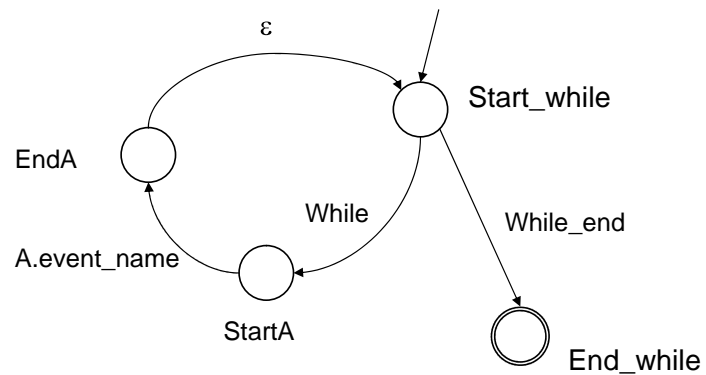
Transitions:  $(Start\_switch) \xrightarrow{SwitchA_i} (StartA_i)$   
 $(StartA_i) \xrightarrow{A_i.event\_name} (EndA_i)$   
 $(EndA_i) \xrightarrow{EndSA_i} (End\_switch)$

Rules:

- $(stateVar = Start\_switch \wedge \neg pre(V_1) \wedge \dots \wedge \neg pre(V_{i-1}) \wedge pre(V_i)) \xrightarrow{SwitchA_i} (stateVar = StartA_i)$
- $(stateVar = Start\_switch \wedge \neg pre(V_1) \wedge \dots \wedge \neg pre(V_{i-1}) \wedge pre(V_i)) \xrightarrow{SwitchA_i} (stateVar = StartA_{n+1})$
- $(stateVar = EndA_i) \xrightarrow{EndSA_i} (stateVar = End\_switch)$

---

# ⟨While⟩



State variables:  $W \subseteq V$ ,

$stateVar \in \{Start\_while, End\_while, StartA, EndA\}$

Events:  $\{While, While\_end, A.event\_name\}$

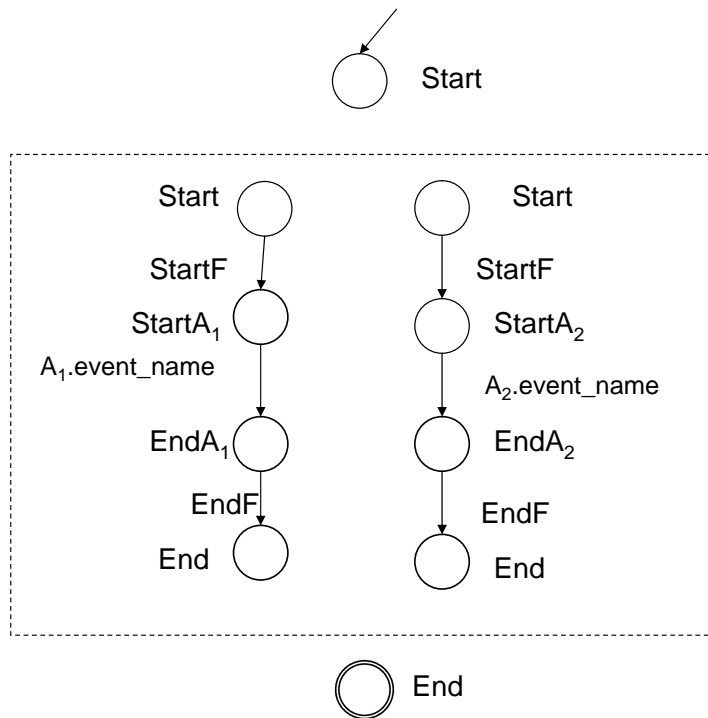
---

Transitions:  $(Start\_while) \xrightarrow{While} (StartA) (StartA) \xrightarrow{A.event\_name} (EndA)$   
 $(EndA) \xrightarrow{\epsilon} (Start\_while) (Start\_while) \xrightarrow{While\_end} (End\_while)$

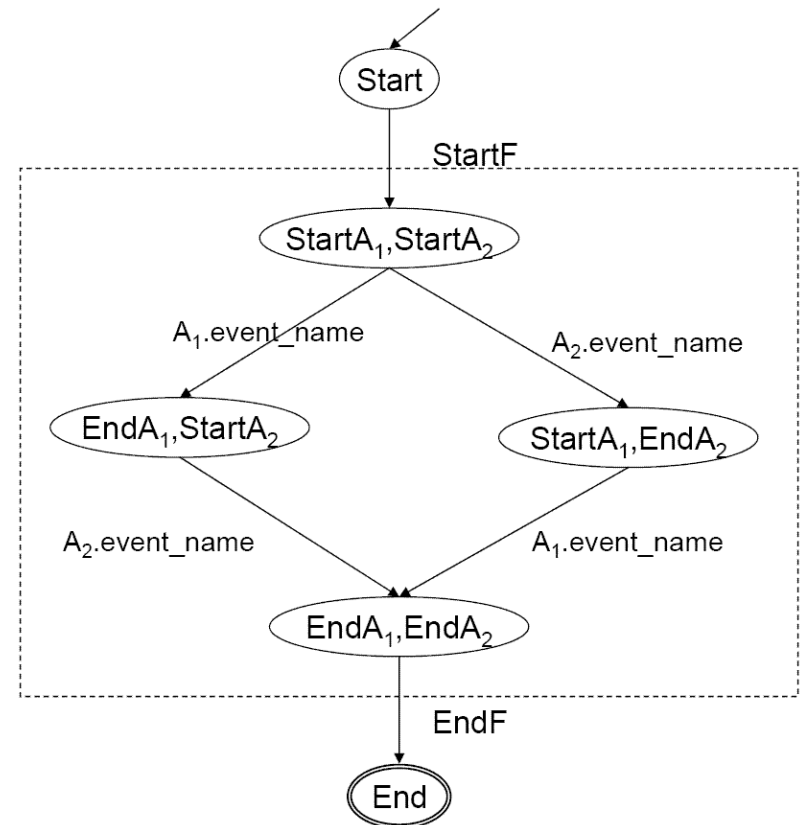
Rules:

- $(stateVar = Start\_while \wedge pre(W)) \xrightarrow{While} (stateVar = StartA)$
- $(stateVar = EndA) \xrightarrow{\epsilon} (stateVar = Start\_while)$
- $(stateVar = Start\_while \wedge \neg pre(W)) \xrightarrow{While\_end} (stateVar = End\_while)$

# Model synchronization links

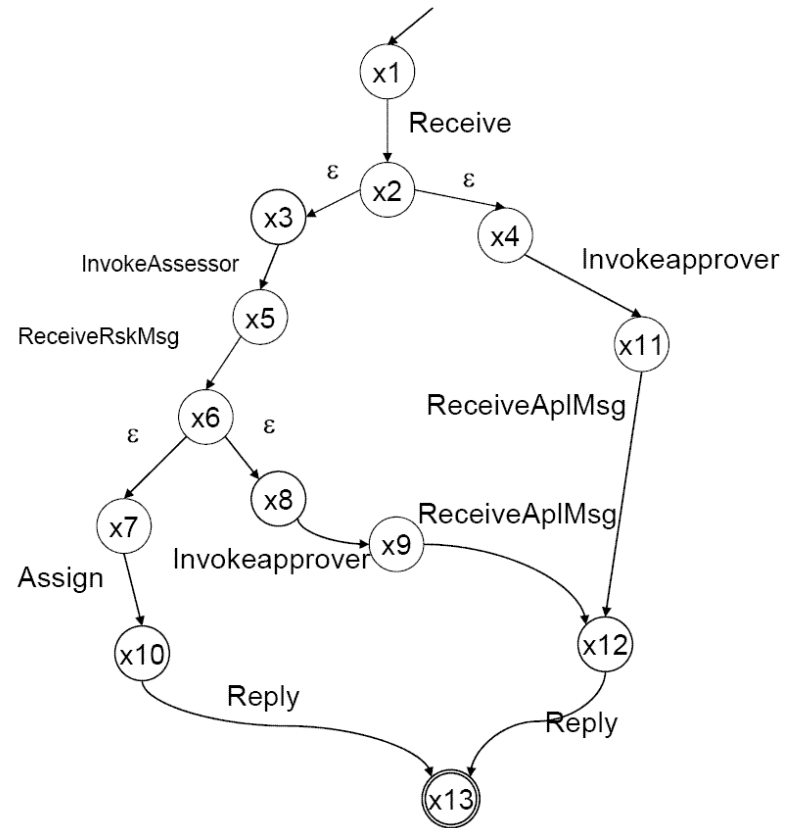


**(a) concurrency branches**



**(b) the joint DES model**

# DES model of the loan approval process



---

## $\langle \text{receive1} \rangle$ in the loan approval process

$\langle \text{receive1} \rangle = \text{Receive}(\{\text{Receive}, \epsilon\}, \text{soapMsg}, \text{request}, \text{stateVar} \in \{\text{Start\_receive}, \text{Post\_receive}, \text{InvokeApprover}, \text{InvokeAssessor}\})$

Transition rules:

- $(\text{stateVar} = \text{Start\_receive} \wedge \text{soapMsg.type} = \text{creditInformationMessage})$   
 $\xrightarrow{\text{Receive}} (\text{request} = \text{soapMsg} \wedge \text{stateVar} = \text{Post\_receive})$
- $(\text{stateVar} = \text{Post\_receive} \wedge \text{request.amount} \geq 1000) \xrightarrow{\epsilon} (\text{stateVar} = \text{InvokeApprover})$
- $(\text{stateVar} = \text{Post\_receive} \wedge \text{request.amount} < 1000) \xrightarrow{\epsilon} (\text{stateVar} = \text{InvokeAssessor})$

---

## $\langle \text{invokeAssessor} \rangle$ in the loan approval process

$\langle \text{invokeAssessor} \rangle = \text{Invoke}(\{\text{InvokeAssessor}, \text{ReceivedRskMsg}, \epsilon\}, \text{request}, \text{risk}, \text{stateVar} \in \{\text{InvokeAssessor}, \text{Wait\_assessor}, \text{Post\_invokeAssessor}, \text{RiskLow}, \text{RiskHigh}\})$

Transition rules:

- $(\text{stateVar} = \text{InvokeAssessor} \wedge \text{exist}(\text{request})) \xrightarrow{\text{InvokeAssessor}} (\text{stateVar} = \text{Wait\_Assessor})$
- $(\text{stateVar} = \text{Wait\_assessor}) \xrightarrow{\text{ReceiveMsg}} (\text{risk} = \text{riskAssessMessage} \wedge \text{stateVar} = \text{Post\_invokeAssessor})$
- $(\text{stateVar} = \text{Post\_invokeAssessor} \wedge \text{risk.level} = \text{high}) \xrightarrow{\epsilon} (\text{stateVar} = \text{RiskHigh})$
- $(\text{stateVar} = \text{Post\_invokeAssessor} \wedge \text{risk.level} = \text{low}) \xrightarrow{\epsilon} (\text{stateVar} = \text{RiskLow})$

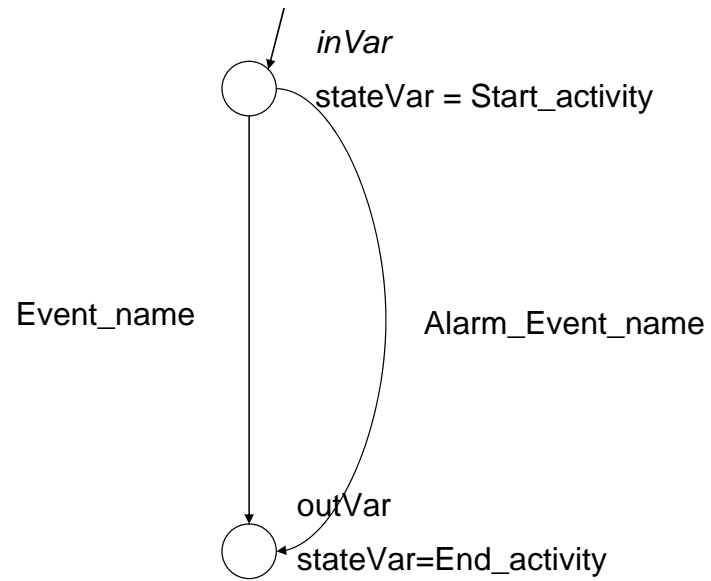
---

# Monitoring Business Process

- We deal with only faults that generate exceptions in this paper (semantic faults are not considered)
- We assume the BPEL engine can record all the messages which indeed is not difficult to implement.
- We trace all the invoked Web services from the record of messages and analyze the variable dependency among the Web services
- We can deduce the Web services responsible for the faults

---

# Model faulty behavior



---

## Reasoning Rules

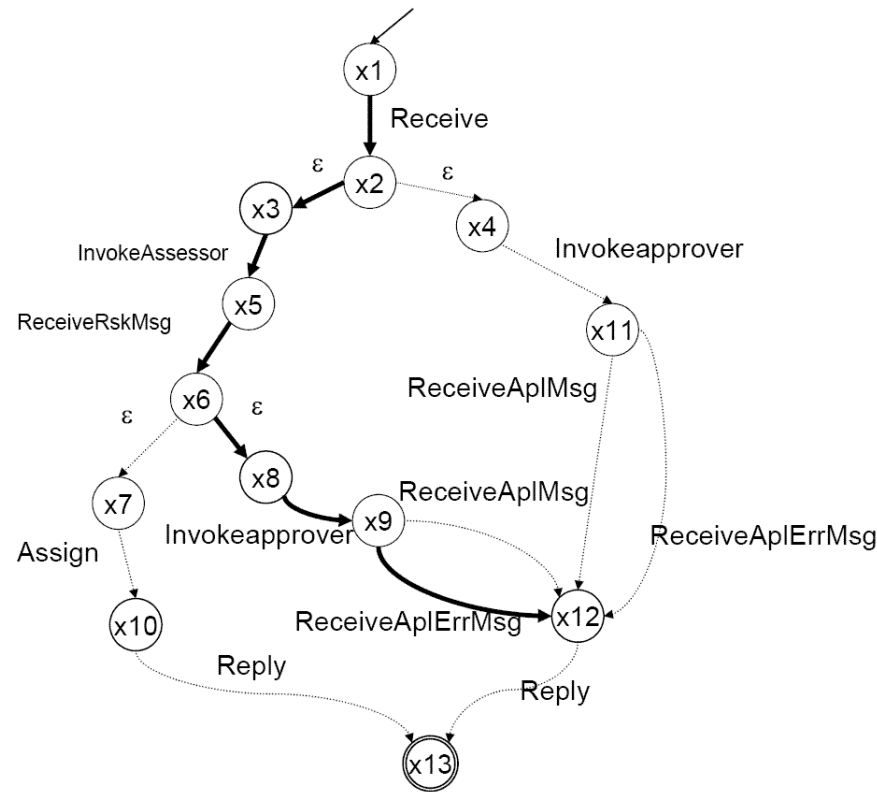
- if A generates alarm, A is faulty or its input  $inVar$  are abnormal  
 $alarm \in \{A.event\} \vdash faulty(A) \vee ab(A.inVar)$
- all the activities  $\{A_i\}$  which generate or change  $A.inVar$  are possibly faulty  
 $ab(A.inVar) \vdash \{faulty(A_i) \vee ab(A_i.inVar) | A_i.outVar = A.inVar\}$
- Propagation Rule  
 $ab(A_i.inVar) \vdash \{faulty(A_j) \vee ab(A_j.inVar) | A_j.outVar = A_i.inVar\}$

---

# Diagnoses

- diagnoses  $\Delta = \{A, A_i \mid \textit{faulty}(A_i) \vee \textit{faulty}(A)\}$

# Trajectory Reconstruction from event sequence



event sequence:  $\{Receive, InvokeAssessor, ReceiveRskMsg, InvokeApprover, ReceiveAplErrMsg\}$

---

# Diagnoses based on trajectory and variable dependency

trajectory

$$\begin{aligned} & (X1) \xrightarrow{Receive} (X2) \xrightarrow{\epsilon} (X3) \xrightarrow{InvokeAssessor} (X5) \dots \\ & \dots (X5) \xrightarrow{ReceiveRskMsg} (X6) \xrightarrow{\epsilon} (X8) \dots \\ & (X8) \xrightarrow{InvokeApprover} (X9) \xrightarrow{ReceiveAplErrMsg} (X12) \end{aligned}$$

from the trajectory, the model of process and reasoning rules  $\vdash$

diagnoses  $\Delta = \{receive1, invokeApprover\}$

---

# Conclusions

- We propose a formal method to model Web service network
- We propose a method to monitor the execution of Web service process and diagnose faults when exceptions are detected.
- Future directions:
  - new diagnosis techniques for Web service networks: e.g. distributed diagnosis and incremental diagnosis
  - reconfiguration vs. planning
  - recovery actions