Monitoring Web Service Networks in a Model-based Approach

Yuhong Yan
National Research Council,
46 Dineen Drive,
Fredericton,
NB E3B 5X9, Canada
yuhong.yan@nrc.gc.ca

Yannick Pencolé
Computer Sciences Laboratory
& National ICT Australia
The Australian National University
Canberra, ACT 0200, Australia
Yannick.Pencole@anu.edu.au

Marie-Odile Cordier
IRISA, Campus de Beaulieu
35042 Rennes Cedex, France
cordier@irisa.fr

Alban Grastien
IRISA, Campus de Beaulieu
35042 Rennes Cedex, France
agrastie@irisa.fr

Abstract

The goal of Web service effort is to achieve universal interoperability between applications by using Web standards: this emergent technology is a promising way to integrate business applications. A business process can then be seen as a set of Web services that could belong to different companies and interact with each other by sending messages. In that context, neither a global model nor a global mechanism are available to monitor and trace faults when the business process fails. In this paper, we address this issue and propose to use model-based reasoning approaches on Discrete-Event Systems (DES). This paper presents an automatic method to model Web service behaviors and their interactions as a set of synchronized discrete-event systems. This modeling is the first step before tracing the evolution of the business process and diagnosing business process faults.

1. Introduction

With Web service technology, one can see the world from a service-oriented point of view. Services are provided by software components over the internet. They are invoked by sending XML-based Simple Object Access Protocol (SOAP) messages to the remote components. Web services rely on internet protocols, such as HTTP, BEEP and XML technology to ensure the interoperability of the components on different platforms and are implemented in different programming languages. W3C accepts the following standards: Simple Object Access Protocol (SOAP), a message-based communication for component interaction [21]; Web Service Description Language (WSDL) for component interface definition [22]; and Universal Description, Discovery Integration (UDDI) for service discovery and integration [15].

Web service technology provides the possibility to integrate business applications and connect business processes across company boundaries. A business process can then be composed of individual Web services that belong to different companies: in other words, a business process is a network of Web services without any global supervision system. Currently, Business Process Execution Language for Web Service (BPEL4WS, denoted BPEL in the following) [12] is the de facto standard to describe the interactions of the individual Web service in both abstract and executable ways.

Like any other system, a business process can fail. In a distributed business environment, it is important to trace faults and recover from their effects. To solve this problem, we propose to develop methods to monitor and diagnose Web service networks, under the condition that only partial behaviors of the network are observable.

Our proposal is based on the fact that the existing Model-Based Diagnosis (MBD) techniques in Artificial Intelligence provide ways to monitor and diagnose static and dynamic systems using partial observations. To use any MBD techniques, a deep-knowledge model is required, i.e. a model that describes the basic behavior of the system. In this paper, we propose to extract business process models from BPEL descriptions and generate a formal DES model that is generally used in the MBD community. We present the methodology to transform the BPEL and WSDL descriptions into a DES model. Then, using the generated

models and the runtime observations, we can apply existing techniques to reconstruct the necessary and unobservable behaviors of the Web services that have been invoked during a business process. This model generation is the first step to achieve our ultimate goal that is to provide fault diagnoses in a business process.

This paper is organized as follows. Section 2 presents an MBD background and motivates the use of those techniques for Web services monitoring. Section 3 formally defines the way to generate a DES model from a BPEL description. Section 4 describes a complete example and Section 5 explains how MBD techniques can be applied to the model we propose and finally, Section 6 presents related work.

2. Background and Motivations

2.1. The Motivation to Use Model-based Diagnosis

MBD is used to monitor and diagnose both static and dynamic systems. The system behavior is modeled symbolically. A diagnosis is performed in order to explain observations in case of a discrepancy between the partial observed behavior of the system and the prediction given by the model. The early results in MBD are collected in [11]. In the following, we focus on a classical model type: Discrete-Event System. DES is a kind of qualitative description of a dynamic system whose behavior is event-driven.

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

- X *is a finite set of states;*
- Σ *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.*

[20] and [4] are fundamental works about DES diagnosis. Since it covers a wide range of systems, both AI and Automatic Control communities are interested in this topic and several recent advances have been made: the decentralized diagnoser approach [16] (a diagnosis system based on several interacting DESs), the incremental diagnosis approach [8] (a monitoring system that online updates diagnosis over time given new observations), active system approaches [2] (approaches that deal with hierarchical and asynchronous DESs), and diagnosis on reconfigurable systems [7].

DES is suitable to model the behavior of a business process since it is composed of Web services which are decentralized and dynamic. The interactions between Web services can be modeled by a synchronized composition of several local models. Consequently, the existing reasoning techniques on decentralized DES and incremental diagnosis can be easily applied to Web services application. The existing techniques, like the decentralized diagnoser approach [17] or the approaches for the diagnosis of active systems [2], reconstruct the unobservable behaviors of the system that are required to compute fault diagnoses.

In order to achieve our ultimate goal, that is to develop a monitoring system for business processes and Web services that is capable of performing fault diagnoses and making the business process recover from the fault effects, the generation of a deep-knowledge model for business processes is the first step. In this paper, we work on the method to build a deep-knowledge model of the business process behavior, more specifically, to transform the behavior description written in BPEL and WSDL into a formal DES.

2.2. Description of the Behavior of Business Processes

BPEL is a standard, recognized by OASIS, that is proposed by IBM and Microsoft along with several other companies to model business processes for Web services [12]. BPEL defines a grammar for describing the behavior of a business process that is based on the interactions between the process instance and its partners. The interactions with each partner occur through Web service interfaces. BPEL is layered on top of several XML specifications: WSDL1.1, XML Schema 1.0, and XPath1.0. WSDL messages and XML Schema type definitions provide the data model for BPEL, XPath provides support for data manipulation, and all external resources/partners are represented by WSDL services. The IBM BPEL4J engine can load BPEL files and invoke individual Web services according to the business processes that are defined in those files.

A BPEL business process is composed of activities. Fifteen activity types are defined, some of them are *basic activities* and the other ones are *structured activities*. Among the basic activities, the most important ones are the following:

1. the ⟨receive⟩ activity is for accepting the triggering message from another Web service;
2. the ⟨reply⟩ activity is for returning the response to its requestor;
3. the ⟨invoke⟩ activity is for invoking another Web service.

The structured activities define the execution orders of the activities inside their scopes. For example:

1. the ⟨sequence⟩ activity defines the sequential order of the activities inside its scope;

2. the ⟨flow⟩ activity defines the concurrent relations of the activities inside its scope.

Execution orders are also modified by defining the synchronization links between two activities (cf. section 3.3).

BPEL does not define how an activity is implemented. Normally BPEL has one entry point to start the process and one point to exit, though multiple entry points are allowed. The variables in BPEL are actually the SOAP messages defined in WSDL. Therefore the variables in BPEL are objects that have several attributes (called *parts* in WSDL). The behaviors of a business process are defined in BPEL and its related WSDL files.

2.3. Example

The loan approval process is an example described in the BPEL specification [12]. It is diagramed in Figure 1. It contains five activities (big shaded blocks). An activity involves a set of input and output variables (dotted box besides each activity). The edges show the execution order of the activities. When two edges are from the same activity, there are conditional (the condition expression is shown on the edge). In this example, the process starts from a ⟨receive⟩ activity called $receive1$. When a request message arrives, the process is triggered. $receive1$ dispatches the request to two ⟨invoke⟩ activities, $invokeAssessor$ and $invokeApprover$, depending on the amount of the loan. If the amount is small ($<$1000), $invokeAssessor$ is called and provides the risk assessment of the loan request. If the risk level is low, then a reply is prepared by an ⟨assign⟩ activity and later sent out by a ⟨reply⟩ activity. If the risk level is not low, $invokeApprover$ is invoked and provides the final decision. The result from $invokeApprover$ is then sent to the client by the ⟨reply⟩ activity.

A BPEL process can be wrapped as a Web service. For example, in the IBM BPEL4J package, which contains the above example, the loan approval process is only one Web service. Its interface is defined in a WSDL file. A client sends a SOAP message to it for the invocation of the business process. In this case, BPEL is the behavior model of a Web service.

3. Modeling Web Services with Discrete-Event Systems

A business process defined in BPEL is a composition of activities. Its model is defined as follows:

Definition 2 *The model of a business process is a tuple* $(V, \mathcal{D}, R)$ *where:*

- V *is a finite set of variables;*

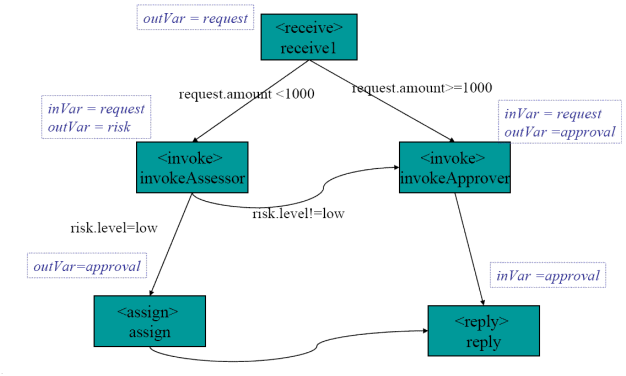


Figure 1. A loan approval process. Activities are represented in shaded boxes. The $inVar$ and $outVar$ are respectively the input and output variables of an activity.

- $\mathcal{D}$ *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).*

Proposition 1 *The model of a business process is a finite discrete-event system.*

This proposition is quite obvious.

In order to model a business process, we need to model each of its activities and the execution order between the activities using variables and rules. In the following subsections, we enumerate the formal model for each BPEL activity type.

3.1. Model of activities

Seven activities in BPEL are basic activities that do not nest other activities. They are the basic building blocks for business processes. Each activity can be translated into the DES formalism as one or several transitions. Each activity type has its own transition rules. This modeling method is inspired by the *tiles* from [5], and follows the extended formation from [9]. $\mathcal{D}$ is a finite variable domain. The empty value, denoted $\emptyset$, is contained in $\mathcal{D}$. Any variable has a domain $\mathcal{D}$. An activity is formally modeled below.

Definition 3 *An activity in a business process can be formally modeled as a transition rule. It transits the system from an initial state* $Start\_activity$ *to an end state* $End\_activity$*.* $inVar$ *and* $outVar$ *are the variables in* V

that are involved in the transition rule. The transition is labeled by an associated $Event\_name$.

⟨activity⟩

State variables: $inVar \in V$, $outVar \in V$, $stateVar = \{Start\_activity, End\_activity\} \in V$

Events: $Event\_name$

Transition rule:

- $(pre(inVar) \wedge stateVar = Start\_activity) \xrightarrow{Event\_name} (post(outVar) \wedge stateVar = End\_activity)$

$Start\_activity$, $End\_activity$, $Event\_name$ can be any strings that are unique to the process. They can contain the ID of the business process instance, if more than one instance are running. For simplicity, we use the below expression to represent an activity with all its states, events and the transition rule.

$$Activity(Event\_name, inVar, outVar, stateVar)$$

Or we can simply use the following notation called an automaton transition (but a state has to satisfy the transition rules in order to trigger the activity):

$$(Start\_activity) \xrightarrow{Event\_name} (End\_activity)$$

Sometimes the definition of the internal behavior of an activity is required. We enrich Definition 3 with internal states and chained transition rules.

Definition 4 *An activity with internal states $\{internalST_i, i \in \{1, \ldots, n\}\}$ and chained transitions rules is described as follows:*

Activity ⟨activity⟩

State variables: $inVar \in V$, $outVar \in V$, $stateVar = \{Start\_activity,\ End\_activity,\ InternalSTi, i \in \{1, \ldots, n\}\} \in V$

Events: $\{Start, End, Event_i, i \in \{1, \ldots, n-1\}\}$

Transition Rules:

- $(pre(inVar) \wedge stateVar = Start\_activity) \xrightarrow{Start} (stateVar = InternalST_1)$
- $(stateVar = InternalST_i) \xrightarrow{Event_i} (stateVar = InternalST_{i+1})$
- $(stateVar = InternalST_n) \xrightarrow{End} (post(outVar) \wedge stateVar = End\_activity)$

For short, it can be denoted:

$Activity(\{Start, End, Event_i\}, inVar, outVar, stateVar)$.

3.2. Modeling basic activities

In the following, we enumerate the model for each basic activity.

Activity ⟨receive⟩

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

Internal variable: $msgType \subseteq String$

Events: $Receive$

Rules:

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

$msgType$ is a predefined message type. If the incoming message has the predefined type, ⟨receive⟩ will initialize $received$.

Activity ⟨reply⟩

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

Events: $Reply$

Rules:

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

$exist(v)$ is the predicate checking that v is initialized.

Activity ⟨invoke⟩

State variables: $inVar$, $outVar$, $stateVar = \{Start\_invoke, End\_invoke, Wait\}$

Events: $Invoke, Receive$

Rules: Synchronous invocation

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

Rules: Asynchronous invocation

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

A synchronous invocation requires both an input variable and an output variable. An asynchronous invocation requires only one input variable because it does not expect a response as part of the operation.

Activity ⟨assign⟩

State variables: $inVar$, $outVar$, $stateVar = \{Start\_assign, End\_assign\}$

Events: $Assign$

Rules:

- $(stateVar = Start\_assign \wedge exist(inVar)) \xrightarrow{Assign} (stateVar = End\_assign \wedge outVar = inVar)$

Activity ⟨throw⟩

State variables: a structured variable *fault* such that $fault.mode \in \{On, Off\}$, $stateVar = \{Start\_throw, End\_throw\}$

Events: $Throw(fault)$

Rules:

- $(stateVar = Start\_throw \wedge fault.mode = Off) \xrightarrow{Throw(fault)} (stateVar = End\_throw \wedge fault.mode = On)$

Activity ⟨wait⟩

State variables: $stateVar = \{Start\_wait, End\_wait\}$

Internal variable: $wait\_mode \in \{On, Off\}$

Events: $Wait, End\_wait$

Rules:

- $(stateVar = Start\_wait \wedge wait\_mode = Off) \xrightarrow{Wait} (wait\_mode = On)$
- $(wait\_mode = On) \xrightarrow{End\_wait} (stateVar = End\_wait \wedge wait\_mode = Off)$

This model is not temporal. We do not consider time, so the notion of delay is not considered in this activity.

Activity ⟨empty⟩

State variables: $stateVar = \{Start\_empty, End\_empty\}$

Events: $Empty$

Rules:

- $(stateVar = Start\_empty) \xrightarrow{Empty} (stateVar = End\_empty)$

3.3. Modeling Structured Activities

Structured activities prescribe the order in which a collection of activities takes place. They describe how a business process is created by composing the basic activities into structures that express the control patterns and data flow. The structured activities of BPEL include:

- Ordinary sequential control between activities is provided by ⟨sequence⟩ , ⟨switch⟩ , and ⟨while⟩ .
- Concurrency and synchronization between activities are provided by ⟨flow⟩ .
- Nondeterministic choice based on external events is provided by ⟨pick⟩ .

Structured activities are modeled by the combination of transition rules that express the behavior of every nested activity and transition rules that express the execution order of those nested activities. In the following, we describe, for each structured activity, the rules that express the execution order. A representation of these rules as an automaton is also described.

Sequence
A ⟨sequence⟩ can nest n ⟨activity⟩ in its scope. The n activities $\{A_i\}$ will be executed in sequential order, if their triggering conditions are satisfied.

Activity ⟨sequence⟩

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

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

Automaton transitions:
$(Start\_sequence) \xrightarrow{Call(A_1)} (StartA_1)$
$(State\_A_1) \xrightarrow{A_1.event\_name} (EndA_1)$
$(EndA_1) \xrightarrow{Call(A_2)} (StartA_2)$
...
$(EndA_i) \xrightarrow{Call(A_{i+1})} (StartA_{i+1})$
...
$(EndA_n) \xrightarrow{End} (End\_sequence)$

Rules for transitions:

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

The transition rule $Call(A_i)$ does not change the values of the state variables except $stateVar$. The states of $EndA_i$ and $StartA_{i+1}$ share the same context. There is no ambiguity if the transition $Call(A_i)$ is abbreviated by connecting two activities $\langle \mathsf{A_i} \rangle$ and $\langle \mathsf{A_{i+1}} \rangle$ directly.

Switch

We assume a $\langle$switch$\rangle$ has n *'case'* branches corresponding to the n activities $\{A_1, \ldots, A_n\}$ and one *'otherwise'* branch corresponding to the activity A_{n+1}. A_i transforms the state $stateA_i$ to the state $EndA_i$ (see Figure 2).

Activity $\langle$switch$\rangle$

State variables: $V_1, \ldots, V_n$ are variable sets on n *'case'* branches, $stateVar = \{Start\_switch, End\_switch, StartA_i, EndA_i, i \in \{1, \ldots, n+1\}\}$

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

Automaton 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 for transitions:

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

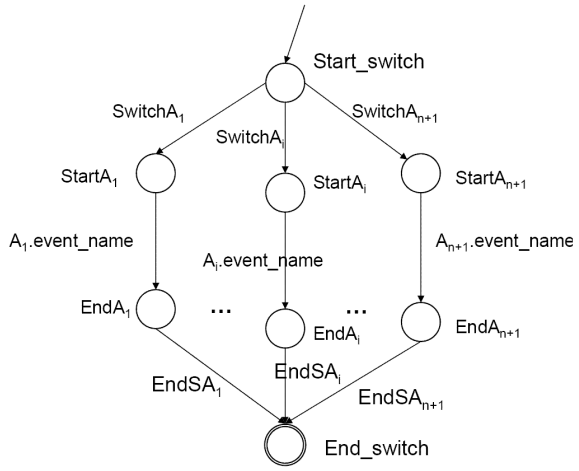


Figure 2. The automaton for $\langle$switch$\rangle$.

While

The activity $\langle$while$\rangle$ nests an activity A (see Figure 3).

Activity $\langle$while$\rangle$

State variables: $W \subseteq V$, $stateVar = \{Start\_while, End\_while, StartA, EndA\}$

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

Automaton 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 for transitions:

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

Flow

$\langle$flow$\rangle$ evaluates all the nested activities $\{A_1, \ldots, A_n\}$ and concurrently runs all triggered activities. Each nested activity A_i contains the input and output variables $\{inVar_i, outVar_i\}$.

Activity $\langle$flow$\rangle$

State variables: $\{inVar_i, outVar_i\}$ for activity $\{A_i\}$, $stateVar = \{Start\_flow, End\_flow\}$

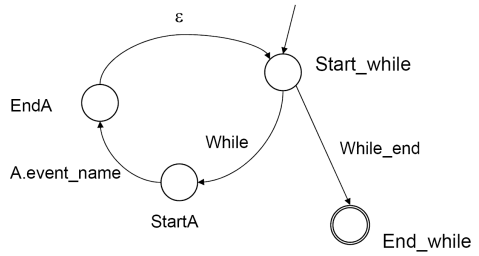


Figure 3. The automaton for ⟨while⟩ .

Internal Variables: $\{internalSTVar_i = \{StartA_i, EndA_i\}, i \in \{1, \ldots, n\}\}$

Events: $\{StartF, A_i.event\_name, EndF, i \in \{1, \ldots, n\}$

Automata transitions:
$(Start\_flow) \xrightarrow{StartF} (StartA_i)$
$(StartA_i) \xrightarrow{A_i.event\_name} (EndA_i)$
$(EndA_i) \xrightarrow{EndF} (End\_flow)$

Rules for transitions:

- $(stateVar = Start\_flow) \xrightarrow{StartF} (\bigwedge internalSTVar_i = StartA_i)$
- $(pre(V_i) \wedge internalSTVar_i = StartA_i) \xrightarrow{A_i.event\_name} (internalSTVar_i = EndA_i \wedge post(outVar_i))$
- $(\bigwedge internalSTVar_i = EndA_i) \xrightarrow{EndF} (stateVar = End\_flow)$

Notice that the semantic of a DES cannot model concurrency very well. So, we actually model the n paralleled branches into several DES pieces and define synchronization events to build their connections. The result of automata synchronization is an automaton defined as follows:

Definition 5 *The synchronized automaton of two automata* $A_1 = (X_1, \Sigma_1, T_1, I_1, F_1)$ *and* $A_2 = (X_2, \Sigma_2, T_2, I_2, F_2)$ *is the automaton* $A = (X, \Sigma, T, I, F)$ *such that:*

- $X = X_1 \times X_2$;
- $\Sigma = \Sigma_1 \cup \Sigma_2$;
- $T \subseteq X \times \Sigma \times X$;
- $I = I_1 \times I_2$;
- $F = F_1 \times F_2$.

Automata synchronization is illustrated in Figure 4. Above, each branch is modeled as an individual DES. The entry state $start$ and the end state end are duplicated for each branch. Events $startF$ and $endF$ are the synchronization events for the two DESs. Below is the joint DES for the concurrent branches. The reasoning on decentralized DES can be found in [17] and [16]. In general, the technique is matured enough to deal with concurrency.

Pick

Compared with ⟨switch⟩ , ⟨pick⟩ is represented by a non-deterministic automaton, i.e. the branch to follow is not predicable in advance. Activities $\{A_1, \ldots, A_n\}$ are corresponding to the n branches accordingly. A_i transforms the state $stateA_i$ to the state $endA_i$, whose transition rules are not included in the below definition.
Activity ⟨pick⟩

State Variables: $V_1, \ldots, V_n$ are variable sets used by the n branches, $stateVar = \{Start\_pick, End\_pick, StartA_i, EndA_i, i \in \{1, \ldots, n\}\}$

Events: $\{Pick, End, A_i.event\_name, i \in \{1, \ldots, n\}\}$

Automaton transitions:
$(Start\_pick) \xrightarrow{Pick} (StartA_i)$
$(StartA_i) \xrightarrow{A_i.event\_name} (EndA_i)$
$(EndA_i) \xrightarrow{EndPick} (End\_pick)$

Rules for transitions:

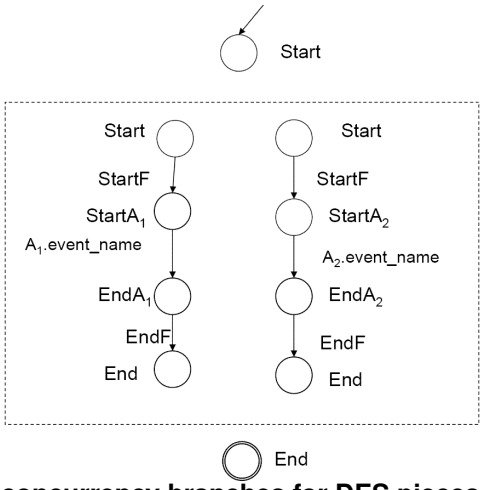
- $(stateVar = Start\_pick \wedge exist(V_i)) \xrightarrow{Pick} (stateVar = StartA_i)$,
- $(stateVar = EndA_i) \xrightarrow{EndPick} (stateVar = End\_pick)$

3.4. Synchronization Links of Activities

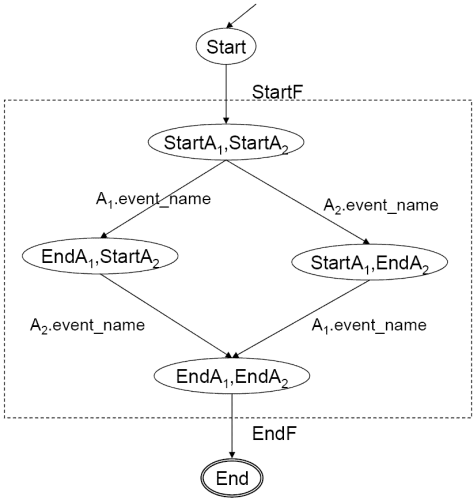
Each BPEL activity has optional nested standard elements ⟨source⟩ and ⟨target⟩ . A pair of ⟨source⟩ and ⟨target⟩ defines a link which connects two activities. The XML grammar is defined as below:

$$< source\ linkName = "ncname"$$
$$transitionCondition = "bool - expr"?/ >$$
$$< target\ linkName = "ncname"/ >$$

An activity may declare itself to be the source of one or more links by including one or more ⟨source⟩ elements. An activity may declare itself to be the target of one or more links by including one or more ⟨target⟩ elements. These



(a) concurrency branches for DES pieces



(b) the joint DES model

Figure 4. Build concurrency as synchronized DES pieces

elements are used for establishing additional sequential order and triggering conditions to the activity. The target activity must wait until the source activity finishes. The link can change the sequential order of activities. For example, if one ⟨flow⟩ contains two activities which are connected by a link, both activities become sequentially ordered. The use of links can express richer logic while causing the process more complex to analyze. For example, one activity can trigger a combination of several selective activities that could run in parallel. This relation can be expressed by DES. The activity containing a ⟨source⟩ with "*transitionCondition*", in addition to its original behaviors, behaves also like ⟨switch⟩ that leads to different activities depending on "*transitionCondition*" is satisfied or not. Formally:

Activity ⟨activity⟩

State variables: $inVar$, $outVar$, $condV$, $stateVar = \{Start\_activity, Post\_activity, End\_activity1, End\_activity2\}$

Events: $Event\_name$

Rules:

- $(pre(inVar) \wedge stateVar = Start\_activity) \xrightarrow{Event\_name} (post(outVar) \wedge stateVar = Post\_activity)$
- $(stateVar = Post\_activity \wedge transCondition(condV)) \xrightarrow{\epsilon} (stateVar = End\_activity_1)$
- $(stateVar = Post\_activity \wedge \neg transCondition(condV)) \xrightarrow{\epsilon} (stateVar = End\_activity_2)$

If the "*transitionCondition*" is empty, the activity model is the same as definition 3. When an activity contains many ⟨target⟩ elements, a join condition is used to specify requirements about concurrent paths reaching the activity. Each activity has optional standard attributes for this purpose: a name, a join condition, and an indicator whether a join fault should be suppressed if it occurs. The default value of *suppressJoinFailure* is no. The XML grammar is as below:

name="ncname"
joinCondition="bool-expr"
suppressJoinFailure="yes—no"

The *joinCondition* can be added as the precondition to trigger the activity. If the condition is not satisfied, the activity is bypassed. A fault is thrown if *suppressJoinFailure* is no. The treatment of *joinCondition* has to use synchronization of concurrent branches. This is not fully discussed in this paper.

4. A Complete Example

In this section, we present the complete DES model for the loan approval process. By using links, all the activities in the ⟨flow⟩ are sequential. For clearness reason, the event caused by ⟨flow⟩ is not shown. For simplicity, we just give the short expressions of the activities and their transition rules. The loan approval in DES is in Figure 5.

⟨receive1⟩ $= Receive(\{Receive, \epsilon\}, soapMsg, request, stateVar = \{Start\_receive, Post\_receive, InvokeApprover, InvokeAssessor\})$
Transition rules:

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

⟨invokeAssessor⟩ $= Invoke(\{InvokeAssessor, ReceivedRskMsg, \epsilon\}, request, risk, stateVar = \{InvokeAssessor, Wait\_assessor, Post\_invokeAssessor, RiskLow, RiskHigh\})$
Transition rules:

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

⟨assign⟩ $= Assign(\{Assign,-, approval, stateVar = \{RiskLow, End\_assign\})$ Transition rules:

- $(stateVar = RiskLow) \xrightarrow{Assign} (stateVar = End\_assign \wedge approval.accept = yes)$

⟨reply⟩ $= Reply(\{Reply,-, approval, stateVar = \{End\_approval, End\_assign, ReplyEnd\})$
Transition rules:

- $(stateVar \in \{End\_approval, End\_assign\} \wedge exist(approval)) \xrightarrow{Reply} (stateVar = ReplyEnd)$

⟨invokeApprover⟩ $= Invoke(\{InvokeApprover, ReceivedAplMsg\}, request, approval, stateVar = \{InvokeApprover, RiskHigh, Wait\_invokeApprover, End\_approval\})$
Transition rules:

- $(stateVar \in \{InvokeAssessor, RiskHigh\} \wedge exist(request)) \xrightarrow{InvokeApprover} (stateVar = Wait\_invokeApprover)$
- $(stateVar = Wait\_invokeApprover) \xrightarrow{ReceivedAplMsg} (approval = approvalMessage \wedge stateVar = End\_approval)$

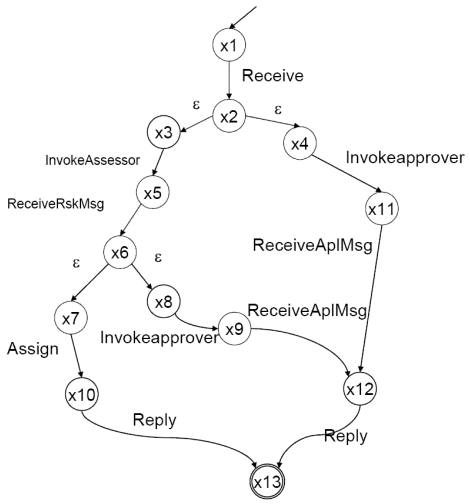


Figure 5. Model of the loan approval process

5. Monitoring Business Processes

We can use our knowledge on MBD for monitoring and diagnosing business processes. In MBD research, the monitoring task consists in deducing the unobserved behaviors from the partial observations and the normal system behavior model. If a discrepancy between the predictions from the normal system behavior model and the observations is detected, diagnostic techniques are then used to find the cause of this discrepancy (faults). A business process is

a dynamic system. We consider a business process is described in BPEL and runs inside a BPEL engine. It is impossible to keep snapshots of system evolution states due to memory or computational resource limitations. We can only record limited events and states when a business process is running. So, in the following analysis, we assume that the BPEL engine records the events when it executes a process. It is reasonable because BPEL engine knows the steps of its execution and this information does not occupy much memory. The fault handling in Web service basically relies on handling exceptions raised by invoked services. No attempt is made to identify the causes of faults. For MBD, the exceptions are alarms which are the symptoms of the faults. An activity which generates an alarm can be modeled as:

Definition 6 *State variables:* $inVar \in V$, $outVar \in V$ $stateVar = \{Start\_activity, End\_activity\}$ *Events:* $\{Event\_name, Alarm\_event\_name\}$ *Transition Rules:*

- $(pre(inVar) \wedge stateVar = Start\_activity) \xrightarrow{Event\_name} (post(outVar) \wedge stateVar = End\_activity)$
- $(pre(inVar) \wedge stateVar = Start\_activity \wedge fault.mode = On) \xrightarrow{Alarm\_event\_name} (stateVar = End\_activity)$

To diagnose is to find which Web services are responsible for the faults. Our method is to unfold the system evolution trajectory, which includes all the possible paths of events and system states that are consistent with the observation records. When observations are not complete, it is not a trivial problem to generate the trajectory [4, 20, 17]. Instead of discussing this problem in this paper, we assume that the BPEL engine records all the events in the system. Therefore trajectory generation is just a recovery from the log file. Assume that an activity A generates alarms, and $\{Ai\}$ is the set of activities involved in its trajectory. Then the fault diagnosis relies on the following insights:

$$alarm \in \{A.event\} \vdash faulty(A) \vee ab(A.inVar) \quad (1)$$

$$ab(A.inVar) \vdash \{faulty(A_i) \vee ab(A_i.inVar) | A_i.outVar = A.inVar\} \quad (2)$$

$$ab(A_i.inVar) \vdash \{faulty(A_j) \vee ab(A_j.inVar) | A_j.outVar = A_i.inVar\} \quad (3)$$

The first rule asserts that, if activity A generates an alarm, it is possible that activity A itself is faulty or its $inVar$ variables are abnormal. The second rule asserts that all the involved activities $\{A_i\}$ which generate $A.inVar$ or change $A.inVar$, are the candidates of the explanation of the alarm. Formula 3 expresses the propagation of the faulty behavior by checking the dependency of the variables. Then a fault diagnosis is a set of activities which are declared faulty.

$$\Delta = \{A, A_i | faulty(A_i) \wedge faulty(A)\}$$

In a business process, we can see a trajectory a sequence of involved activities. According to the diagnosis, some of them are affected by the faults. The fault handling should then undo all the affected activities. The following is a simple example to explain the diagnosis process.

BPEL engine records sequential events

$$\{Receive, InvokeAssessor, ReceiveRskMsg,$$

$$InvokeApprover, ReceiveAplErrMsg\}.$$

$ReceiveAplErrMsg$ is an alarm which informs that there is a type mismatch in the received parameters. We can build the evolution trajectory as follows, trajectory which is also illustrated in Figure 6.

$$(X1) \xrightarrow{Receive} (X2) \xrightarrow{\epsilon} (X3) \xrightarrow{InvokeAssessor} (X5) \cdots$$

$$\cdots (X5) \xrightarrow{ReceiveRskMsg} (X6) \xrightarrow{\epsilon} (X8) \cdots$$

$$(X8) \xrightarrow{InvokeApprover} (X9) \xrightarrow{ReceiveAplErrMsg} (X12)$$

We can easily deduce the dependency relation of the variables. We find that $request$ was used as input variable in activity $invokeAssessor$ but was not changed since it has been received. So the conclusion is either the Web service of $invokeApprover$ is wrong, or the activity $receive1$, which sends this message, is wrong.

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

Here we just give a very simple example about how the model can be used in monitoring and diagnosis. Existing tools can solve more complex problems, for example when several BPEL processes interact with each other in a decentralized system. This will be our future work.

6. Related work

Web services development reinforces the need of tools to improve their reliability. In this paper, we propose a model-based approach to develop a monitoring tool for Web services. The ultimate goal is to get self-healing Web services able to detect abnormal situations, to diagnose the primary faults and to recover from their effects. The closest work is [9] which is devoted to monitoring component-based software systems whose behavior is modeled using a formalism based on Petri nets. The main difference is that we rely on existing BPEL specifications and examine how to translate them into a transition rule formalism. The goal of [1] is also

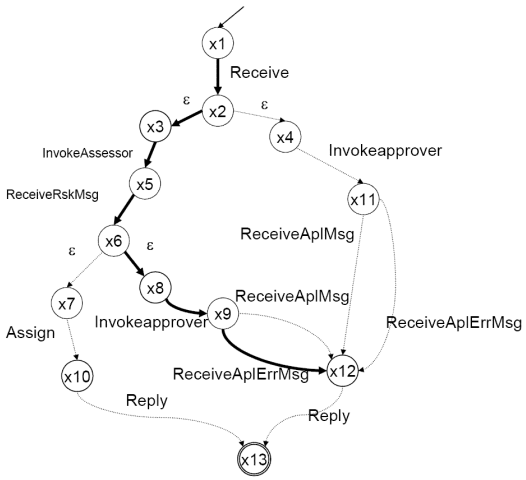


Figure 6. Loan approval example: evolution trajectory.

close to ours, in that they are currently developing a monitoring tool for Web services. They adopt grey-box models. This means that only the correlations between input and output parameters are described rather than the internal behavior of the activities. In our opinion, this abstract view is not sufficient when dealing with highly interacting components.

Literature about Web services monitoring is relatively small. Most of works related to Web services reasoning focus on two related but distinct problems. The first one is the automated composition of Web services to answer a specific request and decide which composition of available services can answer it. One of the proposed approaches is to use planning techniques on behavioral models as in [14, 18, 10, 3, 6]. For instance, [14] starts from DAML-S descriptions and automatically transforms them into Petri nets. Other works, such as [3, 13, 18], rely on transition rule systems. Like us, [18] proposes to build the behavioral models by automatically translating existing process descriptions, such as BPEL ones, into finite state machines. The second kind of problems is the property verification on Web services in order to guarantee that deployed applications satisfy a set of requirements and temporal properties (for instance, the absence of deadlocks). It is usually argued, for instance by [14, 19], that existing automated model-checking tools can support these tasks under the condition that components'behavior and their interactions are described by formal models. In this context, [19] proposes to use process algebras and shows that off-the-shelf tools based on process algebra are effective at verifying that Web services are well designed. Our proposal shares some similarities with this work in that we claim we can benefit from existing monitoring tools. Our proposal is to use distributed approaches that have been developed for telecommunication networks[16, 17]. This leads us to choose a transition rule formalism to model the components.

7. Conclusion

Web services is the emergent technology for business process integration. Existing formal methods provide reasoning tools for these applications. As seen before, depending on the kind of problems which are tackled, different modeling techniques are proposed to build formal models for Web services. In this paper, we aim at proposing a monitoring and diagnosis tool for Web services. The final goal is to give to these components the ability to detect abnormal situations, to identify the causes of these deficiencies, and lastly to decide recovering actions. We propose to start from existing process descriptions given in BPEL and to translate them in order to build a distributed behavioral system model. We examine each activity type and give its translation in term of transition rules. We start with basic activities (definition 3), show how more complex activities with internal states can be translated (definition 4), which enable to consider structure activities. Synchronization links correspond to synchronization of DES. In order to allow diagnosis capabilities, it is shown how alarm propagation can be modeled (definition 6). This modeling task is illustrated on the loan approval example. We argue that, giving the behavioral model, off-the-shelf tools can be used to monitor Web services. Moreover, it seems to us that the decentralized and incremental approach that we experimented on telecommunication networks is well-suited to this kind of systems.

Our method can be easily implemented with an open source BPEL engine to automatically build the model from BPEL specifications. A perspective on fault diagnosis is to augment this model with fault models, again starting from what can be described in BPEL. Lastly, it is important to check whether this abstract way of modeling the components is satisfying with respect to the scalability issue.

References

[1] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, and M. Segnan. Cooperative model-based diagnosis of web services. In *Proceedings of the 16th International Workshop on Principles of Diagnosis (DX-2005)*, pages 125–132, 2005.

[2] P. Baroni, G. Lamperti, P. Pogliano, and M. Zanella. Diagnosis of large active systems. *Artificial Intelligence*, 110(1):135–183, 1999.

[3] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automated composition of e-services that

export their behavior. In *Proceedings of the 1st Int. Conf. on Service-Oriented Computing (ICSOC'03)*, LNCS 2910, pages 43–58, 2003.

[4] M.-O. Cordier and S. Thiébaux. Event-based diagnosis for evolutive systems. In *Proceedings of the Fifth International workshop on Principles of diagnosis(DX'94)*, pages 64–69, 1994.

[5] E. Fabre, A. Aghasaryan, A. Benveniste, R. Boubour, and C. Jard. Fault detection and diagnosis in distributed systems: an approach by partially stochastic Petri nets. *Journal of Discrete Events Dynamic Systems*, 8:203–231, 1998.

[6] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE Int. Conf. on Automated Software Engineering (ASE'03)*, pages 152–161, 2003.

[7] A. Grastien, M.-O. Cordier, and C. Largouët. Extending decentralized discrete-event modelling to diagnose reconfigurable systems. In *Proceedings of the Fifteenth International Workshop on Principles of Diagnosis (DX-04)*, pages 75–80, Carcassonne, France, 2004.

[8] A. Grastien, M.-O. Cordier, and C. Largouët. Incremental diagnosis of discrete-event systems. In *Proceedings of the Sixteenth International Workshop on Principles of Diagnosis (DX-05)*, pages 119–124, Pacific Grove, California, USA, 2005.

[9] I. Grosclaude. Model-based monitoring of software components. In *Proceedings of the 16th European Conf. on Artificial Intelligence (ECAI'04)*, pages 1025–1026, 2004.

[10] R. Hamadi and B. Benatallah. A Petri net-based model for web service composition. In *Proceedings of the Fourteenth Australasian database conference on Database technologies (ADC'03)*, pages 191–200. Australian Computer Society, Inc., 2003.

[11] W. Hamscher, L. Console, and J. .de Kleer, editors. *Readings in model-based diagnosis*. Morgan Kaufmann Publishers Inc., 1992.

[12] IBM and et al. *Business process execution language for web services*, retrieved April 20, 2003. *ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf*.

[13] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. In *Proceedings of the 1st Int. Conf. on Service-Oriented Computing (ICSOC'03)*, LNCS 2910, pages 335–350, 2003.

[14] S. Narayanan and S. McIlraith. Simulation, verification and automated compostion of web services. In *Proceedings of the Eleventh International World Wide Web Conference (WWW-11)*, pages 77–88, 2002.

[15] OASIS. Uddi homepage, 2003, retrieved in 2004. *http://uddi.org/pubs/uddi_v3.htm*.

[16] Y. Pencolé and M.-O. Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence Journal*, 164(1-2):121–170, 2005.

[17] Y. Pencolé, M.-O. Cordier, and L. Rozé. Incremental decentralized diagnosis approach for the supervision of a telecommunication network. In *Proceedings of 41th IEEE Conf. on Decision and Control (CDC'2002)*, pages 435–440, Las Vegas, USA, 2002.

[18] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated composition of web services by planning at the knowledge level. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1252–1260, 2005.

[19] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *Proceedings of the Second IEEE Int. Conf. on Web Services (ICWS'04)*, pages 43–51, 2004.

[20] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

[21] W3C. SOAP specification, 2003, retrieved in 2004. *http://www.w3.org/TR/soap12-part1/*.

[22] W3C. WSDL specification, 2003, retrieved in 2004. *http://www.w3.org/TR/wsdl/*.