Monitoring and diagnosing orchestrated web service processes
Yan, Y.; Dague, P.

National Research Council Canada   Conseil national de recherches Canada

Canada

National Research
Council Canada

Conseil national
de recherches Canada

Institute for
Information Technology

Institut de technologie
de l'information

NRC·CNRC

*Monitoring and Diagnosing Ochestrated
Web Services Processes**

Yan, Y., and Dague, P.
2007

Canada

Modeling and Diagnosing Orchestrated Web Service Processes

Yuhong Yan[1], Philippe Dague[2],
[1]National Research Council, 46 Dineen Drive, Fredericton, NB E3B 5X9, Canada
[2]LRI, University of Paris Sud, CNRS, 91893 Orsay, France
Email: yuhong.yan@nrc.gc.ca

Abstract

Web service orchestration languages describe executable business processes composed of Web services. A business process can fail for many reasons, such as faulty Web services or mismatching messages. It is important to find out which Web services are responsible for a failed business process because we could penalize these Web services and exclude them from the business process in the future. In this paper, we propose a model-based approach to diagnose orchestrated Web service process. We convert the Web service orchestration language, BPEL4WS, into synchronized automata, so that we have a formal description of the topology and variable dependency of the business process. After an exception is thrown, the diagnoser can calculate the business process execution trajectory based on the formal model and the observed evolution of the business process. The faulty Web services are deduced from the variable dependency on the execution trajectory. We demonstrate our diagnosis technique with an example.

1 Introduction

Various Web service process description languages are designed by standard bodies and companies. Among them, Business Process Execution Language for Web Service (BPEL4WS, denoted as BPEL after) [1] is the de facto standard used to describe an executable Web service process. In this paper, we study the behaviours of a business process described in BPEL. As any other systems, a business process can fail. For a Web service process, the symptom of a failure is that exceptions are thrown and the process halts. As the process is composed of multiple Web services, it is important to find out which Web services are responsible for the failure. If we could diagnose the faulty Web services, we could penalize these Web services and exclude them from the business process in the future. The current throw-and-catch mechanism is very preliminary for diagnosing faults. It relies on the developer associating the faults with excep-

tions at design time. When an exception is thrown, we say certain faults occur. But this mechanism does not guarantee the soundness and the completeness of diagnosis.

In this paper, we propose a model-based approach to diagnose faults in Web service processes. We convert the basic BPEL activities and constructs into synchronized automata whose *states* are presented by the values of the *variables*. The process changes from one state to another by executing an *action*, e.g. assigning variables, receiving or emitting messages in BPEL. The emitting messages can be a triggering *event* for another service to take an action. The diagnosing mechanism is triggered when exceptions are thrown. Using the formal model and the runtime observations from the execution of the process, we can reconstruct the unobservable trajectories of the Web service process. Then the faulty Web services are deduced based on the variable dependency on the trajectories. Studying the fault diagnosis in Web service processes serves the ultimate goal of building self-manageable and self-healing business processes.

This paper is organized as follows: section 2 presents Model-based Diagnosis (MBD) background and motivates the use of those techniques for Web services monitoring and diagnosis; section 3 formally defines the way to generate an automata model from a BPEL description; section 4 extends the existing MBD techniques for Web service monitoring and diagnosis; section 5 is the related work, and section 6 is the conclusion.

2 The Principle of Model-based Diagnosis for Discrete Event Systems

MBD is used to monitor and diagnose both static and dynamic systems, such as communication systems, plant processes and automobiles. It is an active topic in both Artificial Intelligence (AI) and Control Theory communities [4]. Let us briefly recall the terminology and notations adopted by the model-based reasoning community.

- *System*: a pair $(SD, COMPS)$, where SD is sym-

bolic system description, e.g in first order logic, $COMPS$ is a finite set of constants to represent the components in the system.

- D: a mode assignment to each component in the system. An assignment to a component is a unary predicate: $ab(c_i)$ means $c_i \in COMPS$ is in an abnormal mode, and $\neg ab(c_i)$ means c_i working properly.
- *Observables*: the variables that can be observed/measured.
- OBS: a set of observations. They are the values of the *Observables*. They can be a finite set of first-order sentences, or value assignments to some variables.
- *Observed system*: $(SD, COMPS, OBS)$.

Diagnosis is a procedure to determine which components are correct and which components are faulty in order to be consistent with the observations and the system description.

Definition 1 *D is a consistency-based **diagnosis** for the observed system $\langle SD, COMPS, OBS \rangle$, if and only if it is a mode assignment and $SD \cup D \cup OBS \nvDash \bot$.*

From Definition 1, diagnosis is a mode assignment D that makes the union of SD, D and OBS logically consistency. D can be partitioned into two parts:

- D_{ok} which is a set of the components which are assigned to the $\neg ab$ mode;

- D_f which is a set of component which are assigned the ab mode.

Usually we are interested in those diagnoses which involve a minimal set of faults, i.e., the diagnoses for which D_f is minimal.

Definition 2 *A diagnosis D is **minimal** if and only if there is no other diagnosis D' for $\langle SD, COMPS, OBS \rangle$ such that $D'_f \subset D_f$.*

When the system description is in first order logic, the computation of diagnoses is rooted in automated reasoning [4].

When applying MBD, a formal system description is needed. As the interactions between Web services are driven by message passing, and message passing can be seen as discrete events, we consider the Discrete Event Systems (DES) suitable to model Web service processes. Many discrete event models, such as Petri nets, process algebras and automata, can be used for Web service process modelling. These models were invented for different purposes, but now they share many common techniques, such as symbolic representation (in addition to graph representation in some models) and similar symbolic operations. In this paper, we present a method to represent Web service processes

described in BPEL as automata in Section 3. Here we introduce some basic concepts and operations for automata. A classic definition of deterministic automaton is as below:

Definition 3 *An **automaton** Γ is a tuple $\Gamma = \langle X, \Sigma, T, I, F \rangle$ where:*

- *X is a finite set of states;*
- *Σ is a finite set of events;*
- *$T \subseteq X \times \Sigma \to X$ 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.*

Definition 4, 5 and 6 are some basic concepts and operations about automata.

Definition 4 *$Synchronization$ between two automata $\Gamma_1 = \langle X_1, \Sigma_1, T_1, I_1, F_1 \rangle$ and $\Gamma_2 = \langle X_2, \Sigma_2, T_2, I_2, F_2 \rangle$, with $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, produces an automaton $\Gamma = \Gamma_1 \| \Gamma_2$, where $\Gamma = \langle X_1 \times X_2, \Sigma_1 \cup \Sigma_2, T, I_1 \times I_2, F_1 \times F_2 \rangle$, with:*
$T((x_1, x_2), e) = (T_1(x_1, e), T_2(x_2, e))$, if $e \in \Sigma_1 \cap \Sigma_2$
$T((x_1, x_2), e) = (T_1(x_1, e), x_2)$, if $e \in \Sigma_1 \backslash \Sigma_2$
$T((x_1, x_2), e) = (x_1, T_2(x_2, e))$, if $e \in \Sigma_2 \backslash \Sigma_1$

Assume $s = \Sigma_1 \cap \Sigma_2$ is the joint event set of Γ_1 and Γ_2, Γ can also be written as $\Gamma = \Gamma_1 \|_s \Gamma_2$.

Example 1 *In Figure 1, Γ_1 and Γ_2 are two automata. The third one Γ_3 is produced by synchronizing Γ_1 and Γ_2.*
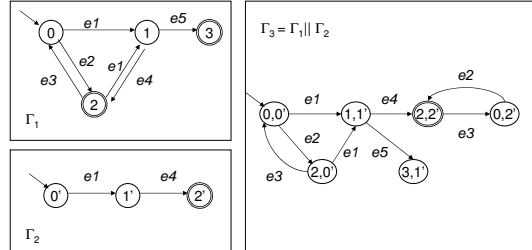
Figure 1. An example of synchronization

Definition 5 *A **trajectory** of an automaton is a path of contiguous states and transitions in the automaton that begins at an initial state and ends at a final state of the automaton.*

Example 2 *The trajectories in the automaton Γ_3 in Figure 1 can be represented as the two formulas below, in which $[\,]^*$ means the content in $[\,]$ repeated 0 or more times:*

$[(0, 0') \xrightarrow{e2} (2, 0') \xrightarrow{e3}]^* [(0, 0') \xrightarrow{e1} (1, 1') \xrightarrow{e4} (2, 2')] [\xrightarrow{e3} (0, 2') \xrightarrow{e2} (2, 2')]^*$,
$[(0, 0') \xrightarrow{e2} (2, 0') \xrightarrow{e3}]^* [(0, 0') \xrightarrow{e2} (2, 0') \xrightarrow{e1} (1, 1') \xrightarrow{e4} (2, 2')]$
$[\xrightarrow{e3} (0, 2') \xrightarrow{e2} (2, 2')]^*$.

Definition 6 *Concatenation between two automata* $\Gamma_1 = \langle X_1, \Sigma_1, T_1, I_1, F_1 \rangle$ *and* $\Gamma_2 = \langle X_2, \Sigma_2, T_2, I_2, F_2 \rangle$, *with* $\Sigma_1 \cap \Sigma_2 = \emptyset$ *and* $F_1 \cap I_2 \neq \emptyset$, *produces an automaton* $\Gamma = \Gamma_1 \circ \Gamma_2$, *where* $\Gamma = \langle X_1 \cup X_2, \Sigma_1 \cup \Sigma_2, T_1 \cup T_2, I_1, F_2 \cup (F_1 \backslash I_2) \rangle$.

3 Modeling Web Service Processes with Discrete-Event Systems

3.1 Description of the Web Service Processes

BPEL is an XML-based orchestration language developed by IBM and recognized by OASIS [1]. BPEL is a so-called executable language because it defines the internal behaviour of a Web service process, as compared to choreography languages that define only the interactions among the Web services and are not executable. BPEL defines fifteen activity types, among those the most important are the following:

$Process ::= Process(Activity_1, \ldots, Activity_n, V)$
$Activity ::= BasicActivity | StructuredActivity$
$BasicActivity ::= receive | invoke | reply | assign | throw |$
$\qquad\qquad\qquad terminate | compensate | wait | empty$
$StructuredActiviy ::= sequence | switch | flow | while |$
$\qquad\qquad\qquad pick | scope$

Example 3 *The loan approval process is an example described in the BPEL Specification 1.1 [1]. It is diagrammed in Figure 2.*
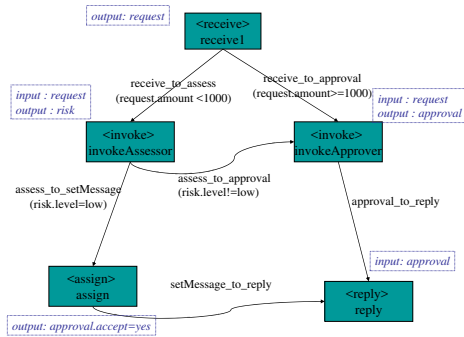

Figure 2. A loan approval process. Activities are represented in shaded boxes. The $inVar$ and $outVar$ are respectively the input and output variables of an activity.

This process contains five activities (big shaded blocks). An activity involves a set of input and output variables (dotted box besides each activity). All the variables are of composite type. The edges show the execution order of the activities. When two edges are issued from the same activity, only one edge that satisfies a triggering condition

(shown on the edge) will be activated. The process is triggered when a $\langle$receive$\rangle$ activity named $receive1$ receives a message of a predefined type. First, $receive1$ initializes a variable $request$. Then, $receive1$ dispatches the request to one of the two $\langle$invoke$\rangle$ activities, $invokeAssessor$ and $invokeApprover$, depending on the amount of the loan. In the case where the amount is large ($request.amount >= 1000$), $invokeApprover$ is called for a decision, otherwise ($request.amount < 1000$), $invokeAssessor$ is called for risk assessment. If $invokeAssessor$ returns with an assessment that the risk level is low ($risk.level = low$), a reply is prepared by an $\langle$assign$\rangle$ activity and later sent out by a $\langle$reply$\rangle$ activity, otherwise, $invokeApprover$ is invoked for a final decision. The result from $invokeApprover$ is sent to the client by the $\langle$reply$\rangle$ activity.

3.2 Modeling Web Services Process with Discrete-Event Systems

A Web service process defined in BPEL is a composition of activities. We are going to model a BPEL activity as an automaton. A BPEL state is associated with an assignment of the variables. A BPEL activity is triggered when its initial state satisfies a finite set of triggering conditions which is a certain assignment of variables. After an activity is executed, the values of the state variables are changed. We need to extend the classic automaton definition to include the operations on state variables.

Assume a BPEL process has a finite set of variables $V = \{v_1, \ldots, v_n\}$, and the domain $\mathcal{D} = \{D_1, \ldots, D_n\}$ for V is real values $\Re$ or arbitrary strings. $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints. A constraint c_j of some arity k is defined as a subset of the cartesian product over variables $\{v_{ji}, \ldots, v_{jk}\} \subseteq V$, i.e. $c_j \subseteq D_{j1} \times \cdots \times D_{jk}$, or a first order formula over $\{v_{ji}, \ldots, v_{jk}\}$. A constraint restricts the possible values of the k variables.

A BPEL state s is defined as an assignment of variables. A BPEL transition t is an operation on the state s_i, i.e., $(s_j, post(V_2)) = t(s_i, e, pre(V_1))$, where $V_1 \subseteq V$, $V_2 \subseteq V$, $pre(V_1) \subseteq \mathcal{C}$ is a set of preconditions that s_i has to satisfy and $post(V_2) \subseteq \mathcal{C}$ is a set of post-conditions that the successor state s_j will satisfy. If a state s satisfies a constraint c, we annotate as $c \wedge s$. Then, the semantics of transition t is also represented as: $t : (s_i \wedge pre(V_1)) \xrightarrow{e} (s_j \wedge post(V_2))$.

Definition 7 *A **BPEL activity** is an automaton* $\langle X, \Sigma, T, I, F, \mathcal{C} \rangle$, *where* $\mathcal{C}$ *is the constraint set on variables that define states X and $T : X \times \Sigma \times 2^{\mathcal{C}} \to X \times 2^{\mathcal{C}}$.*

3.2.1 Modeling Basic Activities

Due to lack of space, only major activities are included in this and the following subsection.

Activity ⟨receive⟩: $\langle\{s_o, s_f\}, \{received\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t : (s_o \wedge SoapMsg.type = MsgType)\xrightarrow{received}(s_f \wedge RecMsg = SoapMsg)$, where
$MsgType$ is a predefined message type. If the incoming message $SoapMsg$ has the predefined type, $RecMsg$ is initialized as $SoapMsg$.

Activity ⟨reply⟩ : $\langle\{s_o, s_f\}, \{replied\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t : (s_o \wedge exists(RepMsg))\xrightarrow{replied}(s_f \wedge SoapMsg = RepMsg)$, where
$exists(RepMsg)$ is the predicate checking that the replay message $RepMsg$ is initialized. $SoapMsg$ is the message on the wire.

Activity ⟨invoke⟩
Synchronous invocation (wait for a return message):
$\langle\{s_o, wait, s_f\}, \{invoked, received\}, \{t_1, t_2\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t_1 : (s_o \wedge exists(InVar))\xrightarrow{invoked}(wait)$, and
$t_2 : (wait)\xrightarrow{received}(s_f \wedge exists(OutVar))$ where
$InVar$ and $OutVar$ are the input and output variables.
Asynchronous invocation (not wait for a return message):
$\langle\{s_o, s_f\}, \{invoked\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t : (s_o \wedge exists(InVar))\xrightarrow{invoked}(s_f)$.

Activity ⟨assign⟩: $\langle\{s_o, s_f\}, \{assigned\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t : (s_o \wedge exists(InVar))\xrightarrow{assigned}(s_f \wedge OutVar = InVar)$

Activity ⟨throw⟩: $\langle\{s_o, s_f\}, \{thrown\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t : (s_o \wedge Fault.mode = Off)\xrightarrow{thrown}(s_f \wedge Fault.mode = On)$

Activity ⟨wait⟩:
$\langle\{s_o, wait, s_f\}, \{waiting, waited\}, \{t_1, t_2\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t_1 : (s_o \wedge Wait\_mode = Off)\xrightarrow{waiting}(wait \wedge Wait\_mode = On)$
$t_2 : (wait \wedge Wait\_mode = On)\xrightarrow{waited}(s_f \wedge Wait\_mode = Off)$
This model is not temporal. We do not consider time, so the notion of delay is not considered in this activity.

Activity ⟨empty⟩: $\langle\{s_o, s_f\}, \{empty\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t : (s_o)\xrightarrow{empty}(s_f)$

3.2.2 Modeling Structured Activities

Sequence A ⟨sequence⟩ can nest n activities ⟨A$_i$⟩ in its scope. These activities are executed in sequential order. Assume ⟨A$_i$⟩ : $\langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i}\rangle$, $i \in \{1, \ldots, n\}$.

Activity ⟨sequence⟩: $\langle\{s_o, s_f\} \cup \bigcup S_{A_i}, \{end\} \cup \bigcup\{callA_i\} \cup \bigcup\Sigma_{A_i}, \{t_i\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup\mathcal{C}_{A_i}\rangle$ with
$t_0 : (s_o)\xrightarrow{callA_1}(s_{A_{1o}})$

$t_i : (s_{A_{if}})\xrightarrow{callA_{i+1}}(s_{A_{i+1o}})$
$t_n : (s_{A_{nf}})\xrightarrow{end}(s_f)$
If assume $s_o = s_{A_{1o}}$, $s_f = s_{A_{nf}}$, and $s_{A_{if}} = s_{A_{i+1o}}$, for $i = [1, \ldots, n-1]$, a short representation of ⟨sequence⟩ is the concatenation of the nested activities $A_1 \circ A_2 \cdots \circ A_n$.

Switch Assume a ⟨switch⟩ has n ⟨case⟩ branches and one ⟨otherwise⟩ branch. Assume ⟨A$_i$⟩ : $\langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i}\rangle$, $i \in \{1, \ldots, n+1\}$.

Activity ⟨switch⟩: $\langle\{s_o, s_f\} \cup \bigcup S_{A_i}, \{end\} \cup \bigcup\{switchA_i\} \cup \bigcup\Sigma_{A_i}, \bigcup\{t_{io}\} \cup \bigcup\{t_{if}\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup\mathcal{C}_{A_i} \cup \bigcup pre(V_i)\rangle$.
Assume $V_1, \ldots, V_n$ are variable sets on n ⟨case⟩ branches. The transitions are defined as below:
$t_{io} : (s_o \wedge \neg pre(V_1) \wedge \cdots \wedge pre(V_i) \cdots \wedge \neg pre(V_n))\xrightarrow{switchA_i}(s_{A_{io}}), \forall i \in \{1, \ldots n\}$
$t_{(n+1)o}(s_o \wedge \neg pre(V_1) \wedge \cdots \wedge \neg pre(V_i) \cdots \wedge \neg pre(V_n))\xrightarrow{swicthA_{n+1}}(s_{A_{(n+1)o}})$
$t_{if} : (s_{A_{if}})\xrightarrow{end}(s_f), \forall i \in \{1, \ldots n+1\}$

While Assume ⟨while⟩ nests an activity ⟨A⟩: $\langle S_A, \Sigma_A, T_A, \{s_{A_o}\}, \{s_{A_f}\}, \mathcal{C}\rangle$.
Activity ⟨while⟩: $\langle\{s_o, s_f\} \cup S_A, \{while, while\_end\} \cup \Sigma_A, \{t_o, t_f, t\} \cup T_A, \{s_o\}, \{s_f\}, \mathcal{C} \cup pre(W)\rangle$.
Assume W is a variable set.
$t_o : (s_o \wedge pre(W))\xrightarrow{while}(s_{A_o})$
$t_f : (s_o \wedge \neg pre(W))\xrightarrow{while\_end}(s_f)$
$t : (s_{A_f})\xrightarrow{\epsilon}(s_o)$

Flow A ⟨flow⟩ can nest n activities ⟨A$_i$⟩ in its scope. These activities are executed concurrently. Assume ⟨A$_i$⟩ : $\langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i}\rangle$, $i \in \{1, \ldots, n\}$.

Activity ⟨flow⟩: $\langle\{s_o, s_f\} \cup \bigcup S_{A_i}, \{start, end\} \cup \bigcup\Sigma_{A_i}, \bigcup\{t_{io}, t_{if}\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup\mathcal{C}_{A_i}\rangle$ with
$t_{io} : (s_o)\xrightarrow{start}(s_{A_{io}})$
$t_{if} : (s_{A_{if}})\xrightarrow{end}(s_f)$

Notice that the semantic of automata cannot model concurrency. We actually model the n-paralleled branches into n automata and define synchronization events to build their connections. The principle is illustrated in Figure 3. Each branch is modeled as an individual automaton. The entry state s_o and the end state s_f are duplicated in each branch. Events $start$ and end are the synchronization events. More complicated case in joining the paralleled branches is discussed in subsection 3.2.3. The key point in reasoning about decentralized automata is to postpone the synchronization until a synthesis result is needed, in order to avoid the state explosion problem. In Web service diagnosis, it is the situation (cf. subsection 4.1).

Pick Assume a $\langle$pick$\rangle$ has n $\langle$onMessage$\rangle$ and one $\langle$onAlarm$\rangle$ branches. The correspondent branches are triggered by predefined events (for $\langle$onMessage$\rangle$) or by a time-out event produced by a timer (for $\langle$onAlarm$\rangle$). Assume $\langle$A$_i$$\rangle$: $\langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i}\rangle$, $i \in \{1, \ldots, n+1\}$.

Activity $\langle$pick$\rangle$: $\langle\{s_o, s_f\} \cup \bigcup S_{A_i}, \bigcup\{start_{A_i}\} \cup \{end\} \cup \bigcup \Sigma_{A_i}, \bigcup\{t_{io}, t_{if}\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup \mathcal{C}_{A_i} \cup \bigcup exists(event_{A_i})\rangle$ with

$t_{io} : (s_o \wedge exists(event_{A_i})) \xrightarrow{start_{A_i}} (s_{A_{io}})$
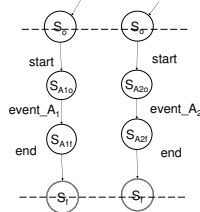
$t_{if} : (s_{A_{if}}) \xrightarrow{end} (s_f)$

Figure 3. Build concurrency as synchronized DES pieces.

3.2.3 Synchronization Links of Activities

Each BPEL activity can optionally nest the standard elements $\langle$source$\rangle$ and $\langle$target$\rangle$:

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

A pair of $\langle$source$\rangle$ and $\langle$target$\rangle$ defines a link which connects two activities. The target activity must wait until the source activity finishes. When one $\langle$flow$\rangle$ contains two parallel activities which are connected by a link, the two activities become sequentially ordered.

$\langle$source$\rangle$ can be modeled similarly like an $\langle$activity$\rangle$, with "*transitionCondition*" as the triggering condition.
Activity $\langle$source$\rangle$: $\langle\{s_o, s_f\}, \{\epsilon\}, \{t\}, \{s_o\}, \{s_f\},$
$transitionCondition\rangle$ with
$t : (s_o \wedge transitionCondition) \xrightarrow{\epsilon} (s_f),$

When an activity is the $\langle$target$\rangle$ of multiple links, a join condition is used to specify how these links can join. The join condition is defined within the activity. BPEL specification defines standard attributes for this activity:

<activityName="ncname", joinCondition="bool-expr", suppressJoinFailure="yes—no">

where joinCondition is the logical OR of the liveness status of all links that are targeted at this activity. If the condition is not satisfied, the activity is bypassed, and a fault is thrown if suppressJoinFailure is no.

In this case, the synchronization event end as in Figure 3 is removed. If the ending state of $\langle$flow$\rangle$ is the starting state s'_o of the next activity, the precondition of s'_o is the joinCondition. For example, either of the endings of the two

branches can trigger the next activity can be represented as: $s'_o \wedge (exists(s_{A_{1f}}) \vee exists(s_{A_{2f}}))$.

3.2.4 Modeling the Loan Approval Process

Example 4 *The loan approval process in Example 3 contains five activities:* $\langle$receive1$\rangle$, $\langle$invokeAssessor$\rangle$, $\langle$invokeApprover$\rangle$, $\langle$assign$\rangle$, $\langle$reply$\rangle$. *The five activities are contained in a* $\langle$flow$\rangle$. *Six links,* $\langle$receive_to_assess$\rangle$, $\langle$receive_to_approval$\rangle$, $\langle$assess_to_setMessage$\rangle$, $\langle$assess_to_approval$\rangle$, $\langle$approval_to_reply$\rangle$, *and* $\langle$setMessage_to_reply$\rangle$, *connect the activities and change the concurrent orders to sequential orders between the activities. In this special case, there are actually no concurrent activities. Therefore, for clarity, the event caused by* $\langle$flow$\rangle$ *is not shown. Assume the approver may return an error message due to an unknown error. The formal representation of the process is shown on Figure 4(b)without highlights (the formulas are eliminated due to lack of space.)*

4 Model-based Diagnosis for Web Service Processes

A Web service process can run down for many reasons. For example, a composed Web service may be faulty, an incoming message mismatches the interface, or the Internet is down. The *symptom*[1] of a failed Web service process is that *exceptions* are thrown and the process is halted. The current fault handling mechanism is throw-and-catch, similar to programming languages. The *exceptions* are thrown at the places where the process cannot be executed. The *catch* clauses process the exceptions, normally to recover the failure effects by executing predefined actions.

The throw-and-catch mechanism is very preliminary for fault diagnosis. The exception reports where it happened and returns some fault information. The exceptions can be regarded as associated with certain faults. When an exception is thrown, we deduce that its associated fault occurred. This kind of association relations rely on the empirical knowledge of the developer. It may not be a real cause of the exceptions. In addition, there may exist multiple causes of an exception which are unknown to the developer. Therefore, the current throw-and-catch mechanism does not provide sound and complete diagnosis. For example, when a Web service throws an exception about a value in a customer order, not only the one that throws the exception may be faulty, but the one that generates these data may

---

[1]In diagnosis concept, *sympton* is an observed abnormal behaviour, while *fault* is the original cause of a sympton. For example, an alarm from a smoke detector is a symptom. The two possible faults, a fire or a faulty smoke detector, are the causes of the symptom.

also be faulty. But a Web service exception can only report the Web service where the exception happens with no way to know who generated these data. In addition, all the services that modified the data should be also suspected. Not all of this kind of reasoning is included in the current fault handling mechanism.

The diagnosis task is to determine the Web services responsible for the exceptions. These Web services will be diagnosed faulty. The exceptions come from the BPEL engine or the infrastructure below. We classify the exceptions into **time-out** exceptions and **business logic** exceptions.

The *time-out* exceptions are due to either a disrupted network or unavailable Web services. If there is a lack of response, we cannot distinguish whether the fault is in the network or at the remote Web service. Since we cannot diagnose which kind of faults prevent a Web service from responding, we can do little with *time-out* exceptions.

The *business logic* exceptions occur while invoking an external Web service and executing BPEL internal activities. For example, mismatching messages (including the type of parameters and the number of parameters mismatching) cause the exceptions to be thrown when the parameters are passed to the remote method. BPEL can throw exceptions indicating the input data is wrong. During execution, the remote service may stop if it cannot process the request. The most common scenarios are the invalid format of the parameters, e.g. the data is not in a valid format, and the data is out of the range. The causes of the exceptions are various and cannot be enumerated. The common thread is that a business logic exception brings back information on the variables that cause the problem. In this paper, our major effort is on diagnosing business logic-related exceptions.

So in our framework, $COMPS$ is made up of all the basic activities of the Web service process considered, and OBS is made up of the exceptions thrown and the events of the executed activities. These events can be obtained by the monitoring function of a BPEL engine. A typical correct model for an activity $\langle A \rangle$ is thus:

$$\neg ab(A) \wedge \neg ab(A.inputs) \Longrightarrow \neg ab(A.outputs) \quad (1)$$

For facilitating diagnosis, the BPEL engine has to be extended for the following tasks: 1) record the events emitted by executed activities; 2) record the input and output SOAP messages; and 3) record the exceptions and trigger the diagnosis function when the first exception is received. ***Diagnosing*** is triggered on the first occurred exception [2]. The MBD approach developed relies on the following three steps with the techniques we introduced in the content above.

---

[2] When a Web service engine supports multiple instances of a process, different instances are identified with a process ID. Therefore, diagnosis is based on the events for one instance of the process.

1) A prior process modelling and variable dependency analysis. All the variables in BPEL are global variables, i.e. they are accessible by all the activities. An activity can be regarded as a function that takes input variables and produces output variables. An activity has three kinds of relation to its input and output variables: initialization, modification and utilization. We use $Init(A, V)$, $Mod(A, V)$ and $Util(A, V)$ to present the relation that activity A initializes V, modifies V or utilizes variable V. An activity is normally a utilizer of its input variables, and is either an initializer or an modifier of its output variables. This is similar to the view point of programming slicing, a technique in software engineering for software debugging. But BPEL can violate this relation by applying some business logic. For example, some variables, such as order ID and customer address, are not changeable after they are initialized in a business process. Therefore, a BPEL activity may be an utilizer of its output variables. In BPEL, it is defined in *correlation sets*. In this case, we use $Util(A, (V1, V2))$ to express that output $V2$ is correlated to input $V1$. In this case, Formula 1 can be simplified as:

$$\neg ab(A.input) \Longrightarrow \neg ab(A.output),$$
$$\text{if } Util(A, (A.input, A.output)) \quad (2)$$

In Example 5, we give a table to summarize the variable dependency for the load approval process. This table can be obtained automatically from BPEL. The approach is not presented due to lack of space.

Example 5 *The variable dependency analysis for the loan approval process is in Table 1.*

2) Trajectories reconstruction from observations after exceptions are detected. As mentioned earlier, the observations are the events and exceptions when a BPEL process is executed. The events can be recovered from the log file in a BPEL engine. The observations are formed in an automaton. The trajectories are calculated by synchronizing the observations with the automaton of the system description:

$$\text{trajectories} = SD \| OBS \quad (3)$$

As trajectories can be recovered from OBS, we do not require to record each event during the execution. It is very useful when some events are not observable and when there are too many events to record. It is our future work to study the minimal observables for diagnosing a fault.

Example 6 *In the loan approval example, assume that OBS={received, invoked_assessor, received_risk, invoked_approver, received_aplErr} (as in Figure 4(a)). received_aplErr is an exception showing that there is a*

Variables	Parts	Initializer	Modifier	Utilizer
request	firstname	receive1		invokeAssessor, invokeApprover
	lastname	receive1		invokeAssessor, invokeApprover
	amount	receive1		invokeAssessor, invokeApprover
risk	level	invokeAssessor		
approval	accept	assign, invokeApprover		reply
error	errorCode	invokeApprover		

Table 1. The variable dependency analysis for the loan approval process.

type mismatch in received parameters. We can build the trajectory of evolution as below, also shown in Figure 4(b).

$$(x_0)\xrightarrow{received}(x_1)\xrightarrow{\epsilon}(x_2)\xrightarrow{invoked\_assessor}(x_4)\xrightarrow{received\_risk}(x_5)$$

$$\xrightarrow{\epsilon}(x_3)\xrightarrow{invoked\_approver}(x_7)\xrightarrow{received\_aplErr}(x_8)$$
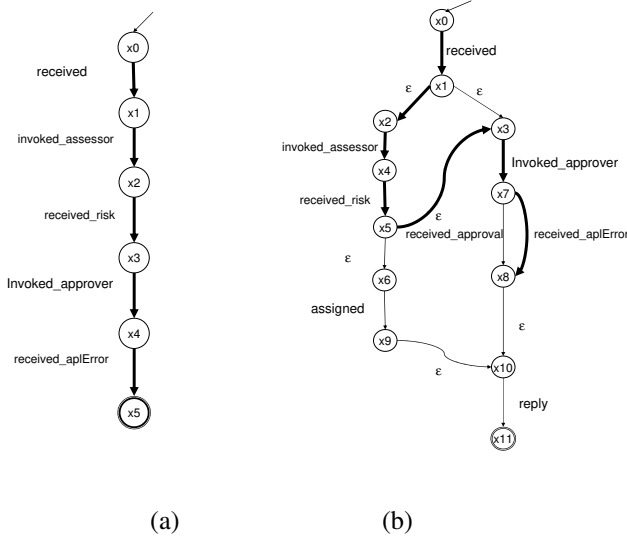

(a)   (b)

Figure 4. (a) the observations; (b) the loan approval process evolution trajectory up to the exception.

3) Accountability analysis for mode assignment

Not all the activities in a trajectory are responsible for the exception. As a software system, the activities connect to each other by exchanging variables. Only the activities which change the attributes within a variable can be responsible for the exception.

Assume that activity A generates exception e_f, and t is a trajectory ending at A. The responsibility propagation rules are (direct consequences of the contraposition of Formula 1 and 2):

$$\forall A.InVar.part \in A.InVar,$$

$$e_f \in \Sigma_A \vdash ab(A) \vee \bigvee ab(A.InVar.part) \quad (4)$$

$\forall A_j \in t, A_j \neq A_i$, A_j is the only activity between A_j and A_i such that $Mod(A_j, A_i.InVar.part) \vee Init(A_j, A_i.InVar.part)$, $\forall A_j.InVar.part \in A_j.InVar$,

$$ab(A_i.InVar.part) \vdash ab(A_j) \vee \bigvee ab(A_j.InVar.part) \quad (5)$$

The first rule in (4) states that if an activity A generates an exception e_f, it is possible that activity A itself is faulty, or any part in its $A.InVar$ is abnormal. Notice a variable is a SOAP message which has several parts. $A.InVar.part$ is a part in $A.InVar$[3]. The second rule in (5) propagates the responsibility backwards in the trajectory. It states that an activity $A_j \in t$ that modifies or initializes a part of $A_i.InVar$ which is known as faulty could be faulty; and its inputs could also be faulty. If there are several activities that modify or initialize a part of $A_i.InVar$, only the last one counts, because it overrides the changes made by the other activities, i.e. A_j is the last activity "between" A_j and A that modifies or initializes $A_i.InVar$, as stated in (5). After responsibility propagation, we obtain a *responsible set* of activities $RS = \{A_i\} \subseteq t$.

Then the diagnosis is that either A or any of A_i in the responsible set is faulty:

$$\{D_f\} = \{\{A\}\} \cup \{\{A_i\}|A_i \in RS\} \quad (6)$$

Each D_f is a single fault diagnosis and the result is the disjunct of D_f. The algorithm is as following. In the worst case, this algorithm checks each activity in t. In the worst case, the function $getFirstModAct(t, A, next\_part)$ checks each activity in t also. Assume k_1 is the maximum number of variables in an activity, and k_2 is the maximum number of parts in a variable. The worst case complexity of this algorithm is $O(k_1 k_2 |t|^2)$, but experiments on examples exhibit lower real complexity.

---

[3]Sometimes, the exception returns the information about the part $A.InVar.part$ is faulty. Then this rule is simplified.

Algorithm 1 Calculate Diagnosis for a Faulty Web Service Process

> **INPUT**: A_0 - the activity generating the exception.
> t - a list of activities in a reserved trajectory whose first element is A_0.
> $Acts$ - a list of candidate activities, initialized as $\{A_0\}$.
> **OUTPUT**: D - the list of faulty activities, initialized as $\{A_0\}$.
> **Notes about the algorithm**: 1) list.next() returns the first element of a list; list.add(element) adds an element at the end of the list; list.remove(element) removes an element from the list. 2) Activity A has a list of variables $A.Vars$ and a variable $var = A.Vars.next()$ has a list of parts $var.Parts$. 3) $getFirstModAct(t, A, next\_part)$ is a function to return the first activity from A in t that modifies or initializes $next\_part$.
>
> **while** $Acts! = null$ **do**
> $A = Acts.next()$
>   **for** $next\_var$ In $A.InVars$ **do**
>   **for** $next\_part$ In $next\_var.parts$ **do**
> $B = getFirstModAct(t, A, next\_part)$
>   **if** $B! = null$ **then**
> $D.add(B)$
> $Acts.add(B)$
> $Acts.remove(A)$
> return D

Example 7 *For the loan approval example, we have the trajectory as in Example 6. We do the responsibility propagation. As invokeApprover generates the exception, according to Formula (4), invokeApprover is possibly faulty. Then its input request is possibly faulty. Among all the activities $\{receive1, invokeAssessor, invokeApprover\}$ in the trajectory, $receive1$ initializes $request$, $invokeAssessor$ and $invokeApprover$ use $request$. Therefore, $receive1$ is possibly faulty, according to Formula (5). $receive1$ is the first activity in the trajectory. The propagation stops. The diagnosis is:*

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

Example 7 has two single faults $\{receive1\}$ and $\{invokeApprover\}$ for the exception $received\_aplErr$, which means either the activity ⟨receive1⟩ or ⟨invokeApprover⟩ is faulty. In an empirical way, an engineer may associate only one fault for an exception. But our approach can find all possibilities. Second, if we want to further identify which activity is indeed responsible for the exception, we can do a further test on the data. For example, if the problem is wrong data format, we can verify the data format against some specification, and then identify which activity is faulty.

4.1 Multiple Exceptions

There are two scenarios where multiple exceptions can happen. The first scenario is the chained exceptions when one exception causes the others to happen. Normally the software reports this chained relation. We need to diagnose only the first occurred exception, because the causal relations for other exceptions are obvious from the chain.

The second scenario is the case when exceptions occur independently, e.g. two parallelled branches report exceptions. As the exceptions are independent, we diagnose each exception independently, the synthesis diagnoses are the union of all the diagnoses. Assume the diagnoses for exception 1 are $\{D_i^1\}$, where $i \in [1, \ldots, n]$, and the diagnoses for exception 2 are $\{D_j^2\}$, where $j \in [1, \ldots, m]$, the synthesis diagnoses are any combinations of D_i^1 and D_j^2: $\{D_i^1 \cup D_j^2 | i \in [1, \ldots, n], j \in [1, \ldots, m]\}$.

What interests us most is the synthesis of the minimal diagnoses. So, we remove the $D_i^1 \cup D_j^2$ that are supersets of other ones. This happens only if at least one activity is common to $\{D_i^1\}$ and $\{D_j^2\}$, giving rise to a single fault that can be responsible for both exceptions. Such activities are thus most likely to be faulty (single faults being preferred to double faults).

5 Related Work

MBD has been used for software debugging. Wotawa, among others, has used consistency-based diagnosis for debugging java and hardware description languages [7]. He has also discussed the relationship between MBD based debugging and program slicing [7] and concluded that MBD in his way and program slicing should draw equivalent conclusions for debugging.

The diagnosis method developed in this paper can be compared to dynamic slicing introduced in [5]. Similar to our method, dynamic slicing considers the bugs should be within the statements that *actually* affect the value of a variable at a program point for a *particular* execution of the program. Their solution, following after Weiser's static slicing algorithm [6], solves the problem using data-flow equations, which is also similar to the variable dependency analysis presented in this paper, but not the same. An external Web service can be seen as a procedure in a program, with unknown behaviour. For a procedure, we normally consider the outputs brought back by a procedure are generated according to the inputs. Therefore, in slicing, the outputs are considered in the definition set (the set of the variables modified by the statement). For Web services, we can know some parts in SOAP response back from a Web service should be unchanged, e.g. the name and the address of a client. Therefore, the variable dependency analysis is

different from slicing. As a consequence, the diagnosis obtained from MBD approach in this paper can be different from slicing, and actually more precise.

Some other people have applied MBD on diagnosing component-based software systems. We found that when diagnosing such systems, the modelling is rather at the component level than translating lines of statements into logic representations. Grosclaude in [3] used a formalism based on Petri nets to model the behaviours of component-based systems. It is assumed that only some of the events are monitored. The history of execution is reconstructed from the monitored events by connecting pieces of activities into possible trajectories. Console's group is working towards the same goal of monitoring and diagnosing Web services like us. In their paper [2], a monitoring and diagnosing method for choreographed Web service processes is developed. Unlike BPEL in our paper, choreographed Web service processes have no central model and central monitoring mechanism. [2] adopted grey-box models for individual Web services, in which individual Web services expose the dependency relationships between their input and output parameters to public. The dependency relationships are used by the diagnosers to determine the responsibility for exceptions. This abstract view could be not sufficient when dealing with highly interacting components. More specifically, if most of the Web services claim too coarsely that their outputs are dependent on their inputs, which is correct, the method in [2] could diagnose all the Web services as faulty. Yan *et al.* [8] was a preliminary work to the present one, focusing on Web service modeling using transition systems. Our work simplifies the model and, more importantly, formalizes the diagnosis principle and presents the diagnosis algorithm for Web service processes.

Other related research includes the studies on Web service monitoring mechanism and the studies to map Web service processes into formal models. The discussion on these studies are eliminated due to lack of space.

6 Conclusions

To identify the Web services that are responsible for a failed business process is important for e-business applications. Existing throw-and-catch fault handling mechanism is an empirical mechanism that does not provide sound and complete diagnosis. In this paper, we develop a monitoring and diagnosis mechanism based on solid theories in MBD. Automata are used to give a formal modelling of Web service processes described in BPEL. We adapt the existing MBD techniques for DES to diagnose Web service processes. Web service processes have all the features of software systems and do not function abnormally until an exception is thrown, which makes the diagnosis principle different from diagnosing physical systems where fault

events are unobservable. The approach developed here reconstructs execution trajectories based on the model of the process and the observations from the execution. The variable dependency relations are utilized to deduce the actual Web services within a trajectory responsible for the thrown exceptions. The approach is sound and complete in the context of modelled behaviour. A BPEL engine can be extended for the monitoring and diagnosis approach developed in this paper.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, and et.al. *Business process execution language for Web Services (BPEL4WS) 1.1*, 2003. *ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf*, retrieved April 10, 2005.

[2] 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.

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

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

[5] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[6] Mark Weise. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[7] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135:125–143, 2002.

[8] Y. Yan, Y. Pencolé, M-O. Cordier, and A. Grastien. Monitoring web service networks in a model-based approach. In *3rd IEEE European Conference on Web Services (ECOWS05)*, Växjö, Sweden, Nov. 14-16, 2005. IEEE Computer Society.