Analysis of BPEL Data Dependencies

Yongyan Zheng, Jiong Zhou, Paul Krause
Department of Computing, University of Surrey
Guildford, GU2 7XH, UK
{y.zheng,j.zhou, p.krause}@surrey.ac.uk

Abstract

BPEL is a de-facto standard language for web service orchestration. It is a challenge to test BPEL processes automatically because of the complex features of the language. The current formal semantics proposed for BPEL can be categorized under three branches: Process Algebra, Petri-nets, and Automata. Our goal is to automate the generation and execution of test cases for composed web services. Model checking is an effective technique for automated test generation, and most mature model checkers such as SPIN and NuSMV use automata as the underlying formal model. Hence, we follow the automata branch. Unfortunately, the current automata based approaches omit the BPEL data dependencies. In order to address this shortcoming, we demonstrate how to model BPEL data dependencies in our proposed web service automata.

Keywords: BPEL, finite state machine, data dependencies, data flow analysis.

1. Introduction

BPEL (Business Process Execution Language) [2] is the de-facto standard language for behavioural modelling in web service orchestration. As is well known, it is tedious and time-consuming to create test cases manually, especially for large and complex models. BPEL is a semi-formal flow-based language with complex features, so it is desirable to provide an automatic test case generation tool for BPEL. In order to verify BPEL rigorously, there are a number of proposals for applying model checking to verify BPEL, by transforming BPEL processes into formal models such as process algebras, Petri nets, and automata [12, 10]. From the model-driven-testing point of view, a BPEL process should be not only a design model but also a test model to derive test cases. In this paper we focus on analyzing BPEL data dependencies. The analysis of BPEL control dependencies is covered in [16], and use of the model checkers SPIN [9] and NuSMV [5] for test case generation from BPEL processes can be found in [17].

The rest of the paper is organized as follows. A motivating example is provided in section 2, the background of BPEL and web service automata is introduced in section 3, the analysis of BPEL data dependencies is provided in section 4, related works are reviewed in section 5, and finally the conclusions are given in section 6.

2. Motivating Example

In BPEL, variables and links may affect the control flow, variables may appear in expressions on the conditions in switch and while, and may also be used in the conditions to fire particular links in the source element. There are two types of variables explicitly declared in a BPEL process: BPEL variables and flow links. BPEL variables are declared in the variables tag of either process or scope activities. Flow links are Boolean variables declared in the links tag of the flow activity. The output link of an activity is defined as *true* if the associated activity completes normally, otherwise the link is defined *false*. BPEL variables and links can be used and defined by the process or scope enclosed activities, and the flow enclosed activities, respectively.

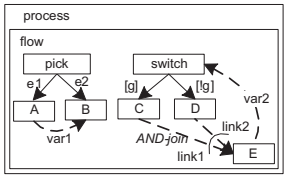

Figure 1. Unreachable and deadlock activities

Fig 1 shows the importance of analysing BPEL data flow. The boxes, the solid arcs, and the dashed arcs denote BPEL activities, control flow, and data flow, respectively. The process encloses a flow, which in turn includes pick, switch, and E running concurrently. The example contains unreachable and deadlock activities. Firstly, *B* never instantiates

33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2007)
0-7695-2977-1/07 $25.00 © 2007 IEEE

variable $var1$, due to the interaction between data flow and control flow. In the *pick*, *A* and *B* are mutually exclusive in control flow, but the output of *A* is the input of *B* in data flow, so *B* cannot instantiates $var1$. E is unreachable due to the faulty design of links. In the *switch*, *C* and *D* are mutually exclusive in control flow, so that link1 and link2 cannot be both true to satisfy the AND-join condition. Therefore, *E* can never be executed. Secondly, there is a deadlock between *swtich* and E, which is caused by the cyclic data flow between them. On one hand, *E* waits for both link1 and link2 to be true but this condition can never be satisfied. On the other hand, the *switch* waits for its input var2 to be defined but var2 cannot be defined by *E* because of the falsity of either link1 or link2. This illustrates the necessity to verify both control and data dependencies of BPEL processes.

3. Background

BPEL consists of two categories of activities: basic and structured activities. Basic activities are atomic actions, including *receive, reply, assign, invoke, throw, terminate, empty*, and *wait*. As with programming languages, the structured activities impose control flow dependency constrains on the executions of either the basic or structured activities within them. A structured activity can contain an arbitrary depth of sub-activities. The structured activities include *pick, switch, while, sequence, flow, scope, eventHandlers, faultHandlers, compensationHandler*.

Definition 1. A **Web Service Automaton** (WSA) M is a sextuple $M = (I_M, S_M, s_{0M}, S_{fM}, T_M, \delta_M)$. As a convention, we omit the subscript of M so that $M = (I, S, s_0, S_f, T, \delta)$.

1) I is the signature of M, denoted as a triple $I = (E, L, O)$, where E, L, O are pair-wise disjoint and represent sets of input, internal, and output events, respectively. Let $Msg = (L \cup E \cup O)$ be the set of events, we assume that L is the disjoint union of a set L_{in} of internal input events and a set L_{out} of internal output events, and the elements of $(E \cup O)$ will be called external events.

2) S is a set of states, $s_0 \in S$ is the initial state, $S_f \subseteq S$ is a set of final states.

3) $T \subseteq (EX \cup \{\Omega\}) \times BX \times (\wp(AX \cup O \cup L_{out}) \cup \{\Omega\})$ is a set of transitions, where EX denotes the event expression. AX, BX are sets of assignments and Boolean expressions respectively.

   - EX is the set of Boolean expressions over input event sets $E \cup L_{in}$. The operators $\wedge, \vee, \neg$ in the input event set correspond to the boolean operators AND, OR, and NOT, respectively.
   - For each transition $t = (ex, g, a) \in T$ (graphically denoted as $ex[g]/a$), $ex \subseteq EX \cup \{\Omega\}$ is the event expression , $g \in BX$ is the guard predicate, and $a \subseteq \wp(AX \cup O \cup L_{out}) \cup \{\Omega\}$ is the action set composed of assignments and output events. Ω indicates the omission of an event expression or an output event.

   The components of transition t are denoted as $t.ex = ex, t.g = g, t.a = a$.

4) $\delta \subseteq S \times T \times S$ is the transition relation (graphically denoted as $s \xrightarrow{t} s'$). If $s \xrightarrow{t} s'$ with $t = (ex, g, a)$, then if the machine is in state s, has received the messages m that satisfy the $t.ex$, and the guard $t.g$ is evaluated to true, then the machine would execute the set of instructions $t.a$ and change state to s'.

Definition 2. We assume that we have available a countable infinite set V of variables together with a set D of values. We define Env to be the set of all functions $\epsilon : V \to D$, an element $\epsilon \in Env$ represents the current values of variables in some system configuration. The **data structure** of machine M is a triple $(V_M, AX_M \cup E_M \cup O_M, BX_M)$, where AX_M, BX_M can be retrieved from the transition set T_M. $AX_M = \{exp \in AX | \exists t \in T.exp \in t.a\}$ and $BX_M = \{exp \in BX | \exists t \in T.exp \in t.g\}$. Let exp denotes an input event, output event, assignment, or Boolean expression. We need three functions:

- $def : (AX \cup E_M) \to \wp(V)$, where $def(exp) \subseteq \wp(V)$ returns the assigned variable, i.e. the variable is the variable on the left hand side of the assignment, and the input parameters of M.
- $cuses : (AX \cup O_M) \to \wp(V)$, where $cuses(exp) \subseteq V$ returns the variables on the right hand side of the assignment, and the output parameters of M.
- $puses : BX \to \wp(V)$, where $puses(exp) \subseteq V$ returns the variables in the Boolean expression.

We define V_M to be the disjoin union of $\bigcup_{exp \in (AX_M \cup E_M \cup O_M)} (def(exp) \cup cuses(exp))$ and $\bigcup_{exp \in BX_M} puses(exp)$.

The machine composition adopts interleaving semantics. Asynchronous execution of web services is achieved by using queues for message processing. The default queueing protocol in WSA is to associate a FIFO queue for each message. WSA communicate by message passing.

Each BPEL activity without flow link corresponds to one WSA. A BPEL activity with flow links will have an associated linkWrapper machine and a core machine. A target

(resp. source) link in an activity is incoming (resp. outgoing) link to the activity. Since *target* and *source* tags are standard element of BPEL activities, a BPEL activity may or may not have flow links. Modelling linkWrapper machines and core machines seperately can simplify the machine structure and clarify machine roles. The linkWrapper machines share a common machine layout, and they are specifically used to handle flow links. As a result, a BPEL process that consists of a set of BPEL activities will be associated with 1..* WSAs.

Since WSA have no hierarchy, we simulate the hierarchical control dependencies of BPEL activities by adding *start* and *done* as common administration messages between machines. A machine can play the role of parent or child. For a machine M_j, if M_i sends a start message to M_j, then M_i is the parent machine of M_j and M_i is the child machine of M_i. A child machine will send a done message to its parent machine when reaching one of its final states. Each machine has $0..1$ parent machines, and $0..*$ child machines. Since the BPEL basic activity is atomic and a BPEL structured activity is hierarchical, the machine for BPEL basic activity has no child, and the machine for a BPEL structured activity has $0..*$ children.

For data handling, BPEL uses a *blackboard* approach, where a set of variables is shared by all activities. By message passing, there are two possible ways to model data exchanges. The centralized approach is to simulate the shared data access by adding data writing to and reading from the *blackboard*. In this case, a machine is required for each variable x to control other machines to write or read x. The decentralized approach is to analyse the BPEL process to discover data dependencies among activities. We use the latter approach because fewer machines will be involved such that the overall state space for a BPEL process can be smaller.

In the following, we use a loan-approval process example from [2] for illustrating how to capture BPEL data dependencies in WSAs and how to generate data flows.

4. BPEL Data Dependencies

In a system, a BPEL process P is seen as a component, and the partnerLinks declared in P correspond to the components interacting with P. In the loan-approval example that will be used later in this paper, the BPEL process *loanapproval* includes three partnerLinks, so the system has four components: loanapproval, customer, assessor, and approver. From the testing point of view, when more than one BPEL process is considered, the system boundary needs to be included. The components within the system boundary are called *service-under-test* (SUT), and a component outside the system boundary is called *tester*. In the following, for a message $msg(x) \in E_M \cup O_M$, msg and x denote the message name and input/output parameter, respectively.

Let $\{M_m..M_n\}$ be the set of machines selected as SUT, a message $msg(v)$ sent from machine M_1 to machine M_2, and a transition t associated with variable x, we have:

- t is annotated with $df(x)$ if a) x is defined in an assignment action of t, i.e.$\{x \in def(exp) | exp \in t.a\}$; or b) $x = v$ is the input parameter of M_2 where M_1 and M_2 are tester and SUT respectively, i.e. $\{x \in def(exp) | t \in T_{M_2}.exp \in t.ex\}$, $M_2 \in \{M_m..M_n\}$, $M_1 \notin \{M_m..M_n\}$.

- t is annotated with $us(x)$ if a) x is used in an assignment action or guard of t, i.e.$\{x \in cuses(exp) | exp \in t.a\}$ or $\{x \in puses(exp) | exp \in t.g\}$; or b) $x = v$ is the output parameter of M_1 where M_1 and M_2 are tester and SUT respectively, i.e. $\{x \in cuses(exp) | t \in T_{M_1}.exp \in t.a \cap O_{M_1}\}$, $M_2 \in \{M_m..M_n\}$, $M_1 \notin \{M_m..M_n\}$.

- Let P_1, P_2 be two BPEL processes. t is annotated with $idf(x)$ if $x = v$ is the input parameter of M_2 where M_1, M_2 belong to different BPEL processes but both are SUT, i.e. $\{x \in def(exp) | t \in T_{M_2}.exp \in t.ex\}$, $M_1, M_2 \in \{M_m..M_n\}$.

- Let P_1, P_2 be two BPEL processes. t is annotated with $ius(x)$ if $x = v$ is the output parameter of M_1 where M_1, M_2 belong to different BPEL processes but both are SUT, i.e. $\{x \in cuses(exp) | t \in T_{M_1}.exp \in t.a \cap O_{M_1}\}$, $M_1, M_2 \in \{M_m..M_n\}$.

The i in $idf(x), ius(x)$ means internal. For simplicity, a transition t is def-x if t can be either annotated with $df(x)$ or $idf(x)$, while t is use-x if t can be either annotated with $us(x)$ or $ius(x)$.

Definition 3. A variable x is **globally defined and used** if there exist transitions $t_1 \in T_{M1}, t_2 \in T_{M2}$, where t_1, t_2 are with def-x and use-x, respectively. The variable x is **locally defined and used** if: 1) there exist transitions $t_1, t_2 \in T_M$, where t_1, t_2 are with def-x and use-x respectively; or 2) there exits a transition $t_1 \in T_M$ where t_1 is with def-x and use-x.

According to the BPEL specification [2], in a WSA the case 1) and 2) will not exist for those variables explicitly declared in BPEL processes (i.e. BPEL variables and flow links). Therefore, we only consider globally defined and used variables, such that a machine can either have a transition with def-x or have a transition with use-x but not both. As a result, the def-x (resp.use-x) annotation of a machine M can be retrieved from the def-x(resp. use-x) of a transition $t \in T_M$. A *exdu-pair* of x is a transition pair (t_i, t_j)

where t_i is with def-x and t_j is with use-x. The pair of machines with def-x and use-x is called *machine-exdu-pairs*.

4.1 Data Exchange Models

An *internal data exchange model* is used for a single BPEL process to specify the relation between inputs and outputs of BPEL activities. An *external data exchange model* is used to capture how messages are transferred from one BPEL process to other BPEL processes. When a single BPEL process is selected as SUT, an internal data exchange model is enough to capture the BPEL data semantics. When multiple BPEL processes are selected as SUT, a *global data exchange model* which is the union of the internal and external data exchange models is required to capture the BPEL data semantics.

A. Internal Data Exchange Model

In this section, we identify different types of data dependencies of BPEL activities, and discuss how to capture these data dependencies in WSAs.

Rule 1. In a BPEL process P, let x be a BPEL variable or a flow link explicitly declared in P, and M_i be the WSA associated with BPEL activity B_i. BPEL activities can be categorized into four types.

1) When B_i receives $msg(x)$ from an external BPEL process, given that the partner who sends $msg(x)$ is a tester and B_i is SUT, M_i is with def-x. The BPEL activities belonging to this type include: *receive* activity, *invoke* activity with x as outputVariable, *pick* activity, and *eventHandler* activity.

2) When B_i uses x in a predicate or assignment, it reads the value of x and M_i is with use-x. The following BPEL activities belong to this type: *assign* activity with x on the right of assignment expressions, *while* activity and *switch*, and an activity with x as a targetLink.

3) When B_i sends a message $msg(x)$ to an external BPEL process, given that the partner who receives $msg(x)$ is a tester and B_i is SUT, M_i is with use-x. The BPEL *invoke* activity with x as inputVariable, and the *reply* activity belong to this type.

4) When B_i defines x in an assignment, it writes a value to x and M_i is with def-x. Two BPEL activities belong to this type: *assign* activity with x on the left of assignment expressions and an activity with x as a sourceLink.

Rule 2. In a BPEL process P, the data can only exchange between two machines $M_1, M_2 \in P$ if one of the following conditions is satisfied:

1) M_1, M_2 have a same parent machine, i.e. they are same-level machines;

2) M_1, M_2 are parent and child, or vice-versa.

For simplicity, condition 1) will be checked first, and condition 2) will be checked when 1) is false.

The rationale behind rule 2 is illustrated by an example in Fig 2. The BPEL process fragment is shown in Fig 2(a), where the solid lines denote the node hierarchy of BPEL activities. It has a *sequence* activity that encloses *flow*, *while*, and *switch* activities. The *flow* activity encloses A and B activities. The *while* activity encloses C activity. The *switch* activity encloses C and D activities. In the example, M_A is used to denote the machine for activity A, and M_f, M_w, M_s are short for $M_{flow}, M_{while}, M_{switch}$. By analyzing model(a) according to rule 1, suppose we have M_A, M_C with def-x, M_w, M_s with use-x, M_B with def-y, and M_D, M_E with use-y.

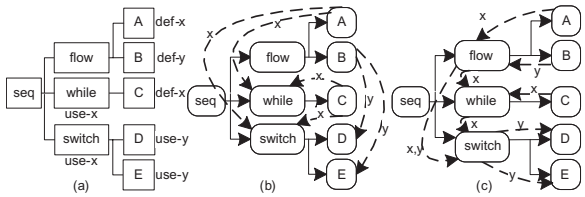

Figure 2. Modelling internal data exchanges

To capture data semantics of the BPEL process in WSAs, Fig 2 (b)(c) show two ways of modelling data exchanges. The dashed and solid lines denote the machine data flows and the control flows, respectively. In (b), for an arbitrary variable v, the machine with def-v will send message $msg(v)$ to the machine with use-v directly. Based on the data exchange model (b), two problems may exist.

1) Sending intermediate data values.

2) Sending data to unreachable machines.

Problem 1) exists in the example; machine M_w receives $msg(x)$ from M_A and uses x in its predicate $pred$. If $pred$ is true, it starts the child machine M_C. M_C re-defines x. On the one hand, M_C sends $msg(x)$ back to M_w for re-evaluating $pred$. On the other hand, M_C sends $msg(x)$ to M_s. The while loop continues until $pred$ becomes false. Since M_C is in a while loop, everytime M_C is executed, it will send message $msg(x)$ to M_s. If the while loop iterates n times, then M_s will receive $n-1$ times of $msg(x)$ with intermediate values of x. Nevertheless, M_s only needs the value of x in the final loop, so the decision of sending $msg(x)$ should be made by M_w.

Problem 2) also exists in the example. M_s has two child machines M_D, M_E, where in a given time only one of the

0-7695-2977-1/07 $25.00

child machines can be executed but not both. Therefore, either M_D or M_E needs to receive $msg(y)$. However, since M_B cannot decide the choice of M_D or M_E, M_B will send $msg(y)$ to M_D and $msg(y)$ to M_E. Suppose M_E is not chosen to execute, M_B will send $msg(y)$ to an unreachable machine. Therefore, M_B should send $msg(y)$ to M_s which can decide which child machine should receive the data. Furthermore, problem 2) may exist for a BPEL activity B with flow links, which will be associated a linkWrapper machine M_{Bwp} and a core machine M_B. Since M_{Bwp} is the parent of M_B, M_{Bwp} decides whether M_B can be started. M_B can be started only when the predicate for the targetLinks is evaluated to be true in M_{Bwp}. So if M_B requires a data v, the message $msg(v)$ should be sent to M_{Bwp} which decides whether to forward it to M_B.

The above two problems can be solved by adding the constraints of rule 2, shown in Fig 2 (c). For the first problem, M_C sends $msg(x)$ to its parent M_w, and M_w forwards $msg(x)$ to the same-level machine M_s. For the second problem, M_B sends $msg(y)$ to its parent M_f, and M_f fowards $msg(y)$ to the same-level machine M_s, which in turn forwards $msg(y)$ to one of M_D, M_E. Comparing (b) and (c), the (c) approach is clearer and more precise even though it requires additional message transfers.

According to rule 2, Fig 3 below shows an algorithm to generate a data exchange path (machine sequence) for a machine-exdu-pair (d_node, u_node). The idea is to find a common ancestor node cp. The message $msg(x)$ is sent upstream from d_node to a child node M_{C1} of cp, while $msg(x)$ is received downstream from a child node M_{C2} of cp to u_node. Finally M_{C1} sends $msg(x)$ to M_{C2}. The worst-case time complexity of the algorithm is $O(n^2)$.

```
1  foo (d_node, u_node) : path {
2    IF (d_node == u_node) RAISE ERROR!
3    d_path.add(d_node);
4    u_path.add(u_node);
5    d_cur = d_node;
6    u_cur = u_node;
7    WHILE ((d_par == d_cur.parent()) != NULL OR (u_par == u_cur.parent()) != NULL) {
8      IF (d_par != NULL) {
9        IF (d_par IN u_path) {
10         IF (d_par == u_path[0]) RETURN d_path.add(d_par);
11         ELSE {
12           u_path = u_path.getNodesBefore(d_par);
13           RETURN d_path.concat(reverse(u_path));
14         }
15       } ELSE {
16         d_path.add(d_par);
17         d_cur = d_par;
18       }
19     }
20     IF (u_par != NULL) {
21       IF (u_par IN d_path) {
22         IF (u_par == d_path[0]) RETURN u_path.insert(u_par,0);
23         ELSE {
24           d_path = d_path.getNodesBefore(u_par);
25           RETURN d_path.concat(reverse(u_path));
26         }
27       } ELSE {
28         u_path.add(u_par);
29         u_cur = u_par;
30       }
31     }
32   }
33   RETURN NULL;
     }
```

Figure 3. Internal data exchange algorithm

In Fig 3, given (d_node, u_node) as input, two sequences d_path, u_path are created (line 2-3). Starting from d_node, u_node, it iteratively gets the parent nodes from the current nodes, denoted by d_par, u_par, until both root nodes are reached (line 7). The while contains two parts. First, it checks whether a common ancestor node is reached. 1.1) d_par is in u_path(line 9-14): if u_node itself is the parent of d_node, then the output path is $\langle d\_node, u\_node\rangle$ (line 10); otherwise, the output path is the reversed elements of the u_path before the common node. 1.2) d_par is not in u_path(line 15-18), d_par is added to d_path and d_par becomes the current node. Second, similarly it checks whether a common ancestor node is reached. 2.1) u_par is in d_path(line 21-26): if d_node itself is the parent of u_node, then the output path is $\langle d\_node, u\_node\rangle$ (line 22); otherwise, the output path is the reversed elements of the u_path before the common node. 2.2) u_par is not in d_path(line 27-30), u_par is added to u_path and u_par becomes the current node.

We use the variable x in Fig 2 as an example. From (a) we get the machine-exdu-pairs for x are (M_A, M_w), (M_A, M_s), (M_C, M_w), and (M_C, M_s). When applying the algorithm to the example, the data exchange paths for the above machine-exdu-pairs would be $\langle M_A, M_f, M_w\rangle$,$\langle M_A, M_f, M_s\rangle$,$\langle M_C, M_w\rangle$, and $\langle M_C, M_w, M_s\rangle$, respectively.

In the loan-approval example below, when the BPEL process *loanapproval* is selected as SUT, the internal data exchange model of this single process is shown in Fig 4, where the control flows are not shown for simplicity.

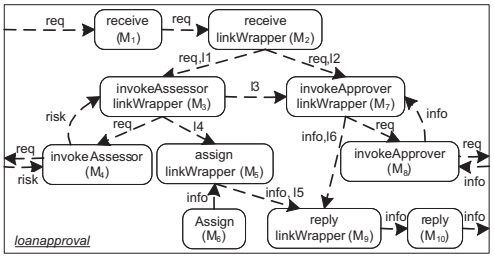

Figure 4. Internal data exchange model

According to rule 1, the machine-exdu-pairs for variable req are $(M_1, M_4), (M_1, M_8)$. By the algorithm of Fig 3, the data exchange paths for the above machine-exdu-pairs are $\langle M_1, M_2, M_3, M_4\rangle$ and $\langle M_1, M_2, M_7, M_8\rangle$ respectively. Similarly for flow link l_1, the machine-exdu-pair is (M_2, M_3) and the data exchange path is $\langle M_2, M_3\rangle$. Note that for a variable x, the additional $msg(x)$ messages are only used to capture the BPEL internal data exchanges.

B. External Data Exchange Model

Since the communication scheme between web services is message passing, BPEL processes exchange data by pass-

ing messages. There exist two ways to capture the interactions between BPEL processes: top-down and bottom-up approaches. The top-down approach is to firstly design a conversation protocol to capture global interactions of BPEL processes, and secondly design the BPEL processes to implement the conversation protocol. The bottom-up approach is to firstly design BPEL processes, and secondly derive the global interactions from the BPEL processes. [8] points out the advantage of the top-down approach over the bottom-up approach. However, from the testing point of view, it is especially important to verify the correctness of the BPEL interactions when a conversation protocol is missing. In our framework, we assume that there is no conversation protocol to guide the design of BPEL process interactions. Instead, we verify the correctness of BPEL process interactions by deriving a BPEL external data exchange model from individual BPEL processes.

For the loan-approval example, three partnerLinks are included in the *loanapproval* process: *customer, assessor*, and *approver*. If *customer* is selected as a tester, then the SUT contains three components: *loanapproval*, *assessor*, and *approver*. The external data exchange model can be easily constructed from BPEL activities by identifying which partnerLink a message is sent to or received from.

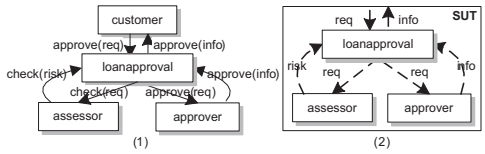

Figure 5. External data exchange model

In Fig 5, the left shows the messages passing between BPEL processes for the loan-approval example, and the right shows the derived BPEL external data exchange model where *customer* component is chosen as tester and the rest components are selected as SUT.

C. Global Data Exchange Model

A global data exchange model is the union of the internal data exchange models of invidual BPEL processes and the external data exchange model of these BPEL processes. In the loan-approval example, when BPEL processes *assessor*, *loanapproval*, and *approver* are selected as SUT, the global data exchange model is shown in Fig 6.

4.2 BPEL Data Flows

After modelling how data exchanges within a BPEL process and across BPEL processes, data flows can be derived for each variable so that we can check whether a defined variable will be later used and whether a used variable has

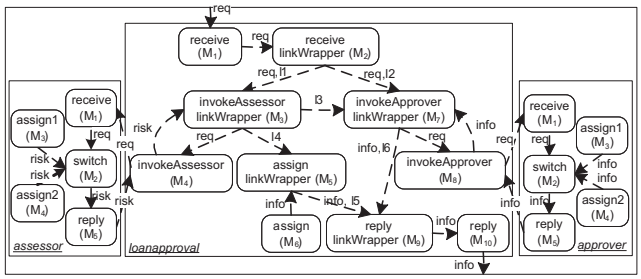

Figure 6. Global data exchange model

been previously defined. In this section, we will discuss two kinds of data flow: 1) the data flows when a single BPEL process is selected as SUT; 2) the data flows when multiple BPEL processes are chosen as SUT.

Let x be a variable. A du-pair of x is a transition pair (t_i, t_j) where t_i is with $df(x)$ and t_j is with $us(x)$. A *def-clear path* with respect to x is a transition sequence $\langle t_1, t_2, .., t_n \rangle$ where there is no $df(x)$ in any transition t_k where $1 < k < n$. A *data flow* (or *du-path*) of x is a transition sequence that (t_i, t_j) is a du-pair and there is a def-clear path from t_i to t_j with respect to x.

Definition 4. Let M_1, M_2 be two machines, there is a **data flow** from M_1 to M_2, i.e. $t_2 \in T_2$ is data dependant on $t_1 \in T_1$, iff there exists a variable x such that

1) t_1 is with $df(x)$.

2) t_2 is with $us(x)$.

3) There exists a def-clear path from t_1 to t_2 for x.

The data flows for a variable x can be constructed by identifying du-pairs, and checking whether there is a def-clear path between the du-pairs. The data flows for a variable x can be derived automatically by model checking techniques, based on the annotations $df(x), us(x)$ of transitions. For each du-pair $\langle t_i, t_j \rangle$ of variable x, where $t_i \in T_{Mi}, t_j \in T_{Mj}$. In t_j, x needs to be asserted that x has been defined previously. For the purpose of illustration, here we use machine-du-pairs retrieved from du-pairs, and use machine sequences retrieved from the transition sequences of data flows. The pair of machines with $df(x)$ and $us(x)$ is called *machine-du-pairs*.

A. Data Flows of Single BPEL Process

When a single BPEL process P is selected as SUT, the data flows can be derived from the internal data exchange model of P. In the loan-approval example, when BPEL process loanapproval is selected as SUT, the internal exchange model is shown in Fig 4. By model checking, the data flows for BPEL variables and flow links can be derived as follows.

0-7695-2977-1/07 $25.00

First, for BPEL variable req, the machine-du-pairs are $(M_1, M_4), (M_1, M_8)$, so the data flows are via machine sequences $\langle M_1, M_2, M_3, M_4\rangle$ and $\langle M_1, M_2, M_7, M_8\rangle$. Second, for BPEL variable $risk$, the machine-du-pair is (M_4, M_3), so the data flow is via machine sequence $\langle M_4, M_3\rangle$. Third, for BPEL variable $info$, the machine-du-pair are $(M_6, M_{10}), (M_8, M_{10})$, so the data flows are via machine sequences $\langle M_6, M_5, M_9, M_{10}\rangle$, and $\langle M_8, M_7, M_9, M_{10}\rangle$. Finally, for flow links $l_1, l_2, l_3, l_4, l_5, l_6$, their data flows are via machine sequences $\langle M_2, M_3\rangle$, $\langle M_2, M_7\rangle$, $\langle M_3, M_7\rangle$, $\langle M_3, M_5\rangle$, $\langle M_5, M_9\rangle$, and $\langle M_7, M_9\rangle$, respectively.

B. Data Flows of Multiple BPEL Processes

For multiple BPEL processes $\{P_1, .., P_n\}$, the data flows for a variable can be derived from the global data exchange model of $\{P_1, .., P_n\}$. In the loan-approval example, when BPEL processes loanapproval, assessor, and approver are selected as SUT, the global exchange model is shown in Fig 6. By model checking, the data flows can be derived in the following.

Let $P.M_i$ denote machine M_i of BPEL process P, and S, L, A are used as shorthands for the BPEL processes *assessor*, *loanapproval*, and *approver*, respectively. First, for BPEL variable req, the machine-du-pairs are $(L.M_1, S.M_2)$ and $(L.M_1, A.M_2)$, so the data flows are via machine sequences $\langle L.M_1, L.M_2, L.M_3, L.M_4, S.M_1, S.M_2\rangle$ and $\langle L.M_1, L.M_2, L.M_7, L.M_8, A.M_1, A.M_2\rangle$. Second, for BPEL variable $risk$, the machine-du-pairs are $(S.M_3, L.M_3)$ and $(S.M_4, L.M_3)$, so the data flows are via machine sequences $\langle S.M_3, S.M_2, S.M_5, L.M_4, L.M_3\rangle$ and $\langle S.M_4, S.M_2, S.M_5, L.M_4, L.M_3\rangle$. Third, for BPEL variable $info$, the machine-du-pairs are $(A.M_3, L.M_{10})$ and $(A.M_4, L.M_{10})$, so the data flows are via machine sequences $\langle A.M_3, A.M_2, A.M_5, L.M_8, L.M_7, L.M_9, L.M_{10}\rangle$, and $\langle A.M_4, A.M_2, A.M_5, L.M_8, L.M_7, L.M_9, L.M_{10}\rangle$. Finally, the data flows of flow links are the same as the data flows derived from a single BPEL process.

5. Related Works

In the literature, most existing work abstracts from data and focuses attention on the control flow. When data is omitted, the transition guards and variables were left out, so selecting one of two control paths, solved by the evaluation of data, needs to be modelled by a nondeterministic choice. Even for work that does consider BPEL data, data dependencies are not modelled in an explicit way. In this section, we review work with consideration of modelling data dependencies in the orchestration models such as BPEL processes. The purpose of analyzing data dependencies is to ensure data is always defined before being used.

In [13], they propose a BioOPera Flow language to model the control dependencies and data dependencies between tasks (BPEL activities) as visual flow graphs. In order to maintain the consistency, they provide a set of constraints when constructing a data flow graph. For instance in a process data flow graph, data always flows from output to input parameters of tasks. The input parameters of a process can only be connected to input parameters of tasks, and output parameters of the process may receive data only from output parameters of tasks. A constant data can be connected to multiple input parameters, but an input parameters bound to a constant data cannot have any other incoming data flow edge. A toolset is developed to support the visualization. Even though their focus is not rigorous verification of design models, they show the importance of considering control and data dependencies in separation.

In the composition language proposed by [15], each task (equally to a BPEL activity) has an inputDependencies section to describe the control dependencies and data dependencies from itself to other tasks. For instance, variable x is the output data of task tk_1 and the input data of task tk_2, tk_2 will declare a data dependency in its input-Dependencies section to specify tk_1 is the source who sends x to it. The task who receives a message from an external web service will send the message to other 'downstream' tasks which have dependencies on this message. Their composition language is mapped to Pi-calculus. In Pi-calculus, a process denotes web service task, channels represent takes data dependencies, and control dependencies are represented implicitly using the operators of Pi-calculus directly. The composition service as a whole is modelled as a parallel composition of all of these processes. Their data dependency modelling makes the data definition and usage clear. With this in mind, our WSA should also be able to capture the data dependencies of BPEL activities in an explicit way.

A grid workflow language is proposed in [7], where each activity may have data-in port and data-out port. The data exchange describes that the data flows from data-out ports to data-in ports. They discuss the constraints added on the data exchanges in conditional activities (e.g. BPEL *switch* and *pick*), in sequential loop activities (e.g. BPEL *while*), and in parallel loop activities (e.g. BPEL *flow*).

The authors in [3] provide a model of data flow in addition to control flow for OWL-S process models. They transform OWL-S to Promela so that SPIN model checker can be used to verify the OWL-S process model. Their scope of the data flow is limited to within a composite process. The processes in a composite process can exchange data among themselves or with the parent process. In OWL-S, a process is similar as a BPEL activity. We also add such *level-based* constraint on the BPEL internal data exchanges.

For external data exchanges between orchestration models, if there is a conversation protocol available, the data dependencies between web services can be directly derived from the conversation protocol; otherwise, one needs to analyze the data exchanges to get the data dependencies. The work of [4] discusses how to analyze data exchanges between YAWL workflow models, so that the resulting data dependencies between web services can be used for service matching. In [6], they propose a OWL-P language to model both the conversation protocol as well as the orchestration models. When composing orchestration models, the designer needs to define a set of *composition axioms* to add constraints on the conversation protocol. A data-flow axiom states the data exchange dependency among the orchestration models. In our test framework, we do not assume a conversation protocol is available, and the data dependencies between BPEL processes need to be analyzed.

In [11], they propose *data nets* to capture data exchange and data manipulation within an orchestration model, as well as data exchange between composition models. The control flow of a orchestration model is modelled by STS (State Transition System) [14]. STS with data is the synchronized product of all the STSs and data nets. A tool is needed to do the experimental evaluation. Since our WSA includes data, there is no need to add a separate data model. Data flows can be derived from WSAs based on existing data flow analysis techniques.

6. Conclusions

In this paper, we analyse the BPEL internal and external data dependencies and illustrate how to capture these dependencies in our previously proposed web service automaton (WSA). Model checkers can be used to automatically generate data flows for each variable, so that data flow testing can be applied to either a single BPEL process or multiple BPEL processes. The proposed WSA is implemented in XML, where a transition annotated with $df(x), idf(x), ius(x), us(x)$ are denoted as XML transition attributes df, idf, ius, us respectively. We developed an Eclipse based tool as a part of the *DBEStudio* deliverable of the EU project [1]. The tool allows users to select SUT, a pre-defined test coverage criterion, and a model checker. Thereafter, the tool can automatically generate JUnit test cases by mapping BPEL processes to WSAs, mapping WSAs to the input languages of the selected model checker, invoking the model checker to generate counterexamples, and retrieving test cases from the counterexamples.

Acknowledgment

This work was supported by the EU FP6 funded project Digital Business Ecosystems.

References

[1] Digital business ecosystem. http://www.digital-ecosystem.org, 2007.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services version 1.1. May 2003.

[3] A. Ankolekar, M. Paolucci, and K. P. Sycara. Towards a formal verification of owl-s process models. In *International Semantic Web Conference*, Lecture Notes in Computer Science, pages 37–51. Springer, 2005.

[4] A. Brogi and R. Popescu. Towards semi-automated workflow-based aggregation of web services. In *ICSOC*, Lecture Notes in Computer Science, pages 214–227. Springer, 2005.

[5] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Proc. of CAV*, pages 495–499. Springer-Verlag, 1999.

[6] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Interaction protocols as design abstractions for business processes. *IEEE Trans. Softw. Eng.*, 31(12):1015–1027, 2005.

[7] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl: an abstract grid workflow language. In *Proc. of CCGRID*, pages 676–685. IEEE Computer Society, 2005.

[8] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.

[9] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley Professional, 2003.

[10] R. Hull and J. Su. Tools for design of composite web services. In *Proc. of SIGMOD*, pages 958–961. ACM Press, 2004.

[11] A. Marconi, M. Pistore, and P. Traverso. Specifying data-flow requirements for the automated composition of web services. In *Proc. of SEFM*, pages 147–156. IEEE Computer Society, 2006.

[12] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 08(6):51–59, 2004.

[13] C. Pautasso and G. Alonso. Visual composition of web services. In *Proc. of HCC*, pages 92–99. IEEE Computer Society, 2003.

[14] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *Proc. of ICWS*, pages 293–301. IEEE Computer Society, 2005.

[15] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. Notations for the specification and verification of composite web services. In *Proc. of EDOC*, pages 35–46. IEEE Computer Society, 2004.

[16] Y. Zheng and P. Krause. Automata semantics and analysis of bpel. In *Proc. of DEST*. IEEE Computer Society, 2007.

[17] Y. Zheng and P. Krause. A model checking based test case generation framework for web services. In *Proc. of ITNG*, pages 715–722. IEEE Computer Society, 2007.