Detecting Concurrency-Related Problematic Activity Arrangement in WS-BPEL Programs

Yitao Ni[1,2], Lu Zhang[1,2], Zhongjie Li[3],Tao Xie[4], Hong Mei[1,2]
[1]Institute of Software, School of Electronics Engineering and Computer Science
[2]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
{niyt08,zhanglu,meih}@sei.pku.edu.cn
[3]IBM Research, China, lizhongj@cn.ibm.com
[4]Department of Computer Science, North Carolina State University, Raleigh, NC 27695
xie@csc.ncsu.edu

Abstract

A composite web service often interacts with several partner web services hosted in different servers. These partner web services are represented as activities in a WS-BPEL program describing the composite web service. Invoking the maximal number of these activities concurrently is essential for improving its performance. But complex message coupling and control dependency between activities in a composite web service may prevent them from being arranged correctly and efficiently. In this paper, we propose an approach for detecting problematic activity arrangement in a WS-BPEL program based on analyzing message propagation and activity dependency in the program. The underlying idea of our approach is to check whether activities with dependency are arranged as concurrent and whether activities without dependency are arranged as sequential. Our preliminary empirical results demonstrate the effectiveness of our approach.

1 Introduction

WS-BPEL [1] is a popular language for integrating multiple web services, which are referred to as partner web services, into a new composite web service. These partner web services are represented as activities in a WS-BPEL program. In order to improve performance, web services are usually hosted in servers with powerful computing capacity. For a composite web service described as a WS-BPEL program, an important factor for its performance is whether the activities in it are arranged to run with as much concurrency as possible. In practice, developers may not arrange the activities correctly or efficiently. As a result, problematic activity arrangement may result in faults in a WS-BPEL program. In this paper, we consider two types of problematic activity arrangement in a WS-BPEL program. First, if a WS-BPEL program contains activities that cannot run concurrently but are arranged to run concurrently, the activity arrangement may result in unexpected run results. Second, if a WS-BPEL program contains activities that can run concurrently but are arranged to run sequentially, the activity arrangement may result in performance issues.

Typically, a developer needs to manually ensure correctness and efficiency of the activity arrangement in a WS-BPEL program. In a WS-BPEL program, it is common that the input message for invoking one activity is composed of different parts of the outputs of running some other activities. These connections between inputs and outputs may form complex dependency between activities. When arranging activities, as the WS-BPEL language allows concurrency of any activities, the developer needs to keep in mind the dependency relationships between activities to produce correct and efficient activity arrangement. Furthermore, besides the parts related to concurrency, a WS-BPEL program also consists of many other elements, such as tags, delimiters, and auxiliary elements. These elements may further complicate the WS-BPEL program and make the program prone to problematic activity arrangement.

Some developers resort to graphical WS-BPEL development tools (such as ActiveBPEL Designer[1] and BPEL Process Manager[2]) to design and implement a WS-BPEL program. These tools may release developers from writing the entire WS-BPEL program manually and the developers can finish their work through drags and drops on the graphical interfaces of the tools. However, these tools can only facilitate programming with WS-BPEL, but cannot ensure correctness and efficiency of the resulting program.

In this paper, we present an approach that detects the two preceding types of problematic activity arrangement related

[1] $http://www.activebpel.org$
[2] $http://otn.oracle.com$

to correctness and performance. There are four steps in our approach. First, given a WS-BPEL program, our approach derives a message propagation graph. Second, based on the message propagation graph, our approach constructs an activity dependency graph. Third, our approach partitions the activities in the program into a series of sets of activity sequences. Finally, our approach checks problematic activity arrangement in the program using two criteria. We also performed preliminary empirical evaluations of our approach, in which we find an instance of problematic arrangement in an exemplary WS-BPEL program. In particular, this problem is of the second type mentioned previously (see the details in Section 5.1). The empirical results indicate that our approach is effective in detecting these two types of problematic activity arrangement.

This paper makes the following main contributions:

- An approach for detecting concurrency-related problematic activity arrangement in a WS-BPEL program based on statically analyzing message propagation.
- Preliminary evaluations that demonstrate the effectiveness of our approach.

We organize the rest of this paper as follows. Section 2 motivates our approach through an example. Section 3 presents our approach. Section 4 discusses our approach. Section 5 presents our evaluations. Section 6 discusses related work, and Section 7 concludes.

2 Motivating Example

In this paper, we use a WS-BPEL program $TravelD$ adapted from $Travel$[3], a well-known and carefully designed sample WS-BPEL program, as a running example to illustrate our approach. $Travel$ has the functionality of searching for information of available flight tickets. $Travel$ accepts a request, processes it, and then returns the information of available flight tickets. When $Travel$ receives a request containing information of a company employee, $Travel$ communicates with a partner service named $Employee$ to query for the travel class that the employee can enjoy. Then $Travel$ communicates with two other partner services providing airline services, namely $American\ Airline$ and $Delta\ Airline$, to find available flight tickets with the lowest price and the specified travel class. Finally, $Travel$ returns the obtained information to its user. All the involved partner services in this process may be hosted in different servers. The detailed description of $Travel$ and its WS-BPEL program can be found elsewhere[3].

The WS-BPEL[4] program $TravelD$ in Figure 1 is expected to implement the same functionality as that of $Travel$, but it may contain problematic activity arrangement. Comparing $TravelD$ with $Travel$, $TravelD$ has the

```
1.<sequence>
2.    <receive partnerLink="client"
3.          portType="trv:TravelApprovalPT"
4.          operation="TravelApproval"
5.          variable="TravelRequest"  createInstance="yes" />
6.   <assign> <copy><from variable="TravelRequest" part="employee"/>
7.      <to variable="EmployeeTravelStatusRequest" part="employee"/>
8.   </copy> </assign>
9.   <invoke partnerLink="employeeTravelStatus"
10.          portType="emp:EmployeeTravelStatusPT"
11.          operation="EmployeeTravelStatus"
12.          inputVariable="EmployeeTravelStatusRequest"
13.          outputVariable="EmployeeTravelStatusResponse" />
14.    <assign> <copy>
15.       <from variable="TravelRequest" part="flightData"/>
16.       <to variable="FlightDetails" part="flightData"/> </copy> <copy>
17.       <from variable="EmployeeTravelStatusResponse" part="travelClass"/>
18.       <to variable="FlightDetails" part="travelClass"/></copy> </assign>
19.   <flow><invoke partnerLink="AmericanAirlines"
20.                  portType="aln:FlightAvailabilityPT"
21.                  operation="FlightAvailability"
22.                  inputVariable="FlightDetails" />
23.           <receive partnerLink="AmericanAirlines"
24.                  portType="aln:FlightCallbackPT"
25.                  operation="FlightTicketCallback"
26.                  variable="FlightResponseAA" /></flow>
27.    <invoke partnerLink="DeltaAirlines"
28.          portType="aln:FlightAvailabilityPT"
29.          operation="FlightAvailability"
30.          inputVariable="FlightDetails" />
31.    <receive partnerLink="DeltaAirlines"
32.          portType="aln:FlightCallbackPT"
33.          operation="FlightTicketCallback"
34.          variable="FlightResponseDA" />
35.    <switch>  <case condition="...">
36.         <assign> <copy> <from variable="FlightResponseAA" />
37.         <to variable="TravelResponse" /> </copy> </assign>  </case> <otherwise>
38.         <assign> <copy> <from variable="FlightResponseDA" />
39.         <to variable="TravelResponse" /> </copy> </assign> </otherwise> </switch>
40.    <invoke partnerLink="client"
41.          portType="trv:ClientCallbackPT"
42.          operation="ClientCallback"
43.          inputVariable="TravelResponse" />
44.</sequence>
```

Figure 1. Fragment of the $TravelD$ program

following parts of concurrency-related problematic activity arrangement:

- Activity $FlightAvailability$ in Line 21 and activity $FlightTicketCallback$ in Line 25 are arranged to run concurrently, but they actually cannot run concurrently.
- Activity $FlightTicketCallback$ in Line 25 and activity $FlightTicketCallback$ in Line 33 are arranged to run sequentially, but actually they can run concurrently.

At a first glance, activity $FlightAvailability$ in Line 21 and activity $Flight\ TicketCallback$ in Line 25 do not explicitly share any common message. Thus, they seemingly may run concurrently as described in $TravelD$. However, by examining them carefully, we find that they share a common partner link (i.e., $AmericanAirlines$ in Lines 19 and 23). Furthermore, $FlightAvailability$ has no output message but is an activity of the *invoke* type as indicated in Line 19, and $FlightTicketCallback$ is an activity of the *receive* type (Line 23). As a result, the information implies that $TravelD$ may send a request to

[3] $http://www.oracle.com/technology/pub/articles/matjaz\_bpel1.html$.

[4] The basics of the WS-BPEL language are described in Section 3.1.

$AmericanAirlines$ and wait a reply from it. Therefore, activity $FlightAvailability$ in Line 21 and activity $FlightTicketCallback$ in Line 25 may not run concurrently, and $TravelD$ may mistakenly arrange them to run concurrently.

When we examine activity $FlightTicketCallback$ in Line 25 and activity $FlightTicketCallback$ in Line 33, we find that the two activities cannot impact each other via messages and they wait for replies from different partner web services. Therefore, they may run concurrently but $TravelD$ arranges them to run sequentially. In other words, the arrangement of the two activities in $TravelD$ may also be problematic, since such arrangement may result in performance issues.

As the $TravelD$ program is simple, it is still feasible to manually check the arrangement of the preceding two pairs of activities. For a complicated WS-BPEL program, checking the arrangement of a pair of activities may be complicated, because such checking may require tracking the flows of messages between activities. Furthermore, as developers may need to check each pair of activities, for a WS-BPEL program involving a large number of partner web services, the number of activity pairs that need checking is also large. As a result, manually checking the whole program is often infeasible.

3 Our Approach

In this section, we propose an approach for detecting problematic activity arrangement in a WS-BPEL program. The basic idea of our approach is to check the arrangement of each pair of activities in the program. When checking a pair of activities, we track messages that flow from and into the two activities to see whether there is a dependency relationship between the two activities; and then we check whether the arrangement of the two activities is consistent with the presence or the absence of dependency relationships between the two activities. To implement this idea, we transform a WS-BPEL program to a graph that models the data flow in the WS-BPEL program and use two concise criteria to check activity-arrangement problems. We next present the basics of the WS-BPEL language in Section 3.1, and details of our approach in Section 3.2.

3.1 Basics of WS-BPEL

The latest WS-BPEL standard is Version 2.0 [1]. WS-BPEL 2.0 provides 21 types of activities, among which 9 activity types are about the mechanism of handling faults and the other 12 types are related to operation and control structure. These 12 activity types include basic types (i.e., $receive$, $reply$, $invoke$, and $assign$) and structural types (i.e., $sequence$, $flow$, $pick$, if, $while$, $repeatUntil$, $forEach$, and $scope$). A basic activity allows a business process to communicate with other services or update local messages. A structural activity is used to form the control structure of a business process. We list the semantics of some activity types in WS-BPEL 2.0 [1] as below:

- A $sequence$ activity is used to define a collection of activities to be performed sequentially in lexical order.
- A $flow$ activity is used to specify activities to be performed concurrently.

The semantics of $receive$, $reply$, $invoke$, $assign$, $pick$, if, $while$, $repeatUntil$, and $forEach$ are self-evident and we refer readers to the WS-BPEL 2.0 standard for detailed descriptions of the semantics of the 12 activity types [1].

3.2 Approach Details

Our approach consists of four steps. First, we construct a $message\ propagation\ graph$ to model the data flow in a WS-BPEL program. Second, based on the $message\ propagation\ graph$, we construct an $activity\ dependency\ graph$ to capture the dependency relationships between activities. When two activities have a dependency relationship, they cannot run concurrently. Third, from the $activity\ dependency\ graph$, we further identify a collection of sets of activity sequences such that two activities in different activity sequences in the same set can run concurrently, and otherwise can run only sequentially. Finally, we check the WS-BPEL program to see whether the actual arrangement in the program is consistent with the criteria defined in the fourth step.

Step 1. Constructing Message Propagation Graph
A message propagation graph models message propagation resulted from both the synchronous and asynchronous communications between partner services and the message flow inside a WS-BPEL program. From the WS-BPEL 2.0 standard [1], we know that messages can propagate in a WS-BPEL program in the following three ways:

- A synchronous communication between a WS-BPEL program and a partner web service is described as an $invoke$ activity with one input message and one output message.
- An asynchronous communication between a WS-BPEL program and a partner web service is described as an $invoke$ activity with one input message only and a $receive$ activity with the same $partnerLink$ as that of the $invoke$, or a $receive$ activity and a $reply$ activity sharing one common $partnerLink$.
- The message propagation inside a WS-BPEL program is described as an $assign$ activity, which is used to copy information from messages to messages. We treat the value of the $from$ element in an $assign$ activity as one of its input messages and the value of the to element as one of its output messages. One $assign$ activity may contain one $froms$ element, which may contain more than one pair of $from$ and to elements. So an $assign$ activity can have more than one input message and more than one output message.

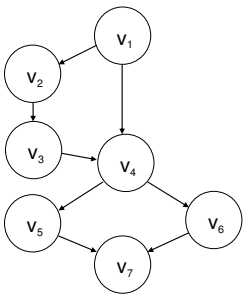

Figure 2. Message propagation graph for $TravelD$ (with labels for messages shown in Table 1)

Table 1. Labels for messages

Label	Message Name
v_1	TravelRequest
v_2	EmployeeTravelStatusRequest
v_3	EmployeeTravelStatusResponse
v_4	FlightDetails
v_5	FlightResponseAA
v_6	FlightResponseDA
v_7	TravelResponse

Based on this observation, we formally define the message propagation relationship in a WS-BPEL program in Definition 1. For simplicity, we use BP to denote the WS-BPEL program, MSG to denote the set of all messages (namely variables) in BP, and ACT to denote the set of all activities of basic types in BP.

Definition 1. (Message Propagation Relationship) Let $m_1, m_2 \in MSG$ ($m_1 \neq m_2$). We say that m_1 propagates immediately to m_2 if one of the following three conditions holds:

1. $\exists\, act \in ACT$, $m_1 \in$ IN(act) $\wedge$ $m_2 \in$ OUT(act).
2. $\exists\, act_1, act_2 \in ACT$ satisfying the following conditions:
   - ATP(act_1) = ATP(act_2);
   - T(act_1) = $invoke$ $\wedge$ T(act_2) = $receive$;
   - IN(act_1) = $\{m_1\}$ $\wedge$ OUT(act_1) = ϕ $\wedge$ OUT(act_2) = $\{m_2\}$.
3. $\exists\, act_1, act_2 \in ACT$ satisfying the following conditions:
   - ATP(act_1) = ATP(act_2);
   - T(act_1) = $receive$ $\wedge$ T(act_2) = $reply$;
   - OUT(act_1) = $\{m_1\}$ $\wedge$ IN(act_2) = $\{m_2\}$.

In Definition 1, IN(act) and OUT(act) are the sets of input and output messages of activity act, respectively; and ATP(act) and T(act) are the partner link and the type of act, respectively.

We denote the relationship that message m_1 propagates immediately to m_2 as $P(m_1, m_2)$, and we define the message propagation graph in Definition 2.

Definition 2. (Message Propagation Graph) Given a WS-BPEL program BP, the message propagation graph of BP is the directed graph $MPG = (MSG, MPR)$, where MSG is the set of all messages in BP and $MPR = \{(m_1, m_2) \mid m_1, m_2 \in MSG \wedge P(m_1, m_2)\}$.

In our approach, Algorithm 1 parses a WS-BPEL program to construct its message propagation graph. As a WS-BPEL program is represented as an XML file and its elements are organized as a tree, Algorithm 1 traverses this tree to collect the set of messages and construct the set of all message propagation relationships.

Algorithm 1. ConstructMPG
Input: A WS-BPEL program (BP)
Output: Message propagation graph (MPG = (MSG, MPR))
$MPR \leftarrow \emptyset$;
$MSG \leftarrow$ GetAllMessage(BP);
for each activity act in BP **do**
  switch (the type of act)
  case $assign$:
    for each pair of $from$ and to in $assign$ **do**
      Let m_1, m_2 be the elements in $from$ and to, respectively;
      if m_1 is a message **then**
        $MPR \leftarrow MPR \cup \{(m_1, m_2)\}$;
    enddo
  case $invoke$:
    if act has both input messages and output messages **then**
      for each $i \in$ IN(act), $o \in$ OUT(act) **do**
        $MPR \leftarrow MPR \cup \{(i, o)\}$;
      enddo
    else if $\exists\, tAct$ in BP, $tAct$ is a $receive$ activity, act and $tAct$ satisfy condition (2) in Definition 1 **then**
      for each $i \in$ IN(act), $o \in$ OUT($tAct$) **do**
        $MPR \leftarrow MPR \cup \{(i, o)\}$;
      enddo
  case $receive$:
    if $\exists\, tAct$ in BP, $tAct$ is a $reply$ activity, act and $tAct$ satisfy condition (3) in Definition 1 **then**
      for each $i \in$ IN(act), $o \in$ OUT($tAct$) **do**
        $MPR \leftarrow MPR \cup \{(i, o)\}$;
      enddo
  case $structural\ activities$:
    Analyze recursively.
enddo
$MPG \leftarrow (MSG, MPR)$;

As the WS-BPEL 2.0 standard treats control structures as activities, Algorithm 1 needs to recursively look into each activity of a structural type to analyze message propagation inside the corresponding structure.

When applying Algorithm 1 to the $TravelD$ program, we obtain the message propagation graph depicted in Figure 2, in which there are seven messages and eight message propagation relationships between them. Table 1 depicts the corresponding messages for the seven nodes in Figure 2.

Step 2. Constructing Activity Dependency Graph

In an activity dependency graph, we consider the following

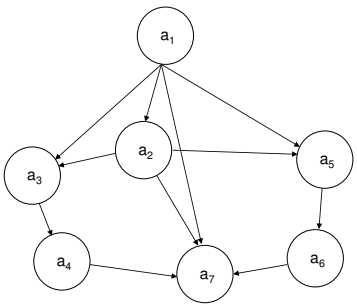

Figure 3. Activity dependency graph for $TravelD$ (with labels for activities shown in Table 2)

Table 2. Labels for activities

Label	(Activity Type, partnerLink, operation)
a_1	(receive, client, TravelApproval)
a_2	(invoke,employeeTravelStatus,EmployeeTravelStatus)
a_3	(invoke, AmericanAirlines, FlightAvailability)
a_4	(receive, AmericanAirlines, FlightTicketCallback)
a_5	(invoke, DeltaAirlines, FlightAvailability)
a_6	(receive, DeltaAirlines, FlightTicketCallback)
a_7	(invoke, client, ClientCallback)

two types of dependency relationship between two activities:

- The result of running one activity impacts that of the other activity.
- The WS-BPEL program explicitly describes a $link$ between the two activities. Based on the WS-BPEL 2.0 standard, if there is a $link$ between activities A and B with A as the source and B as the target, the execution of B should depend on that of A.

We formally define the activity dependency relationship in Definition 3. For simplicity, we use BP to denote the WS-BPEL program, MSG to denote the set of all messages in BP, MPG to denote the message propagation graph, $LINKS$ to denote the set of all $links$ in BP, and $PACT$ to denote the set of all activities with the type of $receive$, $reply$, or $invoke$.

Definition 3. (Activity Dependency Relationship) Let act_1, $act_2 \in PACT$ ($act_1 \neq act_2$), act_2 depends on act_1 if one of the following two conditions holds:

1. $\exists m_1$,..., $m_k \in MSG$ satisfying the following two conditions:
   - (m_i, m_{i+1}) is an edge in MPG $(1 \leq i \leq k-1)$;
   - OUT(act_1) = $\{m_1\} \wedge$ IN(act_2) = $\{m_k\}$.
2. $\exists\, link \in LINKS$ such that $link$ is the source link of act_1 and the target link of act_2.

We denote the relationship that act_2 depends on act_1 as $D(act_1, act_2)$, and we then define the activity dependency graph in Definition 4.

Definition 4. (Activity Dependency Graph) Given a WS-BPEL program BP, the activity dependency graph of BP is the directed graph ADG = ($PACT$, ADR), where $PACT$ is the set of all activities with the type of $receive$, $reply$, or $invoke$ in BP, and ADR = $\{(act_1, act_2) | act_1, act_2 \in PACT \wedge D(act_1, act_2)\}$.

Our approach uses Algorithm 2 to construct the activity dependency graph on the basis of the WS-BPEL program and the message propagation graph.

Algorithm 2. ConstructADG
Input: A WS-BPEL program (BP)
The message propagation graph of BP (MPG)
Output: Activity dependency graph (ADG=($PACT$, ADR))
$PACT \leftarrow \emptyset$;
$ADR \leftarrow \emptyset$;
for each activity act in BP **do**
if the type of act is $receive$, $invoke$, or $reply$ **then**
$PACT \leftarrow PACT \cup \{act\}$;
enddo
for each activity $act_1 \in PACT$ **do**
for each activity $act_2 \in (PACT$ - $\{act_1\})$ **do**
if $\exists m_1 \in$OUT(act_1), $m_2 \in$IN(act_2) and m_1 can reach m_2 in MPG **then**
$ADR \leftarrow ADR \cup \{(act_1, act_2)\}$;
enddo
enddo
for each $link \in LINKS$ **do**
Traverse BP to find activities act_1 and act_2 such that both the source link of act_1 and the target link of act_2 are $link$;
$ADR \leftarrow ADR \cup \{(act_1, act_2)\}$;
enddo
ADG $\leftarrow$ ($PACT$, ADR);

When applying Algorithm 2 to the $TravelD$ program and the message propagation graph depicted in Figure 2, we obtain the activity dependency graph depicted in Figure 3, in which there are 7 activities and 11 activity dependency relationships between them. Table 2 depicts the corresponding activities for the 7 nodes in Figure 3.

Step 3. Identifying Concurrent Activity Sequences

With the activity dependency graph ADG of a WS-BPEL program BP, we generate a collection of sets of activity sequences. Algorithm 3 traverses an $activity\ dependency\ graph$ and constructs these sets. For convenience in describing the algorithm, we give the following definition to capture the property of an activity sequence.

Definition 5. (Maximal Simple Path) We call a path P(u) = $uv_1...v_n$ in a directed graph G = (V, E) a maximal simple path for u with 0 in-degree if the path satisfies any one of following three conditions:

- out-degree(u) $\neq 1 \wedge$ n = 0, namely P(u) = u.
- out-degree(u) = 1 $\wedge$ in-degree(v_k) = 1 $\wedge$ out-degree(v_k) = 1 $\wedge \exists v \in V$ $((v_n, v) \in E \wedge$ in-degree(v) ≥ 2)) $(1 \leq k \leq n)$.

- out-degree(u) = 1 $\wedge$ in-degree(v_k) = 1 $\wedge$ out-degree(v_k) = 1 $\wedge$ in-degree(v_n) = 1 $\wedge$ out-degree(v_n) $\geq$ 2 ($1 \leq k \leq n-1$).

Algorithm 3.
ConstructConcurrentActivitySequenceSets
Input: Activity dependency graph for BP (ADG)
Output: Collection of sets of concurrent activity sequences ($CASS$)

```
tADG ← ADG;
repeat
  tmp ← ∅;
  for each node ∈ tADG do
    if in-degree(node) == 0 then
      Find the maximal simple path P for node in tADG;
      tmp ← tmp ∪ {P};
    endif
  enddo
  CASS ← CASS ∪ {tmp};
  Remove all nodes in tmp and their associated edges from tADG
until tADG is empty;
```

In Algorithm 3, each maximal simple path is presented as an activity sequence in the $CASS$. So applying Algorithm 3 to the activity dependency graph in Figure 3, we obtain $CASS$, the collection of sets of concurrent activity sequences:

$$\{\ \{\langle a_1\rangle\},\ \{\langle a_2\rangle\},\ \{\langle a_3, a_4\rangle,\ \langle a_5, a_6\rangle\},\ \{\langle a_7\rangle\}\ \}$$

For simplicity, we use the labels of the activities here, and the actual activities can be found in Table 2.

Step 4. Detecting Problematic Activity Arrangement

According to Definition 5 and Algorithm 3, we know that activities in the different activity sequences that belong to the same set in $CASS$ can run concurrently, and activities in the same activity sequence or in different activity sequences that belong to different sets in $CASS$ can run only sequentially. Therefore, we formally derive the following criteria for detecting concurrency-related problematic activity arrangement.

- $C1$. Given two different activities act_1 and act_2 arranged as running concurrently in a WS-BPEL program (denoted as ca), let $act_1 \in L_1 \wedge L_1 \in S_1 \wedge S_1 \in CASS$, $act_2 \in L_2 \wedge L_2 \in S_2 \wedge S_2 \in CASS$, if ($L_1 = L_2$) $\vee$ ($S_1 \neq S_2$), then there is a problem with ca.
- $C2$. Given two different activities act_1 and act_2 arranged as running sequentially in a WS-BPEL program (denoted as sa), let $act_1 \in L_1 \wedge L_1 \in S_1 \wedge S_1 \in CASS$, $act_2 \in L_2 \wedge L_2 \in S_2 \wedge S_2 \in CASS$, if ($L_1 \neq L_2$) $\wedge$ ($S_1 = S_2$), then there is a problem with sa.

Criterion $C1$ captures the fact that if two activities belong to the same maximal simple path (see Definition 5) or different set in $CASS$, then they must be arranged to run sequentially; otherwise, running them will produce erroneous results. Thus, arranging such two activities as running concurrently in a WS-BPEL program is problematic. On the other hand, criterion $C2$ captures the fact that if two activities belong to two different maximal simple paths that are contained in the same set in $CASS$, then they can be arranged to run concurrently; otherwise, running them sequentially may compromise performance. Thus, arranging such two activities as running sequentially in a WS-BPEL program is also problematic.

Based on criteria $C1$ and $C2$, we can automatically detect concurrency-related problematic activity arrangement in a WS-BPEL program by leveraging $CASS$ of the program. The basic idea is as follows. For the basic activities in the same $flow$ (or $sequence$) structure, we check them against $CASS$ to determine whether there are activities violating criterion $C1$ (or $C2$). If a violation is found, then we detect a problematic activity arrangement. In addition, a developer can use our approach together with a graphical development tool, such as ActiveBPEL Designer, to further facilitate the development of WS-BPEL programs.

When using our approach to examine $TravelD$ against its $CASS$, we find the following inconsistencies.

- In Figure 1, the $FlightAvailability$ activity (a_3) in Line 21 and the $Flight\ TicketCallback$ activity (a_4) in Line 25 are arranged to run concurrently, since they are in an activity of the $flow$ type. However, as a_3 and a_4 are in the same $activity\ sequence$ $\langle a_3, a_4\rangle$ in $CASS$ of $TravelD$ (see the resulting $CASS$ in Step 3). This case implies that a_3 and a_4 can run only sequentially.
- In Figure 1, the $FlightTicketCallback$ activity (a_4) in Line 25 in the activity of the $flow$ type in Line 19 and the $FlightTicketCallback$ activity (a_6) in Line 33 are arranged to run sequentially due to the activity of the $sequence$ type in Line 1. However, since a_4 is in the activity sequence $\langle a_3, a_4\rangle$, a_6 in $\langle a_5, a_6\rangle$ and these two activity sequences are in the same set in $CASS$ of the $TravelD$ (see the resulting $CASS$ in Step 3). So a_4 and a_6 can run concurrently. Similarly, our approach also detects that the $FlightAvailability$ activity (a_3) in Line 21 and the $FlightAvailability$ activity (a_5) in Line 29 can also run concurrently, but they are arranged to run sequentially in $TravelD$.

These inconsistencies imply that there are activity-arrangement problems in $TravelD$. Examining $TravelD$ carefully, we find that the former is a fault and the latter two are places that can be further optimized. Note that two of the activity-arrangement problems detected by our approach are those discussed in Section 2.

4 Discussion

In this section, we discuss time complexity, assumption, and limitations of our approach.

4.1 Time Complexity

We estimate the time complexity for the algorithms in our approach as follows.

Algorithm $ConstructMPG$ traverses each activity in a WS-BPEL program to construct message propagation relationships. For some specific $invoke$ and $receive$ activities (see Algorithm 1), the algorithm may traverse all other activities in the program to determine whether there exists a message propagation relationship. Thus, in the worst case, its time complexity is $O(n_a^2)$, where n_a is the number of different activities in the WS-BPEL program.

Algorithm $ConstructADG$ checks each pair of activities in a WS-BPEL program to determine activity dependency. The algorithm must traverse the Message Propagation Graph to make the decision. Thus, its time complexity is $O(n_a^2 * n_m)$, where n_a is the number of different activities in the WS-BPEL program, and n_m is the number of different messages used in the program.

Algorithm $ConstructConcurrentActivitySequenceSets$ traverses the Activity Dependency Graph to construct a collection of sets of concurrent activity sequences. Each node in the graph is disposed no more than two times. Thus, its time complexity is O(n_a), where n_a is the number of different activities in the WS-BPEL program.

According to the discussion in Step 4 in Section 3.2, we can easily design an algorithm with time complexity O(n_a^2), where n_a is the number of different activities in the WS-BEPL program, to detect concurrency-related problematic activity arrangement in the WS-BPEL program. Thus, the total time complexity of our approach is $O(n_a^2 * n_m)$, where n_a and n_m have the same meaning as described earlier.

4.2 Approach Assumption

Our approach assumes that both the $message\ propagation\ graph$ and $activity\ dependency\ graph$ for a WS-BPEL program are acyclic directed graphs, and this assumption may not hold for some WS-BPEL programs in practice. However, if a cycle occurs in a $message\ propagation\ graph$, then the cycle may indicate a problem. Based on Definition 1, a cycle in a $message\ propagation\ graph$ indicates that there exist m_1, m_2, ..., m_k such that $P(m_1,$ m$_2)$, ..., $P(m_{k-1}, m_k)$, and $P(m_k, m_1)$, where the activities related to the propagation between m_1 and m_2 and the propagation between m_k and m_1 are not the same. Otherwise, m_1 and m_2 should have the same message type. This case would not occur in a WS-BPEL program in practice. So we could rename m_1 in the latter propagation to a new name distinguished from those of other messages and modify the corresponding affected elements. Doing so could eliminate cycles in the $message\ propagation\ graph$. The similar way can be used to eliminate cycles in an $activity\ dependency\ graph$.

4.3 Limitations

Our approach may have two main limitations. The first main limitation is that our approach may fail to find some $implicit$ activity dependencies in a WS-BPEL program. Wu et al. [12] categorize the dependencies in a business process into four types: $data$, $control$, $service$, and $cooperation\ dependency$. Among these dependencies, the $cooperation\ dependency$ can be recognized by only domain experts and the $service\ dependency$ is imposed by a remote partner service. Our approach treats partner services as black boxes. So if a WS-BPEL program contains such type of implicit dependencies, our approach cannot detect them.

The second main limitation is that our approach may detect in a WS-BPEL program some dependencies that actually do not exist. According to the second condition in Definition 1, if there is an asynchronous "send" activity followed by several "receive" activities from the same partner, then our approach conservatively reports one message dependency for the "send" message and each of the "receive" messages, but not all these reported dependencies may actually exist. This limitation may result in false positives. As our approach reports detected problematic arrangements as warnings, developers can filter out false positives via further inspection.

5 Preliminary Evaluations

We evaluated our approach by applying our approach on six WS-BPEL programs. One is a WS-BPEL program owned by an enterprise. We denote it as PE for convenience. The other five are well-known WS-BPEL programs: $Travel$[3], $marketplace$[1], $loanapprovalcorr$[5], $AsyncEchoClient$[6], and $Correlation$[1]. We present the evaluation results in Section 5.1. Then we discuss threats to validity of our evaluations in Section 5.2.

5.1 Results and Analysis

Table 3 depicts the empirical results in our evaluations. In the table, column "LOC" depicts the number of the lines of code for the six programs, and column "$DAPA$" depicts the number of detected activities with problematic arrangement.

Table 3 shows that our approach detects six activities with problematic arrangement in subject program PE. In fact, program PE contains eight activities that interact with different partner web services. For the confidentiality of the enterprise, we denote these eight activities as a_1, ..., a_8. The eight activities a_1, ..., a_8 were arranged to run sequentially. Our approach finds that three activity sequences (i.e., sequences $\langle a_3 \rangle$, $\langle a_4, a_8 \rangle$, and $\langle a_5, a_6, a_7 \rangle$) in PE can

[5] $http://www.alphaworks.ibm.com/tech/bpws4j$

[6] $http://www.oracle.com/technology/bpel/index.html$

Table 3. Evaluation results

BPEL Program	LOC	DAPA
PE	259	6
Travel	156	0
marketplace	59	2
loanapprovalcorr	122	0
AsyncEchoClient	128	0
Correlation	111	0

run concurrently. But in program PE, the six activities a_3, a_4, a_5, a_6, a_7, and a_8 contained in these three activity sequences were simply arranged to run sequentially. We inspect the program PE carefully and find that these three activity sequences can be arranged to run concurrently. These problematic activity arrangements may compromise program PE performance but would not cause it to produce erroneous execution results.

Table 3 shows that our approach detects two activities with problematic arrangement in subject program $marketplace$. Our approach finds that the activity $submit$ in the partner link $seller$ can run concurrently with the activity $submit$ in the partner link $buyer$, but $marketplace$ arranges them to run sequentially. We inspect the $marketplace$ program carefully and have the following findings. The functionality of $marketplace$ is to serve as a market place. The $marketplace$ program accepts messages from partner web services $seller$ and $buyer$ with common $negotiationIdentifier$, judges whether the $seller$ and the $buyer$ negotiate successfully, constructs an outcome of the negotiation, and finally returns the outcome to the $seller$ and the $buyer$, respectively. The two $submit$ activities described earlier act as the function of returning the negotiation outcome to the $seller$ and the $buyer$, respectively. As the two $submit$ activities do not depend on each other, they actually can run concurrently. This activity-arrangement problem also would not cause $marketplace$ to produce erroneous execution results but may compromise its performance.

Table 3 also shows that our approach reports no problematic activity arrangement in the other four subject programs. We suspect the reason to be that all the subject programs except program PE are examples from the WS-BPEL standard or widely used WS-BPEL development tools, and they are well designed and carefully implemented. Thus, it is not surprising that our approach detects only problematic activity arrangements in two subject programs. In fact, the activity-arrangement problems detected by our approach are minor ones, as the problems would not cause erroneous execution results.

We also conducted another evaluation for our approach. We seeded activity-arrangement problems in the last five programs. For each of these subject programs, we modified the program to obtain several problematic versions by randomly choosing one $sequence$ ($flow$) activity and changing it into a $flow$ ($sequence$) activity. We obtained 19 problematic versions of these subject programs. Our approach detects all the seeded activity-arrangement problems. The results in this evaluation confirm that our approach is actually able to detect activities with dependency misarranged as concurrent and activities without dependency misarranged as sequential.

5.2 Threats to Validity

The threats to external validity primarily include the degree to which the subject WS-BPEL programs, and the problematic activity arrangements are representative of true practice. The five programs in our evaluation are small and widely distributed. The likelihood for them to include activity-arrangement problems is small. They may not be representative for real-world WS-BPEL programs. Moreover, the seeded activity-arrangement problems in the second evaluation may not reflect circumstances in real-world development. These threats could be reduced by more experiments on wider types of subjects, especially real-world WS-BPEL programs.

6 Related Work

Much work in the literature has been done for static analysis of the correctness of composite web services. Researchers used different techniques to model composite web services and checked the resulting models to verify different properties by using model checkers. Comparing with these techniques, our work uses a more lightweight technique without resorting to model checkers to address our specific problem.

Research on verification of composite web services focused on verifying the following properties: safety (Foster et al. [2, 8], Lohmann [6]), liveness (Foster et al. [2, 8]), reachability (Ouyang et al. [9, 10], Qian et al. [11]), deadlocks (Qian et al. [11]), synchronizability and realizability (Xiang et al. [4, 3]), and controllability (Lohmann et al. [7]). Different from these research efforts, our work focuses on detecting concurrency-related problematic activity arrangement in composite web services.

In order to verify the required properties of composite web services, researchers used the following formal representations to model WS-BPEL programs: Petri Nets (Lohmann [6, 7], Ouyang et al. [9, 10]), Guarded Finite State Automata (Xiang et al. [4, 3]), Timed Automata (Qian et al. [11]), Finite State Process (Foster et al. [2, 8, 5]). Our formal representation is based on two directed graphs.

There is also research on identifying parallelism in sequential programs (e.g., Bokhariet [13], Sarkar [14, 15], Yang and Gerasoulis [16]). Although our approach essentially also analyzes parallelism in programs, there are

two main differences. First, our approach works specifically for WS-BPEL programs not for procedural programs. Second, our approach analyzes the contradictions between parallel and sequential constraints in WS-BPEL programs, while existing approaches mainly focus on only parallel constraints.

7 Conclusions and Future Work

In this paper, we present an approach for detecting problematic activity arrangement in WS-BPEL programs. Our approach models data flow in a WS-BPEL program as a $message\ propagation\ graph$ and then constructs an $activity\ dependency\ graph$ of the program from the $message\ propagation\ graph$. Based on the constructed $activity\ dependency\ graph$, our approach deduces a collection of sets of activity sequences (i.e., $CASS$) such that activities belong to different activity sequences in the same set in $CASS$ can run concurrently but can run only sequentially in the other cases. Finally, our approach derives two criteria and detects problematic activity arrangement by examining the activities arrangement in the WS-BPEL program against its $CASS$ based on the criteria. Our preliminary empirical results demonstrate the effectiveness of our approach. We plan to further evaluate our approach by applying it to a variety of WS-BPEL programs in future work.

8 Acknowledgments

The work is supported by the National Basic Research Program of China (973) under Grant No. 2011CB302604, the Science Fund for Creative Research Groups of China under Grant No. 60821003. Tao Xie's work is supported in part by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, and ARO grant W911NF-08-1-0443.

References

[1] A. Alves, A. Arkinand, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, A. Gułzar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. *Web service business execution language v2.0.* Oasis standard, OASIS, 2007.

[2] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *Proc. IEEE International Conference on Web Services (ICWS)*, pages 738–741, 2004.

[3] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. 13th International Conference on World Wide Web*, pages 621–630, 2004.

[4] X. Fu, T. Bultan, and J. Su. Wsat: A tool for formal analysis of web services. In *Proc. 16th International Conference on Computer Aided Verification (CAV)*, pages 510–514, 2004.

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

[6] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. Technical report, Humboldt-Universität zu Berlin Institut für Informatik, 2007.

[7] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting WS-BPEL processes using flexible model generation. *Data & Knowledge Engineering*, 64(1):38–54, 2008.

[8] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs*. John Wiley, Germany, 1999.

[9] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. WofBPEL: A tool for automated analysis of BPEL processes. In *Proc. International Conference on Service Oriented Computing*, pages 484–489, 2005.

[10] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[11] Y. Qian, Y. Xu, Z. Wang, G. Pu, H. Zhu, and C. Cai. Tool support for BPEL verification in ActiveBPEL engine. In *Proc. 8th Australian Software Engineering Conference*, pages 90–100, 2007.

[12] Q. Wu, C. Pu, A. Sahai, and R. Barga. Categorization and Optimization of Synchronization Dependencies in Business Processes. In *Proc. IEEE 23rd International Conference on Data Engineering (ICDE'07)*, pages 306–315, 2007.

[13] S. H. Bokhari. Partitioning Problems in Parallel, Pipelined, and Distributed Computing. IEEE Transactions on Computers, C-37: 48–57, January 1988.

[14] V. Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. IBM Journal of Research and Development, 35(5/6), 1991.

[15] V. Sarkar. Partitioning and Scheduling Parallel Programs on Multiprocessors. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, MA, 1989.

[16] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Transactions on Parallel and Distributed Systems, 5(9): 951–967, 1994.