PAPER

Path Feasibility Analysis of BPEL Processes under Dead Path Elimination Semantics

Hongda WANG[†a)], **Jianchun XING**[†], **Juelong LI**[††b)], **Qiliang YANG**[†], **Xuewei ZHANG**[†], **Deshuai HAN**[†], ***Nonmembers***, ***and*** **Kai LI**[†††], ***Member***

SUMMARY Web Service Business Process Execution Language (BPEL) has become the de facto standard for developing instant service-oriented workflow applications in open environment. The correctness and reliability of BPEL processes have gained increasing concerns. However, the unique features (e.g., dead path elimination (DPE) semantics, parallelism, etc.) of BPEL language have raised enormous problems to it, especially in path feasibility analysis of BPEL processes. Path feasibility analysis of BPEL processes is the basis of BPEL testing, for it relates to the test case generation. Since BPEL processes support both parallelism and DPE semantics, existing techniques can't be directly applied to its path feasibility analysis. To address this problem, we present a novel technique to analyze the path feasibility for BPEL processes. First, to tackle unique features mentioned above, we transform a BPEL process into an intermediary model — BPEL control flow graph, which is proposed to abstract the execution flow of BPEL processes. Second, based on this abstraction, we symbolically encode every path of BPEL processes as some Satisfiability formulas. Finally, we solve these formulas with the help of Satisfiability Modulo Theory (SMT) solvers and the feasible paths of BPEL processes are obtained. We illustrate the applicability and feasibility of our technique through a case study.

key words: *BPEL processes, path feasibility analysis, dead path elimination, BPEL control flow graph, SMT solver*

1. Introduction

Web Service Business Process Execution Language (WS-BPEL or BPEL for short) is one of the most popular standards for developing service-oriented workflow applications [1]. BPEL processes can provide value-added services by compositing Web Services or other BPEL applications. In the path-oriented test case generation of BPEL processes, one commonly assumption is that every BPEL process path is feasible. Such conservative assumption often yields imprecise results, since the existence of infeasible paths for which there is no input data for them to be executed [2]. If a majority of infeasible paths could be detected during static analysis, the performance of software maintenance will be greatly improved, especially the structural testing. Therefore, the analysis of path feasibility is a basis and key issue.

Manuscript received April 5, 2015.
Manuscript revised September 4, 2015.
Manuscript publicized November 27, 2015.

[†]The authors are with College of Defense Engineering, PLA University of Science and Technology, Nanjing, 210007, China.
[††]The author is with Technical Management Office of Naval Defense Engineering, Beijing 100841, China.
[†††]The author is with State Grid Xinjiang Information and Telecommunication Company, Urumqi, 83000, China.
a) E-mail: wanghongda000@126.com
b) E-mail: li_juelong@126.com (Corresponding author)

DOI: 10.1587/transinf.2015EDP7121

Identification of infeasible path in a program is an undecidable problem [2]. To the best of our knowledge, no technique has successfully identified a large number of infeasible program paths. Existing work on detecting infeasible path can be classified into two categories: static analysis and dynamic technique. The former is mainly based on symbolic execution [3]. The techniques based on symbolic execution to identify infeasible paths have been proposed in [4]–[6]. In these techniques, a path is firstly represented by a set of constraints or formulas. The formula is solvable if and only if there are input values which drive the execution of the program down to the path. A theorem prover or a constraint solver is invoked to test the feasibility of the path. However, due to the limitation of symbolic evaluation in handling pointers, arrays and function calls, some of the infeasible paths cannot be detected; a common strategy in dynamic techniques is to limit the number and the depth of search. If the datum that traverses a target path has not been found in the period of the search, the path will be considered infeasible. Generally speaking, dynamic techniques cannot distinguish the feasibility of path precisely. Therefore, dynamic techniques are not suitable for the detection of path feasibility of BPEL processes because of the following reasons [7]: The cost of using a service (for services with access quotas or per-use basis); Service disruptions that might be caused by massive testing; Effects of testing in some systems, such as stock-exchange systems, where each usage of the service means a business transaction.

However, it is believed that, in many production software systems, a large percentage of expressions and predicates are linear [8]. Moreover, our observation is that the variable types related to predicate node of a BPEL process branch are simple and path condition is easily to determine. Therefore, the static analysis is more appropriate for the analysis of path feasibility of BPEL process. Unfortunately, most of the existing static analyses of path feasibility for traditional program languages are also insufficient or inapplicable to BPEL processes, since BPEL processes support both parallelism and *dead path elimination* (DPE). DPE semantic is a technique of propagating the disablement of an activity so that activities downstream do not wait forever for its completion. The propagation is needed, because each activity carries a join condition which is evaluated on the status of the incoming ⟨link⟩. However, it brings difficulties for the analysis of path feasibility. To address this problem, we propose a technique to analyze the path feasibility for

BPEL processes. First, to tackle unique features mentioned above, we transform a BPEL process into an intermediary model — *BPEL control flow graph*, which is proposed to abstract the execution flow of BPEL processes. Second, based on this abstraction, we symbolically encode every path of BPEL processes as some Satisfiability (SAT) formulas. Our SAT formulas are based on transforming BPEL control flow graph into a Concurrent Static Single Assignment (CSSA) form. Finally, we solve these formulas with the help of a Satisfiability Modulo Theory (SMT) solver and the feasible paths of BPEL processes are obtained. A SMT solver solves SAT problems for Boolean formulas containing predicates of underlying theories. Such theories can be, for example, theories of arrays, lists and strings [10]. In addition, a SMT solver can be extended with new theories as shown in [11]. To present our technique, we consider the theory of the linear arithmetic. We illustrate the applicability and feasibility of our technique through a case study.

Our main contributions are two folds. First, considering the unique features of parallelism and DPE semantics, a novel BPEL control flow graph that can abstract the execution flow of BPEL processes is proposed. Second, we apply symbolic encoding to the feasible path analysis of BPEL processes and followed by the subsequent analysis using a SMT solver.

The rest of this paper is organized as follows. In Sect. 2, we give some preliminaries to help readers comprehend our technique. In Sect. 3, we present our technique. In Sect. 4, our technique is evaluated. In Sect. 5, related works are reviewed. In Sect. 6, summary and future works are presented.

2. Preliminaries

Our technique analyzes the path feasibility with SMT solver for BPEL processes under DPE semantics. To make this paper self-contained, in this section, we briefly review the basics of BPEL language [1] and Concurrent Single Static Assignment (CSSA) [15].

2.1 BPEL Language

BPEL is a language that composes partner Web Services into BPEL applications [1]. It defines the process logic through *activities* that can be divided into two classes: basic and structured. Basic activities describe the elemental steps of the process behavior; these activities include ⟨receive⟩, ⟨invoke⟩, ⟨reply⟩, and ⟨assign⟩. Structured activities comprise the basic activities into structures that express control-flow logic of a BPEL application; these activities include ⟨sequence⟩, ⟨flow⟩, ⟨pick⟩, ⟨switch⟩, ⟨while⟩, and ⟨if⟩. Structured activities can contain recursively other basic and/or structured activities.

BPEL describes concurrency and synchronization mechanism by structured activity ⟨flow⟩. A ⟨flow⟩ activity is complete if and only if all activities included in it have been completed. In addition, BPEL expresses synchronization dependencies between activities by ⟨link⟩s. Each activity included in the ⟨flow⟩ activity has optional incoming and/or outgoing ⟨link⟩s. Each ⟨link⟩ associates with a transition condition, and each target activity associates with a join condition. Join condition is a Boolean expression over the status (true, false, unset) of the incoming ⟨link⟩s. Only when all the incoming ⟨link⟩s obtain their status (true or false) can the join condition be evaluated. This activity cannot be executed if the value of the *suppressJoinFailure* attribute is set to "yes" and if the ⟨joincondition⟩ of a target activity (basic activity or structure activity) is false. Then, the status of the target activity's outgoing ⟨link⟩s is set to false. This situation will propagate downstream until a true ⟨joincondition⟩ of an activity is reached and the activity can be executed (although the previous step had been skipped). It is an advanced BPEL mechanism that is somehow "dead path elimination (DPE)" or "awake from the dead".

2.2 Concurrent Static Single Assignment (CSSA)

The Static Single Assignment (SSA) form is an intermediate representation that is used to facilitate program analysis and optimization [12], [13]. The SSA form has the property that each variable is defined exactly once. A *definition* of variable v is an event that modifies v, and a *use* is an event when v appears in an expression (condition or right-hand-side of an assignment).

The SSA form can be characterized through two properties. First, each reference to a name corresponds to the value produced at precisely one definition point giving the single assignment property. The single assignment property is achieved by giving a unique index to each occurrence of the original variable on the left side of an assignment (when it is reassigned). Second, it identifies the points in the computation where values from different control flow paths merge. To ensure the single-assignment property, at a merge point of if-else statements to represent the confluence of multiple definitions in thread-local branches, the construction inserts a new definition at the merge point; its right hand side is a pseudo-function called an φ-function that represents the merge of multiple SSA names. As parameters the φ-function contains all variables written by possible writers. Due to the uniqueness of variable names, there is no need to distinguish between variables and activities. Thus, we use the term "possible writers" also for the variables, which can be uniquely mapped to the corresponding activity.

All concepts of SSA explained so far only hold for sequential programs. Therefore, an extension to the SSA form has been introduced in [14] to handle parallelism in programs — known as *Concurrent Static Single Assignment* (CSSA). The main idea of the CSSA form is that it summarizes the interleaving information for conflicting variables in an explicitly parallel program through the use of π-functions. The values of all conflicting variables are well defined by the π-function at the point where the π-function is placed and is represented via parameters of this function. Like the SSA form, the CSSA form also has the property that all uses of a variable are reached by exactly one assign-

ment to the variable.

3. Path Feasibility Analysis of BPEL Processes

In this section, we elaborate the proposed static analysis method of path feasibility with SMT solver for BPEL processes under dead path elimination semantics.

3.1 BPEL Control Flow Graph (BCFG)

There are many works in path feasibility analysis method for traditional languages. Mostly are based on *Control Flow Graph* (CFG). These models are insufficient for path feasibility analysis of BPEL processes, because BPEL processes have ⟨link⟩ and DPE semantics as mentioned in the previous section.

BPEL Control Flow Graph (BCFG) is proposed as an intermediary model to facilitate the analysis of path feasibility for BPEL processes. It is an extension of CFG, which adds some concurrency related syntax and DPE related semantics. It contains not only structural information, which specifies all control flow information of a BPEL process and data flow information for the analysis of path feasibility, but also semantic information such as DPE semantics. BCFG can be seen as a "partially executed" representation of BPEL processes in the same way as CFG does in traditional sequential program testing. Compared with the BPEL processes, BCFG has the following differences to facilitate path feasibility analysis. First, it unravels the folded structures of BPEL processes (e.g. while loop, dead path elimination) into unfolded structures that are directly used in the path feasibility analysis, especially the dead path elimination. Second, it reduces the quantity of control structure types — some control structures are represented uniformly according to their similar semantics (e.g. ⟨switch⟩ and ⟨pick⟩ both express branching control flow, thus will share the same notation in BCFG). In the transformation from BPEL processes to BCFG, each activity or ⟨link⟩ in BPEL process is transformed into a corresponding node in BCFG or a sub-graph composed by a set of control nodes and edges.

Formally, we define the BCFG as follows:

Definition 1. (BPEL control flow graph, BCFG) *A BPEL control flow graph is a directed graph ⟨N, E⟩, where*

• *N is a set of nodes. There is one entry node and one end node in N while the other nodes denote the activities and predicate expressions.*

• $E \subseteq N \times N$ *is a set of directed edges, and an edge* $\langle n_i, n_j \rangle \in E$ *directed from* n_i *to* n_j *denotes the relationship of control flow between the two activities or predicate expressions represented by* n_i *and* n_j.

Then we map BPEL processes to BCFG structures, which include basic activities (⟨receive⟩, ⟨reply⟩, ⟨invoke⟩, ⟨assign⟩, etc.) and structural activities (⟨sequence⟩, ⟨flow⟩, ⟨while⟩, etc.). The following mapping rules are referenced in the transformation.

Rule 1. The basic activities of BPEL processes are mapped to the BCFG nodes.

Rule 2. The structural activities of BPEL processes are mapped to BCFG nodes and edges. Depending on activity types, several sub-rules are needed, then:

Sub-rule 1: ⟨while⟩.

For loop control flow that may repeat many times, it is common practice to assume a 0-1 criterion in the analysis of path feasibility, where only two samples are used. One is zero repetition, which corresponds to no-execution of the contained activities; the other is one repetition, which corresponds to one execution of the contained activity.

Sub-rule 2: ⟨Switch⟩ and ⟨pick⟩.

First, to facilitate our path feasibility analysis, we should transform the multiple-choice pattern in BPEL processes into exclusive-choice structure in BCFG. In other words, the ⟨switch⟩ and ⟨pick⟩ activities are replaced by the ⟨if⟩ activity. Then, the target ⟨link⟩s of the ⟨switch⟩ activities are mapped to edges that are connected to the decision node, and edges are also added to connect the decision node and the nodes mapped from "case" branches of the switch activity. Similar mapping can be applied to the ⟨pick⟩ activities.

Sub-rule 3: ⟨flow⟩.

The handling of ⟨flow⟩ is the most complex. The ⟨flow⟩-⟨/flow⟩ pair is mapped to a "bar" pair to denote that the including activities of the process execute concurrently. Besides this mapping, there are also similar mappings for activities inside a ⟨flow⟩. In our BCFG, we regard ⟨link⟩ activity as a basic activity and its join condition as decision condition. Given the ⟨link⟩ and DPE semantics [1], there are four cases needed.

Case 1: There is only one ⟨link⟩.

As is depicted in Fig. 1, activities A, B and C denote three basic activities and these three activities are all in the activity ⟨flow⟩. Dotted line denotes ⟨link⟩ and solid line denotes control flow between two activities. The transition condition of B is tc_1, and join condition of B is default respectively. According to the ⟨link⟩ semantics, the case 1 can be depicted in Fig. 1. In this case, we transform ⟨link⟩ into normal control flow nodes and this activity denotes $l_1 = tc_1$, which is followed by a decision activity. This decision activity's condition is l_1.

Case 2: Two ⟨link⟩s connect sequentially.

If two ⟨link⟩s connect sequentially, according to the DPE semantics in the preliminaries mentioned above, the transforming rule can be depicted in Fig. 2.

Case 3: An activity has two incoming ⟨link⟩s.

If an activity has two incoming ⟨link⟩s, according to the ⟨link⟩ semantics, the transforming rule can be depicted

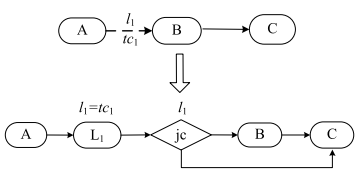

Fig. 1 Illustration of case 1 of sub-rule 3.

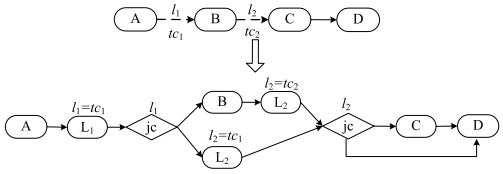

Fig. 2 Illustration of case 2 of sub-rule 3.

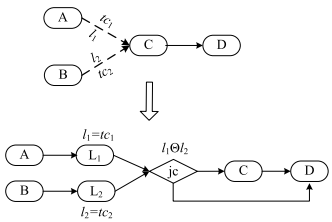

Fig. 3 Illustration of case 3 of sub-rule 3.

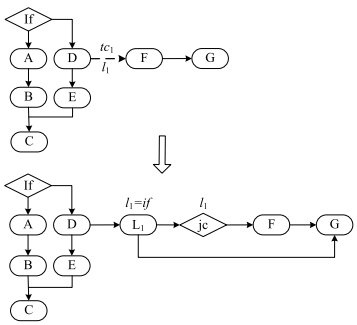

Fig. 4 Illustration of case 4 of sub-rule 3.

in Fig. 3. In this figure, Θ means that the arbitrary relation between l_1 and l_2 (for example, $>$, $\cup$, $\neq$, etc.).

Case 4: A ⟨link⟩'s source activity is included in a predicate structure (⟨while⟩, ⟨if⟩, ⟨pick⟩).

If a ⟨link⟩'s source activity is included in a predicate structure, the transforming rule can be depicted as Fig. 4. In this figure, whether activity F executes or not is decided by a predicate activity (for example ⟨if⟩). Therefore, the activity L_1 denotes that $l_1 = if$ (if denotes the decision condition of predicate activity ⟨if⟩).

The extension of CFG to BCFG is both on syntax (e.g. parallel) and semantics (e.g. DPE) to express concurrency and DPE, which is explained as follows. If the condition associated with a ⟨link⟩ activity is evaluated to be false, which means a dead path, the false value should be propagated downstream till a join node. The condition of the outgoing edges of the join node will be evaluated when all the status of incoming edges have been determined.

Given a BCFG, we symbolically encode every path of BPEL processes as some Satisfiability (SAT) formulas and then apply a SMT solver to determine the path feasibility of BPEL processes. However, our symbolic encoding is only applicable to no choice BPEL processes. Therefore, we need to decompose BCFG into a set of sub-components — *No Choice BPEL control flow graph, NC-BCFG*. NC-BCFG is a graph which has no if-else branch and cycle. The reference [16] has presented an algorithm for this step and readers can refer to [16] for more detailed explanations.

3.2 BPEL Feasible Path Analysis

Given a NC-BCFG, we symbolically analyze its feasible path for formula violations. We express this verification problem as such a SAT formula $\Phi_{\text{NC-BCFG}}$ that it is satisfiable if a feasible path satisfies the formula $\Phi_{\text{NC-BCFG}}$.

1) Constructing CSSA Form for NC-BCFG

Our symbolic analysis is based on transforming a loop-free program (e.g. a NC-BCFG) into a concurrent static single assignment (CSSA) form. Our CSSA form, inspired by [25], has the property that each variable is defined exactly once.

The transformation consists of two steps. Firstly, we rename variables that have more than one definition. Secondly, adding π-functions before shared variable uses to represent the confluence of multiple definitions in different threads. Since each thread in a NC-BCFG has a single thread-local path without branches, φ-functions are not needed in a NC-BCFG.

We construct the CSSA form of a BPEL process as follows:

a) Create unique names for local variables in their definitions.

b) Create unique names for shared variables in their definitions.

c) For each use of a local variable, replace the use with the most recent (unique) definition.

d) For each use of a shared variable v, the most recent definition may not be unique (depending on the interleaving).

• Add a π-function immediately before the use, create a unique name w, and add definition $w \leftarrow \pi(v_1, \ldots, v_k)$;

• Replace the use with the newly defined w.

2) From CSSA to $\Phi_{\text{NC-BCFG}}$

The CSSA form is designed for compiler optimizations where π functions are treated as *nondeterministic choices*. In our SAT encoding, we interpret them precisely in the following paragraphs.

For activities t and t', we use $HB(t, t')$ to express the constraint that t is executed before t'; We define path condition $g(t)$ such that t is executed if $g(t)$ is true.

We construct $\Phi_{\text{NC-BCFG}}$ as follows ($\Phi_{\text{NC-BCFG}}$ = true initially):

a) Program order: For each activity $t \in$ NC-BCFG,

• If t is the first activity in the NC-BCFG, do nothing;

• Otherwise, for each predecessor t' of t in the NC-BCFG, let $\Phi_{\text{NC-BCFG}} = \Phi_{\text{NC-BCFG}} \wedge HB(t', t)$.

b) Actions: For each activity $t \in$ NC-BCFG,

• If t is an activity in the NC-BCFG, let $\Phi_{\text{NC-BCFG}} = \Phi_{\text{NC-BCFG}} \wedge g(t)$.

c) π-function: For each $w \leftarrow \pi(v_1, \ldots, v_k)$, defined in t, let t_i be the activity that defines v_i, let

$$\Phi_{\text{NC-BCFG}} := \Phi_{\text{NC-BCFG}} \wedge \vee_{i=1}^{k}(w = i) \wedge g(t_i) \wedge HB(t_i, t) \wedge \wedge_{j=1, j\neq i}^{k}(HB(t_j, t_i) \vee HB(t_i, t_j))$$

Intuitively, the π-function evaluates to v_i if it chooses the i-th definition in the π-set. Having chosen v_i, all other definitions $j \neq i$ must occur either before t_i, or after this use of v_i in t.

The symbolic encoding of formula $\Phi_{\text{NC-BCFG}}$ directly follows the semantics of NC-BCFG. Therefore, the theorem holds by construction. It is important point that the encoding allows interleavings between different threads to take place, subject only to the HB-constraints added in rules 1 and 3. Since NC-BCFG has a finite size, the formula $\Phi_{\text{NC-BCFG}}$ can be expressed in quantifier-free first-order logic. Note that solutions (interleavings and input) to $\Phi_{\text{NC-BCFG}}$ correspond to feasible paths and test data of BCFG. After encoding all the constraints, we then employ a SMT solver STP [17] to solve them. The SMT solver returns the infeasible paths of each NC-BCFG in the global order. If a solution is found by the solver, it means that we find a feasible path. Since NC-BCFG has a finite size, we can find all the feasible paths through finite times.

4. Evaluation

In this section, let us take the BPEL process borrowed from WS-BPEL 2.0 Primer [1] to demonstrate our approach. The XML-based program segment of this BPEL process is shown in Fig. 5. In this BPEL process, three activities are executed in parallel, i.e. the corresponding Web services would be invoked concurrently.

For intuitive expression, we use UML activity diagrams instituting of BPEL codes (in XML format) to depict this application. In this activity diagram, each node denotes a WS-BPEL activity, and each solid line denotes a transition between two activities. Dashed line denotes synchronization dependency (link with ⟨transitioncondition⟩) between two activities. The UML activity diagram corresponds to Fig. 5 is shown in Fig. 6. All four activities are started concurrently when the ⟨flow⟩ activity starts. The ⟨link⟩ *request-to-approve* has a transition condition that checks whether the part *value* of variable *creditVariable* has a value that is less than 5000. If that is the case, the ⟨link⟩ status of the *request-to-approve* ⟨link⟩ will be set to true, otherwise to false. Since the transition condition of the *request-to-decline* ⟨link⟩ is the exact opposite (greater than or equal to 5000), this means that exactly one of the two successor activities *approveCredit* or *declineCredit* will be executed. Transition conditions offer a mechanism to split the control flow based on certain conditions. Therefore, a mechanism to merge it again must be offered, too. BPEL process does that with join conditions. Join conditions are associated with activities, usually if the activities have any incoming ⟨link⟩s. A *joinCondition* specifies for an activity something like a "start condition", e.g. all incoming ⟨link⟩s must have the status of true in order for the activity to execute, or at least one incoming ⟨link⟩ must have the status true. Let's imagine that an error occurs in activity *approveCredit*, the outgoing ⟨link⟩ of this activity (which is ⟨link⟩ *approve-to-notify*) will be set to false by the execution environment. This may lead to a situation where the join condition is evaluated to false as well.

```
<flow ... >
  <links >
    <link name="request-to-approve" />
    <link name="request-to-decline" />
    <link name="approve-to-notify" />
    <link name="decline-to-notify" />
  </links >
  <receive name="ReceiveCreditRequest"
   createInstance="yes"
   partnerLink="creditRequestPLT"
   operation="creditRequest"
   variable ="creditVariable">
   <sources>
     <source linkName ="request-to-approve" >
       <transitionCondition>
         $creditVariable/value &lt; 5000
       </transitionCondition>
     </source>
     <source linkName ="request-to-decline" >
       <transitionCondition>
         $creditVariable/value >= 5000
       </transitionCondition>
     </source>
   </sources>
  </receive>
  <invoke name="approveCredit" ... >
   <source linkName ="approve-to-notify" />
   <targets>
     <target linkName ="request-to-approve" />
   </targets>
  </invoke>
  <invoke name="declineCredit" ... >
   <source linkName ="approve-to-notify" />
   <targets>
     <target linkName ="request-to-decline" />
   </targets>
  </invoke>
  <reply name="notifyApplicant" ... >
   <targets>
     <joinCondition >
       $approve-to-notify or $decline-to-notify
     </joinCondition >
     <target linkName ="approve-to-notify" />
     <target linkName ="decline-to-notify" />
   </targets>
   </invoke>
  </reply >
</flow>
```

Fig. 5 XML-based BPEL program segment.

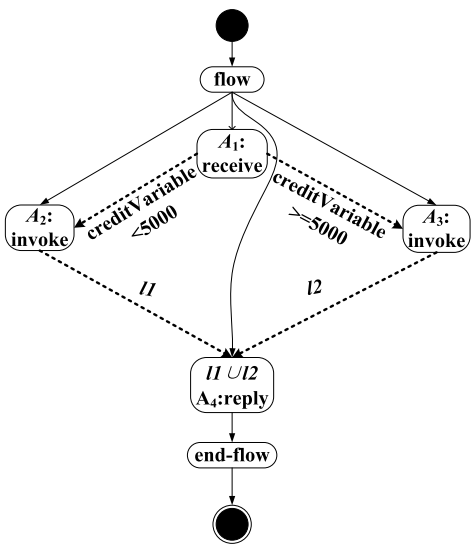

Fig. 6 UML activity diagram of Fig. 5.

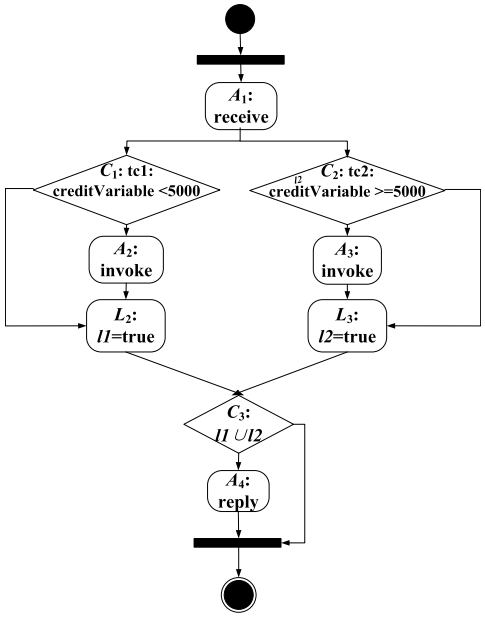

Fig. 7 BCFG with *suppressJoinFailure* is set to false.

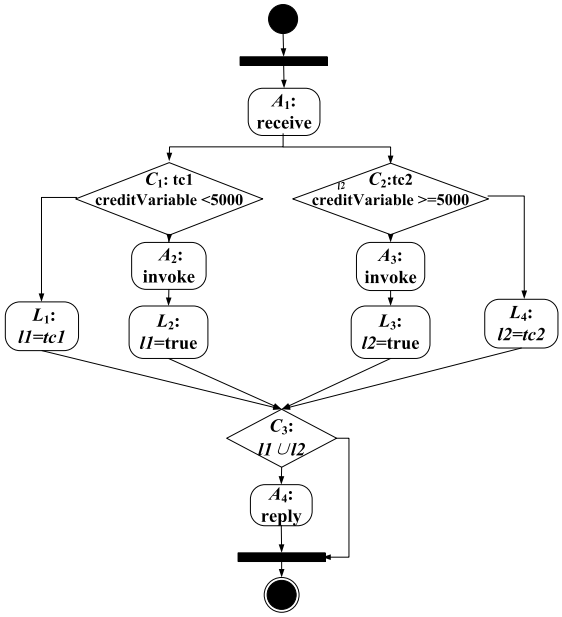

Fig. 8 BCFG with *suppressJoinFailure* is set to true.

By default, a *joinFailure* fault is thrown that may be caught by an appropriate fault handler. Alternatively, when the attribute *suppressJoinFailure* on the process or an enclosing activity is set to yes, the activity associated with the false join condition is skipped and the false ⟨link⟩ status is propagated along ⟨link⟩s leaving that activity. In other words, a false ⟨link⟩ status will be propagated transitively along entire paths formed by successive ⟨link⟩s until a join condition is reached that evaluates to true.

In our experiments, if the attribute *suppressJoinFailure* on the process or an enclosing activity is set to false, we suppose that its ⟨link⟩ status is not propagated along ⟨link⟩s and the activity is skipped. Furthermore, its next activity still can be executed. We separately analyze the attribute by which *suppressJoinFailure* is set to false or true. If not considering DPE (*suppressJoinFailure* is set to false), the BPEL control flow graph can be depicted in Fig. 7.

If considering of DPE (*suppressJoinFailure* is set to true), its corresponding BPEL control flow graph can be depicted in Fig. 8. We employed the traditional technique (Yan et al. [15] and Liu et al. [24]) and our technique (Symbolic encoding every test path of BPEL processes into some formulas as proposed in the last section and then solve it with a SMT solver) to analyze the two BPEL control flow graph. Table 1 lists the statistic result of infeasible paths. In Table 1, the second column means the number of feasible paths of this program; feasible paths refer to the sequential activities contained in the feasible paths. These feasible paths are represented in an implicit way. To facilitate understanding, an informal representation is used. Here, we use A_1A_2 to denote that two activities A_1 and A_2 execute sequentially; $(A_1 \| A_2)$ to denote that two activities A_1 and A_2 execute concurrently.

Observing Table 1, we can find that the feasible paths

Table 1 Result of compare analysis about a typical program.

Technique	Number of feasible path	Feasible path
Traditional technique *suppressJoinFailure=false*	6	$A_1(L_2 \| A_3L_3)A_4$; $A_1(L_3 \| A_2L_2)A_4$
Our technique *suppressJoinFailure=true*	6	$A_1(L_1 \| A_3L_3)A_4$; $A_1(L_4 \| A_2L_2)A_4$

of the two techniques detected contain different activities. In fact, due to DPE propagate the disablement of an activity so that activities downstream do not wait forever for its completion, some activities are "dead". Therefore, our technique outperforms than existing techniques to the path feasibility analysis of BPEL processes because we have taken DPE semantics into consideration. Furthermore, the result also demonstrate that in the path feasibility of BPEL processes, we take the DPE semantics into consideration is necessary.

4.1 Discussion

In our experiments, the external validity mainly originates from the following aspects. First, this paper only chooses some typical processes to test the feasibility of our approach. The test results show that our approach is promising. Like most other empirical studies, the result of our empirical study may not be generalized to cover all cases. With this consideration in mind, we plan to apply our approach to large-scale, evolving BPEL processes in the future.

Another disadvantages of our approach is that it only covers some basic structures and elements of WS-BPEL 2.0. Some advanced activities, e.g., ⟨scope⟩ with fault handing, which are designed to undo the partial and unsuccessful work of a ⟨scope⟩ in which a fault has occurred, have not

been fully considered so far. This is another future work for us to be done.

5. Related Work

In this section, we review some related work on the analysis of path feasibility, and make a comparison between those studies and our work.

First, let's look at some representative strategies of path feasibility analysis techniques on programming-in-the-small languages (e.g. C, C++, and Java) and classify them into two categories, i.e., static method and dynamic method. The static method [3], [18], [19] is regarded as the strategies that mainly based on the symbolic execution and the static branch correlations. The symbolic execution-based technique to detecting path feasibility was proposed by [3]. A path is represented by a set of equations and the equations are solvable if and only if there are some inputs that drive the execution of the program down to the path. Path feasibility is generally detected when these equations are inconsistent with each other. However, the above technique is expensive; furthermore, it cannot behave well for a BPEL program. Since BPEL process has loops, synchronization. The purpose of branch correlations is to determine the feasibility of a target path by investigating the branch correlation of different conditional statements [18], [19]. Although the branch correlation of some conditional statements can be easily determined, it is a difficult task and cannot be done timely and exactly for many cases. It is reported that only about 13% of the analyzable conditional statements show some correlations during compilation. The dynamic method [2], [20]–[22] is regarded as the strategies that if the datum that traverses a target path has not been found in the final period of the search, the path will be considered infeasible. Generally speaking, dynamic techniques cannot distinguish the feasibility of path precisely. Dynamic techniques are not suitable for the detection of path feasibility of BPEL process because of the cost of testing [7].

Second, we review some works on BPEL process testing. There have been many works in path feasibility analysis and they are mostly based on Control Flow Graph. Several works focus on providing systematic methods to generate BPEL process test cases. Yan et al. [23] propose an extended Control Flow Graph (XCFG) to represent a BPEL specification. From the XCFG, all the sequential test paths are generated and then combined to form concurrent test paths. A symbolic execution method is used to extract a set of constraints of the test paths. Finally, a constraint solver is employed to solve the constraints and generate feasible test cases. Yuan et al. [15] describe a graphical model, called BPEL Flow Graph (BFG), to represent the control flow of a BPEL process. By traversing the BFG, concurrent test paths for the BPEL process can be derived. The test data for each path are then generated using a constraint solving method. Liu et al. [24] proposed a structural testing technique for Web service compositions implemented with WS-BPEL process. The technique uses a BPMN-based BCFG to represent the control flow of a WS-BPEL process. Test paths of the process can be derived by traversing the BCFG. Our work is similar to the works of Yan et al. [15] and Liu et al. [24]. All of our techniques from the view of flow graph of BPEL process. The major difference is that we model Dead Path Elimination (DPE) semantics in the BPEL flow graph. Moreover, our modeling method makes our flow graph more intuitive for testers. Our model can facilitate the understanding and analyze path feasibility of BPEL process.

In addition, many intensive studies referred to this problem have been done using the Petri nets, process algebra, or model checking tools. For detecting dead lock and reachability, Dong et al. [26] proposed to use HPNs to describe WS-BPEL applications. They also implemented a tool called Poses++, which was used for automated translation from BPEL to HPN and was also capable of generating test cases. Hummer et al. [27] considered data dependencies between services as potential points of failure and introduced the k-node data flow test coverage metric. They insisted that their approach can help to significantly reduce the number of test combinations. To verify the properties data-bound they defined, Huang et al. [28] proposed the application of model checking to workflow applications. Their model checking technique for workflow applications was based on the process model of OWL-S (Web Ontology Language for Web Services) and the model checker BLAST. Dynamic techniques are used to limit the number and the depth of search. If the datum that traverses a target path has not been found in the period of the search, the path will be considered infeasible. Generally speaking, these dynamic techniques are not suitable for the detection of path feasibility of BPEL processes because of the cost of using a service (for services with access quotas or per-use basis) or service disruptions. Besides, DPE semantics is not considered by these dynamic approaches.

6. Conclusion

Web Service Business Process Execution Language (BPEL) has gradually become the de facto standard for developing instant applications in open environment. BPEL workflow applications are one of the most popular service-oriented workflow applications, and their correctness and reliability have gained increasing concerns. Path feasibility analysis is the basis of BPEL testing, especially the test case generation of BPEL processes. Considering the unique features of parallelism and DPE semantics, this paper presents a technique to analyze the path feasibility of BPEL processes. The case study illustrates that our technique is applicable and feasible.

Our current work only focuses on path feasibility analysis of BPEL processes. Based on this work, in the future, we will broaden our work to address test data generation problem for BPEL processes.

Acknowledgments

The authors are very grateful to the anonymous reviewers for their insightful remarks and helpful comments on an earlier draft. This work was partially supported by the National Natural Science Foundation of China under Grant NO. 61321491 and the Foundation of State Key Laboratory for Novel Software Technology under Grant NO. KFKT2014B12.

References

[1] "WS-BPEL 2.0 Specification" 2007.
Available http://www.casisopen.org download.php/2046/
BPEL process%20V1-1%20Ma%y%205%202003%20Final.pdf.
[2] M.N. Ngo and H.B.K. Tan, "Heuristics-based path feasibility detection for dynamic test data generation," Information and Software Technology, vol.50, no.7-8, pp.641–655, 2008.
[3] J.C. King, "Symbolic execution and program testing," Commun. ACM, vol.19, no.7, pp.385–394, 1976.
[4] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," Int. J. Softw. Eng. Knowl. Eng., vol.11, no.2, pp.139–156, 2001.
[5] A. Santone and G. Vaglini, "Formula-based abstractions and symbolic execution for model checking programs," Microprocessors and Microsystems, vol.28, no.2, pp.69–76, 2004.
[6] B. Pourvatan, M. Sirjani, H. Hojjat, and F. Arbab, "Automated analysis of Reo circuits using symbolic execution," Electronic Notes in Theoretical Computer Science, vol.255, pp.137–158, 2009.
[7] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: A survey," Software Testing, Verification and Reliability, vol.23, no.4, pp.261–313, 2013.
[8] L.J. White and E.I. Cohen, "A domain strategy for computer program testing," IEEE Trans. Softw. Eng., vol.6, no.3, pp.247–257, 1980.
[9] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: Automatically generating inputs of death," ACM Trans. Information and System Security, vol.12, no.2, pp.1–38, 2008.
[10] B. Beckert, T. Hoare, R. Hahnle, D. Smith, C. Green, S. Ranise, C. Tinelli, T. Ball, and S. Rajamani, "Intelligent systems and formal methods in software engineering," IEEE Intell. Syst., vol.21, no.6, pp.71–81, 2006.
[11] G. Nelson and D.C. Oppen, "Simplification by cooperating decision procedures," ACM Trans. Programming Languages and Systems, vol.1, no.2, pp.245–257, 1979.
[12] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," ACM Trans. Programming Languages and Systems, vol.13, no.4, pp.451–490, 1991.
[13] B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Global value numbers and redundant computations," Proc. 15th Symposium on Principles of Programming Languages, pp.12–17, 1988.
[14] J. Lee, S.P. Midkiff, and D.A. Padua, "A constant propagation algorithm for explicitly parallel programs," International Journal of Parallel Programming, vol.26, no.5, pp.563–589, 1998.
[15] Y. Yuan, Z. Li, and W. Sun, "A graph-search based approach to BPEL4WS test generation," International Conference on Software Engineering Advances, pp.1–14, 2006.
[16] J.Q. Li, Y.S. Fan, and M.C. Zhou, "Performance modeling and analysis of workflow," IEEE Trans. Syst., Man, Cybern. A, Syst., Humans, vol.34, no.2, pp.229–242, 2004.
[17] V. Ganesh and D.L. Dill, "A decision procedure for bit-vectors and arrays," Proc. 19th International Conference on Computer Aided Verification, Lecture Notes in Computer Science, vol.4590, pp.519–531, Springer Berlin Heidelberg, 2007.
[18] R. Bodik, R. Gupta, and M.L. Soffa, "Refining data flow information using infeasible paths," Foundations of Software Engineering, Proc. 6th European Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.361–377, 1997.
[19] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra, "Exploiting branch constraints without exhaustive path enumeration," Proc. 5th International Workshop on Worst-Case Execution Time Analysis, pp.46–47, 2005.
[20] N. Malevris, "A path generation method for testing LCSAJs that restrains infeasible paths," Information and Software Technology, vol.37, no.8, pp.435–441, 1995.
[21] P.M.S. Bueno and M. Jino, "Automatic test data generation for program paths using genetic algorithms," Int. J. Softw. Eng. Knowl. Eng., vol.12, no.6, pp.691–709, 2002.
[22] G. Balakrishnan, S. Sankaranarayanan, F. Ivančić, O. Wei, and A. Gupta, "SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement," Proc. International Symposium on Static Analysis, Lecture Notes in Computer Science, vol.5079, pp.238–254, Springer Berlin Heidelberg, 2008.
[23] J. Yan, Z. Li, Y. Yuan, et al., "BPEL4WS unit testing: Test case generation using a concurrent path analysis approach," IEEE International Symposium on Software Reliability Engineering, pp.75–84, 2006.
[24] C.-H. Liu, S.-L. Chen, and X.-Y. Li, "A WS-BPEL based structural testing technique for Web service compositions," Proc. International Symposium on Service-Oriented System Engineering, pp.135–141, 2008.
[25] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," Proc. 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol.6015, pp.328–342, Springer Berlin Heidelberg, 2010.
[26] W.-L. Dong, H. Yu, and Y.-B. Zhang, "Testing BPEL-based Web service composition using high-level petri nets," Proc. 10th International Conference on Enterprise Distributed Object Computing, pp.441–444, 2006.
[27] W. Hummer, O. Raz, O. Shehory, P. Leitner, and S. Dustdar, "Testing of data-centric and event-based dynamic service compositions," Software Testing, Verification and Reliability, vol.23, no.6, pp.465–497, 2012.
[28] H. Huang, W.-T. Tsai, R. Paul, and Y. Chen, "Automated model checking and testing for composite Web services," Proc. 8th International Symposium on Object-Oriented Real-Time Distributed Computing, pp.300–307, 2005.

Hongda Wang received the MS degrees in power system and automation from PLA University of Science and Technology, Nanjing, in 2013. He is currently working toward the PhD degree at PLA University of Science and Technology. His research interests include service computing and workflow technology. E-mail: wanghongda000@126.com

Jianchun Xing received his B.Sc. and M.Sc. degrees in electric system and automation from Engineering Institute of Engineering Corps, China, in 1984 and 1987, respectively, and the Ph.D. degree in Information System Engineering from PLA University of Science and Technology (PLA UST), China, in 2006. He is currently a professor in PLA UST. His research interests include intelligent control, artificial intelligence and information processing. He is a member of the IEEE and the CCF. E-mail: xjc@893.com.cn

Juelong Li received his B.Sc. and M.Sc. degrees in electric system and automation from Engineering Institute of Engineering Corps, China. He is currently a professor in Technical Management Office of Naval Defense Engineering, China. His research interests include intelligent control and information processing. He is a member of the IEEE and the CCF. E-mail: li_juelong@126.com

Qiliang Yang born in 1975, Ph.D. of Nanjing University. He is also an associate professor in PLA University of Science and Technology. His research interests include self-adaptive software systems, mission-critical system and software and pervasive computing, Cyber-Physical System. He is a member of CCF and IEEE. E-mail: yql@893.com.cn

Xuewei Zhang received the MS degrees in power system and automation from PLA University of Science and Technology, Nanjing, in 2014. He is currently working toward the PhD degree at PLA University of Science and Technology. His research interests include service computing and workflow technology. E-mail: zxw19880707@163.com

Deshuai Han born in 1990. He is currently working toward the MS degree at PLA University of Science and Technology. His research interests include self-adaptive software systems, Cyber-Physical System.

Kai Li received the B.S. from Northeastern University, China in 2010. and received M.S. degree from North China Electric Power University, China in 2013. He is currently an engineer of State Grid Xinjiang Information and Telecommunication Company. He is a member of CCF, IEICE.