BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach

Jun Yan[1,2,3], Zhongjie Li[2], Yuan Yuan[2], Wei Sun[2] and Jian Zhang[1]
yanjun@ios.ac.cn, {lizhongj,yyuan,weisun}@cn.ibm.com, zj@ios.ac.cn
[1]Laborary of Computer Science, Institute of Software, Chinese Academy of Sciences
[2]IBM China Research Lab
[3]Graduate University of Chinese Academy of Sciences

Abstract

BPEL is a language that could express complex concurrent behaviors. This paper presents a novel method of BPEL test case generation, which is based on concurrent path analysis. This method first uses an Extended Control Flow Graph (XCFG) to represent a BPEL program, and generates all the sequential test paths from XCFG. These sequential test paths are then combined to form concurrent test paths. Finally a constraint solver `BoNuS` *is used to solve the constraints of these test paths and generate feasible test cases. Some techniques are proposed to reduce the number of combined concurrent test paths. Some test criteria derived from traditional sequential program testing are also presented to reduce the number of test cases. This method is modularized so that many test techniques such as various test criteria and complex constraint solvers can be applied. This method is tested sound and efficient in experiments. It is also applicable to the testing of other business process languages with possible extension and adaption.*

1 Introduction

With the advent of SOA age, business process design, analysis, development and testing is becoming more and more important. Business Process Execution Language for Web Services [1] (abbreviated to BPEL in this paper) defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partner processes.

Mission-critical business solutions need comprehensive testing to ensure that it could perform correctly and reliably in operation. However, in current industrial practice, business process testing focuses on system and user acceptance testing, whereas unit testing [8] has not gained much attention. This is strange, given the fact that unit testing has been prevalent in object-oriented software development, and test generation for both sequential and concurrent programs has a long research history. A possible reason is that BPEL is still a new language, although it is gaining momentum in industry. We expect that BPEL process testing will draw more attention along with the maturation and adoption of SOA and BPEL specification. BPEL unit testing treats an individual BPEL process as the unit under test, and tests its internal logic thoroughly. This paper proposes a method of test case generation for an individual BPEL process.

BPEL has built-in constructs expressing concurrency and synchronization, thus can be seen as a kind of concurrent programs. In the area of testing concurrent programs, many existing research works are based on reachability analysis. A common method is to construct a reachability graph (RG) of the program under test [9]. It represents all the possible execution scenarios resulted from non-deterministic run of the concurrent program. This method is limited in practice due to state space explosion problem. To overcome this problem, our approach will not construct a RG and cover all the serialized paths; instead, we only cover "basic paths", which differ with each other by at least one activity. Our approach is more applicable to programs without complex variable sharing or process interaction patterns, and conforms to BPEL programming common practices.

Some test generation methods based on path analysis are also proposed for testing concurrent programs [6, 13]. These methods are applicable to programs consisting of communicating processes or tasks, like those written in Ada or CSP, but inappropriate for BPEL, which has neither explicit separation of individual processes (tasks) nor synchronization via rendezvous.

Furthermore, BPEL has unique features in both syntax (for example, flow with activity synchronization, join condition) and semantics (for example, dead-path-elimination) that need special treatment.

We introduce a novel BPEL test case generation method in this paper. This method first uses an Extended Control Flow Graph (XCFG) to represent a BPEL program, and generates all the sequential test paths from XCFG. These sequential test paths are then combined to form concurrent test paths. Finally we adopt the idea of symbolic excution method [7] to extract a set of constraints from the test paths and employ a constraint solver `BoNuS` to solve the constraints of these test paths and generate feasible test cases. This method can handle almost all the features of BPEL.

The paper is organized as follows. Some BPEL basics and characteristics will be introduced in the next section, then the proposed test generation method is elaborated in section 3. Some experimental results are shown in Section 4. Section 5 follows with related works and the recapitulation of their differences with our work. Section 6 concludes the paper with future work predictions.

2 BPEL Language

2.1 BPEL Basics

Like any programming language, BPEL has typical control structures including `sequence`, `switch`, `while`, etc. It defines `pick` to await the occurrence of one of a set of events and then performs the enclosed activities, similar to `switch` in branching different control flows. In addition, BPEL uses the `flow` construct to provide concurrency and synchronization. Synchronization between concurrent activities is provided by means of links. Each `link` has a source activity and a target activity. Furthermore, a transition condition (a Boolean expression) is associated with each `link` and is evaluated when the source activity terminates. As long as the transition condition of a `link` has not been evaluated, the value of the `link` is undefined.

Each activity of a `flow` has a join condition. This condition consists of incoming links of the activity combined by Boolean operators. Only when all the values of its incoming links are defined and its join condition evaluates to true, an activity can start. The join conditions can be defined by the following BNF production:

$$c ::= TRUE | FALSE | l | \neg c | c \wedge c | c \vee c | (c)$$

where l is the name of a link. For example, $\neg(\neg l_1 \wedge l_2) \vee \neg l_3$ is a join condition. The join condition is set to be the disjunction of the incoming edges by default.

As a consequence, if its join condition evaluates to false then the activity never starts. If the attribute `suppressJoinFailure` is set as `"true"`, the semantics of a false join condition are to skip the activity and set the status of all outgoing links from that activity to negative and propagate the link status along the entire paths instead of throwing out an exception. This is called dead-path-elimination (DPE).

This `flow` construct brings in a major complexity of BPEL testing with its flexible control flow expressing powers and special semantics.

Figure 1 is an example of BPEL program describing a loan approval process. This process begins by receiving a loan request. For low amounts (less than $10,000) and low-risk individuals, approval is automatic. For high amounts or medium and high-risk individuals, each credit request needs to be studied in greater detail. The use of risk assessment and loan approval services is represented by `invoke` elements, which are contained within a `flow`, and their (potentially concurrent) behavior is staged according to the dependencies expressed by corresponding `link` elements. Note that the transition conditions attached to the links determine which links get activated, all the join conditions use default setting. Finally the process responds with either a “loan approved” message or a “loan rejected” message. Because the loan approval service can return a fault of type “loanProcessFault”, a fault handler is provided in this process. When a fault occurs, control is transferred to the fault handler, where a `reply` element is used to return a fault response of type `unableToHandleRequest` to the loan requester.

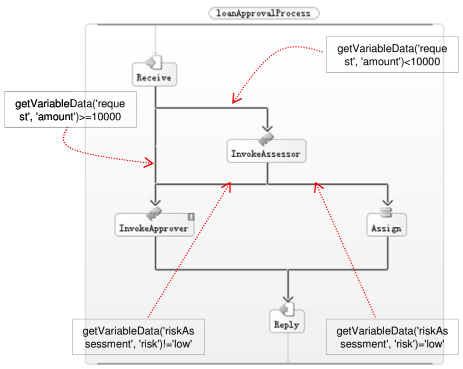

loanApprovalProcess
getVariableData('request', 'amount')<10000
Receive
getVariableData('request', 'amount')>=10000
InvokeAssessor
InvokeApprover
Assign
Reply
getVariableData('riskAssessment', 'risk')!='low'
getVariableData('riskAssessment', 'risk')='low'

Figure 1. A BPEL Example

2.2 BPEL Characteristics

Here we list some BPEL characteristics that are important to our test generation approach. Their relevance will be discussed in later sections.

Event handlers

The whole process as well as each scope can be associated with a set of event handlers that are invoked concurrently if the corresponding event occurs. The business process is enabled to receive such events concurrently with the normal activity of the scope to which the event handler is attached. This allows such events to occur (or not occur) at an arbitrary number of times while the corresponding scope is active. Particularly, a specific message event can occur multiple times.

Shared variables and serializable scopes

When the `variableAccessSerializable` attribute is set to "yes", the scope provides concurrency control in governing access to shared variables between "threads" inside a flow. For those shared variables that are not protected as such, we assume that their read and write order is nondeterministic at process design time, thus the order can be determined at test generation phase. This assumption is viable because it conforms to common use (random, unprotected access to shared variables is dangerous and not desired in process design). However, general handling of shared variables protected by `variableAccessSerializable` attribute is difficult and will not be covered in this paper.

Control cycle

BPEL specifies that a link must not create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity, where an activity A logically precedes an activity B if the initiation of B semantically requires the completion of A. Therefore, directed graphs created by links are always acyclic. So in BPEL program there is no loop of control flow except "while" loop.

3 A Concurrent Path Analysis Approach

Our approach can be divided into four main steps or tasks: XCFG construction, sequential paths enumeration, sequential paths combination and constraint processing. Some existing test coverage criteria and constraint solvers can be used in our testing approach. The approach is visualized in Figure 2.

3.1 Extended Control Flow Graph (XCFG) Construction

The original flow graph of BPEL such as Figure 1 is a direct visualization of the BPEL source code, but it is highly compact and does not express dynamic execution alternatives resulted from concurrency, DPE, exceptions and so on. This is why an Extended Control Flow Graph (XCFG) model is proposed to represent a BPEL program, for the benefit of test path searching.

3.1.1 XCFG

A key point of XCFG is that all the BPEL activities are stored in edges and each edge has a predicate attribute to guard its execution. All the nodes in XCFG are homogeneous, instead of falling into different categories such as control nodes and normal nodes. We give the definition of XCFG similar to EFSM in [15]:

Definition 3.1 (XCFG). The XCFG is defined as a 4-tuple $\langle N, E, s, f\rangle$ where

- N is a set of nodes that represent the program states.
- E is a set of edges that represent the transitions between two nodes. Each transition is a tuple $\langle cn, nn, pr, ac\rangle$ where cn and nn denote the current node and the next node of the transition respectively; pr is a predicate, i.e., a set of conditional expressions. The ac is an action, which records BPEL receiving activities (the receiving direction of a 2-way invocation, receive, etc) and assignment statements that give values to the variables.
- s is the start node and f is the final node, $s, f \in N$.

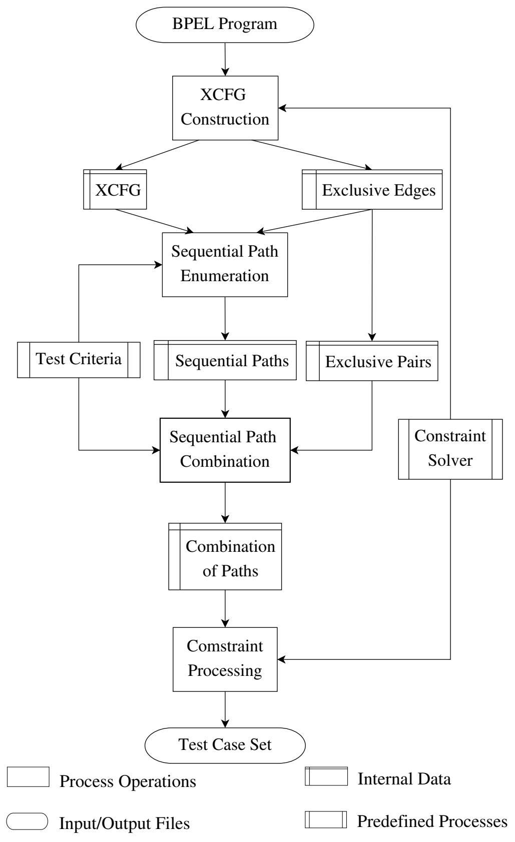


Figure 2. Test Method Workflow

Note that since the XCFG model can express concurrent flow, a state of an execution in XCFG may correspond to more than one node. While in EFSM, each node represents a state.

3.1.2 XCFG Transformation

As stated in Section 3.1.1, BPEL basic activities will be mapped to XCFG edges. Now we discuss the detailed transformation.

Normal logic

The most difficult and important part of the transformation is that of `flow` and `link` constructs. Each activity inside a `flow` could have two kinds of execution states at runtime: executed and skipped, depending on the evaluation of its join condition. Similarly each link could have two kinds of execution states at runtime: normal and DPE, depending on the evaluation of its transition condition. In transformation, each activity or link will be mapped to either a pair of edges - one is **normal edge**, the other is **dead edge**, or a single normal edge. The normal edge and its paired dead edge (denoted as E and E' respectively in this paper) connect the same two nodes. The sketch map is shown in Figure 3, where graph (a) is the original BPEL flow graph, graph (b) is the resulting XCFG segment. For simplification the dead edges are not shown.

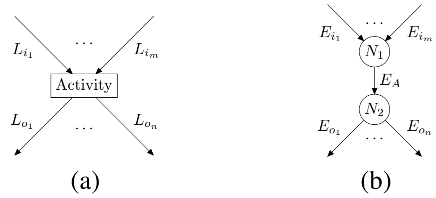


Figure 3. Activity and Link Translation

For a BPEL activity, the ac element of a mapped normal edge is the actions of the BPEL activity, while the ac of a mapped dead edge is empty. The ac of an edge mapped from a link is always empty. In addition, we need to introduce a binary-domain variable es to denote the edge state. We add the assignment $es = TRUE$ to the normal edge's ac and $es = FALSE$ to the dead edge's ac.

For a normal edge E_A mapped from a BPEL activity, the predicate pr_A is translated from the join condition of the activity by replacing the variable l_i with es_{l_i}. For a normal edge E_L mapped from a BPEL link, the predicate is: $pr_L = tr_L \wedge es_A$, where tr_L is the transition condition of the link, and A is the source activity of the link. The predicate of a dead edge is the negation of its paired normal edge.

If an activity has only one incoming link, the mapped edges for the activity and the link can be merged, because the state of the activity is defined by the state of the only incoming link. Consider the graph (a) of Figure 4, the transition condition of link L is tr_L, the join condition of activity A is l_A. Firstly according to the above descriptions, it can be transformed into graph (b), where the dashed lines represent dead edges. Then E_A and E_L, E'_A and E'_L can be merged respectively, resulted in graph (c). The new edge $E_A + E_L$'s predicate is $pr = tr_L$, while the new edge $E'_A + E'_L$'s predicate is $\neg pr$. The ac of a new edge is the mergence of the original two edges. If the join condition is $\neg L$, we need to merge E_A and E'_L, E'_A and E_L respectively.

Since the external services are black-box for us, the ac of an edge mapped from an `invoke` activity is a set of input clauses for the modified variables and some user defined assertions for test oracles.

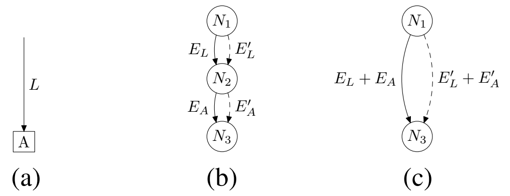


Figure 4. Edges Mergence

The BPEL `switch` construct could be transformed as follows. Firstly, a virtual entry node is added. Secondly an edge is added for each `case` branch, where pr is the expression of the branch, and ac is empty.

The `pick` is transformed in the same way as the `switch`, because it is similar to `switch` in its semantics of selecting one from several branches to execute. The difference is: `pick` does not introduce predicate elements to the XCFG edges; rather, it introduces action elements into the XCFG edges.

The BPEL `while` could cause paths of infinite length. In our work, the loop is transformed to a structure that has no backward edges with 0-1 loop test criterion.

Event handling

The event handlers in a scope, as described in Section 2.2, add concurrent event processing "threads" to the normal activity of the contained scope. Therefore, this is similar in effect to a "thread" in a `flow` construct, and could be transformed in a similar way. The difference is: event handlers allow a specific message event to occur multiple times, possibly in parallel. Multiple enablements of a single message event will bring in complexity in variable sharing. To simplify the discussion, this paper limits its maximum enablement to one time. More complex handling will be our future work.

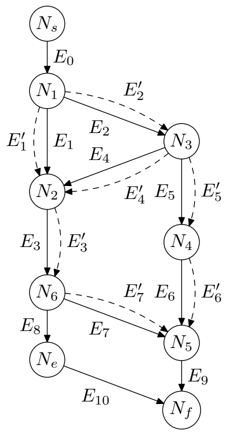
Fault handling

For an activity that can throw a fault, we add an extra edge from the destination node of the edge mapped from the activity to the source node of the edge mapped from the related `faultHandlers` activity. Often, the exception handling logic will cause a termination of the whole program, and the states of other running "threads" are undeterminate. In this paper, to simplify discussion and focus on main points, we assume that the exception handling logic will not affect the other running "threads", which will run to completion undisturbedly.

3.1.3 An XCFG Example

Figure 5 shows the XCFG of the example process in Figure 1. All dashed edges are dead edges. N_s and N_f represent the virtual start and final node respectively.

Compared with the orignal BPEL flow graph, XCFG has the following differences to facilitate test path searching.



```
E0:  receive
E1:  receive-to-approver
link
E2:  InvokeAssessor
E3:  InvokeApprover
E4:  assessor-to-approver
link
E5:  Assign
E6:  Assign-to-reply link
E7:  approver-to-reply link
E8:  fault thrown
E9:  reply
E10:  fault handler
```

Figure 5. XCFG

Firstly, it unravels the folded structures of BPEL (e.g. while loop, dead path elimination) into unfolded structures that are directly traversable in graph searching. Secondly, it turns implicit, disjoint control flows (e.g. exception handling) into explicit, connected control flows. Thirdly, it eliminates all the controlling structures including `while`, `switch`, `flow` and so on. All the nodes of XCFG are homogeneous. What control logic (switch, concurrent, etc) is feasible will be determined based on data constraints analysis or simple rules.

3.2 Exclusive Edges

The relation of two edges in XCFG is determined by their predicates. A simple relation is **Exclusive**. Two edges are exclusive if they cannot be executed in parallel at runtime. We use (E_1, E_2) to denote exclusive edges. Exclusive edges can be identified during the XCFG construction. There are two types of exclusive edges.

The first type exists between two edges s.t. one edge's terminus is the other one's origin. Consider, for example, a node of graph (a) in Figure 6 that has only one outgoing normal edge. If the join condition of the original activity is in a disjunctive form jc (i.e. the edge's $pr = jc$): $jc = l_1 \vee \ldots \vee l_m$ where $l_1 \ldots l_m$ are the incoming links of the activity, we have $(\neg jc \wedge l_1) \vee \ldots \vee (\neg jc \wedge l_m) = FALSE$. This implies that the dead activity edge E'_A is exclusive with all the incoming normal link edges. Otherwise, if the join condition is in conjunctive form, the normal activity edge E_A is exclusive with all the incoming dead edges. Similarly, for a node of graph (b) with a single incoming edge, the dead edge E'_I of the activity or the incoming link is exclusive with all the outgoing normal edges.

The second type exists between two edges with the same origin node. The following rules could be used to collect this type of exclusive edges: 1) The common origin is a

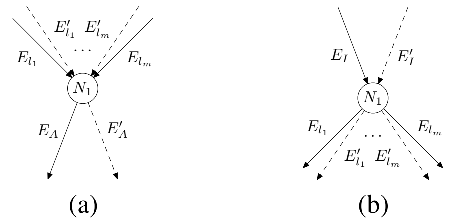


Figure 6. The First Type Exclusive Edges

node mapped from a `switch` or `pick` activity; 2) One edge is the other one's paired dead edge; 3) One edge is the other one's exception edge; 4) The predicate conjunction of the two edges is reported to be unsatisfiable by a constraint solver.

Example 3.2. For example, in the XCFG of Figure 5, (E_1, E'_3),(E'_2, E_4) are exclusive edges of the first type. If N_3 corresponds to a switch node, then E_4 and E_5 are exclusive. E'_1 and E_1 are exclusive since E'_1 is the dead edge of E_1. Also E_8 is exclusive with E_7 since it's an exception edge. If the transition conditions of E_1 and E_2 are `"getVariableData('request', 'amount') >= 10000"` and `"getVariableData('request', 'amount') <= 20000"` respectively, then E'_1 and E'_2 are exclusive. This conclusion can be drawn by manual calculation or invoking a constrain solver (Please refer to Section 3.6.3 for more about constraint solving).

3.3 Sequential Path Enumeration

An XCFG is a directed graph and a sequential path is defined as in the graph theory. Each path begins with the start node and ends with the final node. Note the "sequential path" here is only a graphical concept and it may be not a executable one. With some restrictions (e.g. test criteria, refer to section 3.5), a simple BFS (Breadth First Search) or DFS (Depth First Search) algorithm could be used to find a finite set of sequential paths.

The first type of exclusive edges are used to filter some invalid sequential paths. For instance, any path that begins with $E_0E_1E'_3$ in Figure 5 is invalid for E_1 and E'_3 are exclusive.

3.4 Sequential Path Combination

The **combination** of m paths $\{p_1, \ldots, p_m\}$ means that the state transitions of these m paths in XCFG should be executed jointly at runtime while the state transition of the other paths are not. We use $p_1 | \ldots | p_m$ to denote the combination of these m sequential paths. The value m is called the **scale** of this combination. In other words, we can think the test path $p_1 | \ldots | p_m$ as the "concurrent" running of these m paths at run time. Therefore, the combined paths could be called concurrent test paths (note that this is actually a simplified statement, the sequential paths of a combined path are not necessarily concurrent, as can be seen later).

3.4.1 Exclusive Pairs

There may be some conflicts in combination. For instance, if two paths correspond to different branch of a switch node, they cannot run concurrently and thus cannot be combined. we can use Exclusive Pairs to describe such a confliction. Two sequential paths p_1 and p_2 that contain two exclusive edges are **Exclusive Pairs**, denoted as (p_1, p_2).

3.4.2 Combination Algorithm

Since a set of sequential paths containing exclusive pairs cannot be combined, we can use a dynamic combination procedure to make use of exclusive pairs to filter many invalid combinations. The pseudo-code of the combination algorithm is shown in Figure 7, where **IsCombinable()** is a function that judges whether a sequential path can be combined with a composed path by utilizing the **exclusive pairs** rule set in Section 3.4.1. Here we use Test Set to store the

```
Sequential Paths $SP[MAX\_PATH]$;
Test Set $TS = \{\epsilon\}$;
void PathComb($SP$, $TS$) {
    for($i = 0; i < MAX\_PATH; i++$){
        $TS_i = \phi$;
        for each element $p$ of $TS$
            if (IsCombinable($p$, $SP[i]$)) add $p|SP[i]$ to $TS_i$;
        $TS = TS \cup TS_i$;
    }
}
```

Figure 7. Sequential Path Combination

combined paths. The element ϵ is an empty path. For any path p, we have $p|\epsilon = p$. This element is only used for convenience of the combination algorithm.

The naive combination will inevitably encounter path number explosion. Let N_{SP} to denote the number of sequential paths. We have to combine $2^{N_{SP}}$ times in the worst situation. It will be beneficial to put some additional restrictions on the combination in order to reduce the combination times.

3.4.3 Concurrent Path Scale Restriction

In fact, we do not need to combine arbitrary number of sequential paths from 1 to N_{SP}. Consider C sequential paths which originate from a node and can run concurrently, if the origination node is inside a `sequence`, then only one outgoing edge can be executed, if it's inside a `flow`, each normal edge or the paired dead edge must be executed. For an exception node, the exception edge or the other following activities will be executed. We can use the following algorithm to estimate the Lower-bound C_{LB} and the upper-bound C_{UB} of the scale. We use BPEL program to calculate the values instead of XCFG.

First we estimate the lower-bound. Note that the parameter "node" designates a basic activity, a structured activity, a scope construct, an event handlers construct, a fault handlers construct, an OnMessage clause or a switch branch. For the present, we omit compensation handlers. In counting the child nodes of event handlers and pick, we'll not include OnAlarm clauses for simplification.

```
int CLB(node $N$){
    **switch**($N$){
        **case** a basic activity without followers:
            /* terminate, final reply */
            return 1;
        **case** scope, OnMessage, a switch branch:
            return the $C_{LB}$ value of its sequence or flow child
            node plus the $C_{LB}$ value of its event handlers child
            nodes;
        **case** flow, event handlers:
            return the sum of $C_{LB}$ values of its child nodes;
        **case** a basic activity:
            return the sum of the $C_{LB}$ values of its followers;
        **otherwise**:
            /* sequence, switch, while, fault handlers, pick,
            other basic activity */
            return the minimum $C_{LB}$ of its child nodes;
    }
}
```

The upper-bound C_{UB} can be estimated with a similar method. CLB(s) and CUB(s) return the lower- and upper-bound of the valid test paths' scale, where s is the start node.

The strategy of scale can be applied to our combination algorithm by setting a restriction to each item p of TS. If $C_{UB} \ll N_{SP}$, the total number of test paths will be reduced to (without consideration of exclusive pairs)

$$\sum_{C_{LB} \le r \le C_{UB}} \binom{N_{SP}}{r} = O({N_{SP}}^{C_{UB}}).$$

Taking into account the effect of exclusive pairs, this number will decrease more (Please refer to Section 4 for experimental results).

3.5 Test Coverage Criteria

The default test coverage criterion used in this paper is to test all the possible paths, with a 0-1 criterion for the loop construct. When testing big programs, we can make use of some traditional test coverage criteria for sequential programs to reduce the number of sequential paths. Here we only describe two test criteria.

3.5.1 Basis Path Coverage

McCabe etc. [11] defined a structural complexity called **Cyclomatic Complexity** for sequential programs based on the control flow graph (CFG). The complexity is defined to be $\sum(m_i - 1) + 1$ where m_i is the out-degree of each decision node. A subset of all the paths (the number equals to the cyclomatic complexity) is selected out to achieve a so-called **basis path coverage**. This subset is called a basis path set. This coverage criterion can use very few tests to cover all the branches of the tested program.

This criterion can be extended for BPEL testing. We first enumerate a basis path set of the XCFG, and then combine these basis paths to concurrent test paths. The resulted coverage can be called **basis concurrent path coverage**. The size of this test set will be much smaller than that of the all path coverage criterion.

Note due to the exclusive edges, most combinations are infeasible. Thus the feasible test path may not cover all the edges of XCFG. Please refer to Section 4 for experimental results.

3.5.2 User Directed Path Coverage

Instead of using an existing test coverage criterion, users may want to prescribe coverage goals by themselves. For example, a common goal is to find all the test paths that cover the exception handling logic. We can easily apply these criteria to our method using the following steps. Some experiment results can be found at Section 4.

1. Collect the sequential paths covering those edges into a set CP, the other sequential paths which are not exclusive with the elements of CP form a set NCP.
2. PathComb(CP, TS);
3. Remove ϵ from TS, then we will not get any combined path that contains no paths from CP;
4. PathComb(NCP, TS).

3.6 Constraint Processing

The combined test paths cannot be executed yet and not all of these paths are feasible. A test path is **infeasible** if there are no initial values of variables that can make the program run along this path. All the variables in the paths are described as a set of constraints and assignments. Whether a test path is feasible can be determined by solving the constraints, and if there exists a solution, an initial set of values could be assigned to the variables for test execution.

3.6.1 Variable Sharing Treatment

The XCFG expresses a partial order relation of some edges. For those edges that have no ordering relation, i.e., concurrent, if these edges contain write and read operation to the same variable, it will cause nondeterminism of execution. To overcome this problem, a method is to predefine the r-w order of involved edges. Consider an example of a program

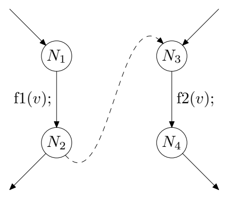


Figure 8. R-W order

segment in Figure 8. Suppose that at least one of the functions f1 and f2 modify the variable v. If f1 should be executed before f2, we can add an edge from N_2 to N_3 in the XCFG to ensure this ordering is used in test data generation. This edge is displayed as a dashed line in the figure.

If the r-w order is not predefined, suppose there are n threads visiting a variable wherein m threads modify the value, then there are $m!(m+1)^{n-m}$ possible r-w orderings, and during test data generation, we will generate different test data for the same test path according to different r-w orderings. However, uncontrolled variable sharing is prone to program problems, thus should be avoided in programming.

3.6.2 Constraints Collection

The paper [14] introduces a backward substitution method to collect the constraints from sequential paths. That method processes the variable assignments one by one from the last to the first, and replaces all the affected variables in the predicates after the assignments. Then it collects all the conditions as a constraint set. The solution for this constrain set is the initial set of values for the variables that makes the path feasible. For example, consider an assertion `x >= 0` following an assignment `x = x + 1`, using the backward substitution, we can get `x + 1 >= 0`.

We extend this method to our concurrent paths. Firstly, the predicates of XCFG edges are stored to its nodes, as a preparation for the main procedure of collecting constraints, using the following "PredicatesConcentration" algorithm.

```
void PredicatesConcentration(node $N$, test path $p$){
    for each outgoing edge $E$ of $N$
        if ($E \in p$) add $pr_E$ to $N$'s constraint set;
        else add $\neg pr_E$ to $N$'s constraint set;
}
```

This algorithm is run for each concurrent path to process all the nodes on this path, resulting in a path whose nodes contain all the predicate information.

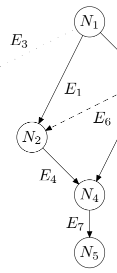
Then we use the backward substitution algorithm in Figure 9 to collect constraints for a test path. Recall that an edge's action is a list of assignments. We use each assignment of an edge to substitute the affected variable in the constraint sets of the nodes after the edge. The "nodes after an edge" include the terminus node of the edge and all the nodes reachable from the terminus node. The main idea of this algorithm is that an assignment cannot be processed until all the followed assignments have been used for substitution.

```
Constraint Set CCC(test path $p$){
    Constrain Set $S = \Phi$;
    $N$ = the final node, mark $N$ as ENABLED;
    while(TRUE){
        for each incoming edge $E$ of $N$ {
            use the assignments of $E$, from last to first, to
                substitute the affected variables in the constraint sets of the nodes after $E$;
            mark $E$ as PROCESSED;
            if ($E$'s origin $N_E$'s all outgoing edges are PROCESSED)
                mark $N_E$ as ENABLED;
        }
        if (there exists an ENABLED node $N'$ that have incoming edges marked with not PROCESSED)
            $N = N'$;
        else break; /* All the incoming edges of all the ENABLED nodes become PROCESSED */
    }
    merge all the constraint sets of nodes to $S$;
    return $S$;
}
```

Figure 9. Constrains Collection

This algorithm processes each edge exactly once, and each edge's assignments are used to process all the nodes after the edge, therefore the time complexity of this algorithm is $O(ne)$ where n and e are the number of nodes and edges respectively.

Example 3.3 *(Constraints Collection).* Consider a path in Figure 10, the dotted edge E_3 represents the XCFG edges that are not contained in the path. E_6 is an empty edge that controls the r-w order. We use the C-like expressions to



```
E1 = {@(x + y >= 0)};
E2 = {@(y > 0), x++};
E3 = {@(x + y < 3)};
E4 = {x++, y++};
E5 = {@(x < 5)};
E6 = {};
E7 = {@(x + 2y > 0)};
```

Figure 10. A Test Path

describe the edge predicates and actions, where those expressions followed by an "@" represent predicates.

The constraint sets of the nodes are:

```
S_N1 = {x + Y >= 0, y > 0, !(x + y < 3)};
S_N2 = {}; S_N3 = {x < 5};
S_N4 = {x + 2y > 0};
```

Then the substitution can begin. After processing these edges in the order: $E_7, E_4, E_5, E_6, E_1, E_2$, we will get the constraint set:

```
S = {x + y >= 0, y > 0, !(x + y < 3), x + 1
< 5, x + 1 + 1 + 2(y + 1) > 0}
```

We can get a suite of initial values for the variables {x,y} by solving the constraint set S to make the BPEL program executed along this path.

3.6.3 Constraint Solving

The purpose of constraint solving for program testing lies in two aspects: judge the feasibility of a test path, and generate a set of initial values for the variables. The solver used to check the feasibility should be complete, i.e., if the constraint is satisfiable, the solver can definitely give us a set of solutions; if the constraints set is unsatisfiable, the tool can tell us that it cannot find any solution. For most of BPEL programs, each constraint is a Boolean combination of primitive constraints, while a primitive constraint is a Boolean variable or a comparison between two numeric expressions. Deciding the satisfiability of this problem is NP-complete. However, for many real instances, this problem can be solved in a reasonable amount of time. The paper [14] introduces a extension of a Boolean satisfiability checker `BoNuS` which calls `LP_Solve` [2] to solve linear numeric constraints. For example, we can apply this solver to the result of Example 3.3 and get a solution {`x = 1, y = 3`} in less than 1 millisecond on a Pentium IV 3GHz PC.

4 Experimental Results

We applied the proposed test generation method to the example of Figure 1. We introduced an error by replacing the transition condition of the incoming link of `InvokeAssessor` with `getVariableData('request','amount') < 20000`. This modification introduced an additional concurrency into the original program. Now the two links originating from the `Receive` activity would become non-exclusive because their transition conditions have overlay now so that they could be enabled in parallel.

Firstly we tried to get all the possible test paths. We got totally 14 sequential paths and $C_{LB} = C_{UB} = 3$. Then we obtained 57 combined paths. Only 9 combinations' scales are 3. All these 9 paths are feasible and 4 feasible paths cover the bug pattern $E_1|E_2$ that is deliberately introduced.

Then we used a user directed coverage criterion that prescribes the coverage of the exception edge E_8. Four sequential paths contain this edge. The valid test path number is 4.

At last we tried the basis path criterion. We got 9 basis sequential paths and 3 valid test paths. Many combinations are infeasible. Two feasible test paths cover the bug pattern $E_1|E_2$.

5 Related Works

Testing concurrent programs presents new testing problems and difficulties that cannot be solved by regular sequential program testing techniques. Concurrent programs are characterized by parallel computation and event synchronization. There could be multiple control flows in a program run, thus a test path could also be concurrent.

A Control Flow Graph (CFG) is a static representation of a sequential program that represents all alternatives of control flow. Based on CFG, various forms of control-flow and data-flow testing metrics have been defined, [3]. However, for concurrent programs, to our best knowledge, there are no such simple graphs for analysis. Petri-Nets cannot easily represent data handling logic. The work [5] uses UML 2.0 activity diagram as a formal base for model analysis rather than code analysis. For a specific programming language, UML activity diagram tends to be limited in that some program features cannot be expressed in a straightforward way. For example, BPEL flow construct allows multiple-choice style workflow pattern [10], which cannot be represented by a simple UML activity structure.

Due to runtime randomness of the scheduler, communication, and other factors, the exact behavior of a concurrent program is usually nondeterministic. For the same set of input data, several runs of the program can produce different permutations of concurrent computation activities, generally called interleavings. Some bugs, e.g. race conditions and deadlocks, can only reveal themselves under a specific interleaving that is hard to be forced to execution. Most research works focus on generating all the interleavings. The work [9] extends the notion of structural testing criteria of sequential programs to concurrent programs, and proposes a hierarchy of coverage criteria including concurrency state coverage, state transition coverage and synchronization coverage. These criteria are defined based on selecting a set of paths from a concurrency graph This approach is limited in practice by state space explosion. The paper [12] presents four different test generation methods, to effectively generate a small set of test sequences that cover all the nodes (edges) in a RG. Although this method achieves test reduction, it still requires constructing a RG. The paper [4] is a typical work on using timing heuristic to force context switches at concurrent events based on some decision function designed to find a well-classified concurrent bug patterns. It assumes that testers have created a good test suite that covers program input, so it does not deal with test case generation, but it is complementary to ours in that the test cases derived by our method can be used as input to their work.

Our approach doesn't construct a RG or enumerate all the interleavings; we only cover some "basic paths". The benefit of our approach is three-fold: 1) the state space explosion problem is avoided by some techniques. 2) it is especially applicable for programs in which concurrent computation units have only very few or no shared variables or other types of synchronization, thus there is little interference between the parallel activities and permutation is useless. 3) Basic paths can be executed directly with the support of test notation and execution environment . 4) Also basic path could act as a foundation for advanced refinement to detect more concurrency-related bugs. One refinement is to rely on the runtime but add assertion-like business logic constraints to enhance the verification logic in a test case (effect: reduce the valid behavior space) . Another is to disturb the runtime scheduler to force different interleavings, as is done in [4] (effect: increase the chances of exercising more interleavings). These two refinements are better to be used in combination.

As far as we know, there are no other BPEL test case generation work. Li et al. [8] proposes a BPEL unit test framework, but does not touch test case generation.

6 Conclusion

In this paper, we have proposed a novel BPEL test case generation method that can effectively handle BPEL concurrent features. An Extended Control Flow Graph (XCFG) is defined to represent a BPEL program. Then all the possible sequential paths are searched from the XCFG, and combined into concurrent test paths. We have introduced three techniques: exclusive edges, exclusive pairs and combination scale strategy to alleviate the path number explosion problem. We also provide a technique to process constraints to generate test data for test paths.

A significant advantage of our method is that it is modularized so that it can be used together with other testing technologies. For example, different test coverage criteria are supported. In addition, it has the potential to be applied to other business process languages including BPMN, WfXML, XPDL, XLANG, WSFL, etc [1], with possible adaption and extension to deal with their specific features. Also our work can be used in regression testing by reusing the sequential paths and combined paths of the old version of the program under test.

Future works are needed to improve the proposed method. Firstly, more techniques are needed to rule out the redundant path combinations for complex programs and improve the combination algorithm. Secondly, some advanced BPEL features (mainly the complex scope nesting) are left to be explored in future.

[1]Process Markup Languages. `http://www.ebpml.org/status.htm`

References

[1] *Business Process Execution Language for Web Services (BPEL4WS)*. Available at `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`.

[2] M. Berkelaar. `lp_solve`, a public domain Mixed Integer Linear Program solver, available at `http://groups.yahoo.com/group/lp_solve/`.

[3] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[4] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS'03*, page 286, 2003.

[5] V. Garousi, L. Briand, and Y. Labiche. Control flow analysis of UML 2.0 sequence diagrams. Technical report. Available at `http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-05-09.pdf`.

[6] T. Katayama, E. Itoh, and Z. Furukawa. Test-case generation for concurrent programs with the testing criteria using interaction sequences. In *Proceedings of the 6th Asian-Pacific Software Engineering Conference*, pages 590–597, December 1999.

[7] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, July 1976.

[8] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS unit testing: framework and implementation. In *Proceedings of 2005 IEEE International Conference on Web Services (ICWS'2005)*, volume 1, pages 103 – 110, 11-15 July 2005.

[9] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March, 1992.

[10] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[11] A. H. Watson and T. J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Computer Systems Laboratory, National Institute of Standards and Technology, September 1996. NIST Special Publication 500-235.

[12] W. E. Wong, Y. Lei, and X. Ma. Effective generation of test sequences for structural testing of concurrent programs. In *Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 539–548, 2005.

[13] R. D. Yang and C. G. Chung. A path analysis approach to concurrent program testing. *Information and Software Technology*, 34(1):43–56, 1992.

[14] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering & Knowledge Engineering*, 11(2):139–156, 2001.

[15] J. Zhang, C. Xu, and X. Wang. Path-oriented test data generation using symbolic execution and constraint solving techniques. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 242–250, 2004.

Appendix: Experiments

We tested the modified version of the BPEL program of Figure 1. Here we list the results. We get 14 sequential paths:

$$\begin{array}{ll}
p_1 = E_0E_1E_3E_7E_9; & p_9 \ = E_0E_2E_4'E_3E_8E_{10}; \\
p_2 = E_0E_1E_3E_8E_{10}; & p_{10} = E_0E_2E_4'E_3'E_7'E_9; \\
p_3 = E_0E_1'E_3E_7E_9; & p_{11} = E_0E_2E_5E_6E_9; \\
p_4 = E_0E_1'E_3E_8E_{10}; & p_{12} = E_0E_2E_5'E_6'E_9; \\
p_5 = E_0E_1'E_3'E_7'E_9; & p_{13} = E_0E_2'E_4'E_3'E_7'E_9 \\
p_6 = E_0E_2E_4E_3E_7E_9; & p_{14} = E_0E_2'E_4'E_3E_7E_9; \\
p_7 = E_0E_2E_4E_3E_8E_{10}; & p_{15} = E_0E_2'E_4'E_3E_8E_{10}; \\
p_8 = E_0E_2E_4'E_3E_7E_9; & p_{16} = E_0E_2'E_5'E_6'E_9;
\end{array}$$

Using the combination algorithm in Figure 7, we get 57 combined paths: $\{\epsilon,\ p_1,\ p_2,\ p_3,\ p_4,\ p_5,\ p_6,\ p_1|p_6,\ p_3|p_6,\ p_7,\ p_2|p_7,\ p_4|p_7,\ p_8,\ p_1|p_8,\ p_9,\ p_2|p_9,\ p_{10},\ p_5|p_{10},\ p_{11},\ p_1|p_{11},\ p_2|p_{11},\ p_3|p_{11},\ p_4|p_{11},\ p_5|p_{11},\ p_8|p_{11},\ p_9|p_{11},\ p_{10}|p_{11},\ p_1|p_8|p_{11},\ p_2|p_9|p_{11},\ p_5|p_{10}|p_{11},\ p_{12},\ p_1|p_{12},\ p_2|p_{12},\ p_3|p_{12},\ p_4|p_{12},\ p_5|p_{12},\ p_6|p_{12},\ p_7|p_{12},\ p_1|p_6|p_{12},\ p_3|p_6|p_{12},\ p_2|p_7|p_{12},\ p_4|p_7|p_{12},\ p_{13},\ p_1|p_{13},\ p_2|p_{13},\ p_{14},\ p_1|p_{14},\ p_3|p_{14},\ p_{15},\ p_2|p_{15},\ p_4|p_{15},\ p_{16},\ p_1|p_{16},\ p_2|p_{16},\ p_{14}|p_{16},\ p_1|p_{14}|p_{16},\ p_{15}|p_{16},\ p_2|p_{15}|p_{16}\}$

Using the scale restriction technique proposed in Section 3.4.3, we get $C_{LB} = C_{UB} = 3$, and then 9 valid combinations can be selected out of the total 57. The constraints of all these paths can be solved by `BoNuS` in less than 1 millisecond. $\{p_1|p_8|p_{11},\ p_2|p_9|p_{11},\ p_5|p_{10}|p_{11},\ p_1|p_6|p_{12},\ p_3|p_6|p_{12}, p_2|p_7|p_{12},\ p_4|p_7|p_{12},\ p_1|p_{14}|p_{16},\ p_2|p_{15}|p_{16}\}$

These combined test paths are represented in a not so intuitive way. To facilitate understanding, an informal representation based on the original BPEL flow graph can be used. Here, we use `(A1|A2)` to denote that two segments `A1` and `A2` run concurrently. We also use abbreviated denotations of BPEL activities as follows: `Rc: Receive, Ap: Approver, ApE: Approver Exception, As: Assessor, An: Assign, Rp: Reply`. Then the 9 combined test paths equal to `Rc-(Ap|As-An)-Rp`, `Rc-(ApE|As-An)-Rp`, `Rc-As-An-Rp`, `Re-(Ap|As-Ap)-Rp`, `Rc-As-Ap-Rp`, `Rc-(ApE|As-ApE)-Rp`, `Rc-As-ApE-Rp`, `Rc-Ap-Rp` and `Rc-ApE-Rp` respectively.

Suppose that instead of using the default all-path coverage goal, a user prescribes that he wants to cover only the exception handling logic (which corresponds to E_8 in the XCFG graph). Then the sequential paths $p_2, p_4, p_7, p_9, p_{15}$ contain E_8. There are totally 4 test paths left for this coverage goal: $\{p_2|p_9|p_{11},\ p_2|p_7|p_{12},\ p_4|p_7|p_{12},\ p_2|p_{15}|p_{16}\}$.

Suppose that a user chooses the basis-path-coverage, then firstly we can get 9 basis sequential paths instead of 14:

$$\{p_1,\ p_2,\ p_3,\ p_5,\ p_6,\ p_9,\ p_{11},\ p_{12},\ p_{16}\}$$

There are totally 3 valid combinations according to the scale restriction: $\{p_2|p_9|p_{11},\ p_1|p_6|p_{12},\ p_3|p_6|p_{12}\}$.