BPEL Integration Testing

Seema Jehan, Ingo Pill, and Franz Wotawa

Institute for Software Technology
Graz University of Technology, Austria
{sjehan,ipill,wotawa}@ist.tugraz.at

Abstract. Service-oriented architectures, and evolvements such as clouds, provide a promising infrastructure for future computing. They encapsulate an IP core's functionality for easy access via well-defined business and web interfaces, and in turn allow us to flexibly realize complex software drawing on available expertise. In this paper, we take a look at some challenges we have to face during the task of testing such systems for verification purposes. In particular, we delve into the task of test suite generation, and compare the performance of two corresponding algorithms. In addition, we report on experiments for a collection of BPEL processes taken from the literature, in order to identify performance trends with respect to fault coverage metrics. Our results suggest that a structural reasoning might outperform a completely random approach.

1 Introduction

Today, service oriented architectures (SOAs) are an important instrument for software design [28], and they might become ubiquitous in future computing - be it mobile apps, cloud applications, or the realization of public and private business processes. As their backend, web services encapsulate an intellectual property (IP) core's individual functionality and provide a web interface for easy and flexible access of our own or a third party's expertise and developments. In this context, BPEL originated almost a decade ago as OASIS[1] standard for modeling and executing such business processes that implement a desired functionality drawing on web services. Non-functional requirements can be defined, for instance, in Service Level Agreements (SLAs).

Aside exacerbated issues regarding controllability and observability, the fact that we have only partial knowledge about some system parts (i.e. third party web services) - as Friedrich et al. discussed in [24] - makes diagnosis and repair of these systems a cumbersome and very complex task. In [31], we envisioned an integrated testing and diagnosis approach that considers such a system's BPEL processes. Extracting a control-flow graph from a BPEL model, and annotating it with (most likely partial) knowledge about invoked web services in the form of pre- and postconditions, we proposed to use constraint representations as reasoning model for such an approach [19].

[1] Organization for the Advancement of Structured Information Standards, see `https://www.oasis-open.org`.

A. Egyed and I. Schaefer (Eds.): FASE 2015, LNCS 9033, pp. 69–83, 2015.
DOI: 10.1007/978-3-662-46675-9_5

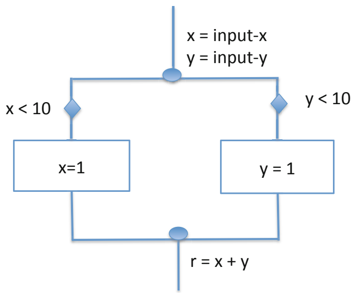

Fig. 1. Flow Example

In our current work, we take a closer look at the task of creating an efficient test suite for functional verification purposes. Complementing earlier work (see Section 4), we focus on path extraction and symbolic execution. Extending [17], we experiment in Section 3 with two path-oriented variants of systematically generating test suites for synchronous, executable BPEL processes (see Section 2). In particular, we compare the performance of test suites generated (a) in an entirely random fashion with (b) a structural approach of exploring the paths in a process' control structure. In this context, we define a test suite's efficiency in terms of its capability to detect an original BPEL process' mutations (specifically in relation to consumed resources). Extending our earlier work and definitions, we support also the flow activity (limited to branches that do not share variables) that allows designers to model also concurrent computations.

Since considering all of a system's feasible runs in the control structure is likely to be infeasible for practical purposes, i.e., in terms of test case generation and execution, specific questions of ours were whether a random approach achieves the desired performance, and how (and to which extent) the variation of certain parameters affects a verdict. Using several example BPEL processes from the literature, we aimed at discovering common trends by bringing several designs to the picture, elaborating on our first ideas proposed in [17].

The general idea of the algorithms compared in Section 3 is to traverse a BPEL process' flow graph in order to extract the necessary inputs and expected outputs for real executions and in turn generate test cases. Our work can be understood best via illustrations for a simple concurrent BPEL process like the one depicted in Figure 1. For our example, an execution starts with assigning the input values to the respective variables, and the subsequent flow activity defines two branches that are triggered if their respective guards ($x < 10$ and $y < 10$) are activated. In both branches, individual activities then offer new value assignments, where after the branches join again, the variables x and y are summed up. An execution then follows a *run* in the graph, where, in contrast to a path as we have been using in our earlier work for sequential programs [17,31], more than one branch may be active simultaneously. Deriving a flow graph from the BPEL process, and annotating it with our partial knowledge about called web services (and other available knowledge) in the form of pre- and postconditions (to be added as conditions and assignments), our test case generation algorithms still select paths, but to a corresponding run's model we add also all the parallel branches that might be traversed as well (depending on the actual assignment and the corresponding evaluation of the guards). Deriving a satisfying variable assignment for a constraint representation of this model, we can derive a corresponding test-case, and in turn, following different strategies for choosing paths, test suites.

Since we have only partial knowledge about invoked external services, we consider our work to fall in the category of active grey-box testing approaches [19]. In our title, we use the term integration testing, as from a certain and important point of view, we test the integration of all a system's components when orchestrated by the BPEL process.

Presently, most research in the field of SOA testing focuses on passive, rather than active testing. Factors here are related costs (i.e. for invoking external web services), execution times (see our experiments in Section 3), and the impact of an environment's dynamic effects (e.g. of the network). We, however, believe that active testing issues in the SOA domain should not be neglected either. In fact, they have become more critical in emerging concepts such as clouds and similar distributed architectures, where faults affect performance for a multitude of users and trust in correctness is an issue of utmost importance.

2 Basic Definitions and Test Case Generation

For our reasoning, we use a specific control-flow graph and annotate it with our knowledge about web services using the concept of pre- and postconditions. In order to support BPEL's flow constructs, we extend our BPEL Flow Graph introduced in [19] as follows. Such flow constructs allow for possibly concurrent branches that are activated by individual and optional guards. If such guards are not specified, we assume them to be *True* for simplifying our description.

Definition 1. *An* Extended BPEL Flow Graph *G is a tuple $(V, B, E, v_0, F, \gamma_C(v \in V), \gamma_A(v \in V), \gamma_P(v \in V), \gamma_G(v \in B), \gamma_B(v \in V \setminus B))$, where V is a finite set of vertices representing BPEL process activities, $B \subset V$ is the finite set of fork activity vertices (where a run might branch), $E \subseteq V \times V$ is a finite set of directed edges representing the connections between BPEL activities (edge $e = (v_1, v_2) \in E$ connects v_1 to v_2), $v_0 \in V$ is the start vertex, $F \subseteq V$ is the set of leaf vertices (with no outgoing edges), and the functions $\gamma_C(v)$ and $\gamma_A(v)$ map vertices $v \in V$ to activity conditions and assignments respectively. If v is in B, $\gamma_P(v \in V)$ returns the complementing join activity vertice (and vice versa), and $\perp$ otherwise. Function $\gamma_G(v \in B)$ returns a list of tuples (e_i, TG_{e_i}) for all of a fork vertice v's outgoing edges e_i and their transition guards TG_{e_i} (if there is no guard specified, we assume True so that this branch is always enabled). For any vertice v in $V \setminus B$, the function $\gamma_B(v \in V \setminus B)$ returns the closest predecessor in B if there is such a node, and $\perp$ otherwise.*

An extended BPEL flow graph thus covers a process' structural concept, and via a vertex' labels defined by $\gamma_C(v \in V)$ and $\gamma_A(v \in V)$, we are able to annotate vertices by specifying additional (likely partial) knowledge about activities.

If there are no concurrent computations, an actual execution follows a path in the flow graph, a fact that we exploited in earlier work [17,19] in order to derive corresponding test cases by searching for a satisfying variable assignment to the conditions and assignments encountered along a path.

Definition 2. *A finite path π of length n in an Extended BPEL flow graph G as of Def. 1 is a finite sequence $\pi = \pi_1\pi_2...\pi_n$ such that (1) for any $0 < i \leq n$: $\pi_i \in V$, (2) $\pi_1 = v_0$, (3) for any $0 < i < n$, the edge $e = (\pi_i, \pi_{i+1})$ is in E, and (4) $\pi_n \in F$. $|\pi|$ denotes the length of a path π. We use $f(\pi)$ to refer to the last vertex in π.*

Definition 3. *A finite path segment π in an Extended BPEL flow graph G is defined like a path, but does not have to start in G's initial state v_0, and neither is $f(\pi)$ required to be in F of G.*

For parallel computations, an execution does not follow a single path but features parallel branches, so that we introduce the following definition of a run.

Definition 4. *A finite* run *r of length n in an Extended BPEL Flow Graph G as of Def. 1 is a finite sequence $r = r_1r_2...r_n$ such that (1) for any $0 < i \leq n$: $r_i \in V$, (2) $r_1 = v_0$, (3) $r_n \in F$, and (4) for any $0 < i < n$, either the edge $e = (r_i, r_{i+1})$ is in E, or if $\gamma_B(p_i) \neq \bot$ then there has to be some $i < j \leq n$ such that (a) there is no $i < k < j$ with $r_k = \gamma_P(\gamma_B(r_i))$ and (b) edge $e = (r_i, r_j)$ is in E. $|r|$ denotes the length of run r. With $f(r)$ we refer to the last vertex in r.*

Obviously, the activities in parallel branches may interleave, as usually there is no defined total order of all the events in the parallel branches, but only a partial one within each individual branch. Part (4) of Def. 4 ensures this partial order. As we will see by the following two definitions, while an actual execution defines a run, in general this is not the case in the other direction.

For one, due to some conflict in the conditions and assignments along a run r, r might actually be infeasible. Accordingly, we have to check a run's collected assignments and conditions for a satisfying assignment in order to determine if such a run is even possible.

Definition 5. *A* feasible *run r is a run as of Def. 4 s.t. the conditions and assignments encountered along the run are feasible. It is* complete*, iff for all satisfied transition guards TG_{e_i} at all $v \in B$ visited by r, the corresponding branch started by edge e_i is present in r.*

A corresponding satisfying assignment for a complete run r defines a valid test case. In earlier work neglecting concurrent constructs, we computed test cases by choosing paths in the flow graph and deriving test cases directly from a satisfying assignment for a path's collected conditions and assignments. Now, via the triggered guards, an individual assignment defines which branches are actually executed, so that pre-selecting which branches are to be taken (specifically if there are many, broad, and/or nested flow activities) and then asking for a satisfying assignment might lead to a lot of infeasible instances and thus bad test suite generation performance. Therefore, we still choose paths π in G, but for identifying an actual *execution*, we derive the following run-constraints model for path π (in analogy to our path constraints in earlier work). Complementing π's constraints, all branches of encountered flow constructs are to be modeled, where only the ones being part of π are specifically required to be active.

Definition 6. *For a path $\pi = \pi_1\pi_2 ... \pi_n$ in some Extended BPEL flow graph G as of Def. 1, we create the* run-constraints $C(\pi)$ *as follows. For each $l \in \gamma_G(\pi_i)$ of a $\pi_i \in B$, we define a branching variable b_l. Let scope be an initially empty list of these branching variables, where we can append a variable b_l via append(scope, b_l), and ask for the last variable with b_l = last(scope) (which will be $\perp$ if the list is empty) as well as remove the last variable via drop(scope). Furthermore, let stop be an initially empty list of vertices in G which we can access with the same functions as scope. Then let $C(\pi)$ be the union of the constraints as derived by traversing π from π_1 to $f(\pi)$ (possibly recursively) as of Def. 7, where in recursive calls the original path can be referred to as π^o, and where variables are replaced by indexed variables in order to implement a static single assignment form (see [10]).*

Definition 7. *For a given path segment π in G, its branching variables and lists scope and stop, we do the following: Let Π be an initially empty list of tuples (v, b_m, π') such that v is a vertex, b_m is a branching variable, and π' is a path segment in G. Then, traversing π from π_1 to $f(\pi)$ do as follows.*

1. *if π_i = last(stop), then for each (π_i, b_m, π') in Π do: First, remove (π_i, b_m, π') from Π, and then add constraints for π' as of this Definition for a local scope having b_m as it sole element, computing the local branching variables for π', and assuming a local empty stop list. When there is no more (π_i, b_m, π') in Π, call drop(scope) and drop(stop).*
2. *if $\pi_i \notin B$ then (a) add constraints $\gamma_C(\pi_i) \cup \gamma_A(\pi_i)$ if last(scope) $= \perp$ and proceed with Step 1 for π_{i+1}, or (b) add constraints $(b_l \rightarrow \gamma_C(\pi_i)) \cup (b_l \rightarrow \gamma_A(\pi_i)) \cup (\neg b_l \rightarrow \gamma'_A(\pi_i))$ for b_l = last(scope) $\neq \perp$ and $\gamma'_A(\pi_i)$ replacing every assignment of a variable in $\gamma_A(\pi_i)$ with an assignment of the variable's old value (so that we are always synchronized in respect of the SSA indices when arriving at the join activity, regardless of which branch was active).*
3. *if $\pi_i \in B$ then do as follows. For $l = ((\pi_i, \pi_{i+1}), TG_l) \in \gamma_G(\pi_i)$, add the constraints $b_l \rightarrow TG_l$ and $TG_l \rightarrow b_l$, and append b_l to scope, append $\gamma_P(\pi_i)$ to stop, but add the constraint b_l only if $\pi = \pi^o$. Then find for each $m = ((\pi_i, v), TG_m) \in \gamma_G(\pi_i)$ s.t. $m \neq l$ a path segment π' leading from v to $\gamma_P(\pi_i)$, and add the tuple $(\gamma_P(\pi_i), b_m, \pi'')$ s.t. π'' equals π' but with the last vertex ($\gamma_P(\pi_i)$) removed to Π, as well as add constraints $b_m \rightarrow TG_m$ and $TG_m \rightarrow b_m$.*

In detail, the static single assignment form means that we use indexed variables (i.e. "temporal" variable instances clocked by assignments), such that, whenever a variable is assigned a value, the index is incremented for further referrals along the run. This process, described in principle in earlier work [19], ensures that every variable along a run is defined only once, but might be referred to many times. Note that our approach is similar to symbolic execution, which is a very well-known technique in testing [20]. Similar to symbolic execution, we compute conditions that belong to a particular execution run. In our case, we convert each run condition into a constraint satisfaction problem.

The basic step in Def. 7 is the second one, since there we collect a visited node's conditions and assignments in order to model the run's constraints. The scope variable (as assigned in Step 3) tells us if a node is visited in a branch of a fork activity, so that the corresponding conditions and assignments are of interest only if the branch is active. Note that (only) for branches being part of the original path π^o, the corresponding branch variables are required to be true (active). Since we model relevant path segments that might not be part of the run determined by the actual assignment, we have to make sure that in the SSA form all the branches (that is, the variable indices) get synchronized. To this end, for an inactive branch (s.t. the branching variable is false), each assignment is "replaced" with a propagation of the last (in a temporal sense) valid value known, so that regardless of the actually active branches, the correct value is assigned to the variables referenced afterwards. Step 3 is responsible for establishing a node visit's scope, and choosing which concurrent branches will have to be modeled (stored in Π). Whenever we reach the end of a branch, we ensure in Step 1 that the necessary complementing branches in Π are contained in the model.

Via run constraints for a path π, we can derive an assignment for a feasible, complete run to be stored as testcase. The only thing that has do be done is to filter the propagation assignments synchronizing the branches, which can be done easily since the assignment contains the branching variables.

Definition 8 (Test Case and Test Suite). *A test case for a BPEL Flow Graph G is a variable assignment that makes a complete run r in G feasible. A test suite TS is a set of test cases.*

For assessing a test suite's quality, we consider its effectiveness in identifying mutated versions of the original BPEL flow graph. Therefore, we introduce the concept of mutants, which are variants of the original BPEL program.

Definition 9. *A* Mutant *is an altered version G' of an original program G. The mutant G' is* equivalent *to G, if and only if they do not differ in their behavior.*

We use a Mutation tool [4] to generate faulty versions of BPEL processes. Those versions we use to check whether a test suite is able to "kill" the mutants.

Definition 10. *A mutant G' is killed by a test suite TS, if there is some test case $t \in TS$ such that the output triggered for G' differs from that for G.*

Note that a mutant not killed might either be equivalent [16] to the original process (which would have to be checked manually), or the test suite is simply

```
1: procedure StructRuns($G$, $MaxLen$)
2:     initialize test suite $S \leftarrow \emptyset$
3:     compute the set $P$ of all paths $\pi$ s.t. $|\pi| \leq MaxLen$, where for vertices $v \in B$,
   we create for each $(e_i, TG_{e_i})$ in $\gamma_G(v)$, a path s.t. $TG_{e_i}$ is enabled.
4:     for each path $\pi \in P$ do
5:         check the satisfiability of run-constraints $C(\pi)$ as of Def. 6
6:         if $C(\pi)$ is satisfiable then
7:             add a corresponding test case to $S$
8:         end if
9:     end for
10:    return test suite $S$.
11: end procedure
```

Fig. 2. Our structural TCG algorithm StructRuns

not able to trigger the mutant in the right way in order to make it unveil itself as a mutant. The higher the mutation score (the percentage of killed mutants), the better we consider a test suite's effectiveness at fault detection to be.

In earlier work [19], we introduced the AllPaths test suite generation algorithm that considers all possible paths through a BPEL flow graph (with no parallel computations) up to a certain path length. The algorithm takes two inputs, i.e., the flow graph G and the maximum length *MaxLen*, and traverses the flow graph using a depth-first search strategy, returning a test suite covering all the corresponding feasible paths. Assuming there is no interaction between the parallel branches, we can derive the variant given in Figure 2, supporting also parallel computations. The StructRuns algorithm is thus also search based, and the only difference is that we derive *run-constraints* as of Def. 6 instead of collecting only the assignments and conditions along the path itself (path-constraints) as in [19]. If such a run-constraints model is satisfiable, the relevant corresponding variable assignments are saved as test case.

Considering limited resources, agile testing requirements, and the fact that computing all paths might be too time consuming for larger BPEL processes (as could be executing the tests), we previously proposed also a random algorithm [17]. This algorithm extracts a desired number of random testcases, limited in length by a given parameter. Like for the AllPath algorithm, we can easily derive a variant RandomRuns (see Fig. 3) supporting flow constructs via considering a path's run-constraints as of Def. 6 instead of its path constraints [17].

3 Empirical Evaluation

Our empirical evaluation's goal was to analyze the performance of the StructRuns and the RandomRuns algorithms. In particular, we have been interested in the coverage and mutation scores obtained when using said algorithms for test suite generation. In the latter context, our main objective was to compare the algorithms. The initial underlying surmise was that StructRuns should perform better than RandomRuns with respect to coverage and mutation scores.

```
1: procedure RandomRuns($G$, $Len$, $numTC$)
2:     initialize test suite $S \leftarrow \emptyset$
3:     while $|S| < numTC$ do
4:         initialize path $\pi \leftarrow v_0$
5:         while $|\pi| < maxLen$ do
6:             pick random $v \in V$ s.t. $\exists e = (f(\pi), v) \in E$
7:             add $v$ to $\pi$: $\pi \leftarrow \pi v$
8:             if $f(\pi) \in F$ then
9:                 if run-constraints $C(\pi)$ (see Def. 6) are satisfiable then
10:                    add a corresponding test case to $S$
11:                end if
12:            else
13:                increment infeasible paths
14:            end if
15:        end while
16:    end while
17:    return test suite $S$
18: end procedure
```

Fig. 3. TCG algorithm RandomRuns based on random paths

For our empirical evaluation, we considered ten examples of synchronous BPEL processes. These examples include activities like Receive, Reply, Assign, If, Else if, While, Invoke, Sequence and Flow. The three SOA processes *Loan*, *LoanCov* and *SquaresS* are available from the mutation tool repository [4], where we used the *Loan* example also in the evaluation presented in [18]. *LoanCov* is a slight variant of the *Loan* example, and *SquaresS* computes the obvious arithmetic function. The SOA example *ATM* is a simplified version of the process discussed in [3]. With *Triangle*, we implemented also a typical example from software engineering studies (see [21]). This *Triangle* process decides for a given triangle whether it is equilateral, isosceles, or scalene. Similarly, *Bmi* is another famous example taken from software testing papers. *Calc* is our last example with Sequence activity that implements basic calculator functionalities, i.e., addition, subtraction, multiplication, and division for given input values. The examples *Flow* and *Flow3* are simple hand-crafted examples using Flow construct, whereas *Order* is a variant of an Ordering Service from the BPEL specification document.

We implemented both algorithms, StructRuns and RandomRuns, in Java. All the experiments ran on a 13" MacBook Pro (Late 2011) with a 2.4 GHz Intel Core i5, 4 GB 1333 MHz DDR3, running under OS X 10.7.2.

Tables 1 and 2 offer the experiments' details when using the StructRuns and RandomRuns algorithms respectively. The number of a BPEL process' activities is given by n, the desired maximum path length by mL, the number of derived paths is labeled p, the minimum and maximum lengths of derived paths are given in columns labeled miP and maP respectively, and the minimum and maximum numbers of constraints derived for any path are reported as miC and maC respectively. $GenT$ defines the total time in milliseconds it took us to derive a corresponding test suite S. The most interesting values, however, are

Table 1. Experimental results for the StructRuns TCG algorithm

Prog	n	mL	p	miP	maP	miC	maC	GenT	Cov	Mut	ExecT
Loan	16	10	2	8	9	9	11	176	76.9	68.50	629,271
		15	3	8	12	9	13	227	100.0	87.64	966,228
		20	3	8	12	9	13	247	100.0	87.64	962,418
Atm	27	10	1	9	9	12	12	68	35.2	21.77	468,613
		15	5	9	15	12	20	569	100.0	80.64	2,411,368
		20	5	9	15	12	20	985	100.0	80.64	2,442,709
SquareS	7	10	3	7	10	12	17	200	100.0	88.51	726,955
		15	5	7	15	12	32	566	100.0	89.65	1,279,305
		20	8	7	19	12	42	830	100.0	89.65	2,421,234
LoanCov	27	10	3	8	10	11	11	293	64.0	52.54	1,346,072
		15	5	8	13	11	15	433	100.0	71.03	1,971,916
		20	5	8	13	11	15	467	100.0	71.03	1,971,476
Triangle	22	10	1	7	7	9	9	354	38.0	12.34	870,823
		15	4	7	15	9	25	477	92.0	66.04	2,289,147
		20	5	7	16	9	26	718	100.0	71.03	2,651,180
Bmi	15	10	5	7	9	9	9	485	100.0	90.00	1,081,270
Calc	30	10	4	5	10	6	20	248	40.0	38.63	1,633,480
		15	9	5	15	6	32	591	100.0	98.37	3,496,140
Flow	11	15	1	11	11	14	14	156	83.3	54.00	450,463
Flow3	11	15	2	11	11	11	11	243	100.0	83.30	326,788
OrderFlow	24	25	2	24	24	41	41	378	100.0	61.00	366,954

Table 2. Experimental results for the RandomRuns TCG algorithm with len = 40

Prog	n	rP	miP	maP	miC	maC	GenT	miCov	maCov	avgCov	miMut	maMut	avgMut	stdev	ExecT
Loan	16	1	8	12	9	13	451.0	46.1	69.2	51.49	24.71	52.80	39.21	15.28	526,905
		2	8	12	9	13	445.0	46.1	76.9	66.89	24.71	68.53	55.16	17.51	696,221
		3	8	12	9	13	641.0	46.1	100.0	86.14	24.71	87.64	67.41	17.13	884,560
Atm	27	1	9	14	11	20	479.0	41.1	52.9	50.54	21.77	45.96	33.70	12.58	447,316
		3	9	15	11	20	644.4	35.2	76.4	54.66	21.77	66.12	48.22	15.03	1,111,243
		5	9	15	11	20	897.5	47.0	100.0	65.85	48.38	80.64	59.59	9.74	1,821,139
SquareS	7	3	7	13	12	27	405.3	100.0	100.0	100.00	83.90	89.65	88.50	1.71	808,907
		5	7	25	12	57	728.2	100.0	100.0	100.00	83.90	89.65	88.85	1.80	1,232,068
		8	7	21	12	47	872.9	100.0	100.0	100.00	88.50	89.65	89.54	0.36	1,930,244
LoanCov	16	3	8	13	11	15	926.3	47.3	89.4	64.15	33.33	55.73	43.98	11.77	1,325,958
		5	8	13	11	15	811.3	47.3	94.7	75.74	34.97	61.74	52.89	7.82	1,884,014
Triangle	22	1	7	12	9	15	475.0	33.3	66.6	50.00	12.96	45.37	26.47	15.04	1,002,234
		4	7	15	9	26	654.0	33.3	83.3	74.20	12.96	58.95	44.41	14.39	1,818,729
		5	7	15	9	26	646.0	75.0	91.6	85.00	37.96	69.75	59.35	9.86	2,722,881
		7	7	15	9	26	915.4	75.0	91.6	84.10	37.34	65.74	57.75	7.78	2,800,813
Bmi	15	1	7	9	9	9	158.7	60.0	60.0	60.00	29.09	52.72	42.09	9.99	406,481
		3	7	9	9	9	737.7	60.0	80.0	74.00	45.45	72.72	64.27	10.61	733,752
		5	7	9	9	9	700.0	70.0	90.0	83.00	60.90	81.81	73.18	7.68	1,049,588
		7	7	9	9	9	797.4	80.0	100.0	87.00	62.72	90.00	77.72	7.59	1,436,189
		10	7	9	9	9	816.8	90.0	100.0	92.00	80.90	90.00	83.00	3.71	2,240,604
		12	7	9	9	9	678.5	90.0	100.0	97.00	80.90	90.00	87.27	4.39	2,645,961
		15	7	9	9	9	852.7	90.0	100.0	98.00	80.90	90.00	88.18	3.83	2,737,402
		17	7	9	9	9	877.1	90.0	100.0	97.00	80.90	90.00	87.27	4.39	3,726,332
Calc	30	4	5	25	6	38	703.6	35.0	70.0	46.00	29.73	57.18	39.90	10.28	1,529,496
		9	5	24	6	56	822.7	40.0	100.0	68.00	38.23	84.64	59.31	15.72	2,930,746
Flow	11	1	11	11	13	23	768.4	83.3	83.3	83.30	54.54	54.54	54.54	1.17	374,245
Flow3	11	1	11	11	11	17	545.5	50.0	83.3	66.65	42.66	52.00	43.60	6.56	283,808
		2	11	11	13	19	884.7	83.3	100.0	91.65	52.00	70.66	67.73	5.65	333,727
Order	24	1	23	23	42	42	826.6	47.0	94.1	65.84	40.84	52.11	45.35	5.81	264,348
		2	23	23	42	42	1059	47.0	100.0	82.33	42.25	60.56	52.53	7.99	348,207

those for *Cov* and *Mut* that give us the percentage of covered activities and killed mutants, respectively. The overall test execution time the mutation tool took to compute mutation coverage (in milliseconds) is given in the columns labeled *ExecT*. That is, for the RandomRuns algorithm, we computed 10 samples per row and report the minimum, maximum, and average values for coverage and mutation scores respectively, with also *GenT* referring to the average value over these 10 samples. For the mutation score, we also report on the standard deviation *stdev*. In Table 2, *rP* defines the desired amount of test cases.

Taking a look at Tables 1 and 2, we obtained the following general observations. First, for both algorithms, the test suite generation time was always a fraction of the time needed for executing the mutation tool. That is, whereas the test suite generation time never exceeded 1 second for any example, test execution took up to slightly over an hour. Hence, in the SOA domain, test execution seems to be very time consuming, even for smaller examples. Second, when considering the time required to come up with a test suite of the same size, overall generation time is almost equivalent for both algorithms. Finally, when taking average values into account (but even for the best cases of RandomRuns), the StructRuns algorithm performs better in terms of coverage and mutation scores for most of the examples.

We recorded also the number of infeasible paths encountered, where we report on corresponding results for the *Calc* example in Table 3. Please note that up to a path length of 20, we saw no infeasible paths for the StructRuns algorithm. For the random approach, however, we see how the number of infeasible paths increases if we raise the desired number of test cases (for a given, fixed maximum path length). The longer test cases allowed for this variant, very early ran into issues in this respect. That is, for $|S| = 4$ we already had to dismiss 3 infeasible paths at most (over 10 samples), increasing to 8 paths for a test suite of size 9. As also for 9 test cases, we could reach only a mutation score of 80%, this raises the question of how many random paths we would need in order to achieve the same mutation score as with the StructRuns algorithm.

To the end of answering this question, we considered the *BMI* BPEL process, and test suite sizes *rP* of 1, 3, 5 7, 10, 12 ,15, and 17 (see Table 2). The coverage attained for 12 to 17 paths was roughly the same, with the execution time increasing from approximately 45 minutes to 62 minutes on average. In Figure 4 we summarize our findings using a box plot diagram, where the grey box indicates the bounds given by the average value and the standard deviation. We see that the results are converging to the same mutation score as obtained when using StructRuns. That is, for 7 test cases or more, the random algorithm provided the same 100% activity coverage and maximum mutation score of around 90% as the StructRuns algorithm did. For 12 test cases and above, also the average mutation score got in the same range. However, the time needed for generating a test suite with similar average performance almost doubled for RandomRuns. Moreover, also due to the higher number of test cases, the execution time was higher for the random approach. That is, for test suite sizes of 10 and above, and using the StructRuns algorithm, it was less than half of that as when using

Table 3. Infeasible paths for the RandomRuns TCG algorithm

Prog	**rP**	**min InP**	**max InP**	**avg InP**
Calc	4	0	3	0.4
Calc	9	0	8	1.4

the RandomRuns algorithm. The random approach took 700 milliseconds on average for computing five paths, achieving an average mutation score of 73.18% (ranging between 60.9 and 81.81 percent), while the StructRuns took (smaller) 485 milliseconds to compute 5 paths for a given maximum length of 10, offering a (higher) mutation score of 90% percent.

It is also worth mentioning that there were many surviving mutants for each of the examples used in our empirical evaluation. The highest achieved mutation score was 90%. In order to investigate the non killed mutants, we inspected the surviving mutants manually for the *BMI* example. We found out that by adding five additional test cases, we were able to reach a mutation score of 95%. The remaining mutants were equivalent ones. Hence, we conclude that there is still room for improving the test case generation process, in order to deliver an algorithm with an improved performance.

4 Related Research

Exploiting constraints for software testing is an attractive concept. Gotlieb et al. [15] extracted test cases from programs via a constraint representation of its source code. Whereas our work is quite close to this in principle, the application domain and constraint extraction process are different. In our case we exploit also a component's pre- and postconditions, because, as a matter of fact, a SOA's services' actual implementations are hardly available. In this respect we differ also from [7], where, in contrast to our BPEL flow graph, Bentakouk et al. translate a BPEL model into a symbolic transition system (STS) used to extract test cases from. They issue specific warnings for situations where their approach is incomplete due to time-out violations in the construction of the STS. Similar work was presented in [29].

In the context of web service testing, literature reports on mainly three model based testing techniques, i.e., *symbolic execution*, *petri nets*, and *model checking* [9]. Yuan et al. [34] presented a graph search based test case generation of BPEL processes that exploits matrix transformations of control flow graphs, path coverage, and a node classification depending on incoming and outgoing edges. While their approach is close, we differ in the use of pre- and postconditions added to the test paths, aiming to solve the test oracle problem. A slight difference is also in the use of the MINION constraint solver [14] rather than Lp.

The underlying idea behind [32], which relies on an extended Control Flow Graph (XCFG), is to extract all sequential paths from the XCFG, and to combine them into concurrent test paths. Constraints are then collected from these

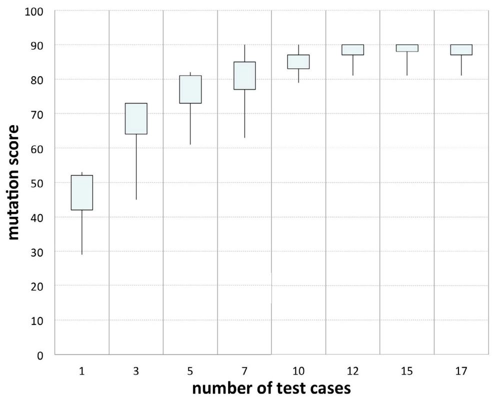

Fig. 4. RANDOMRUNS: Mutation score as function of the number of test cases for *BMI*

concurrent test paths via backward substitution. In contrast, we transform each sequential path directly into a set of constraints, each set independently checked for satisfiability. For unsatisfiable constraints, the corresponding path is discarded, satisfiable ones produce the corresponding variables used to execute the path. These values define a test case to be included in the test suite.

Also model checking [9] can be exploited in the context of web service testing. For this, BPEL specifications are converted into a formal modeling language like PROMELA [13]. Defining test criteria as formal properties, in a language such as LTL [30], a model checker can be used to search for violations of the properties by the BPEL model. The actual test cases then are derived from the counter examples provided by the model checker for such violations. Zhen et al. [36] applied the same idea to web services and BPEL processes, addressing the state space explosion problem inherent with model checking. Moreover, they also developed a tool for the generation of JUnit test cases for automated test execution. Model based testing techniques using Petri Nets have also been explored extensively. Petri Nets are attractive for modeling concurrent processes and their synchronization, and can be categorized into Plain Petri Nets [27], Colored Petri Nets [33] and High-level Petri Nets. Dong [11] developed a tool for test case generation of BPEL processes using High-level Petri Nets. The basic approach is to build a reachability graph from which test cases can be extracted. The approach has a very high space complexity.

For test case execution, we use [23], where we convert the abstract test cases manually to executable ones accepted by the BPELUnit tool [22]. This tool supports simulated as well as real-life testing, accommodating many BPEL engines like Active VOS [1], Oracle BPEL Process Manager [5], and Apache ODE [2]. For simulated testing, a BPEL process is not deployed, rather the intended engine is called through a debug API. In real-life testing mode, a business process is actually deployed and the partner web services are tested using mocks.

The survey of Zakaria et al. [35] gives a very good comparison of different unit testing approaches applied to BPEL processes. One key issue pointed out there is the lack of an empirical evaluation. Surprisingly, only 1 out of 27 considered studies provides results on real-life BPEL processes.

Random testing has been successfully used in practice. For example, Faigon [12] reports on experience gained in random testing of a compiler. He mentions that a simple Random IR (Intermediate Representation) Generator was able to find more than half of all the bugs reported by customers in just one night. Other applications include testing graphical user interfaces (GUIs). In GUI testing, dump and smart monkeys [26,6] are used, where the latter makes use of a simplified model of the application in order to guide testing to some extent. Arnold [6] reports that monkeys are able to find more than 25 % of the bugs when used early in the development cycle. Hence, random testing can be effectively used in practice. In contrast to these papers, we apply random testing on a model obtained from the source code directly.

5 Conclusions

Summarizing, we report on experiments with two test suite generation algorithms that implement orthogonal strategies for creating functional tests for synchronous executable BPEL processes. That is, the StructRuns variant derives test cases covering all feasible paths up to a given length, and the RandomRuns variant derives a desired number of test cases covering a random selection of feasible paths (and successor states in a path), also limited by a predetermined length. Our initial surmise that by construction, the StructRuns variant should offer better performance in terms of achievable mutation scores and activity coverage, but might be infeasible in practical terms (s.t. a random approach might be favorable) was confirmed by our experiments. However, the experiments showed also that the random approach was inferior in terms of generation time (and execution time). That is, for the BMI example, even computing 3 random paths took us 50 % longer to construct, than all the five paths for a maximum length of 10. While we saw that for the SOA domain, construction time is negligible in comparison to test execution time, our random setup was quite inferior also in the achieved mutation scores for the BMI example and a comparable execution time. We draw several conclusions from this. First, our observations regarding execution times show that one definitely has to employ some strategic reasoning for designing test suites in the context of BPEL process testing. Our tests also showed that, at least our examples might not favor the use of longer paths as allowed for the random approach, in comparison to the shorter paths derived for the StructRuns variant. For a completely random approach, longer path lengths must be allowed however, due to the random choice of the successor states (i.e. in the context of loops). Thus an approach mixing a random component with some structural reasoning should be the subject of future research. Such research will also aim at accommodating those five manual tests that allowed us to improve the mutation score for the BMI example, and will be subject to stimuli from realistic [8] and search-based test-case generation [25].

Acknowledgement. The research leading to these results has received funding from the Austrian Science Fund (FWF) under project references P23313-N23 and P22959-N23.

References

1. Active VOS engine, `http://www.activevos.com`
2. Apache ODE, `http://ode.apache.org/`
3. JBoss example, `http://docs.jboss.com/jbpm/bpel/v1.1/userguide/tutorial.atm.html`
4. MuBPEL- a mutation testing tool for WS-BPEL, `https://neptuno.uca.es/redmine/projects/sources-fm/wiki/MuBPEL`
5. Oracle BPEL Process Manager, `http://www.oracle.com/technetwork/middleware/bpel`
6. Arnold, T.R.: Visual Test 6 Bible. IDG Books Worldwide, Inc., Foster City (1998)
7. Bentakouk, L., Poizat, P., Zaïdi, F.: A formal framework for service orchestration testing based on symbolic transition systems. In: Proc. of the 21st IFIP WG 6.1 Int. Conf. on Testing of Software and Communication Systems and 9th Int. FATES Workshop, pp. 16–32 (2009)
8. Bozkurt, M., Harman, M.: Automatically generating realistic test input from web services. In: International Symposium on Service-Oriented System Engineering (SOSE), pp. 13–24 (December 2011)
9. Bozkurt, M., Harman, M., Hassoun, Y.: Testing Web Services: A Survey. Tech. Rep. TR-10-01, Dep. of Computer Science, King's College London (January 2010)
10. Brandis, M.M., Mössenböck, H.: Single-pass generation of static assignment form for structured languages. ACM TOPLAS 16(6), 1684–1698 (1994)
11. Dong, W.: Test case generation method for BPEL-Based Testing. In: Int. Conf. on Computational Intelligence and Natural Computing, vol. 2, pp. 467–470 (June 2009)
12. Faigon, A.: Testing for zero bugs, `http://www.yendor.com/testing/`
13. Garcia-fanjul, J., Tuya, J., Riva, C.D.L.: Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN (2006), `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.9287`
14. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proceedings of ECAI 2006, Riva del Garda, pp. 98–102. IOS Press (2006)
15. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 53–62 (1998)
16. Grün, B.J.M., Schuler, D., Zeller, A.: The impact of equivalent mutants. In: Proceedings of the IEEE Int. Conf. on Software Testing, Verification, and Validation Workshops, ICSTW 2009, pp. 192–199 (2009)
17. Jehan, S., Pill, I., Wotawa, F.: SOA testing via random paths in BPEL models. In: 10th Workshop on Advances in Model Based Testing; 2014 IEEE Seventh Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), pp. 260–263 (2014)
18. Jehan, S., Pill, I., Wotawa, F.: Functional SOA testing based on constraints. In: 8th Int. Workshop on Automation of Software Test (AST), pp. 33–39 (2013)

19. Jehan, S., Pill, I., Wotawa, F.: SOA grey box testing - a constraint-based approach. In: 5th Int. Workshop on Constraints in Software Testing, Verification and Analysis; 2013 IEEE Sixth Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), pp. 232–237 (2013)
20. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976), `http://doi.acm.org/10.1145/360248.360252`
21. Langdon, W.B., Harman, M., Jia, Y.: Efficient Multi-objective Higher Order Mutation Testing with Genetic Programming. J. Syst. Softw. 83(12), 2416–2430 (2010)
22. Lübke, D.: Bpel Unit (2006), `http://bpelunit.github.com`
23. Mayer, P., Lübke, D.: Towards a BPEL unit testing framework. In: Workshop on Testing, Analysis, and Verification of Web Services and Applications, pp. 33–42 (2006)
24. Mayer, W., Friedrich, G., Stumptner, M.: On computing correct processes and repairs using partial behavioral models. In: European Conf. on Artificial Intelligence (ECAI), pp. 582–587 (2012)
25. McMinn, P., Shahbaz, M., Stevenson, M.: Search-based test input generation for string data types using the results of web queries. In: 5th Int. Conf. on Software Testing, Verification and Validation (ICST), pp. 141–150 (April 2012)
26. Nyman, N.: Using monkey test tools. Software Testing & Quality Enineering Magazine (January/February 2000)
27. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. 67(2-3), 162–198 (2007)
28. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. of Cooperative Information Systems 17(2), 223–255 (2008)
29. Paradkar, A., Sinha, A.: Specify once test everywhere: Analyzing invariants to augment service descriptions for automated test generation. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 551–557. Springer, Heidelberg (2008)
30. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)
31. Wotawa, F., Schulz, M., Pill, I., Jehan, S., Leitner, P., Hummer, W., Schulte, S., Hoenisch, P., Dustdar, S.: Fifty shades of grey in SOA testing. In: 9th Workshop on Advances in Model Based Testing; 2013 IEEE Sixth Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), pp. 154–157 (2013)
32. J., Li, Z., Yuan, Y., Sun, W., Yan, J.Z.: BPEL4WS Unit Testing: Test case generation using a concurrent path analysis approach. In: 17th Int. Symp. on Software Reliability Engineering (ISSRE), pp. 75–84. IEEE Computer Society (2006)
33. Yang, Y., Tan, Q., Xiao, Y.: Verifying web services composition based on hierarchical colored petri nets. In: 1st Int. Workshop on Interoperability of Heterogeneous Information Systems, IHIS 2005, pp. 47–54. ACM (2005)
34. Li, Z., Yuan, W.S.Y.: A graph-search based approach to BPEL4WS test generation. In: Int. Conf. on Software Engineering Advances, p. 14 (October 2006)
35. Zakaria, Z., Atan, R., Ghani, A.A.A., Sani, N.F.M.: Unit testing approaches for BPEL: A systematic review. In: 16th Asia-Pacific Software Engineering Conference, pp. 316–322. IEEE Computer Society (2009)
36. Zheng, Y., Zhou, J., Krause, P.: A model checking based test case generation framework for web services pp. 715–722 (April 2007)