Test suite minimization for mutation testing of WS-BPEL compositions

Francisco Palomo-Lozano
Escuela Superior de Ingeniería
University of Cádiz
Puerto Real, Spain
francisco.palomo@uca.es

Antonia Estero-Botaro
Escuela Superior de Ingeniería
University of Cádiz
Puerto Real, Spain
antonia.estero@uca.es

Inmaculada Medina-Bulo
Escuela Superior de Ingeniería
University of Cádiz
Puerto Real, Spain
inmaculada.medina@uca.es

Manuel Núñez
Faculty of Computer Science
Complutense University of Madrid
Madrid, Spain
mn@sip.ucm.es

ABSTRACT

This paper presents an exact search-based technique to minimize test suites while maintaining their mutation coverage. The minimization of test suites is a hard problem whose solution is important both to reduce the cost of mutation testing and to precisely assess the quality of existing test suites. This problem can be addressed with Search-Based Software Engineering (SBSE) techniques, including metaheuristics and exact techniques. We have applied Integer Linear Programming (ILP) as an exact technique to reduce the effort of testing with very promising results. Our technique can be adapted to different formalisms but this paper focuses on testing WS-BPEL compositions, as it poses several interesting problems. Despite the fact that web service compositions are relatively small, as they just orchestrate web services, their execution can be very expensive because the deployment and execution of web services, and the underlying infrastructure, are not trivial. Therefore, although test suites for the compositions themselves are also usually small, it is fundamental to reduce, as much as possible and without losing coverage, their size.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; *Software testing and debugging*; *Orchestration languages*; • **Theory of computation** → *Linear programming*;

KEYWORDS

SBSE, WS-BPEL, web service compositions, mutation testing, test suite minimization, exact techniques, ILP

ACM Reference Format:
Francisco Palomo-Lozano, Antonia Estero-Botaro, Inmaculada Medina-Bulo, and Manuel Núñez. 2018. Test suite minimization for mutation testing of
WS-BPEL compositions. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3205455.3205533

1 INTRODUCTION

Software testing represents a major part of the development effort that in some projects can raise up to 40% of the total development cost even assuming a conservative estimate. Therefore, it is necessary to carefully select which test cases to execute, or in which order they should be executed, to detect potential errors with the least possible cost [46]. There exist two situations where the execution cost of tests is very high, though for different reasons.

In the first situation, and this is the most common scenario, there is a vast number of test cases, which are likely to come from legacy test suites that have been evolving and growing throughout different versions of the system under test (SUT). Thus, the accumulated cost of maintaining and executing these test suites, particularly for regression testing, can be very high. There is another situation, maybe not so common but even more challenging, where the associated costs can be also very high. It happens when the test suites are relatively small but they exercise a large number of features of the SUT, what might imply a high overall cost too.

The latter situation is exactly the case with mutation testing as every mutant, which is a potentially faulty variant of the SUT, is executed against each test case in the test suite in an attempt to kill the mutant [22]. The number of mutants is usually large. Moreover, this situation is also prevalent in the development of web service compositions [40]. These compositions combine services that are tested apart or whose testing is not among the responsibilities of the organization

developing or maintaining the composition. In fact, several web services conforming the same functional specification[1] may be available. Services can be automatically *discovered* and it might be the case that the service to be executed is dynamically selected or even hot-swapped at a given moment, depending on certain conditions.

In summary, web services should be seen as *black boxes* where requests are formulated and results are returned in exchange. The goal of the test team primarily consists in designing effective test cases for the web service composition itself. It usually happens that the number of test cases is not high, at least, in comparison with the number of test cases needed to test the underlying services. However, their execution cost is generally very high: in particular, it includes the execution costs of the services involved. The situation worsens in the presence of mutation testing, as a large number of mutants has to be considered.

Nevertheless, the elimination of test cases cannot be properly addressed without incorporating metrics to assess the effectiveness of the selected test cases. As there is not just one possible criterion to choose these metrics [10] and different criteria may even represent conflicting goals, the context has to be clearly specified.

In this work we present an approach to the minimization of test suites based on mutation coverage [22, 35] for WS-BPEL compositions. WS-BPEL is an OASIS standard for web service compositions [29]. We use exact techniques based on Integer Linear Programming (ILP) [23]. The problem is reduced to the minimization of a linear function with binary decision variables and linear constraints. Decision variables represent which test cases are selected and constraints ensure that mutation coverage does not decrease.

This paper is, to the best of our knowledge, the first work providing test-suite minimization for web service compositions and the first time that an exact technique is used to minimize test suites in the context of mutation testing. Besides, it builds the individual execution costs of test cases into the model instead of assuming, as usual, that they are executed at unit cost. Finally, it preserves mutation coverage, which is important in the light of recent results [6]: exactly the same mutants are killed by the resulting test suite.

The structure of the rest of the paper is as follows. Section 2 contains a discussion of related work. Then, a minimal background on mutation testing of web service compositions is provided in Section 3. Afterwards, Section 4 introduces the minimization problem in this context and the techniques used to solve it. Next, Section 5 contains a description of the experiments conducted and discusses the results obtained. Threats to validity are presented in Section 6. Finally, conclusions and future work follow in Section 7.

2 RELATED WORK

There is a vast literature on metaheuristic techniques for solving the test-suite minimization problem and other related problems. The interested reader is referred to the extensive survey by Yoo and Harman [46] and the work cited therein. This line of work will not be discussed here because, although interesting and widely used, it lies outside the focus of this paper. On the one hand, this paper develops an exact technique, which can be used to guarantee optimal results, unlike metaheuristics, which most of the time just produce best-effort approximations. On the other hand, exact techniques, which in the past have been overlooked in SBSE, are gaining momentum in the last few years. In particular, it is worth to mention the recent trend on requirements selection [16, 26, 42] and the work on test-suite minimization cited below.

Most authors employing exact techniques resort to some kind of reduction from an $\mathcal{NP}$-hard covering problem to a target problem. These target problems are also $\mathcal{NP}$-hard, but general enough to allow the encoding of a vast amount of problems [13, 24]. As a consequence, they enjoy *very efficient* optimizers that have been developed and improved for decades, sometimes with a strong investment from the private sector.[2] These optimizers are indeed exponential in the worst case, but they can be quite fast for small or reasonable size instances, instances with certain structure, or even many large instances.

Quite often the target problem is ILP [23]. Actually, ILP is close to the covering problems arising in test suite minimization in the sense that its language is expressive enough to efficiently encode them. The application of ILP to the minimization of test suites has been the subject of a number of works in the literature [5, 11, 44, 47]. An alternative approach consists in using SAT, instead of ILP, as the target problem. An extension of SAT with pseudo-boolean constraints is in fact used as an intermediate representation [3, 27].

More recently, Li et al. [25] proposed power consumption as a metric for assessing the cost of the test process. They measured the electrical energy consumed when executing the individual test cases against the SUT and reduced the test suite by applying ILP, minimizing the overall power consumption.

Regarding the evaluation of the approach, much of the previous work is based on the well-known SIEMENS suite of C programs [7, 19] (http://sir.unl.edu/portal) where the application of heuristics have a major impact. For example, under the unit-cost assumption, the most complex instance resulting after preprocessing with straightforward heuristics is a test suite for `replace` with 215 test cases and 208 characteristics [3].

3 BACKGROUND

In this section we introduce the more important aspects of the two application domains that we combine in this paper: mutation testing and WS-BPEL compositions.

3.1 Mutation testing

Mutation testing [34] is a testing technique based on fault injection. The idea is injecting faults in the code implementing

[1] Usually these alternative services present different non-functional features (e.g. run-time or quality of service).

[2] http://www-03.ibm.com/press/us/en/pressrelease/26403.wss

the SUT by using *mutation operators*. Mutation operators produce *mutants* as a result, which are slight variants of the original code. Upon execution of the SUT and one of its mutants against the same test case, two outcomes are possible. When the results of the execution do not agree, the mutant has been correctly identified as a faulty piece of code. Then, it is said that it has been *killed* by the corresponding test case and, so, it is *dead*. This gives evidence that the test case is useful, as it has served the purpose of detecting the fault inside the mutant. However, if the results always agree for every test case in the test suite, then the mutant remains *alive*.

In its simplest form, each mutation introduces a single syntactic change in a program. For example, if a program contains the arithmetic expression `x + y + z` and there is a mutation operator available to replace the arithmetic operator `+` by the arithmetic operator `-`, mutants containing the expressions `x - y + z` and `x + y - z` can be produced.

There are two reasons why a mutant can remain alive after the execution of the test suite. First, maybe the test suite is not good enough to detect the difference between the mutant and the original code. For example, the fault can be located in a statement that is not covered by the test suite. Second, and more interesting, the difference may not represent a real fault. A mutation may happen to produce a variation of a program with exactly the same functional behavior. In the latter case, it is said that an *equivalent mutant* has been generated. However, distinguishing equivalent mutants from mutants that simply stay alive after execution is undecidable and, even when the particular instance is, it can be very costly.

Mutation testing exhibits two main difficulties: the inherent computational cost associated to the execution of a huge number of mutants and the detection of equivalent mutants. Regarding the latter difficulty, many authors have proposed different techniques to reduce the burden of detecting equivalent mutants [2, 4, 8, 14, 15, 17, 31–33, 36–39, 43]. However, in practice, this detection is done by hand, being a time-consuming and error-prone task. Besides, a terminating program can be mutated into a non-terminating one. As termination (and non-termination) is undecidable, it is not possible, in general, to determine in finite time whether a program is going to terminate its execution for a given input.

These drawbacks worsen when there are many mutation operators involved, as it is usual when mutation testing is applied to mainstream languages [34]. Each mutant has to be executed against every test case in the test suite and a high number of mutants can be generated even for an SUT of modest size.

Traditional mutation, as described above, is also known as *strong mutation*. Strong mutation requires that certain conditions hold [30]. In particular, we have to wait until the execution terminates to check whether the mutant and the original program outputs differ. This is certainly a strong requirement that can be waived to reduce the computational cost of mutation testing in different ways.

Weak mutation [18] is a cost-reducing technique where the outputs are compared right after the execution of the mutated sentence instead of at the end of the execution. Clearly, this can greatly reduce the execution time, though is far more difficult to implement than strong mutation. *Firm mutation* [45] has the same aim and goes one step forward. Again, this technique differs from strong mutation in the way the mutant behavior is inspected to decide whether the mutant has been killed by the execution. The idea is similar to weak mutation, but taking advantage of the state of the program (not just its output) and giving more flexibility about when to perform the comparison. Thus, testing a mutant in the context of firm mutation can be seen as inspecting a set of designated variables, which are called the *inspection set*. A mutant is killed by a test case on an inspection set when the mutant and the original program produce different values for any variable in the inspection set at some location after the mutated sentence.

Firm mutation is, therefore, a compromise between strong and weak mutation. Let l be the location where the mutation is injected. Under firm mutation, the original program and the mutant can be compared on a given inspection set at any convenient position located after l in its execution path. Therefore, weak and strong mutation are generalized by firm mutation. In weak mutation, l is just the location of the mutated sentence, while in strong mutation l is the exit location. Therefore, firm mutation subsumes both weak and strong mutation [15]. Firm mutation is a relevant technique in the context of WS-BPEL compositions as we will see in Section 4.1.

3.2 WS-BPEL compositions

WS-BPEL 2.0 is the OASIS standard [29] for web service composition. WS-BPEL is an XML-based programming language describing the behavior of business processes and how they interact with other web services. The specification of a business process with WS-BPEL consists of four steps:

(1) Declaring the *process relations* with the *external partners*.
(2) Declaring the *process variables*.
(3) Declaring the *process handlers*.
(4) Describing the business logic or *process behavior*.

The external partners include the client invoking the business process and the services invoked by the process itself. These processes typically include event handlers and fault handlers too.

WS-BPEL processes are built from *activities*. Although basic activities perform only one purpose, they can be grouped into structured activities, which define the business logic. Basic activities include tasks like assigning data to a variable, invoking a web service, receiving a message, replying to a formerly received message, etc. Structured activities include classical conditional and loop statements, different flavors of parallel execution, blocking, etc. Besides, WS-BPEL processes can be further structured into different components through *scopes*. Scopes usually contain local declarations

```
codewrapperfontupper=,coltext=green!50!black,colframe=red!95!black,colback=red!5!white
...
<!- Structured activity for parallel execution ->
<flow>
    <!- Links for specifying synchronization dependencies ->
    <links>
        <link name = "checkFlight-to-bookFlight" />              <!- Link element ->
    </links>
    <!- Basic activities invoking the web services ->
    <invoke name = "checkFlight" ... >
        <!- Source container for synchronization ->
        <sources>
            <source linkName = "checkFlight-to-bookFlight" />    <!- Source element
->
        </sources>
    </invoke>
    <invoke name = "checkHotel" ... />
    <invoke name = "checkRentCar" ... />
    <invoke name = "bookFlight" ... >
        <!- Target container for synchronization ->
        <targets>
            <target linkName = "checkFlight-to-bookFlight" />    <!- Target element
->
        </targets>
    </invoke>
</flow>
...
```

Figure 1: WS-BPEL 2.0 snippet of a travel reservation process.

as above, event handlers and fault handlers. Activities may have *attributes* as well as *containers* associated to them. Of course, these containers can include elements with their own attributes too.

WS-BPEL provides concurrency and synchronization mechanisms between activities. The basic idea is illustrated in Figure 1. The `flow` activity initiates the execution of a set of activities in parallel. As these activities can be linked together, a dependence graph is created and concurrent execution is controlled by the partial order induced on activities by their dependence graph. Web services `checkFlight`, `checkHotel`, and `checkRentCar` begin their execution in parallel. However, `bookFlight` can only be invoked upon `checkFlight` completion. This synchronization between these two activities is achieved by creating a `link`: the link target will be eventually executed only when the source activity of the link is completed.

4 MINIMIZATION OF WS-BPEL TEST SUITES

In this section we introduce our approach to the mimimization of test suites. Our technique is quite general and can be applied in practice to almost every SUT and test framework capable of producing the relevant information that is necessary for the minimization process. In this paper we focus on developing this technique for the special case of WS-BPEL compositions with an appropriate test framework because it presents some challenges and peculiarities.

4.1 Mutation of WS-BPEL compositions

It is always convenient to automate the testing process. However, in the context of web service compositions, in particular in the application of mutation testing to WS-BPEL 2.0, this is a must. An automated tool has to be able to generate the mutants according to a well-defined and representative set of mutation operators, execute them against a test suite, and decide whether a mutant has been killed or not by comparing its behavior to the original program for each test case. In addition, it has to incorporate logging and reporting capabilities and take advantage of concurrent or parallel execution to reduce the testing time.

Given these requirements, we have chosen MuBPEL [10]. This tool implements 26 mutation operators for WS-BPEL 2.0 [9]. Table 1 shows their names alongside a brief description. These mutation operators have been classified in 5 categories and those essentially different from mutation operators employed in mainstream languages have been marked with ★.

MuBPEL proceeds roughly as follows. First, the *analyzer* parses the original composition and generates a list of mutation operators and program locations where they can be applied. With this information, the *mutant generator* generates every possible mutant. Then, the *execution engine*

Table 1: Mutation operators for WS-BPEL 2.0.

		Identifier mutation
ISV		Replaces a variable identifier by another of the same type
		Expression mutation
EAA		Replaces an arithmetic operator by another of the same kind.
EEU		Removes the unary minus operator from an expression.
ERR		Replaces a relational operator by another of the same kind.
ELL		Replaces a logical operator by another of the same kind.
ECC		Replaces a path operator by another of the same kind.
ECN		Modifies a numerical constant by incrementing or decrementing its value, or by adding or removing one digit.
EMD		Modifies a duration expression, replacing it by 0 or by half of its initial value.
EMF	★	Modifies a deadline expression, replacing it by 0 or by half of its initial value.
		Activity mutation (parallel)
ACI	★	Changes the `createInstance` attribute from an inbound message activity to *no*.
AFP	★	Replaces a sequential `forEach` activity by a parallel one.
ASF	★	Replaces a `sequence` activity by a `flow` activity.
AIS	★	Changes the `isolated` attribute of a `scope` to *no*.
		Activity mutation (sequential)
AEL		Deletes an activity.
AIE		Deletes an `elseif` element or the `else` element from an `if` activity.
AWR		Replaces a `while` activity by a `repeatUntil` activity and vice versa.
AJC	★	Removes the `joinCondition` attribute from an activity.
ASI	★	Exchanges the order of two child activities in a `sequence` activity.
APM	★	Removes an `onMessage` element from a `pick` activity.
APA	★	Removes the `onAlarm` element from a `pick` activity or from an event handler.
		Event and fault mutation
XMF		Removes a `catch` element or the `catchAll` element from a fault handler.
XMC	★	Removes a compensation handler definition.
XMT	★	Removes a termination handler definition.
XTF		Replaces the fault thrown by a `throw` activity.
XER	★	Removes a `rethrow` activity.
XEE	★	Removes an `onEvent` element from an event handler.

deploys the original composition, which is executed against the given test suite. Finally, the same process is applied to each mutant and the behavior of the mutant and the original program are compared to determine whether the mutant has been killed or stays alive.

MuBPEL builds on a fork of ActiveBPEL [1], a WS-BPEL 2.0 standard compliant execution engine. It also integrates BPELUnit [28], a library for unit testing that can be used with any compliant engine. XML files are used to describe the test suite. A remarkable feature of BPELUnit and MuBPEL is that external web services can be replaced by *mocks*, which behave following a predefined pattern and can be used to mimic real web services in a controlled environment. This way is possible to execute a composition though some web services are not available. This is very convenient for our purposes. The use of mocks allows us to repeat an experiment under exactly the same conditions. We follow a firm mutation [45] approach according to recent work on the formalization of mutation testing in the context of WS-BPEL compositions [10] that includes the technical details about firm mutation in MuBPEL.

4.2 Execution and cost matrices

The *execution matrix*, E, is extracted from the logs produced by MuBPEL for the WS-BPEL composition. If the proper command-line options are supplied, MuBPEL will also log the execution time corresponding to each test case. Therefore, a *cost matrix*, C, can be generated too. In order to get reproducible results, web services have to be fixed in advance and prepared in a local environment or even the same machine, so that external factors like network latency, discovery time, hot-swapping of services, or differences in alternative services are kept aside during the experiments.

Regarding the execution matrix definition, if $e_{ij} = 2$ for some j then m_i is an *invalid mutant* that could not be properly executed. Otherwise, if mutant m_i is killed by t_j then $e_{ij} = 1$, else $e_{ij} = 0$. As for the cost matrix, c_{ij} is simply

defined as the execution time of mutant m_i against test case t_j.

Henceforth, rows corresponding to invalid mutants are removed both from E and C. Therefore, it can be assumed that E is a binary matrix and that the times in C correspond to valid executions. E and C will be $m \times n$ matrices, with $m = |M|$ and $n = |T|$, where M is the set of valid mutants and T is the set of test cases.

4.3 Exact minimization

Our goal is to minimize the number of test cases in WS-BPEL compositions using the *mutation coverage* and the *execution cost* as metrics to guide the search. Mutation coverage is a measure of the number of mutants killed by the test suite. First, we introduce some relevant concepts. A test case is *redundant* with respect to a test suite if the set of mutants killed by the test suite does not change when the test case is included in the test-suite. A test suite is *non-redundant* if it does not contain redundant test cases. A test suite may contain different non-redundant subsets. These subsets may contain a different number of test cases while retaining the same *testing power* in the sense that they are able to kill the same mutants. Therefore, it is natural to ask for a *minimum size test suite*, or *minimal test suite*, which preserves mutation coverage.

A first scenario arises when execution costs are not available or they can be assumed identical, which is equivalent to the hypothesis that the test cases can be executed at unit cost. The problem is reduced to the minimization of a linear function with binary decision variables and linear constraints. Binary decision variables $x_1, \ldots, x_n$ represent which test cases are selected while constraints ensure that mutation coverage does not decrease. Consequently, the following BILP (Binary ILP) can be stated:

$$\arg\min \left\{ \sum_{1 \leq j \leq n} x_j \;\middle|\; \forall i \in [1,m] \sum_{1 \leq j \leq n} e_{ij} x_j \geq 1 \right\} \qquad (1)$$

Thanks to the constraints, it is guaranteed that each valid mutant that can be killed is really killed by at least one selected test case.

A different, more realistic scenario, implies minimizing the overall execution cost of the selected test cases. This cost-aware minimization can be represented through the following BILP:

$$\arg\min \left\{ \sum_{1 \leq j \leq n} \left(x_j \sum_{1 \leq i \leq m} c_{ij} \right) \;\middle|\; \forall i \in [1,m] \sum_{1 \leq j \leq n} e_{ij} x_j \geq 1 \right\} \qquad (2)$$

This is clearly a generalization of the first problem because it is enough to fix $c_{ij} = 1/m$ to obtain the previous formulation. In fact, assuming the same cost for every execution is equivalent to the unit-cost hypothesis in Equation (1).

We have implemented an algorithm in C++ performing the reduction from E and C to the corresponding BILP, solving the BILP with CPLEX [20], checking whether it finds an optimal solution (otherwise, reporting the situation), and producing the expected output along with some simple statistics. During this reduction, rows corresponding to mutants that remain alive are removed[3], as with rows for invalid mutants. Optimal solutions are guaranteed if enough computational resources, specially memory, are granted.

The reduction is linear in the matrix size on the Word RAM model of computation [12] because it traverses the matrices and encodes the equations on the fly, which is clearly $\Theta(mn)$. Such an efficient encoding is key, as ILP is $\mathcal{NP}$-hard [24] and no algorithm with subexponential worst-case time is known for any $\mathcal{NP}$-hard problem. In fact, though "not too inefficient" superpolynomial algorithms for these problems are not precluded by the $\mathcal{P} \neq \mathcal{NP}$ hypothesis, such algorithms cannot exist under the Exponential Time Hypothesis (ETH) [21]. Linear reductions are thus much preferred over, say, square or other polynomial reductions.

5 EXPERIMENTS AND DISCUSSION

As with any complex software it is not generally possible, or sometimes it is simply not convenient, to conduct the tests for WS-BPEL compositions in a *production environment*. Therefore, a *test environment* has been arranged to conduct the experiments. This test environment includes a job queue managed by Condor, a software framework for coarse-grained distributed parallelization of computationally intensive tasks [41]. Different computing infrastructure can be used with Condor. Besides, when necessary, we prepare mock web services that simulate the behavior of the real web services involved so that their compositions can be fully executed in the test environment and the experiments reproduced.

In our case, the testing time ranges from minutes to days, depending on the underlying computing infrastructure and the web service composition, which determine the number of test cases and mutants to execute. In fact, the overall experimentation time can be considerable.

Typically, experiments with WS-BPEL compositions are repeated several times and results are averaged to better cater for variability. Sometimes, we noticed that some executions failed, took too long or their execution times were not consistent. Upon investigation, several causes were identified:

(1) Resource exhaustion under heavy duty.
(2) Deployment errors produced by invalid mutants.
(3) Non-termination induced by valid mutants.
(4) Flaky tests.

Resource exhaustion can be tackled by reducing the number of concurrent threads per machine and restarting the WS-BPEL execution engine periodically to obtain a fresh environment. WS-BPEL execution engines and the surrounding web and application servers can reveal themselves as

[3] Otherwise, they would produce infeasible constraints ($0 \geq 1$).

Table 2: Characteristics of the WS-BPEL compositions under test and results.

	Description	LOC	$\|I\|$	$\|M\|$	$\|D\|$	$\|T\|$	$\|R\|$	Time (s)	Reduction
LA	Loan Approval Service [10, 29]	110	1	60	53	8	3	0.01	62.5%
COMBO2	Artificial combination of several compositions [10]	1281	19	890	646	34	17	0.03	50.0%
TRS	Travel Reservation Service [10]	384	2	213	152	19	12	0.02	36.8%
MS	MetaSearch Engine [10, 28]	633	0	508	424	37	8	0.02	78.8%
LAE	Loan Approval Service Extended [10]	1533	0	3647	2891	95	64	2.80	32.6%

memory-hungry processes that are better kept monitored during experiments.[4]

As for deployment errors, certain constraints have to be enforced to avoid the generation of invalid mutants. However, WS-BPEL is a complex language and invalid mutants may appear even if mutation operators are implemented with great care. In traditional languages (like C, C++, Java, etc.) if a mutation operator produces mutations that are correct at the syntactic and type-system level, the mutant can be compiled and executed. Any further problems are delayed to run time. However, WS-BPEL constructs are riddled with semantic constraints whose violation can prevent proper deployment. These violations are not usually detected in advance by WS-BPEL execution engines. In our experiments, invalid mutants are produced when the order of two activities is reversed inside a `sequence` activity, an `onAlarm` element is removed, or an activity is deleted (operators `ASI`, `APA` and `AEL`, respectively, in Table 1).

Non-termination generally happens when well-formed terminating compositions are mutated into non-terminating compositions. Since it is undecidable to determine whether the execution will eventually halt, a timeout is introduced and those mutants exceeding the grace period are terminated.

Regarding flaky tests, they may appear as a consequence of an improper use of non-deterministic constructs in WS-BPEL leading to random behavior. Sometimes they are induced by mutation. For example, a sequential `forEach` activity can be mutated into a parallel one or into a `flow` activity (operators `ASF` and `AFP`, respectively, in Table 1). This can introduce race conditions and synchronization problems. Flaky tests should be removed when detected.

Once we have completed the experiments and the execution and cost matrices are available, we can proceed to minimize the test suites. We have used five WS-BPEL compositions. Key characteristics of these compositions are included in Table 2 alongside the results of our experiments. In this table, LOC stands for the number of lines of WS-BPEL code, I is the set of invalid mutants, M is the set of valid mutants, D is the set of dead mutants, T is the original test suite, and R is the reduced test suite. The last two columns represent, respectively, the time needed to compute R from matrix E, and the percentage of reduction obtained in the size of the test suite. Times were measured on a single core of a laptop featuring 8 GiB DDR3L and an Intel Core i5 5200U CPU at 2.20 GHz. Our algorithm guarantees that D is kept invariant when T is replaced by R and, therefore, mutation coverage is preserved: the same mutants are killed by the original test suite and the optimal one.

First of all, it is important to remark that the size of R is *guaranteed* to be optimal. In contrast, metaheuristics can only produce reasonable approximations for combinatorial search problems, unless the particular instance is small or simple enough. We face to a constrained discrete combinatorial optimization problem of size $|M| \cdot |T|$, where $|M|$ determines the number of constraints and $|T|$ determines the number of variables and the size of the landscape. In fact, when confronted to a new instance there is no measure of how close or far we are from the optimal solution.

Second, as the WS-BPEL compositions at hand are rather small and the numbers of test cases and mutants are modest, it is difficult to assess in advance whether a cost-aware minimization of T really helps. The most complex composition available for this study is LAE and we have conducted an experiment to precisely measure the execution time of each of its 3647 mutants against the 95 test cases available. The experiment takes about one week of CPU time in our computing infrastructure, but once the cost information is available, the minimization of the test suite is computed in less than 3 seconds in the aforementioned laptop, roughly the same time employed without using the execution cost as a metric.

Finally, the mutation testing time of LAE has been cut from 185.80 hours to 117.32 hours without loosing mutation coverage, as a result of this minimization effort. In a regression testing context this represents significant savings during the whole life cycle.

The source code for the five WS-BPEL compositions, WSDL specifications and mocks for web services, test suites, and scripts for the experiments are available at the UCASE WS-BPEL repository,[5] which is maintained by the UCASE Software Engineering Research Group at the University of Cádiz. MuBPEL is freely available at https://ucase.uca.es/mubpel under an open-source license.

6 THREATS TO VALIDITY

The main threats to the validity of our conclusions are concerned with external validity, i.e. the possibility of generalizing our findings to other web service compositions with

[4] Actually, a severe memory leak was detected in the underlying WS-BPEL execution engine during the execution of a preliminary experiment.

[5] https://neptuno.uca.es/redmine/projects/wsbpel-comp-repo/wiki

different features. Since we have only five case studies available, the natural way to add evidence to our findings is by replicating our results with different compositions drawn from different application domains. This is not as easy as it may seem, as WS-BPEL compositions are scarce. Freely available WS-BPEL compositions are indeed very rare. On the one hand, compositions are of economic importance for the enterprises that develop them. On the other hand, business logic and information of strategic value can be buried inside the composition. Besides, it is not easy to reproduce the environment in which an enterprise web service composition is executed.

Scalability can be also an issue but, as mentioned before, the size of WS-BPEL compositions is usually not high when compared with applications written in mainstream programming languages. In our experience, the time devoted to the optimization of the test suite (seconds) is negligible when compared to the testing time (minutes, hours, or even days, depending on the WS-BPEL composition and the number of test cases). Unless the number of test cases is really high, we do not foresee scalability problems. We have easily managed around one hundred test cases and thousands of mutants for a composition. However, as mentioned in Section 4, the underlying optimization problem is $\mathcal{NP}$-hard and we cannot exclude the possibility of stumbling on an insidiously hard instance of moderate size. After all, the search space for even one hundred test cases is really huge.

7 CONCLUSIONS AND FUTURE WORK

We have introduced an exact search-based technique to minimize test suites while maintaining their mutation coverage. The minimization of test suites is a hard computational problem whose solution is relevant both to reduce the cost of mutation testing and to precisely assess the quality of existing test suites. This problem can be addressed with SBSE techniques, including exact techniques. We have applied ILP as an exact technique to reduce the testing effort with excellent results. Our technique can be adapted to different formalisms, though this paper focused on testing WS-BPEL compositions, as it poses several interesting problems. Even when web service compositions are relatively small, as they just orchestrate web services, their execution can be very expensive because the deployment and execution of web services and the underlying infrastructure are not trivial. Therefore, although test suites for the compositions themselves are generally small, it is important to reduce their size as much as possible without losing coverage.

This paper deals with testing in the context of WS-BPEL compositions and a particular technique, mutation testing, but the methodology that we have followed and the lessons learned are general enough to be of application to other programming languages and test frameworks. Actually, we just require that the execution of individual test cases allows us to distinguish which characteristics of the SUT are exercised. In particular, the minimization of the test suite can be done whenever an execution matrix is available.

Let us remark that the performance is excellent even for the most complex composition available in this study (LAE) for which a cost-aware minimization of the test suite has been performed too.

One highlight of our work is that, in contrast to most authors, we do not need to assume the same execution cost for each test case. This assumption is equivalent to consider a *unit cost* for every test case, which is indeed an extremely simplifying assumption. In fact, such an assumption enables certain reductions in the number of test cases that are not possible in its absence. This can be achieved by preprocessing the test suites with very simple heuristics (e.g. subsumption). Thus, heuristics can be applied to reduce the size of the instance at hand before addressing the minimization effort with other algorithms. Sometimes the execution costs for individual test cases are not readily available and this assumption can be justified. However, when the execution costs are known or predictable in advance, assuming unit costs is unrealistic and can clearly lead to suboptimal results, specially in the context of regression testing.

Future work will be devoted to validating the techniques described above on new WS-BPEL compositions and extending the domain of application to other languages. In particular, we are interested in reducing the cost of testing C++ programs. Furthermore, a comparison with other SBSE techniques might also prove interesting.

ACKNOWLEDGMENTS

This work is partially funded by the Spanish Ministry of Economy and Competitiveness through the National Program for Research, Development and Innovation, with funds of the European Union (European Regional Development Fund - ERDF), projects TIN2015-65845-C3-1-R, TIN2015-65845-C3-3-R (DArDOS) and, also for the first author, project TIN2014-60844-R (SAVANT). The fourth author is also partially funded by the Comunidad de Madrid, project S2013/ICE-3006 (SICOMORo-CM).

REFERENCES

[1] ActiveVOS. 2009. Version 5.0.2 of the ActiveBPEL engine. (Jan. 2009). Retrieved February 1, 2018 from http://sourceforge.net/projects/activebpel502

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. 2004. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *GECCO 2004: Proceedings of the Genetic and Evolutionary Computation Conference*. 1338–1349.

[3] F. Arito, F. Chicano, and E. Alba. 2012. On the Application of SAT Solvers to the Test Suite Minimization Problem *(LNCS)*, Vol. 7515. Springer, 45–59.

[4] D. Baldwin and F. G. Sayward. 1979. *Heuristics for Determining Equivalence of Program Mutations*. Techreport 276. Yale University, New Haven, Connecticut.

[5] J. Black, E. Melachrinoudis, and D. Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. In *Proc. 26th Int. Conf. on Soft. Engineering*. 106–115.

[6] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable

Clean Program Assumption. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 597–608. https://doi.org/10.1109/ICSE.2017.61

[7] H. Do, S. Elbaum, and G. Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Software Engineering* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[8] M. Ellims, D. Ince, and M. Petre. 2007. The Csaw C Mutation Tool: Initial Results. In *TAICPART-Mutation'07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - Muation*. IEEE Computer Society Press, 185–192.

[9] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. 2008. Mutation Operators for WS-BPEL 2.0. In *ICSSEA 2008, 21th International Conference on Software & Systems Engineering and their Applications*. 7.

[10] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Domínguez-Jiménez, and A. García-Domínguez. 2015. Quality metrics for mutation testing with applications to WS-BPEL compositions. *Softw. Test., Verif. Reliab.* 25, 5–7 (2015), 536–571.

[11] K. Fischer. 1977. A test case selection method for the validation of software maintenance modifications. In *Proceedings of International Computer Software and Applications Conference*. IEEE Computer Society Press, 421–426.

[12] M. L. Fredman and D. E. Willard. 1993. Surpassing the Information Theoretic Bound with Fusion Trees. *J. Comput. Syst. Sci.* 47, 3 (1993), 424–436. https://doi.org/10.1016/0022-0000(93)90040-4

[13] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman.

[14] B. J. M. Grün, D. Schuler, and A. Zeller. 2009. The Impact of Equivalent Mutants. In *Mutation'09: 4th International Workshop on Mutation Analysis*. IEEE Computer Society Press, 192–199.

[15] M. Harman, R. Hierons, and S. Danicic. 2001. *Mutation Testing for the New Century*. Kluwer Academic Publishers, Chapter The relationship between program dependence and mutation analysis, 5–13.

[16] M. Harman, J. Krinke, I. Medina-Bulo, F. Palomo-Lozano, J. Ren, and S. Yoo. 2014. Exact scalable sensitivity analysis for the next release problem. *ACM Trans. Softw. Eng. Methodol.* 23, 2 (2014), 19:1–19:31. https://doi.org/10.1145/2537853

[17] R. Hierons, M. Harman, and S. Danicic. 1999. Using Program Slicing to Assist in the Detection of Equivalent Mutants. *Software Testing, Verification and Reliability* 9 (1999), 233–262.

[18] W. E. Howden. 1982. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering* 8, 4 (1982), 371–379.

[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. 1994. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*. 191–200. https://doi.org/10.1109/ICSE.1994.296778

[20] IBM. 2018. IBM ILOG CPLEX Optimization Studio 12.8.0. (Jan. 2018). Retrieved February 1, 2018 from http://www-01.ibm.com/support/docview.wss?uid=swg27050618

[21] R. Impagliazzo and R. Paturi. 1999. Complexity of k-SAT. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity*. 237–240. https://doi.org/10.1109/CCC.1999.766282

[22] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[23] M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey (Eds.). 2010. *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer. https://doi.org/10.1007/978-3-540-68279-0

[24] R. M. Karp. 1972. *Complexity of Computer Computations*. Plenum Press, Chapter Reducibility among Combinatorial Problems, 85–103.

[25] D. Li, Y. Jin, C. Sahin, J. Clause, and W. G. J. Halfond. 2014. Integrated Energy-directed Test Suite Optimization. In *ISSTA 2014: Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 339–350.

[26] L. Li, M. Harman, F. Wu, and Y. Zhang. 2017. The Value of Exact Analysis in Requirements Selection. *IEEE Trans. Software Eng.* 43, 6 (2017), 580–596. https://doi.org/10.1109/TSE.2016.2615100

[27] R. E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. 2013. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *29th IEEE Int. Conf. on Software Maintenance*. IEEE, 404–407.

[28] P. Mayer and D. Lübke. 2006. Towards a BPEL unit testing framework. In *TAV-WEB'06, 2006 workshop on Testing, analysis, and verification of web services and applications*. ACM, 33–42.

[29] OASIS. 2007. Web Services Business Process Execution Language 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. (2007).

[30] A. J. Offutt. 1988. *Automatic test data generation*. Ph.D. Dissertation. Georgia Institute of Technology, Atlanta, GA, USA.

[31] A. J. Offutt and W. M. Craft. 1994. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification and Reliability* 4, 3 (1994), 131–154.

[32] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology* 5 (1996), 99–118. Issue 2.

[33] A. J. Offutt and J. Pan. 1997. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability* 7, 3 (1997), 165–192.

[34] A. J. Offutt and R. H. Untch. 2001. *Mutation Testing for the New Century*. Kluwer Academic Publishers, Chapter Mutation 2000: Uniting the Orthogonal, 34–44.

[35] A. J. Offutt and J. M. Voas. 1996. *Subsumption of condition coverage techniques by mutation testing*. Technical Report ISSE-TR-96-01. Department of Computer Science, George Mason University.

[36] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*. 936–946. https://doi.org/10.1109/ICSE.2015.103

[37] D. Schuler, V. Dallmeier, and A. Zeller. 2009. Efficient Mutation Testing by Checking Invariant Violations. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*. 69–80.

[38] D. Schuler and A. Zeller. 2010. (Un-)Covering Equivalent Mutants. In *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 45–54.

[39] B. Schwarz, D. Schuler, and A. Zeller. 2011. Breeding High-Impact Mutations. In *ICSTW '11: Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. 382–387.

[40] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu. 2014. Web services composition: A decade's overview. *Information Sciences* 280 (2014), 218–238. https://doi.org/10.1016/j.ins.2014.04.054

[41] D. Thain, T. Tannenbaum, and M. Livny. 2005. Distributed Computing in Practice: The Condor Experience: Research Articles. *Concurr. Comput.: Pract. Exper.* 17, 2-4 (Feb. 2005), 323–356. https://doi.org/10.1002/cpe.v17:2/4

[42] N. Veerapen, G. Ochoa, M. Harman, and E. K. Burke. 2015. An Integer Linear Programming approach to the single and bi-objective Next Release Problem. *Information and Software Technology* 65 (2015), 1–13. https://doi.org/10.1016/j.infsof.2015.03.008

[43] J. Voas and G. McGraw. 1997. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons.

[44] L. Wang, R. Wan, M. Wang, and M. Li. 2009. Generating Small Combinatorial Test Suite via LP. In *WISA'09: Proc. of the 2009 Int. Symp. on Web Information Systems and Applications*. 226–229.

[45] M. R. Woodward and K. Halewood. 1988. From Weak to Strong, Dead or Alive? an Analysis of Some Mutationtesting Issues. In *TVA'88: 2nd Workshop on Software Testing, Verification, and Analysis*. IEEE Computer Society, 152–158.

[46] S. Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[47] H. Zhong, L. Zhang, and H. Mei. 2008. An experimental study of four typical test suite reduction techniques. *Information and Software Technology* 50, 6 (2008), 534–546.