SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–36
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Quality metrics for mutation testing with applications to WS-BPEL compositions

Antonia Estero-Botaro[1]*, Francisco Palomo-Lozano[1], Inmaculada Medina-Bulo[1], Juan José Domínguez-Jiménez[1] and Antonio García-Domínguez[1]

[1] *University of Cádiz. Escuela Superior de Ingeniería, Spain.*

SUMMARY

Mutation testing is a successful testing technique based on fault injection. However, it can be very costly and several cost-reduction techniques for reducing the number of mutants have been proposed in the literature. Cost reduction can be aided by an analysis of mutation operators, but this requires the definition of specialized metrics. Several metrics have been proposed before, though their effectiveness and relative merits are not easy to assess. A step-ahead in the evaluation of mutation-reduction techniques would be a better metric to determine objectively the quality of a set of mutants with respect to a given test-suite. This work introduces such a metric, which is naturally extended to mutation operators and may be used to reduce the number of mutants, particularly of equivalent mutants. Finally, a firm mutation analysis tool for WS-BPEL service compositions is presented and experimental results obtained by comparing different metrics on several compositions are presented. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Mutation testing; firm mutation; quality metrics; quality of mutation operators; service compositions; WS-BPEL

1. INTRODUCTION

Mutation testing [1] is a testing technique injecting simple faults in the program under test through the use of *mutation operators*. As a result, *mutants* are obtained: variants of the original program. The original program and its mutants are executed on a given test-suite. When the output of a mutant for a test case does not agree with the output of the original program for the same test case, the mutant has been *killed* by that test case and, so, it is *dead*. This means that the test case has served a purpose: detecting the fault that is present in the mutant. If the output is always the same as the original program output for every test case in the test-suite, then the mutant remains *alive*.

Each mutation operator corresponds to a category of typical errors that the developer could incur. Thus, if a program contains the arithmetic expression $a + b + c$ and there is a mutation operator available to replace the arithmetic operator $+$ by the arithmetic operator $-$, mutants containing the expressions $a - b + c$, $a + b - c$, and $a - b - c$ can be produced. However, the first two are simplest in a sense than the latter, as they amount to a single syntactic change. They are *first-order mutants*, while the latter is a *higher-order mutant*.† Just to illustrate the simplest case, let us consider an extension of the previous example: the expression $a_1 + \cdots + a_n$ can be mutated into $n - 1$ different

*Correspondence to: antonia.estero@uca.es
†Please ensure that you use the most up to date class file, available from the STVR Home Page at
http://www3.interscience.wiley.com/journal/13635/home
†In particular, a second-order one.

mutants by the sole consideration of the "replace $+$ by $-$" mutation operator. This exponentially increases to $2^{n-1}-1$ different mutants when mutations can be accumulated resulting in higher-order mutants.

Last, but not least, a mutation may happen to produce a variation of the original program with the same behavior. The number of such *equivalent mutants* plays its role in mutation metrics. Distinguishing them from mutants that simply stay alive after executing a test-suite is not decidable, and, even when so, it is usually done by hand.

Mutation testing presents two main problems: the computational cost associated to the execution of a huge number of mutants and the detection of equivalent mutants. Moreover, a terminating program can be mutated into a non-terminating one. However, 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 problems worsen in the presence of a great number of mutation operators, as it is customary when mutation testing is applied to real-life languages [1]. Even if attention is restricted to first-order mutants, as it will be done henceforth, every mutant produced has to be executed on every test case in the test-suite. Even modest-size units under test may need a considerable number of test cases. Testing complexity is not a mere question of size, but it is more related to the number of potential paths in a computation.

In order to reduce the overall cost, several techniques have been devised, which are well-reviewed by Jia and Harman [2], and Polo and Reales [3]. Of course, a common approach is to focus on the reduction of the number of mutants. This reduction can be achieved by a number of techniques, including *mutant sampling* [4, 5], *mutant clustering* [6], *selective mutation* [7], *higher-order mutation* [2], and *evolutionary mutation* [8].

The computational cost of mutation testing can also be reduced by optimizing the execution of mutants. Techniques such as weak [9] and firm [10] mutation have been proposed for this purpose. Nowadays, traditional mutation is known as strong mutation. Strong mutation requires three conditions defined by Offutt [11]: *reachability*, *necessity*, and *sufficiency*. Differences between these three techniques are based on the way the mutant behavior is inspected to decide whether the mutant is killed by a test case.

With *strong mutation*, a mutant is considered to be killed by a test case when the mutant and the original program outputs differ for this test case. That is, a mutant m of a program p is killed by the test case t in the strong sense if m and p produce different outputs when executed on t. However, with *weak mutation* the outputs are compared right after the mutation execution instead of at the end of execution.

Woodward and Halewood [10] suggested using the final state of a program in place of the output. Harman et al. [12] identify this approach based on the final state as a generalization of the original approach. They consider that testing a mutant consists of inspecting a set of variables, which they call the *inspection set*, IS. A mutant m of a program p is killed by the test case t on the inspection set IS, when m and p produce different values for some $x \in IS$.

Woodward and Halewood [10] also introduced *firm mutation* as a compromise between strong and weak mutation. Let k be the point where p is mutated into m. Under firm mutation, the original program and the mutant can be compared on some inspection set IS at some point of the execution path, but not before k, which is the execution point of the mutated instruction. Weak and strong mutation are then particular cases of firm mutation. In weak mutation, k is just the mutation point, while in strong mutation k is the exit point of the program. Therefore, firm mutation subsumes both weak and strong mutation [12], in the sense that both can be regarded as special cases within a firm mutation analysis tool.

One of the drawbacks of firm mutation is that it does not specify where the comparisons must be done. For this reason, only a few number of mutation tools use firm mutation. Jackson and Woodward [13] consider Java and, in general, object-oriented languages specially amenable to firm mutation. They propose a system which applies firm mutation to Java. Their system mutates Java methods and compares the outputs of original and mutated methods.

DOI: 10.1002/stvr

Regarding the equivalent mutant problem many authors have proposed different techniques to automatically detect equivalent mutants [14, 15, 7, 16, 12, 17, 18, 19, 20, 21, 22, 23, 24]. However, in practice, this detection is done by hand, being a time-consuming and error-prone task. A different approach tries to detect those operators contributing most to the existence of equivalent mutants.

This work introduces new metrics to measure the quality of mutants and mutation operators. These metrics can be used for reducing the number of mutants by discarding those operators whose quality is under an appropriate threshold. Moreover, they punish those operators producing a high number of equivalent mutants. This allows us to obtain a set of high-quality operators that is smaller than the original and generates less mutants and less equivalent mutants, while maintaining similar capabilities.

The main contribution in this work is a precise definition of metrics to measure the quality of mutants and mutation operators. A quality metric for mutation operators will allow us to establish which mutation operators deserve more attention in a language, and even discard some of them as ineffective. In order to assess the effectiveness of the proposed metrics, they will be applied to different service compositions developed in the WS-BPEL language.

Another contribution is the introduction of a new mutation analysis tool, which has been named MuBPEL, to provide the framework for experimentation with mutation in WS compositions. MuBPEL adds to the short list of tools capable of firm mutation analysis.

The choice of WS-BPEL is not casual. Testing software as services is not an easy task. Service compositions represent a maximum of flexibility, enabling the creation of new web-services from existing ones in a distributed environment, while tackling with non-conventional programming concerns at the same time. These compositions model complex business processes from simpler ones, offering the resulting component as a new web-service. WS-BPEL 2.0 is an OASIS standard [25] and an industrial-strength reference. There is a need for efficient and effective testing techniques for this kind of software and this need is increasing as the relevance and economic impact of service compositions are growing. Finally, testing WS-BPEL compositions is especially costly because of the very nature of the underlying technology: services have to be deployed, a BPEL engine has to be started, etc.[‡] This makes WS-BPEL an excellent target for cost reduction techniques.

The proposed metrics contribute to reduce the cost of mutation testing for WS-BPEL by identifying those operators producing low quality mutants, which are later discarded or improved accordingly.

The structure of the rest of the paper is as follows. Section 2 introduces a formalization of the theory of mutation testing. Next, Section 3 formally defines the proposed metrics for mutant quality and mutation operator quality. Previous work focusing on mutation operator quality and the relation of our metrics to other metrics is described in Section 4. Mutation testing in a WS-BPEL context with a new firm mutation analysis tool, MuBPEL, is introduced in Section 5. In Section 6 the metrics are applied to evaluate the WS-BPEL mutation operators. The experimental method followed is also described in this section. Then, in Section 7, some operators are improved with the help of the results obtained in the previous section. An experimental comparison between metrics is provided by Section 8. Finally, Section 9 presents conclusions and future research lines.

2. MUTATION: THEORETICAL BACKGROUND

Now a formalization of a significant fragment of the theory of mutation testing is introduced. The objective is twofold. On the one hand, a consistent but clear notation is developed to represent several concepts whose definitions are sparse, missing or even slightly different in the literature. On the other hand, precise definitions of weak mutant, resistant mutant, and hard to kill mutant

[‡]Just the testing infrastructure needed for BPEL compositions is far from trivial, at least when compared to traditional languages.

DOI: 10.1002/stvr

are introduced. This will pave the way to the definitions of quality of a mutant and of a mutation operator in Section 3. This formalization will allow for reasoning accurately about these concepts.

2.1. Basic Notions

Let p be a program under test. A first-order mutant of p is a simple variant obtained by modifying, inserting or deleting a single syntactic unit of p. The following sets are defined:

- I_p: The input space of p.
- T_p: A test-suite (a set of test cases) for p.
- M_p: A set of valid mutants of p.[§]

Let $m \in M_p$ and $t \in T_p \subseteq I_p$.[¶] The following relation is also defined, where $q(t)\downarrow$ means that program q halts on test case t, while $q(t)\uparrow$ has the opposite meaning.

Definition 1 (Mutant equivalence under a test case for strong mutation)
Let $\approx_S$ be the binary relation such that $m(t) \approx_S p(t)$ iff mutant m is equivalent to program p under test case t and strong mutation:

$$m(t) \approx_S p(t) \equiv (m(t)\downarrow \wedge p(t)\downarrow \wedge m(t) = p(t)) \vee (m(t)\uparrow \wedge p(t)\uparrow) \tag{1}$$

This means that a mutant m is equivalent to its original program p under test case t if, upon termination, they agree on the result of their computations or if none of them terminates.

Termination conditions can be relaxed in a number of ways, giving us different notions of equivalence. However, the formalization developed here is highly independent of this issue.

In order to introduce firm mutation the above definition of mutant equivalence is generalized by just using an inspection set IS consisting of state variables $(s_1, s_2, \dots)$ and output variables $(o_1, o_2, \dots)$.

$$p(t) = \langle s_1, s_2, \dots, o_1, o_2, \dots \rangle \qquad\qquad m(t) = \langle s'_1, s'_2, \dots, o'_1, o'_2, \dots \rangle$$

Thus, $s_1, s_2, \dots, o_1, o_2, \dots$ are the values taken by the variables included in IS for p and $s'_1, s'_2, \dots, o'_1, o'_2, \dots$ are their counterparts for m. While output variables can only be established once execution has been completed, state variables can have their values at some point of the execution. Thus, technically, equivalence of p and m under t in the firm sense can be refuted in many cases by paying attention to the state variables, without having to complete the execution of m. This is the advantage of firm mutation.

Definition 2 (Mutant equivalence under a test case for firm mutation)
Let $\approx_F$ be the binary relation such that $m(t) \approx_F p(t)$ iff mutant m is equivalent to program p under test case t and firm mutation:

$$\begin{aligned} m(t) \approx_F p(t) \equiv (&m(t)\downarrow \wedge p(t)\downarrow \wedge m(t) \simeq_s p(t) \wedge m(t) \simeq_o p(t)) \vee \\ &(m(t)\uparrow \wedge p(t)\uparrow \wedge m(t) \simeq_s p(t)) \end{aligned} \tag{2}$$

where

$$m(t) \simeq_s p(t) \equiv \bigwedge_k (s_k = s'_k) \qquad\qquad m(t) \simeq_o p(t) \equiv \bigwedge_k (o_k = o'_k)$$

This means that the behavior of the mutant m and the original program p are the same under a test case t if, upon termination, they agree on the values of the variables included in the inspection set. As soon as there is a difference between a state variable value in m and p, execution can be stopped,

[§]Mutation operators can produce invalid mutants that fail to execute because of syntactic or semantic errors. Invalid mutants are not executed, thus only the mutants of p that can be executed are considered.
[¶]Test cases are just defined as particular inputs, because in mutation testing the original program acts as the oracle.

DOI: 10.1002/stvr

as $m(t) \not\simeq_s p(t)$ and this would imply that $m(t) \not\approx_F p(t)$. This allows for non-terminating systems where p, and consequently m, may not terminate.

It can be shown that, with the above formal definitions, the equivalence of mutants under strong mutation is a particular case of the equivalence of mutants under firm mutation. Definition 1 is subsumed by 2 when the inspection set consists just of the output variables. When there are no state variables, $m(t) \simeq_s p(t)$ holds, as it is an empty conjunction, which is true. The values of the output variables can be obtained when $m(t)\downarrow$ and $p(t)\downarrow$, and then $m(t) \simeq_o p(t)$ amounts to $m(t) = p(t)$, which is a simple comparison of the results of both computations. Therefore, both definitions agree.

In the rest of the paper, the dependency on p will be omitted, assuming all these concepts implicitly defined for a given program under test. Moreover, several notions depending on I, T, or M will be defined. For the sake of readability, these dependencies will be also omitted from the notation, as they will be clear from their contexts.

Definition 3 (Killing a mutant)
Let kills be the binary relation such that t kills m iff mutant m is killed by test case t:

$$t \text{ kills } m \equiv m(t) \not\approx p(t) \tag{3}$$

Please, notice that $\approx \in \{\approx_S, \approx_F\}$, the particular relation depending on the kind of mutation used.

Definition 4 (Test cases killing mutant m)
Let K_m be the set of test cases that kill mutant m:

$$K_m = \{t \in T \mid t \text{ kills } m\} \tag{4}$$

Therefore, obviously, $\varnothing \subseteq K_m \subseteq T$.

Definition 5 (Dead mutants)
Let D be the set of of dead mutants:

$$D = \{m \in M \mid \exists t \in T \; t \text{ kills } m\} \tag{5}$$

Definition 6 (Potentially equivalent mutants)
Let P be the set of potentially equivalent mutants:

$$P = \{m \in M \mid \neg\exists t \in T \; t \text{ kills } m\} \tag{6}$$

Potentially equivalent mutants are just those mutants staying alive, i.e, the survivors after executing M against the test-suite T:

$$D = \{m \in M \mid K_m \neq \varnothing\} \qquad\qquad P = \{m \in M \mid K_m = \varnothing\}$$

Therefore, it is clear that $M = D \cup P$, $D \cap P = \varnothing$, $D = M - P$, and $P = M - D$.

Definition 7 (Equivalent mutants)
Let E be the set of equivalent mutants:

$$E = \{m \in M \mid \neg\exists t \in I \; t \text{ kills } m\} \tag{7}$$

Therefore, obviously, $E \subseteq P \subseteq M$.

The characteristic predicate of E is, in general, undecidable. This means that there can be no algorithm taking two arbitrary programs and telling us whether they are or not equivalent. In fact, as a consequence of the unsolvability of Turing's Halting Problem it cannot even be determined if an arbitrary program terminates its execution for a given input. Of course, this does not prevent these problems to be solved for particular instances.

Definition 8 (Mutation score)
Let $\mathcal{S}$ be the mutation score:

$$\mathcal{S} = \frac{|D|}{|M| - |E|} \tag{8}$$

DOI: 10.1002/stvr

2.2. Adequacy and Redundancy of Test-suites

Definition 9 (Adequate test-suite)
T is *adequate* for M if every non-equivalent mutant in M is killed by some test case in T.

$$T \text{ is adequate iff } \forall m \in M - E \; \exists t \in T \; t \text{ kills } m \tag{9}$$

When T is adequate, $P = E$ and $D = M - E$. Moreover, being $E \subseteq M$, it is clear that $|D| = |M| - |E|$. Therefore, $\mathcal{S} = 1$ for adequate test-suites.

Definition 10 (Mutants covering test case t)
Let C_t be the set of mutants killed by test case t:

$$C_t = \{m \in M \mid t \text{ kills } m\} \tag{10}$$

Therefore, obviously, $\varnothing \subseteq C_t \subseteq M$. Of course, this definition can be extended to test-suites in a natural way:

$$C_T = \bigcup_{t \in T} C_t \tag{11}$$

Definition 11 (Non-redundant test-suite)
A test case t is *redundant* with respect to a test-suite T if $C_T = C_{T \cup \{t\}}$. T is *non-redundant* if it contains no test case t such that t is redundant with respect to $T - \{t\}$.

A test-suite may contain different non-redundant subsets. These subsets are interesting, as they may contain far fewer test cases while retaining the same testing power.

2.3. Resistance of Mutants

Usually, mutation testing classifies mutants executed against a test-suite as alive or dead. Nevertheless, to study the quality of mutants the authors think that it is necessary to rank dead mutants by the number of test cases that kill them. In order to do so, the following concepts are proposed: *weak mutant*, *resistant mutant*, and *hard to kill mutant*. Their formal definitions follow.

Definition 12 (Weak mutant)
A mutant is *weak* when killed by every test case in the test-suite. Let W be the set of weak mutants:

$$W = \{m \in M \mid K_m = T\} \tag{12}$$

Definition 13 (Resistant mutant)
A mutant is *resistant* when killed by a single test case. Let be R the set of resistant mutants:

$$R = \{m \in M \mid |K_m| = 1\} \tag{13}$$

Definition 14 (Hard to kill mutant)
A resistant mutant is *hard to kill* when it covers just the test case that kills it. Let H be the set of hard to kill mutants:

$$H = \{m \in R \mid \forall t \in K_m \; |C_t| = 1\} \tag{14}$$

Please, notice that a resistant mutant can also be weak in the degenerate case that $|T| = 1$. This is not a defect of the above definitions, it is just that no much significant information can be extracted from a mutation system with just one test case.

3. FORMAL DEFINITION OF QUALITY

The classification of mutants into weak, resistant, and hard to kill is by no means exhaustive. Clearly, there is a whole spectrum between being killed by every test case and being killed by a single test case. Although this distinction is important and captures important groups of mutants, a finer measure would be desirable.

DOI: 10.1002/stvr

3.1. Quality of Mutants

In order to detect the defects introduced by mutants, resistant mutants are more interesting than weak mutants because they need more specific test cases to detect them. From this point of view, hard to kill mutants are the most interesting of all. The metric proposed for measuring the quality of a mutant depends on the numbers of test cases killing the mutant and the number of mutants covering these test cases. As any other metric for mutation, this metric is not independent of the test-suite used, so the authors consider that it is very important for test-suites to accomplish two properties: they must be adequate (Definition 9) and non-redundant (Definition 11). Henceforth, it will be assumed that both properties hold. Later, in Section 4, it will be explained why minimality is also an important property to take into account.

Definition 15 (Quality of a mutant)

$$Q_m = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{(|M| - |E|) \cdot |T|} \displaystyle\sum_{t \in K_m} |C_t|, & m \in D \end{cases} \tag{15}$$

Now, the behavior of mutants will be analyzed with respect to the above definition of quality.

The least significant mutants to determine the correct behavior of a program are the weak ones, which are killed by every test case. If all the mutants happen to be weak, their quality will be 0, because $\sum_{t \in K_m} |C_t| = (|M| - |E|) \cdot |T|$.

Resistant mutants are the most interesting ones, especially if they are hard to kill. When one of the mutants is hard to kill, its quality is $1 - 1/((|M| - |E|) \cdot |T|)$.

A resistant mutant is killed by a single test case. Several degrees of resistance can be established by taking into account the number of mutants killed by this test case. For example, the worst resistant mutant must be killed by a single test case that also kills the rest of the mutants. Therefore, $\sum_{t \in K_m} |C_t| = |D| = |M| - |E|$ and the quality of the worst resistant mutant reduces to $1 - (|M| - |E|)/((|M| - |E|) \cdot |T|) = 1 - 1/|T|$. Analogously, other quality thresholds for a resistant mutant can be established, for example, when the test case killing the mutant also kills half of the rest of the mutants, one fourth of them, etc.

3.2. Quality of Mutation Operators

The definition of quality of a mutant can be generalized to a set of mutants in a natural way.

Definition 16 (Quality of a set of mutants)

$$Q_M = \frac{1}{|M|} \sum_{m \in M} Q_m \tag{16}$$

This definition can be used to measure the quality of the mutants generated by a particular mutation operator. So, this metric can be regarded as a means to assess the quality of operators straightforwardly.

Definition 17 (Quality of a mutation operator)

$$Q_O = \frac{1}{|M_O|} \sum_{m \in M_O} Q_m = Q_{M_O} \tag{17}$$

It is clear that the quality of operator O is just the quality of the set of mutants that this operator generates, M_O.

Definition 17 computes the quality of all the mutants produced by a mutation operator, including the equivalent mutants. In some situations it can be interesting to compute the quality of just the dead mutants generated by the mutation operator. The following definition addresses this issue:

DOI: 10.1002/stvr

Definition 18 (Quality of the dead mutants generated by a mutation operator)

$$Q_{D_O} = \frac{1}{|D_O|} \sum_{m \in D_O} Q_m \tag{18}$$

Q_{D_O} and Q_O will agree when mutation operator O does not generate equivalent mutants. Otherwise, Q_{D_O} will be greater than Q_O.

As it was warned in Section 2, dependencies on M and T have been consistently omitted from notation. In fact, $Q_M = Q_M(T) = Q(M,T)$, that is, the quality of a mutation system $\langle M, T \rangle$ is unsurprisingly a function of both M and T. When analyzing an operator, attention will be restricted to the set of mutants generated by that operator, $M_O \subseteq M$, and the set of test cases used to kill them, $T_O \subseteq T$. Consequently, in this setting, Q_O and Q_{D_O} really stand for $Q(M_O, T_O)$ and $Q(D_O, T_O)$, respectively. This distinction will be important in the next section, which will develop the relation between the proposed metrics and other existing quality metrics.

4. RELATION TO OTHER QUALITY METRICS

Derezińska [26] assesses the quality of object-oriented mutation operators in the C# language. She calculates the *effectiveness* of test cases for an operator as the ratio of the number of test runs killing mutants generated by the operator to the total number of test runs performed on non-equivalent mutants generated by the operator.

Estero et al. [27] have shown that Derezińska's effectiveness can be alternatively defined as:

$$\mathcal{E} = \frac{\sum_{m \in M} |K_m|}{(|M| - |E|) \cdot |T|}$$

where M is the set of mutants generated, T is the test-suite, E is the set of equivalent mutants, and K_m is the set of test cases killing mutant m.

Smith et al. [28] define the *utility* of a mutation operator. They classify their mutants into *dead on arrival* (DOA, those killed by the initial test-suite), *killed* (those killed by a specific test case added to the initial test-suite), *crossfire* (those killed by a test case specifically designed to kill a different mutant), and *stubborn* (those that could not be killed by adding additional test cases).[‖] They define the operator utility function as a linear combination of the percentages of mutants in each class:

$$\mu = \alpha \cdot \text{Killed} + \beta \cdot \text{DOA} + \beta \cdot \text{Crossfire} - \alpha \cdot \text{Stubborn}$$

and execute the mutants generated by this operator against an initial test-suite to calculate its utility. All the mutants killed by this test-suite are DOA. Then, they select one living mutant to create a new test case intended to kill this, and only this, mutant. They execute the new test-suite against the remaining living mutants. The selected mutant is recorded as killed. If other mutants are killed by the same test case they are recorded as crossfire.

One major drawback of this approach is that the utility definition does not just depend on the initial test-suite, but also on the order in which the living mutants are selected after the initial test-suite execution. Moreover, it is not easy to justify a particular choice of the parameters α and β in the definition.

Estero et al. [27] have evaluated WS-BPEL 2.0 mutation operators quantitatively. They have measured the effectiveness of adequate and non-redundant test-suites and computed the percentage of equivalent, invalid, resistant and weak mutants produced by each operator.

Hu et al. [29] have studied empirically Java class mutation operators implemented in muJava [30]. This work defines the *mutation operator strength* as:

$$MOS = \frac{\#minTests}{\#muts - \#notKilled}$$

[‖] This terminology differs from that commonly accepted.

DOI: 10.1002/stvr

where $\#minTests$ is the minimal number of tests needed to kill all killable mutants generated by the mutation operator studied, $\#muts$ is the total number of mutants generated by the mutation operator, and $\#notKilled$ is the number of mutants alive. As they use non-redundant test-suites, the size of test-suites is bounded by the number of mutants. If a single unique test is needed to kill each mutant, the ratio will be 1 and the operator is strong. From the results obtained they draw conclusions about which mutation operators are more or less useful.

The present work introduces an alternative approach to measure mutation operators quality. First, the quality of an individual mutant is defined. Then, this definition is lifted to a set of mutants. Finally, it is observed that this set of mutants can be formed with just the mutants generated by a particular mutation operator.

Another major difference between this work and those of Derezińska [26] and Smith et al. [28] is that the work presented here uses adequate and non-redundant test-suites. Moreover, unlike Hu et al. [29], minimal test-suites are obtained. Why these properties are fundamental to achieve fair comparisons will be clear in Section 4.4.

It is interesting to compare different metrics analytically. The proposed quality metrics will be compared with effectiveness and mutation strength.

4.1. Preliminaries

Next, it will be argued that Q_O is a finer metric than effectiveness and mutation strength in the sense that it can distinguish operators that, from the viewpoint of the other metrics, are similar in goodness. The point is that, as shown next, the factor:

$$\mathcal{I} = \sum_{m \in D_O} \sum_{t \in K_m} |C_t| \tag{19}$$

modulates the value of effectiveness and mutation strength when computing Q_O from them. This double sum plays a key role as it represents the interactions between mutants and test cases for the whole mutation system $\langle M_O, T_O \rangle$ corresponding to the mutation operator under study, O. Test-suites will be assumed adequate. Please, remember than when T_O is adequate $P_O = E_O$, $D_O = M_O - E_O$, and $|D_O| = |M_O| - |E_O|$, where $|D_O|$, $|P_O|$, and $|E_O|$ are the corresponding restrictions of $|D|$, $|P|$, and $|E|$ to mutants generated by O. Obviously, $M_O = D_O \cup E_O$ too. Finally, it will be discussed why minimality matters.

4.2. Quality versus Effectiveness

Derezińska developed the notion of *effectiveness* by observing how good test cases are at killing the mutants generated by a given operator [31]. In a sense, this is dual to mutant quality: the more effective the test-suite is, the worse the mutants are. She computes this effectiveness as the ratio of the number of test runs killing mutants generated by a given operator to the total number of test runs done on the killable mutants produced by that operator. This can, of course, be generalized to an arbitrary set of mutants [27].

Definition 19 (Derezińska's effectiveness)

$$\mathcal{E} = \frac{\sum_{m \in M} |K_m|}{(|M| - |E|) \cdot |T|} \tag{20}$$

Thanks to this formalization, the authors have derived a simple and handy relationship between Derezińska's effectiveness and mutation score. Manipulating the previous formula yields:

$$\mathcal{E} = \frac{|D|}{|M| - |E|} \cdot \frac{\sum_{m \in M} |K_m|}{|D| \cdot |T|} = \mathcal{S} \cdot \frac{\sum_{m \in M} |K_m|}{|D| \cdot |T|}$$

However, as $M = D \cup P$, it is clear that:

$$\sum_{m \in M} |K_m| = \sum_{m \in D} |K_m| + \sum_{m \in P} |K_m| - \sum_{m \in D \cap P} |K_m|$$

DOI: 10.1002/stvr

But, $\sum_{m \in P} |K_m| = 0$ and $D \cap P = \varnothing$. Thus:

$$\sum_{m \in M} |K_m| = \sum_{m \in D} |K_m|$$

Let us define the average number of test cases killing *dead* mutants:

$$\overline{K}_D = \frac{\sum_{m \in D} |K_m|}{|D|} \tag{21}$$

Therefore, finally:

$$\mathcal{E} = \mathcal{S} \cdot \frac{\overline{K}_D}{|T|} \tag{22}$$

This is a general formulation, but when T is adequate, as it is the case, $\mathcal{S} = 1$ and the above equation simplifies accordingly.

Nevertheless, the relation between both metrics can be established by comparing the definitions of quality of a mutation operator (Definition 17) and the restriction of the above definition of effectiveness to that operator. Thus, once the set of mutants and test cases corresponding to a mutation operator O have been fixed, the effectiveness restricted to operator O is:

$$\mathcal{E}_O = \frac{\sum_{m \in M_O} |K_m|}{(|M_O| - |E_O|) \cdot |T_O|}$$

As T_O is adequate, $M_O = D_O \cup E_O$. From Definition 17:

$$Q_O = \frac{1}{|M_O|} \sum_{m \in D_O \cup E_O} Q_m = \frac{1}{|M_O|} \left(\sum_{m \in D_O} Q_m + \sum_{m \in E_O} Q_m \right)$$

According to Definition 15, the quality of an equivalent mutant is 0. Thus, the last sum vanishes:

$$Q_O = \frac{1}{|M_O|} \sum_{m \in D_O} \left(1 - \frac{1}{(|M_O| - |E_O|) \cdot |T_O|} \sum_{t \in K_m} |C_t| \right) \tag{23}$$

Finally, pushing the first sum inside and completing for $\mathcal{E}_O$:

$$\begin{aligned} Q_O &= \frac{1}{|M_O|} \left(|D_O| - \frac{\mathcal{E}_O}{\sum_{m \in M_O} |K_m|} \sum_{m \in D_O} \sum_{t \in K_m} |C_t| \right) \\ &= \frac{1}{|M_O|} \left(|D_O| - \frac{\mathcal{E}_O}{\sum_{m \in M_O} |K_m|} \cdot \mathcal{I} \right) \end{aligned} \tag{24}$$

In conclusion, it can be observed that the impact of effectiveness in the quality of the set of mutants produced by a particular operator is smoothed by the ratio of I to the number of times that a mutant is killed by a test case.

4.3. Quality versus Mutation Operator Strength

Hu et al. [29] define the *mutation operator strength*. They consider that a mutation operator is *strong* if the mutants generated by the operator are killed by few test cases. On the other hand, the operator will be *weak* if the mutants generated are killed by many test cases. Mutation operator strength is defined by the following equation:

$$MOS = \frac{\#minTests}{\#muts - \#notKilled}$$

Following the above notation, mutation operator strength can be defined in an alternative way, more amenable to formal reasoning.

DOI: 10.1002/stvr

Definition 20 (Mutation operator strength)

$$MOS = \frac{|T_O|}{|M_O| - |P_O|} = \frac{|T_O|}{|D_O|} \tag{25}$$

The relation between quality of a mutation operator and mutation operator strength can be established by proceeding in an analogous way to effectiveness. Beginning with equation 23:

$$Q_O = \frac{1}{|M_O|} \sum_{m \in D_O} \left(1 - \frac{1}{(|M_O| - |E_O|) \cdot |T_O|} \sum_{t \in K_m} |C_t| \right)$$

it can be observed that $|M_O| - |E_O| = |D_O|$, because T_O is adequate:

$$Q_O = \frac{1}{|M_O|} \sum_{m \in D_O} \left(1 - \frac{1}{|D_O| \cdot |T_O|} \sum_{t \in K_m} |C_t| \right)$$

Finally, pushing the first sum inside and completing for MOS:

$$Q_O = \frac{1}{|M_O|} \left(|D_O| - \frac{MOS}{|T_O|^2} \sum_{m \in D_O} \sum_{t \in K_m} |C_t| \right) = \frac{1}{|M_O|} \left(|D_O| - \frac{MOS}{|T_O|^2} \cdot \mathcal{I} \right) \tag{26}$$

In conclusion, it can be observed that the impact of mutation operator strength in the quality of the set of mutants produced by a particular operator is smoothed by the ratio of I to the square of the number of test cases.

4.4. Some Remarks

Equations 24 and 26 show how $\mathcal{E}_O$ and MOS are fine-tuned by $\mathcal{I}$ when obtaining Q_O from them. In 24, $\mathcal{E}_O$ is divided by $\sum_{m \in M_O} |K_m|$ instead of the cruder denominator $|T_O|^2$ affecting MOS in 26. Consequently, MOS is expected to be less sensitive to interactions between mutants and test cases and thus, less able to accurately distinguish interesting scenarios.

Please, notice that adequate and non-redundant test-suites are not present in the work by Derezińska [31]. Moreover, not every non-equivalent mutant is killed in the work by Hu et al. [29], though the importance of adequate test-suites is acknowledged. Thus, the adequacy of the test-suites used is not guaranteed in practice. However, this is an important issue, at least when assessing the quality of mutation operators, as metrics are impacted by every alive mutant that should be instead dead. Although this could be justified in the general case, because detecting whether a mutant is really equivalent is an undecidable question, utmost care has been taken to kill every non-equivalent mutant in this work, thus providing a solid and fair framework for comparisons.

For similar reasons, minimal test-suites are used in the experiments to better isolate the dependency of the metric on the size of T_O. This is very important for any mutation metric, as different non-redundant test-suites corresponding to an initial redundant test-suite can be of rather different sizes and the particular choice is likely to affect the measure. This would be clearly undesirable. This feature is not present in the aforementioned works.

5. MUTATION TESTING WITH THE WS-BPEL LANGUAGE

The preceding theory of mutation and the formalization of mutation metrics presented have been made independent of the programming language purposely. However, in order to show the utility of the proposed metrics a concrete language is needed and experiments must be conducted to validate our intuitions on an empirical ground. The choice of WS-BPEL is motivated by the difficulty of testing software as services and, in particular, service compositions, which are developed in a distributed environment and are subject to some unconventional restrictions.

DOI: 10.1002/stvr

```
<flow> ↩ Structured activity
  <links> ↩ Container
    <link name="checkFlight-To-BookFlight" ↩ Attribute /> ↩ Element
  </links>
  <invoke name="checkFlight" ... > ↩ Basic activity
    <sources> ↩ Container
      <source linkName="checkFlight-To-BookFlight" ↩ Attribute /> ↩ Element
    </sources>
  </invoke>
  <invoke name="checkHotel" ... />
  <invoke name="checkRentCar" ... />
  <invoke name="bookFlight" ... >
    <targets> ↩ Container
      <target linkName="checkFlight-To-BookFlight"/> ↩ Element
    </targets>
  </invoke>
</flow>
```

Figure 1. WS-BPEL 2.0 activity sample

5.1. A Brief Introduction to WS-BPEL

WS-BPEL 2.0 is an OASIS standard [25] and an industrial-strength programming language for WS compositions. WS-BPEL is an XML-based language which allows to specify the behavior of a business process based on its interactions with other WS. The structure of a WS-BPEL process is divided into four sections: declaration of relationships with external partners (the client invoking the business process and the different WS invoked by the process), declaration of variables used by the process itself, declaration of handlers that the process can use (like fault and event handlers), and description of the business process behavior.

All the elements that have been defined above are global if they are declared within the process. Nevertheless, they can be declared in a local way through a `scope` container. Scopes allow us to split the business process into different, logically related, portions.

The major building blocks of a WS-BPEL process are *activities*. There are two types: basic and structured activities. Basic activities only perform one purpose (receiving a message from a partner, sending a message to a partner, assigning to a variable, etc.). Structured activities define the business logic and can contain other activities.

Activities may have both attributes and a set of containers associated to them. Of course, these containers can include elements with their own attributes too. This can be illustrated with the example skeleton in Figure 1.

WS-BPEL provides concurrency and synchronization between activities. An example is the `flow` activity, which launches a set of activities in parallel and allows to specify the synchronization conditions between them. The aforementioned example includes a `flow` activity that invokes three WS in parallel: `checkFlight`, `checkHotel`, and `checkRentCar`. Moreover, there is another WS, `bookFlight`, that will be invoked upon `checkFlight` completion. This synchronization between activities is achieved by establishing a `link`, so that the target activity of the link will be eventually executed only if the source activity of the link has been completed.

5.2. The MuBPEL Tool

The automated application of mutation analysis to programs written in a concrete programming language requires a tool to generate the mutants, execute them against the test cases, and deciding whether a mutant has been killed or not by comparing its behavior to the original program for each test case. With this aim, a new analysis tool, MuBPEL, has been implemented by our research group, integrating 26 mutation operators for WS-BPEL 2.0 defined by Estero et al. [32]. MuBPEL can be used with GAmera [8], an evolutionary mutation system for service compositions.

DOI: 10.1002/stvr

Table I. Mutation operators for WS-BPEL 2.0

OPERATOR		DESCRIPTION
		IDENTIFIER MUTATION
ISV		Replaces a variable identifier by another of the same type
		EXPRESSION MUTATION
EAA		Replaces an arithmetic operator (`+`, `-`, `*`, `div`, `mod`) 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 (`and`, `or`) by another of the same kind
ECC		Replaces a path operator (`/`, `//`) by another of the same kind
ECN		Modifies a numerical constant by incrementing/decrementing it by one, or adding/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
		CONCURRENT ACTIVITY MUTATION
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*
		NON-CONCURRENT ACTIVITY MUTATION
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
		EXCEPTION AND EVENT 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

Table I shows the name and a brief description of each operator proposed. Boubeta et al. [33] have compared mutation operators defined for WS-BPEL [32] and those defined for C [34], Fortran [35], Ada [7], C++ [36], C# [26], ASP.NET [37], Java [38, 39, 40, 30, 41, 42], SQL [43], and XSLT [44]. This study concludes that 50% of WS-BPEL 2.0 mutation operators are specific to this language. They appear marked with ☆.

5.3. *MuBPEL Architecture*

MuBPEL let us apply mutation analysis to standard WS-BPEL 2.0 compositions and it is the first tool with this capability. It includes the following features:

DOI: 10.1002/stvr

- Analysis of WS-BPEL 2.0 compositions, resulting in the list of mutation operators applicable and the locations where they can be used inside the composition.
- Generation of every possible mutant produced by the mutation operators defined by Estero et al. [32].
- Execution of the original composition against a given test-suite.
- Execution of mutants against the selected test-suite and comparison of the behavior of each mutant to the original composition according to the firm mutation technique, which generalizes both traditional strong mutation and weak mutation.

In order to achieve this, MuBPEL builds on ActiveBPEL [45], a WS-BPEL 2.0 standard compliant engine, and BPELUnit [46], a library for unit testing that can be used with any compliant engine. A remarkable feature of BPELUnit and MuBPEL is that external WS can be replaced by *mocks*, which behave following a predefined pattern. This way is possible to execute a composition though some WS are not available or when scenarios where the WS produce determined outputs arise. The use of mocks allows us to repeat an experiment under exactly the same conditions. XML files are used to describe the test cases. MuBPEL consists of three main components:

- Analyzer
- Mutant generator
- Execution system

Figure 2 shows the relations between these components. The analyzer receives the original composition and generates the information about which mutation operators can be applied and in which locations. This information is delivered to the mutant generator, which generates every possible mutant. Then, the execution engine runs the original composition and its mutants against the set of test cases, and compares their behaviors to determine whether the mutants have been killed or stay alive.

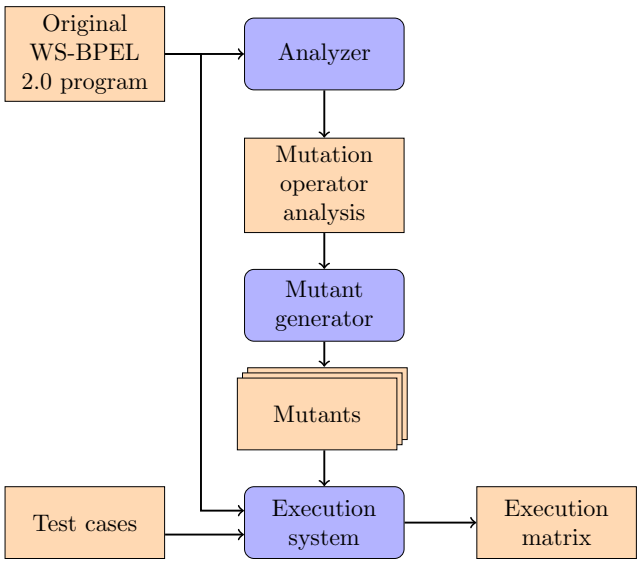
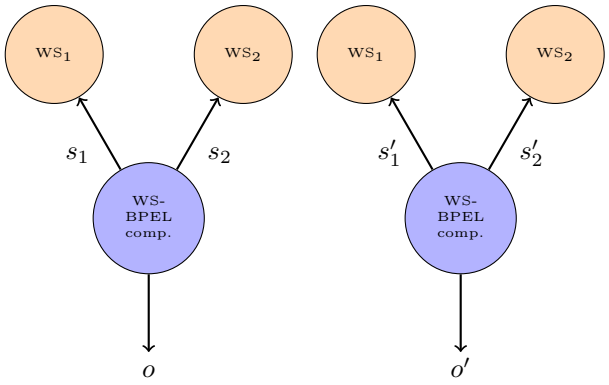


Figure 2. MuBPEL architecture

5.4. Firm Mutation with MuBPEL

One of the drawbacks of firm mutation is that it does not specify where the comparisons between the original program and its mutants should be done. This is a handicap when a tool based on firm mutation must be implemented, as non-trivial technical issues must be addressed. Jackson and Woodward [13] consider that Java and object-oriented languages in general, are suitable for

DOI: 10.1002/stvr



(a) Original program (b) Mutant

Figure 3. Message passing in a WS-BPEL composition

the application of firm mutation. The authors think that WS-BPEL is suitable too. The reason is that BPEL compositions define a business process in which several WS communicate. From this viewpoint, both the messages sent by the composition to the WS that integrate it and the output message, which represents the result of the composition execution, are interesting. Figure 3 represents a composition interacting with two WS. While s_1 and s_2 are the messages sent by the original composition to the WS that it communicates with, s'_1 and s'_2 are the corresponding messages sent by the mutated composition to the WS. Besides, o represents the output of the original composition and o' the output of its mutant.

Thus, in this example, an inspection set $IS = \{s_1, s_2, o\}$ can be formed with the message variables, where s_1 and s_2 are the state variables and o is the output variable. Figure 4 shows the different possibilities relative to the behavior of the original composition and its mutants by taking into account the messages sent to the partner WS and the final output produced.

5.5. *Execution Matrix*

The results of comparing the behavior of the original compositions and its mutants against the test cases are represented in an *execution matrix*. This is an $|M| \times |T|$ matrix as in the following example:

$$X = \begin{bmatrix} 0 & 0 & 1 & \dots & 1 \\ 1 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 2 & 2 & 2 & \dots & 2 \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{27}$$

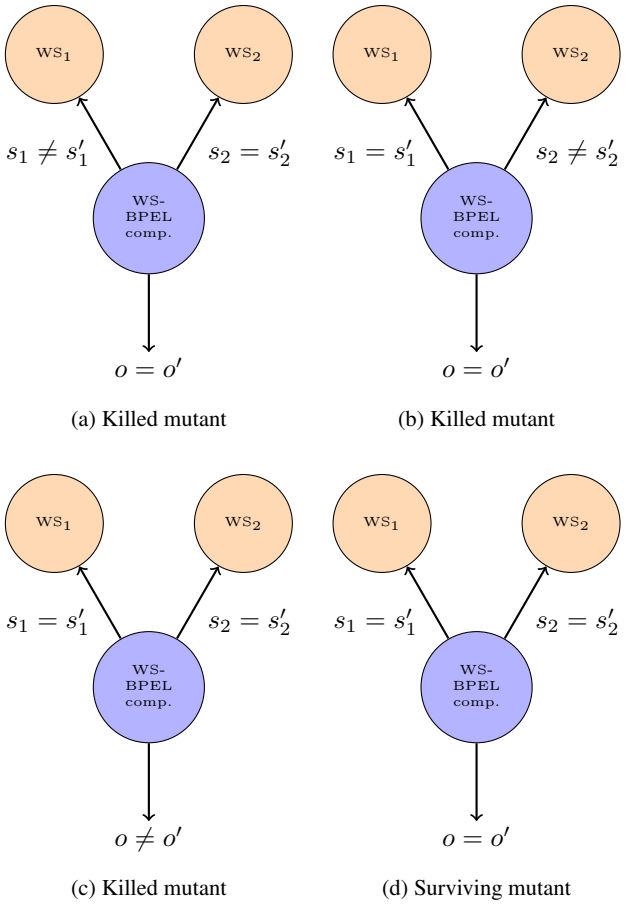
where x_{ij} is 1 or 0 for most of the mutants, depending on whether the ith mutant was killed or not by the jth test case, respectively.

Some mutants may fail to execute because they violate some static restriction imposed by the language. For example, a mutated composition may fail when being parsed or deployed. Such a mutant is *invalid* and can be recognized in the execution matrix because the corresponding row entries contain 2, instead of 0 or 1.

Therefore, all the relevant quantities to conduct a mutation analysis and compute the metrics can be determined from the execution matrix. A quick look suffices to know which mutants are dead and the test cases that killed them, which mutants stay alive and which are not valid.

MuBPEL can execute a mutant until it is killed by a test case or it can continue its execution against the full test-suite. The latter option is used to obtain a full execution matrix. The use of each option depends on the kind of study desired.

DOI: 10.1002/stvr



(a) Killed mutant (b) Killed mutant

(c) Killed mutant (d) Surviving mutant

Figure 4. Firm mutation with MuBPEL

6. APPLYING METRICS TO THE EVALUATION OF MUTATION OPERATORS

In this section the quality metrics defined previously will be applied to the evaluation of the set of mutation operators for WS-BPEL 2.0.

6.1. Objectives

The goal pursued with this evaluation is threefold:

1. To determine if all of the mutation operators are valid or if some of them can be eliminated, for example, because they only produce invalid mutants.
2. To determine if the definition or implementation of some operator should be modified, for example, because it produces many invalid or equivalent mutants.
3. To determine if some mutation operators are more interesting than others according to their quality.

Several variables have been measured in this study:

1. Number of equivalent mutants.
2. Number of invalid mutants.
3. Quality of a mutation operator.
4. Percentage of resistant mutants.
5. Percentage of weak mutants.

DOI: 10.1002/stvr

In order to evaluate the mutation operators defined for WS-BPEL in Table I the authors would like to answer the following research questions:

RQ1. *Does any operator generate invalid mutants?* If an operator generates invalid mutants the reason must be studied, and it must be decided whether to discard it or modify its definition or implementation.
RQ2. *Does any operator generate a high percentage of equivalent mutants?* In this case the cause must be determined, and if necessary, its definition must be modified.
RQ3. *How many weak and resistant mutants does each operator produce?* A weak mutant is killed by many test cases, while a resistant mutant is killed only by one test case. It is interesting to compute which proportions of the dead mutants they represent.
RQ4. *Which is the quality of each mutation operator?* If there is much variation in the qualities of mutation operators a threshold value can be established to discard mutation operators exhibiting a low quality.

6.2. Experimental Method

Four WS-BPEL 2.0 compositions have been used: LoanApproval (LA), TravelReservationService (TRS), MetaSearch (MS), and COMBO. The LA composition has been taken from the WS-BPEL 2.0 Standard [25], though it has been modified to exercise more mutation operators. TRS is a modified and extended version of the composition with the same name in NetBeans 6.1 [47] documentation. The MS composition is a modified version of the MetaSearch composition that is distributed with BPELUnit [46]. The last composition, COMBO, is an artificial composition comprising four smaller compositions exercising some exotic BPEL operators.

In order to evaluate the quality of mutation operators, the following procedure has been used for each composition:

1. Generate all the mutants for the composition.
2. Execute the composition and its mutants against an initial hand-made test-suite. This execution allows us to know which mutants are invalid and to determine, by inspection of the surviving mutants, which of them are equivalent and which are not.
3. Complete, if necessary, the initial test-suite with additional test cases killing some non-equivalent mutant. This step and the previous one are iterated until an adequate test-suite is found.
4. Reduce the resulting test-suite to produce a non-redundant test-suite of minimum size, that is, a minimal test-suite.
5. Obtain minimal test-suites for each operator.
6. Compute the quality of each mutation operator and the percentages of dead mutants which are weak and resistant.

The generation and execution of mutants have been carried out with MuBPEL. The minimization of test-suites is a well-known hard problem that has been solved with the help of specialized algorithms that the authors have implemented and tailored to their specific needs. They will not be discussed here as it is out of the scope of this paper and it is a topic deserving a deep digression on its own.

6.3. Results

Table II shows the operators that can be applied to each composition, the number of mutants that each operator produces for every composition, and the total number of mutants generated by each operator. It can be noticed that there are some operators that produce very few mutants and can only be applied to some compositions. These operators are rather specific to BPEL.

Table III shows the main features for each composition used: the total number of source code lines and mutants generated, and the sizes of the original and the minimal test-suite. As it can be seen in the table the size of the original test-suite can be much larger than the minimal test-suite size.

DOI: 10.1002/stvr

Table II. Mutants generated by compositions

	LA	COMBO	TRS	MS	Total
ISV	0	7	0	116	123
EAA	0	16	0	44	60
EEU	0	0	0	2	2
ERR	10	15	30	65	120
ELL	0	3	0	2	5
ECC	13	4	24	41	82
ECN	4	12	24	100	140
EMD	0	2	6	0	8
EMF	0	0	2	0	2
ACI	0	2	0	0	2
AFP	0	0	0	2	2
ASF	4	9	14	10	37
AIS	0	1	0	0	1
AEL	17	44	62	73	196
AIE	2	3	0	11	16
AWR	0	1	0	0	1
AJC	0	1	0	0	1
ASI	12	25	34	39	110
APM	0	4	0	0	4
APA	0	1	3	0	4
XMF	0	2	5	4	11
XMC	0	2	0	0	2
XMT	0	0	1	0	1
XTF	0	0	6	0	6
XER	0	0	4	0	4
XEE	0	0	2	0	2
Total	62	154	217	509	942

Table III. Main features of tested compositions

	LA	COMBO	TRS	MS
LOC	110	537	384	633
Mutants	62	154	217	509
$\|T\|$ (original)	8	26	19	37
$\|T\|$ (reduced)	3	10	12	8

BPEL compositions tend to be smaller than traditional programs in the sense that they define the logic of the composition of the external WS or partners, while the bulk of the code is in the WS themselves. Therefore, the number of mutants obtained is comparatively much lower than in traditional programming languages. On the other hand, this does not implies a reduction of cost at all. WS compositions can be very heavy applications. They are typically executed by BPEL engines on top of application servers. Even the deployment and undeployment time devoted to every mutant can be noticeable. Communication between different partners in the composition is achieved by message passing. Timeouts can even be defined for inbound messages.

Tables IV, V, VI and VII include for each composition and operator the total number of mutants generated, the number of invalid and equivalent mutants, the quality of the mutation operator (Q_O), the quality of the dead mutants generated by the operator (Q_{D_O}), the proportion of resistant mutants in the set of dead mutants (ρ), and the same for weak mutants (ω).

DOI: 10.1002/stvr

Table IV. LoanApproval Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ERR	10	0	0	0.60	0.60	80.0	20.0
ECC	13	0	7	0.39	0.84	83.3	0.0
ECN	4	0	0	0.75	0.75	100.0	0.0
ASF	4	1	0	–	–	100.0	0.0
AEL	17	1	0	0.74	0.74	62.5	12.5
AIE	2	0	0	–	–	100.0	0.0
ASI	12	0	0	0.59	0.59	75.0	25.0

Table V. COMBO Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ISV	7	0	0	0.00	0.00	–	–
EAA	16	0	4	0.00	0.00	–	–
ERR	15	0	2	0.82	0.95	85.0	0.0
ELL	3	0	0	–	–	100.0	0.0
ECC	4	0	2	0.00	–	–	–
ECN	12	0	2	0.55	0.66	100.0	0.0
EMD	2	0	0	–	–	–	–
ACI	2	2	0	–	–	–	–
ASF	9	3	5	0.00	–	–	–
AIS	1	0	0	–	–	–	–
AEL	44	6	2	0.92	0.96	81.0	0.0
AIE	3	0	0	–	–	100.0	0.0
AWR	1	0	0	–	–	–	–
AJC	1	0	0	–	–	–	–
ASI	25	3	5	0.71	0.92	100.0	0.0
APM	4	0	0	0.00	0.00	–	–
APA	1	0	0	–	–	–	–
XMF	2	0	1	–	–	–	–
XMC	2	0	0	–	–	–	–

Henceforth, some table entries have been marked as – (not significant). In particular, a quality value is regarded as "not significant" when there are less than four mutants involved in its computation. In fact, when an operator generates just one valid mutant, its quality would be zero but what really happens is that the value obtained for the quality metric has no significance as it has only been obtained from a single data. – has been also used for ρ and ω when the number of test cases in the minimal test-suite is 1.

Table VIII shows a summary of the results obtained for all the compositions analyzed. The values shown for Q_O and Q_{D_O} are the arithmetic means of those obtained for every composition. When computing each mean, values marked as – have been discarded.

One of the objectives of this study is to determine if some operator should be discarded or if it is necessary to modify their definition, or implementation, because it produces many invalid or equivalent mutants. In this sense, only four operators produce invalid mutants: ACI, ASF, AEL and ASI. Two of them (AEL and ASI) in a low percentage and under circumstances that are fully justified by the kind of transformations that they perform to the code. The operator ASF produces a 16.2% of invalid mutants and the authors conclude that it is necessary to study the reason to try to reduce this percentage. The operator ACI generates 100.0% of invalid mutants, but it is only used by one of the compositions, which generates just two mutants.

DOI: 10.1002/stvr

Table VI. TravelReservationService Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ERR	30	0	6	0.67	0.84	33.3	8.3
ECC	24	0	23	0.00	–	–	–
ECN	24	0	0	0.94	0.94	75.0	0.0
EMD	6	0	1	0.73	0.88	100.0	0.0
EMF	2	0	1	–	–	–	–
ASF	14	1	13	0.00	–	–	–
AEL	62	2	8	0.82	0.95	61.5	0.0
ASI	34	0	8	0.69	0.90	50.0	3.8
APA	3	0	1	–	–	100.0	0.0
XMF	5	0	0	0.84	0.84	80.0	20.0
XMT	1	0	0	–	–	–	–
XTF	6	0	0	0.89	0.89	100.0	0.0
XER	4	0	0	0.94	0.94	100.0	0.0
XEE	2	0	0	–	–	100.0	0.0

Table VII. MetaSearch Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ISV	116	0	8	0.59	0.64	18.5	22.2
EAA	44	0	0	0.55	0.55	38.6	38.6
EEU	2	0	2	–	–	–	–
ERR	65	0	9	0.67	0.78	48.2	0.0
ELL	2	0	0	–	–	–	–
ECC	41	0	40	0.00	–	–	–
ECN	100	0	15	0.62	0.73	25.9	0.0
AFP	2	0	1	–	–	–	–
ASF	10	1	4	0.00	0.00	–	–
AEL	73	0	0	0.71	0.71	39.7	9.6
AIE	11	0	2	0.64	0.78	77.8	0.0
ASI	39	0	3	0.54	0.58	36.1	5.6
XMF	4	0	0	0.81	0.81	75.0	0.0

ACI modifies the `createInstance` attribute of an inbound message activity from *yes* to *no* whenever there is more than one such activity in the composition. That is, it transforms an initial activity that serves as an entry point into one that does not. This transformation can only be valid if previously there is another activity that can also serve as an entry point. After finding the reason why the invalid mutants are generated, it is necessary to modify the implementation of the operator to avoid generating them. On the other hand, evaluating the quality of this operator requires a composition that contains two or more entry points in different locations.

It is necessary to take into account that some operators obtain a zero value for Q_O and Q_{D_O} because they produce a small number of mutants, and all of them can be killed by the same test case, or it mainly produces equivalent mutants, e.g., ASF. It would be desirable to have available specific compositions generating a higher number of mutants for these operators.

The need to identify equivalent mutants is one of the main drawbacks of mutation analysis. For this reason, it is useful to classify operators according to the percentage of equivalent mutants that they generate. Table IX shows this classification by ranking the operators by increasing percentages of equivalent mutants that they generate.

Judging from the percentages of equivalent mutants, the best operators are at the top of Table IX. These operators do not generate equivalent mutants (ELL, AIS, AWR, AJC, APM, XMC, XMT,

DOI: 10.1002/stvr

Table VIII. Summary

	% Mutants	% Invalid	% Equiv.	Q_O	Q_{D_O}	ρ (%)	ω (%)
ISV	13.1	0.0	6.5	0.30	0.32	17.4	20.9
EAA	6.4	0.0	6.7	0.28	0.28	30.4	30.4
EEU	0.2	0.0	100.0	–	–	–	–
ERR	12.7	0.0	14.2	0.69	0.79	52.4	3.9
ELL	0.5	0.0	0.0	–	–	60.0	0.0
ECC	8.7	0.0	87.8	0.10	0.84	50.0	0.0
ECN	14.9	0.0	12.1	0.72	0.77	43.9	0.0
EMD	0.8	0.0	12.5	0.73	0.88	71.4	0.0
EMF	0.2	0.0	50.0	–	–	–	–
ACI	0.2	100.0	0.0	–	–	–	–
AFP	0.2	0.0	50.0	–	–	–	–
ASF	3.9	16.2	59.5	0.00	0.00	33.3	0.0
AIS	0.1	0.0	0.0	–	–	–	–
AEL	20.8	4.6	5.1	0.80	0.84	56.5	5.1
AIE	1.7	0.0	12.5	0.64	0.78	85.7	0.0
AWR	0.1	0.0	0.0	–	–	–	–
AJC	0.1	0.0	0.0	–	–	–	–
ASI	11.7	2.7	14.5	0.63	0.75	57.1	6.6
APM	0.4	0.0	0.0	0.00	0.00	–	–
APA	0.4	0.0	25.0	–	–	66.7	0.0
XMF	1.2	0.0	9.1	0.83	0.83	70.0	10.0
XMC	0.2	0.0	0.0	–	–	–	–
XMT	0.1	0.0	0.0	–	–	–	–
XTF	0.6	0.0	0.0	0.89	0.89	100.0	0.0
XER	0.4	0.0	0.0	0.94	0.94	100.0	0.0
XEE	0.2	0.0	0.0	–	–	100.0	0.0

XTF, XER and XEE). Q_O, Q_{D_O}, ρ and ω could not be computed for the ELL, AIS, AWR, AJC, APM, XMC, XMT, and XEE operators, as they generate very few mutants. The remaining operators in this group (XTF and XER) obtain high quality values and produce 100% of resistant mutants. However, as they produce few mutants, their excellent results must be interpreted cautiously.

Conversely, the worst operators would be those appearing at the bottom of Table IX. EEU produces 100% of equivalent mutants, but it is only used by the MS composition and thus it requires further study. ECC produces 87.8% of equivalent mutants: this can be explained by the fact that it is a path mutation operator. For the simple paths that are used in the compositions under study, it is unlikely that its mutations have any effect over the result. ECC requires further analysis with compositions exhibiting more complex paths.

ASF, AFP and EMF produce about 50% of equivalent mutants. ASF is an operator that replaces a `sequence` activity with a `flow` activity. Unless there is a data dependency between the affected activities, the generated mutants will be equivalent. AFP replaces a sequential `foreach` activity with a parallel one, and thus suffers from the same problem as ASF. Lastly, an analysis of EMF reveals that these issues would be solved by a careful modification of its definition.

Operators AEL, ISV, EAA, XMF, ECN, EMD, AIE, ERR, and ASI produce less than 15% of equivalent mutants. Among them, AEL, XMF, ECN, EMD, AIE, ERR, and ASI do not yield values of Q_{D_O} below 0.75. Moreover, this latter group produce resistant mutants and only AEL, XMF, ERR, and ASI produce weak mutants (less than 10%). Therefore, they are regarded as useful operators. ISV and EAA get values of Q_{D_O} below 0.5 and produce both resistant and weak mutants in a low percentage. Additional data is needed to establish the usefulness of these operators.

DOI: 10.1002/stvr

Table IX. Operators classified by the percentage of equivalent mutants

% Equiv. mutants	Operators	Q_O	Q_{D_O}	ρ (%)	ω (%)
0	ELL	–	–	60.0	0.0
	AIS	–	–	–	–
	AWR	–	–	–	–
	AJC	–	–	–	–
	APM	0.00	0.00	–	–
	XMC	–	–	–	–
	XMT	–	–	–	–
	XTF	0.89	0.89	100.0	0.0
	XER	0.94	0.94	100.0	0.0
	XEE	–	–	100.0	0.0
≤ 15	AEL	0.80	0.84	56.5	5.1
	ISV	0.30	0.32	17.4	20.9
	EAA	0.28	0.28	30.4	30.4
	XMF	0.83	0.83	70.0	10.0
	ECN	0.72	0.77	43.9	0.0
	EMD	0.73	0.88	71.4	0.0
	AIE	0.64	0.78	85.7	0.0
	ERR	0.69	0.79	52.4	3.9
	ASI	0.63	0.75	57.1	6.6
> 15	APA	–	–	66.7	0.0
	EMF	–	–	–	–
	AFP	–	–	–	–
	ASF	0.00	0.00	33.3	0.0
	ECC	0.10	0.84	50.0	0.0
	EEU	–	–	–	–

7. MUTATION OPERATOR OPTIMIZATION

Results obtained on the evaluation of the WS-BPEL 2.0 mutation operators in the previous section indicate that while some operators can already be considered useful, two groups of operators deserve further research. On one hand, some operators require improvements in their definition or implementation to reduce the number of invalid and equivalent mutants that they generate. On the other hand, some operators need to be evaluated with new compositions, as the available compositions have not provided enough insight upon them. This section describes the improvements applied to both groups of operators and their results.

7.1. Operators Requiring Changes

EMF This operator replaces a deadline expression with the current date or with the average between the current date and the original date in the process definition. Including deadline expressions in the process definitions presents several issues for testing. If the expression describes a future date, the composition will have to wait until that date to check the behavior of the composition. To avoid these potentially very long waits, a 20 second time limit is enforced through BPELUnit on the execution of any BPEL process instance. The first evaluation of EMF shows that it produces 50% of equivalent mutants. A detailed study of the two mutants produced by EMF in each of the involved locations reveals that the mutant where the original date is replaced with the average is always equivalent. This is because of the time limit imposed on the execution of a BPEL process instance. Consequently, the definition and implementation of EMF has been modified so

Prepared using stvrauth.cls
DOI: 10.1002/stvr

that it only produces a single mutant for every location: the mutant that replaces the original date with the current date.

ASF This operator replaces a `sequence` activity with the `flow` activity. The initial evaluation of ASF shows that it produces 16.2% of invalid mutants and 59.5% of equivalent mutants. A close examination of the mutants generated by ASF reveals that the invalid mutants are generated when replacing a `sequence` at the beginning of the composition that has an inbound message activity (`receive` or `pick`) with the `createInstance` set to "yes", serving as an entry point into the composition. When this `sequence` is replaced with a `flow`, it may allow an activity that is not an entry point to run before one that is. This is explicitly disallowed by WS-BPEL and statically checked at deployment time. Accordingly, the implementation of ASF has been modified so that it does not produce these invalid mutants. On the other hand, ASF also produces a high percentage of equivalent mutants. These equivalent mutants originate from the fact that there are no data dependencies among the activities in the `sequence`, and replacing them with `flow` has no effect on the output of the process instance. This problem cannot be handled just by modifying the operator.

ACI This operator changes the attribute `createInstance` of an entry point from "yes" to "no". Among the four compositions used for evaluating the operators, ACI could only be applied to one of them, COMBO, producing two invalid mutants. Both mutants show that the restriction imposed on ACI requiring it to be applied only when more than one entry point existed was not enough. In this optimization stage, another restriction to ACI has been added so that it does not produce the above invalid mutants. Therefore, its new implementation considers the following cases:

```
<process>
  <sequence>
    <flow>
      <receive createInstance="yes"/> ↩ discarded
      <receive createInstance="yes"/> ↩ discarded
    </flow>
  </sequence>
</process>
```

```
<process>
  <sequence>
    <flow>
      <receive createInstance="yes"/> ↩ discarded
      <sequence>
        <receive createInstance="yes"/> ↩ discarded
        ...
      </sequence>
    </flow>
  </sequence>
</process>
```

```
<process>
  <sequence>
    <receive createInstance="yes"/> ↩ discarded
    ...
    <flow>
      <receive createInstance="yes"/> ↩ produces the first mutant
      <receive createInstance="yes"/> ↩ produces the second mutant
    </flow>
```


Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table X. Main features of tested compositions

	LA	COMBO2	TRS	MS	LAE
LOC	110	1281	384	633	1533
Mutants	61	909	221	508	3647
$\|T\|$ (original)	8	34	19	37	95
$\|T\|$ (reduced)	3	17	12	8	64

```
    ...
  </sequence>
</process>
```

Now, ACI does not produce any mutants in the first and second fragments, as they would be invalid according to the static checks imposed by WS-BPEL. However, when applied to the third fragment it produces two mutants, since the `flow` that contains the two last `receive` activities is not the first child of its `sequence` container.

7.2. Operators Requiring Additional Compositions

ACI This operator could not produce valid mutants from the compositions in the previous study. In order for the revised version of ACI to produce valid mutants, compositions should meet the following requirements:

- The composition should define an entry point.
- Later in the process definition, it should define n intermediate entry points, with control dependencies on the previous entry point.
- Every entry point should belong to the same `correlationSet`.

After implementing a composition that meets these requirements, the ActiveBPEL engine was verified to be even more restrictive than the WS-BPEL 2.0 standard in this particular aspect. ActiveBPEL generates deployment errors whenever there are entry points in the main `sequence` of the composition and any other entry point is added. For this reason, the ACI operator was discarded.

EEU, ELL, EMF, AFP, AIS, AWR, AJC, APM, APA, XMC, XMT, and XEE These operators could not be evaluated in the previous study, as they produced a very low number of mutants. Two new compositions, COMBO2 and LAE, have been introduced to try to obtain sufficient data.

ISV and EAA These operators have obtained values of Q_{D_O} below 0.5 in the previous study and they have produced both resistant and weak mutants. Therefore, additional data from other compositions is required to extract conclusions about the usefulness of these operators. COMBO2 and LAE will prove valuable for all of them.

7.3. Results after Optimizing

In order to evaluate the modified operators and those needing further study with additional compositions, the experiments in Section 6 have been repeated, replacing COMBO with COMBO2, which is an extended version of COMBO, and adding a new composition, LAE. LAE is an extended version of LA which takes into account more personal details from various sources (current assets, debts, job history and so on) and may demand endorsement from a third party if the applicant does not meet the requirements set by the bank.

Table X presents the characteristics of the compositions used, including their total number of source code lines and mutants generated, and the sizes of their original and minimal test-suites. Table XI shows the operators used by the compositions in this study.

DOI: 10.1002/stvr

Table XI. Mutants generated by compositions after operator optimization

	LA	COMBO2	TRS	MS	LAE	Total
ISV	0	201	0	116	1684	2001
EAA	0	116	0	44	204	364
EEU	0	2	0	2	6	10
ERR	10	75	30	65	475	655
ELL	0	5	0	2	20	27
ECC	13	142	24	41	103	323
ECN	4	68	24	100	548	744
EMD	0	4	6	0	0	10
EMF	0	1	1	0	0	2
AFP	0	1	0	2	2	5
ASF	3	20	13	9	26	71
AIS	0	4	0	0	2	6
AEL	17	145	62	73	291	588
AIE	2	6	6	11	55	80
AWR	0	3	0	0	4	7
AJC	0	2	0	0	1	3
ASI	12	86	34	39	159	330
APM	0	6	0	0	0	6
APA	0	3	3	0	0	6
XMF	0	7	5	4	24	40
XMC	0	3	0	0	1	4
XMT	0	2	1	0	0	3
XTF	0	4	6	0	32	42
XER	0	1	4	0	10	15
XEE	0	2	2	0	0	4
Total	61	909	221	508	3647	5346

Table XII. LoanApproval Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ERR	10	0	0	0.60	0.60	80.0	20.0
ECC	13	0	7	0.39	0.84	83.3	0.0
ECN	4	0	0	0.75	0.75	100.0	0.0
ASF	3	0	0	–	–	100.0	0.0
AEL	17	1	0	0.74	0.74	62.5	12.5
AIE	2	0	0	–	–	100.0	0.0
ASI	12	0	0	0.59	0.59	75.0	25.0

Tables XII, XIII, XIV, XV, and XVI show, for every composition and applicable operator, the total number of invalid and equivalent mutants, the quality of the operator considering all the generated mutants (Q_O), the quality of the operator considering only killed mutants (Q_{D_O}) and the percentages of resistant (ρ) and weak (ω) mutants within the set of all killed mutants.

Table XVII shows a summary of the results obtained for every composition. Quality is shown to have been improved in several cases.

After the changes introduced in the definition and implementation of some operators and the addition of new compositions to the study, the following results have been achieved:

ACI This operator has been discarded after checking that it is not possible to write a composition for which it can produce valid mutants.

DOI: 10.1002/stvr

Table XIII. TravelReservationService Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ERR	30	0	6	0.67	0.84	33.3	8.3
ECC	24	0	23	0.00	–	–	–
ECN	24	0	0	0.94	0.94	75.0	0.0
EMD	6	0	1	0.73	0.88	100.0	0.0
EMF	1	0	0	–	–	–	–
ASF	13	0	13	0.00	–	–	–
AEL	62	2	8	0.82	0.94	75.0	0.0
ASI	34	0	8	0.69	0.90	50.0	3.8
APA	3	0	1	–	–	100.0	0.0
XMF	5	0	0	0.84	0.84	80.0	20.0
XMT	1	0	0	–	–	–	–
XTF	6	0	0	0.89	0.89	100.0	0.0
XER	4	0	0	0.94	0.94	100.0	0.0
XEE	2	0	0	–	–	100.0	0.0

Table XIV. MetaSearch Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ISV	116	0	8	0.59	0.64	18.5	22.2
EAA	44	0	0	0.55	0.55	38.6	38.6
EEU	2	0	2	–	–	–	–
ERR	65	0	9	0.67	0.78	48.2	0.0
ELL	2	0	0	–	–	–	–
ECC	41	0	40	0.00	–	–	–
ECN	100	0	15	0.62	0.73	27.1	0.0
AFP	2	0	1	–	–	–	–
ASF	9	0	4	0.00	0.00	–	–
AEL	73	0	0	0.71	0.71	39.7	9.6
AIE	11	0	2	0.64	0.78	77.8	0.0
ASI	39	0	3	0.54	0.58	36.1	5.6
XMF	4	0	0	0.81	0.81	75.0	0.0

EMF Equivalent mutants have been reduced from 50% to 0% thanks to improvements in its definition and implementation, but it produces just a few mutants and it has not been possible to measure its quality.

ASF Invalid mutants represented 16.2% of the mutants produced by this operator. After refining its implementation, this figure has been reduced to 0%.

EEU, ELL, AIS, AWR, and APM These operators have been evaluated thanks to data supplied by the new compositions.

ISV, EAA, and XMF Increasing the number of generated mutants with the new compositions has improved the quality metrics produced by ISV, EAA and XMF.

EMD The new compositions determine that the quality of the EMD operator is not as high as estimated, as many equivalent mutants are now introduced.

AFP, AJC, APA, XMC, XMT, and XEE It still has not been possible to evaluate these exotic operators as their mutants are generated in rather small quantities.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table XV. COMBO2 Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ISV	201	0	32	0.65	0.78	34.3	0.0
EAA	116	0	5	0.74	0.77	56.8	0.0
EEU	2	0	1	–	–	–	–
ERR	75	0	17	0.71	0.92	67.2	0.0
ELL	5	0	1	0.70	0.88	100.0	0.0
ECC	142	0	100	0.23	0.77	38.1	0.0
ECN	68	0	23	0.59	0.90	100.0	0.0
EMD	4	0	2	0.00	–	–	–
EMF	1	0	0	–	–	–	–
AFP	1	0	0	–	–	–	–
ASF	20	0	15	0.23	0.90	80.0	0.0
AIS	4	0	0	0.75	0.75	100.0	0.0
AEL	145	10	21	0.81	0.96	59.6	0.0
AIE	6	0	3	0.44	–	100.0	0.0
AWR	3	0	0	–	–	100.0	0.0
AJC	2	0	0	–	–	100.0	0.0
ASI	86	8	16	0.74	0.93	62.9	0.0
APM	6	0	0	0.72	0.72	100.0	0.0
APA	3	1	0	–	–	100.0	0.0
XMF	7	0	2	0.66	0.93	100.0	0.0
XMC	3	0	0	–	–	100.0	0.0
XMT	2	0	1	–	–	–	–
XTF	4	0	3	0.00	–	–	–
XER	1	0	0	–	–	–	–
XEE	2	0	2	–	–	–	–

Table XVI. LAE Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	ρ (%)	ω (%)
ISV	1684	0	325	0.52	0.64	16.6	14.3
EAA	204	0	37	0.59	0.72	23.4	0.0
EEU	6	0	0	0.89	0.89	83.3	16.7
ERR	475	0	90	0.76	0.94	26.2	0.8
ELL	20	0	2	0.67	0.74	44.4	0.0
ECC	103	0	101	0.00	–	–	–
ECN	548	0	118	0.70	0.90	32.6	0.0
AFP	2	0	1	–	–	–	–
ASF	26	0	18	0.21	0.68	50.0	37.5
AIS	2	0	0	–	–	–	–
AEL	291	0	17	0.85	0.90	43.4	1.1
AIE	55	0	6	0.86	0.97	65.3	0.0
AWR	4	0	0	0.44	0.44	50.0	50.0
AJC	1	0	1	–	–	–	–
ASI	159	0	40	0.48	0.64	22.7	2.5
XMF	24	0	0	0.99	0.99	95.8	0.0
XMC	1	0	0	–	–	–	–
XTF	32	0	0	0.99	0.99	100.0	0.0
XER	10	0	0	0.99	0.99	100.0	0.0

DOI: 10.1002/stvr

Table XVII. Summary

	% Mutants	% Invalid	% Equiv.	Q_O	Q_{D_O}	ρ (%)	ω (%)
ISV	37.4	0.0	18.2	0.59	0.69	18.6	13.4
EAA	6.8	0.0	11.5	0.63	0.68	37.0	5.3
EEU	0.2	0.0	30.0	0.89	0.89	83.3	16.7
ERR	12.3	0.0	18.6	0.68	0.82	34.3	1.3
ELL	0.5	0.0	11.1	0.69	0.81	50.0	0.0
ECC	6.0	0.0	83.9	0.12	0.81	42.0	0.0
ECN	13.9	0.0	21.0	0.72	0.84	39.1	0.0
EMD	0.2	0.0	30.0	0.37	0.88	71.4	0.0
EMF	< 0.1	0.0	0.0	–	–	–	–
AFP	0.1	0.0	40.0	–	–	–	–
ASF	1.3	0.0	70.4	0.11	0.53	52.4	14.3
AIS	0.1	0.0	0.0	0.75	0.75	100.0	0.0
AEL	11.0	2.2	7.8	0.79	0.85	57.3	3.5
AIE	1.5	0.0	13.8	0.65	0.88	69.8	0.0
AWR	0.1	0.0	0.0	0.44	0.44	71.4	28.6
AJC	0.1	0.0	33.3	–	–	100.0	0.0
ASI	6.2	2.4	20.3	0.61	0.73	39.6	3.5
APM	0.1	0.0	0.0	0.72	0.72	100.0	0.0
APA	0.1	16.7	16.7	–	–	100.0	0.0
XMF	0.8	0.0	5.0	0.83	0.89	92.1	2.6
XMC	0.1	0.0	0.0	–	–	100.0	0.0
XMT	0.1	0.0	33.3	–	–	97.0	0.0
XTF	0.8	0.0	7.1	0.63	0.94	100.0	0.0
XER	0.3	0.0	0.0	0.97	0.97	100.0	0.0
XEE	0.1	0.0	50.0	–	–	100.0	0.0

7.4. Answers to Research Questions

In Section 6.1, four research questions were stated. Next, answers to these questions are provided in the light of the previous experimental results.

RQ1. *Does any operator generate invalid mutants?* Yes, all the mutants produced by the ACI operator are invalid for the compositions under study. Although specific compositions were designed for testing this operator, it has not been possible to produce valid mutants from it. Consequently, this operator has been discarded. Regarding the ASF operator, 16.2% of its mutants were invalid mutants. Having modified its implementation, now it generates no invalid mutants. Two other operators, AEL and ASI, generate a small percentage of invalid mutants (< 5%), but this is because of the nature of the fault that they inject into the code and, thus, those invalid mutants cannot be avoided by modifying the operator.

RQ2. *Does any operator generate a high percentage of equivalent mutants?* Yes, 50% of mutants generated by the EMF operator were equivalent mutants. Once its definition and implementation have been refined no equivalent mutants are produced. Other operators generating a percentage of equivalent mutants not less than 50% are ECC (83.9%), ASF (70.4%), and XEE (50%). However, in their case the equivalent mutants are inherent to the faults injected and particular features of the compositions that they are applied to. Thus, for example, operator ASF, which replaces a `sequence` activity by a `flow` one, always produces equivalent mutants when there are no data dependencies between child activities.

RQ3. *How many weak and resistant mutants does each operator produce?* Weak mutants are killed by every test case. Resistant mutants are killed just by one test case. The proportion of both quantities in the set of dead mutants is an interesting figure that has been computed and

DOI: 10.1002/stvr

Table XVIII. LoanApproval Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	$\mathcal{E}_O$	MOS
ERR	10	0	0	0.60	0.60	0.60	0.20
ECC	13	0	7	0.39	0.84	0.39	0.50
ECN	4	0	0	0.75	0.75	0.50	0.50
ASF	3	0	0	–	–	–	–
AEL	17	1	0	0.74	0.74	0.50	0.19
AIE	2	0	0	–	–	–	–
ASI	12	0	0	0.59	0.59	0.62	0.17

presented in Table XVII. However, these numbers alone are a rough measure of the goodness of an operator. Thus, other metrics, as those defined in Section 3 should be considered. This is the purpose of the next question.

RQ4. *Which is the quality of each mutation operator?* Quality metrics of mutations operator after improvements are presented in Table XVII. By observing the variation in the resulting values, a threshold to eliminate low quality operators can be established. The authors propose studying whether the operators where both Q_O and Q_{D_O} are below their medians should be removed. Thus, ISV, EAA, ASF, AWR, and ASI, whose $Q_O < 0.65$ and $Q_{D_O} < 0.81$, would be discarded.

Although EMF, AFP, AJC, APA, XMC, XMT, and XEE could not be evaluated, they do no harm, as they hardly produce a significant number of mutants. Therefore, discarding these operators a priori is not recommended.

7.5. Threats to Validity

Threats to *external validity* include the use of small WS-BPEL compositions that do not allow us to generate a large number of mutants for most of the operators. This has been mitigated by creating bigger artificial compositions (COMBO and COMBO2) from smaller compositions.

Another threat is that the most specific mutation operators are evaluated on the results produced by a small number of compositions. Since they are very specific, they can only be applied in very special situations. In fact, some of these operators produce very few mutants, and the values obtained cannot be considered fully significant.

Regarding the initial test-suite, the importance of using minimal adequate test-suites has been discussed. However, such test-suites may not be unique and, of course, they depend on the original test-suite. It is implicitly assumed that there will be no significant differences when replacing one test-suite by another, as long as they share the required properties.

8. COMPARING THE QUALITY OF AN OPERATOR WITH OTHER METRICS

In this section, the evaluated operator quality metrics are compared to other metrics proposed in the literature, such as the effectiveness [26] or the strength of an operator (MOS) [29].

The quality of a mutation operator, Q_O (see Definitions 17 and 15) depends on how resistant its generated mutants are. When an operator only generates hard to kill mutants, its quality will be higher than when it generates plain resistant mutants, which will be in turn higher than when it generates weak mutants. Q_O also depends on the number of equivalent mutants produced by the O operator: the more equivalent mutants that are generated, the lower its quality will be. To study the quality of the operator regardless of the equivalent mutants that it generates, the authors have proposed the Q_{D_O} metric (definition 18) that only takes into account the quality of the killed mutants. When an adequate test-suite is used, Q_O and Q_{D_O} will be equal if O does not produce equivalent mutants. Otherwise, Q_{D_O} will be higher than Q_O.

DOI: 10.1002/stvr

Table XIX. TravelReservationService Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	$\mathcal{E}_O$	MOS
ERR	30	0	6	0.67	0.84	0.40	0.25
ECC	24	0	23	0.00	–	–	–
ECN	24	0	0	0.94	0.94	0.24	0.25
EMD	6	0	1	0.73	0.88	0.33	0.60
EMF	1	0	0	–	–	–	–
ASF	13	0	13	0.00	–	–	–
AEL	62	2	8	0.82	0.94	0.23	0.15
ASI	34	0	7	0.69	0.90	0.30	0.31
APA	3	0	1	–	–	–	–
XMF	5	0	0	0.84	0.84	0.40	0.80
XMT	1	0	0	–	–	–	–
XTF	6	0	0	0.89	0.89	0.33	0.50
XER	4	0	0	0.94	0.94	0.25	1.00
XEE	2	0	0	–	–	–	–

Table XX. MetaSearch Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	$\mathcal{E}_O$	MOS
ISV	116	0	8	0.59	0.64	0.60	0.04
EAA	44	0	0	0.55	0.55	0.67	0.07
EEU	2	0	2	–	–	–	–
ERR	65	0	9	0.67	0.78	0.42	0.07
ELL	2	0	0	–	–	–	–
ECC	41	0	40	0.00	–	–	–
ECN	100	0	15	0.62	0.73	0.50	0.07
AFP	2	0	1	–	–	–	–
ASF	9	0	4	0.00	0.00	1.00	0.20
AEL	73	0	0	0.71	0.71	0.47	0.05
AIE	11	0	2	0.64	0.78	0.41	0.33
ASI	39	0	3	0.54	0.58	0.56	0.08
XMF	4	0	0	0.81	0.81	0.42	0.75

$\mathcal{E}_O$ measures the effectiveness of the test cases regarding the mutants produced by a mutation operator. Please, remember that, contrary to mutant quality, the more effective the test-suite is, the worse the mutants are, and vice versa. According to equation 22, the minimum value of $\mathcal{E}_O$ is $1/|T_O|$. When $\mathcal{E}_O$ reaches this minimum value, it means that it only produces resistant mutants. On the other hand, the maximum value of $\mathcal{E}_O$ for adequate test-suites is 1. $\mathcal{E}_O = 1$ means that all the mutants generated by the operator are weak.

For adequate test-suites, MOS is the ratio between the number of test cases required to kill the mutants generated by an operator and the number of non-equivalent mutants generated, $|T_O| \,/\, |D_O|$. MOS will be equal to 1 when the mutants are hard to kill. In the opposite case when all the mutants can be killed with a single test case, MOS will be equal to $1/|D_O|$. Therefore, an operator will be stronger when MOS is closer to 1.

From these observations, Q_O and $\mathcal{E}_O$ could be expected to deviate in opposite directions, as Q_O measures the quality of the operator and $\mathcal{E}_O$ measures the effectiveness of the test-suite against the mutants generated by the operator. That is to say, whenever Q_O increases $\mathcal{E}_O$ should decrease and the other way around. In contrast, Q_O and MOS should increase or decrease at the same time.

Let us analyze the behavior of these three metrics in some particular cases:

DOI: 10.1002/stvr

Table XXI. COMBO2 Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	$\mathcal{E}_O$	MOS
ISV	201	0	32	0.65	0.78	0.39	0.03
EAA	116	0	5	0.74	0.77	0.48	0.03
EEU	2	0	1	–	–	–	–
ERR	75	0	17	0.71	0.92	0.18	0.14
ELL	5	0	1	0.70	0.88	0.33	0.75
ECC	142	0	100	0.23	0.77	0.42	0.14
ECN	68	0	23	0.59	0.90	0.25	0.09
EMD	4	0	2	0.00	–	–	–
EMF	1	0	0	–	–	–	–
AFP	1	0	0	–	–	–	–
ASF	20	0	15	0.23	0.90	0.30	0.80
AIS	4	0	0	0.75	0.75	0.50	0.50
AEL	145	10	21	0.81	0.96	0.15	0.11
AIE	6	0	3	0.44	–	–	–
AWR	3	0	0	–	–	–	–
AJC	2	0	0	–	–	–	–
ASI	86	8	16	0.74	0.93	0.23	0.15
APM	6	0	0	0.72	0.72	0.50	0.33
APA	3	1	0	–	–	–	–
XMF	7	0	2	0.66	0.93	0.25	0.80
XMC	3	0	0	–	–	–	–
XMT	2	0	1	–	–	–	–
XTF	4	0	3	0.00	–	–	–
XER	1	0	0	–	–	–	–
XEE	2	0	2	–	–	–	–

Table XXII. LAE Composition

	Total	Invalid	Equivalent	Q_O	Q_{D_O}	$\mathcal{E}_O$	MOS
ISV	1684	0	325	0.52	0.64	0.58	0.02
EAA	204	0	37	0.59	0.72	0.48	0.05
EEU	6	0	0	0.89	0.89	0.33	0.83
ERR	475	0	90	0.76	0.94	0.23	0.08
ELL	20	0	2	0.67	0.74	0.42	0.22
ECC	103	0	101	0.00	–	–	–
ECN	548	0	118	0.70	0.90	0.31	0.13
AFP	2	0	1	–	–	–	–
ASF	26	0	18	0.21	0.68	0.56	0.50
AIS	2	0	0	–	–	–	–
AEL	291	0	17	0.85	0.90	0.29	0.09
AIE	55	0	6	0.86	0.97	0.18	0.47
AWR	4	0	0	0.44	0.44	0.75	0.50
AJC	1	0	1	–	–	–	–
ASI	159	0	40	0.48	0.64	0.53	0.14
XMF	24	0	0	0.99	0.99	0.07	0.58
XMC	1	0	0	–	–	–	–
XTF	32	0	0	0.99	0.99	0.08	0.41
XER	10	0	0	0.99	0.99	0.10	1.00

DOI: 10.1002/stvr

Table XXIII. Comparative summary of metrics including all the compositions

	Q_O	Q_{D_O}	$\mathcal{E}_O$	MOS
ISV	0.59	0.69	0.52	0.03
EAA	0.63	0.68	0.54	0.05
EEU	0.89	0.89	0.33	0.83
ERR	0.68	0.82	0.41	0.15
ELL	0.69	0.81	0.38	0.49
ECC	0.12	0.81	0.41	0.32
ECN	0.72	0.84	0.36	0.21
EMD	0.37	0.88	0.33	0.60
EMF	–	–	–	–
AFP	–	–	–	–
ASF	0.11	0.53	0.62	0.50
AIS	0.75	0.75	0.50	0.50
AEL	0.79	0.85	0.33	0.12
AIE	0.65	0.88	0.30	0.40
AWR	0.44	0.44	0.75	0.50
AJC	–	–	–	–
ASI	0.61	0.73	0.45	0.17
APM	0.72	0.72	0.50	0.33
APA	–	–	–	–
XMF	0.83	0.89	0.29	0.73
XMC	–	–	–	–
XMT	–	–	–	–
XTF	0.63	0.94	0.21	0.46
XER	0.97	0.97	0.18	1.00
XEE	–	–	–	–

- *What happens when all the mutants generated by an operator are equivalent?* Q_O would be 0, indicating that it is a low quality operator. However, $\mathcal{E}_O$ and MOS are undefined (also marked as – in table entries).
- *What happens when an operator generates a single mutant?* Q_O would be 0. In this case this means that there is not enough information to establish the quality of the operator. By contrast, $\mathcal{E}_O$ and MOS yield 1, though these values are interpreted in the same way, as they have been obtained from a single data point and they cannot be considered significant.
- *What happens when an operator generates several mutants that are all killed with the same test case?* Q_O would be 0, indicating that it is a low quality operator, as a single test case can kill all of its mutants. $\mathcal{E}_O$ is equal to 1, indicating the same fact. MOS is equal to $1/\left|D_O\right|$: its value depends on the number of killed mutants that the operator generates.
- *What happens when all the mutants produced by an operator are hard to kill?* This kind of operator should produce the highest values of quality and strength, and conversely, the lowest values of effectiveness. For such an operator, Q_O would be equal to $1 - 1/(\left|D_O\right| \cdot \left|T_O\right|)$, effectiveness would reach its minimum value, $1/\left|T_O\right|$, and MOS would be equal to 1. While MOS reach its absolute maximum, both Q_O and $\mathcal{E}_O$ take into account the number of data available. The higher the number of mutants and test cases, the closer to the optimum value. In this sense, they allow for finer-grain analysis than MOS, distinguishing between apparently similar scenarios.

The results obtained for the three metrics across the available compositions confirm that the relation between Q_O and $\mathcal{E}_O$ is as expected. However, the relative values of Q_O and MOS do not follow the expected pattern. For some operators Q_O is high and MOS is low, while for others Q_O is low and MOS is high. Nevertheless, Q_O is finer for distinguishing operators than $\mathcal{E}_O$ and MOS.

DOI: 10.1002/stvr

There are pairs of operators for which the values of $\mathcal{E}_O$ are similar but the values of Q_O are different (see EMD–XTF in Table XIX and AIE–XMF in Table XX). This happens, for example, when the proportions of equivalent mutants generated by the operators are significantly different.

MOS behavior seems sometimes erratic, producing identical strengths for operators with very different quality and effectiveness measures (see ECN–ERR in Table XIX and ECC–ECN in Table XVIII), and very different strengths for operators with similar quality measures and effectiveness measures (see, for example, ECN–AEL in Table XVIII).

An advantage of Q_O over $\mathcal{E}_O$ and MOS is that it takes into account the equivalent mutants produced by the operator, penalizing those producing many equivalent mutants.

A graphical comparison of the metric behaviors using the data contained in Table XXIII is illustrated by Figure 5. From this figure, many of the previous observations can be confirmed and new insights can be derived:

1. *Comparison of Q_O versus Q_{D_O}*. Clearly, Q_{D_O} is never below Q_O, as it focus on a subset of the mutants considered in Q_O. Differences imply the existence of equivalent mutants. ECC and ASF are the two operators that generate the largest proportions of equivalent mutants and they are easily spotted by the bigger gap between both plots.
2. *Comparison of Q_O versus $\mathcal{E}_O$*. No clear relation can be visually established as Q_O takes into account the equivalent mutants while $\mathcal{E}_O$ explicitly omits them in its formulation. However, Q_O tends to increase when $\mathcal{E}_O$ decreases and vice versa, confirming that they deviate in opposite directions.
3. *Comparison of Q_{D_O} versus $\mathcal{E}_O$*. Q_{D_O} is analogous to Q_O when equivalent mutants are not considered. Now, Q_{D_O} and $\mathcal{E}_O$ exhibit almost perfect symmetry around the dotted line. As previously commented, they are, in a sense, complementary. This can be easily appreciated thanks to these plots.
4. *Comparison of Q_O versus MOS*. Unlike Q_O, MOS does not take into account the equivalent mutants, though this does not explain its behavior, as it produces identical strengths for operators with very different Q_O (ASF–AIS) and remarkably different strengths for operators with similar Q_O (ERR–ELL). Other plots comparing MOS with different metrics confirm this trend, as well as the above discussion.
5. *Comparison of Q_{D_O} versus MOS*. Even when Q_{D_O} and MOS are on equal grounds regarding equivalent mutants, the same behavior is observed again. Peaks in MOS do not seem to follow a consistent pattern.
6. *Comparison of $\mathcal{E}_O$ versus MOS*. Again, no relation with MOS can be stated. For example, sometimes low strengths correspond to low effectiveness, when intuition suggest that it should be just the opposite (AEL, ECN).

9. CONCLUSIONS AND FUTURE WORK

This work introduces generic but precisely defined metrics that can be used to assess the quality of mutants and the quality of mutation operators in a mutation testing framework. Armed with these metrics, the effectiveness of mutant reduction techniques and of mutation operators themselves can be studied for a particular language. WS-BPEL 2.0 has been chosen as the target language and the MuBPEL mutation analysis tool has been developed for this purpose. MuBPEL is one of the few tools capable of performing firm mutation analysis.

The quality metrics for mutation operators were first applied to four WS-BPEL compositions. Then, the behavior of the operators was evaluated and the useful information obtained about them guided us when deciding which operators should be modified or discarded.

After performing these changes, experiments were repeated, and metrics for the mutation operators were recomputed after replacing one of the compositions with an extension and introducing a new composition to obtain additional information about some operators that could not be evaluated in the first study.

DOI: 10.1002/stvr

Quality metrics depend both on the mutants and the test-suites used to measure them. In order to ensure that the values obtained for the various compositions and operators are comparable with each other, test-suites with well-defined requirements have been used: they should be adequate, non-redundant, and minimal. These minimal test-suites have been computed both for the set of mutants generated by a composition and for the subsets generated by each particular mutation operator.

These metrics can be used for reducing the number of mutants by discarding those mutation operators whose quality is under an appropriate threshold or that do not meet other suitable quality criteria, maybe taking into account different quality metrics. Consequently, this allows us to obtain a set of high-quality operators that is smaller than the original and generates less mutants, while maintaining a similar effectiveness.

For example, one approach would be to select only operators for which $Q_{D_O} \geq 0.85$ or $Q_{D_O} - Q_O \leq 0.2$, excluding ECC and ASF, the two operators that generate the largest proportions of equivalent mutants. But more restrictive criteria can be defined. In particular, we could discard operators ISV, EAA, ASF, AWR and ASI, where both $Q_O < 0.65$ and $Q_{D_O} < 0.81$, that is, below their respective median values. These operators are responsible for 51.8% of the total number of mutants generated.

The results of this study show that six operators can be discarded. One of these operators, ACI, was discarded after several failed attempts to define a composition from which it could generate valid mutants and after studying the deployment-time static restrictions imposed by the WS-BPEL standard and the WS-BPEL engine employed. The remaining discarded operators ISV, EAA, ASF, AWR, and ASI have been chosen by applying reasonable thresholds to the quality metrics that the authors have defined.

Finally, the proposed metrics were compared to two related ones: Derezińska's effectiveness [26] and mutation operator strength [29]. One advantage of Q_O over $\mathcal{E}_O$ and MOS is that it punishes those operators producing many equivalent mutants. It was also found that Q_O is a finer metric than effectiveness and mutation strength in the sense that it can distinguish operators that, from the viewpoint of the other metrics, are similar in goodness.

Regarding future work, the authors would like to study new compositions exercising the whole set of mutation operators for WS-BPEL. Unfortunately, open-source and freely available WS-BPEL compositions are very scarce. This is because of the economic importance of those compositions for the enterprises developing them, as well as the strategic value of the information that is sometimes buried in their internal logic.** Moreover, enterprises are (for good reasons) reluctant to allow researchers to play with their compositions in a production environment and it is not that easy to reproduce the environment in which the real compositions are executed. In order to overcome this difficulties, a repository of compositions reflecting different parts of the WS-BPEL language is being developed. This is an ongoing effort and the authors plan to make this repository freely available for the research community.

If the thresholds for the quality values are set too low, useless operators will be included and increase the cost of running all the mutants. Conversely, using thresholds that are too high will discard useful mutants. The thresholds presented above were just two plausible examples, but many different sets could be defined. The authors plan to conduct a study to fine-tune the threshold values by comparing the effectiveness of the reduced operator set with the effectiveness of the full set.

Another way of taking advantage of the metrics defined is to compare how different reduction techniques perform for the same programming language. The technique consistently producing the highest percentage of quality mutants will be the best. It would be interesting to study the effectiveness of mutant reduction techniques as a function of the quality of the subset of mutants that they generate, thus establishing their relative strength.

Finally, another research line would be studying the degree with which one WS-BPEL mutation operator interacts with another. The proposed metrics can be used to measure the quality of the operators by themselves or in relation with others, depending on the particular test-suite used. By

**A composition usually reflects a business process.

DOI: 10.1002/stvr

comparing operators pairwise, it could be determined whether they are orthogonal or they exhibit some interactions instead.

ACKNOWLEDGEMENTS

This work was funded by the Spanish Ministry of Science and Innovation under the National Program for Research, Development and Innovation, project MoDSOA (TIN2011-27242), and by the University of Cádiz under its Research Promotion Plan, project PR2011-004.

REFERENCES

1. Offutt AJ, Untch RH. *Mutation Testing for the New Century*, chap. Mutation 2000: Uniting the Orthogonal. Kluwer Academic Publishers, 2001; 34–44.
2. Jia Y, Harman M. Higher order mutation testing. *Information and Software Technology* 2009; **51**(10):1379–1393.
3. Polo Usaola M, Reales Mateo P. Mutation testing cost reduction techniques: A survey. *Software, IEEE* 2010; **27**(3):80–86.
4. Acree AT. On mutation. PhD Thesis, Georgia Institute of Technology 1980.
5. Budd TA. Mutation analysis of program test data. PhD Thesis, Yale University 1980.
6. Hussain S. Mutation clustering. Master's Thesis, King's College London 2008.
7. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology* 1996; **5**:99–118.
8. Domínguez-Jiménez J, Estero-Botaro A, García-Domínguez A, Medina-Bulo I. Evolutionary mutation testing. *Information and Software Technology* 2011; **53**(10):1108–1123.
9. Howden WE. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* 1982; **8**(4):371–379.
10. Woodward MR, Halewood K. From weak to strong, dead or alive? an analysis of some mutationtesting issues. *TVA'88: 2nd Workshop on Software Testing, Verification, and Analysis*, IEEE Computer Society, 1988; 152–158.
11. Offutt J. Automatic test data generation. PhD Thesis, Georgia Institute of Technology, Atlanta, GA, USA 1988.
12. Harman M, Hierons R, Danicic S. *Mutation Testing for the New Century*, chap. The relationship between program dependence and mutation analysis. Kluwer Academic Publishers, 2001; 5–13.
13. Jackson D, Woodward MR. *Mutation Testing for the New Century*, chap. Parallel firm mutation of Java programs. Kluwer Academic Publishers, 2001; 55–61.
14. Baldwin D, Sayward FG. Heuristics for determining equivalence of program mutations. *Techreport 276*, Yale University, New Haven, Connecticut 1979.
15. Offutt AJ, Craft WM. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 1994; **4**(3):131–154.
16. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability* 1997; **7**(3):165–192.
17. Hierons R, Harman M, Danicic S. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability* 1999; **9**:233–262.
18. Voas J, McGraw G. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.
19. Adamopoulos K, Harman M, Hierons RM. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. *GECCO 2004: Proceedings of the Genetic and Evolutionary Computation Conference*, 2004; 1338–1349.
20. Ellims M, Ince D, Petre M. The Csaw C mutation tool: Initial results. *TAICPART-Mutation'07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - Muation*, IEEE Computer Society Press, 2007; 185–192.
21. Grün BJM, Schuler D, Zeller A. The impact of equivalent mutants. *Mutation'09: 4th International Workshop on Mutation Analysis*, IEEE Computer Society Press, 2009; 192–199.
22. Schuler D, Dallmeier V, Zeller A. Efficient mutation testing by checking invariant violations. *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009; 69–80.
23. Schuler D, Zeller A. (un-)covering equivalent mutants. *ICST '10: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, IEEE Computer Society, 2010; 45–54.
24. Schwarz B, Schuler D, Zeller A. Breeding high-impact mutations. *ICSTW '11: Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011; 382–387.
25. OASIS. Web Services Business Process Execution Language 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html` 2007.
26. Derezińska A. Quality assessment of mutation operators dedicated for C# programs. *QSIC 2006: Sixth International Conference on Quality Software*, IEEE Computer Society Press, 2006; 227–234.
27. Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I. Quantitative evaluation of mutation operators for WS-BPEL compositions. *IEEE International Conference on Software Testing Verification and Validation Workshops*, IEEE Computer Society, 2010; 142–150.
28. Smith BH, Williams L. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering* 2009; **14**(3):341–369.
29. Hu J, Li N, Offutt J. An analysis of OO mutation operators. *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011; 334–341.

DOI: 10.1002/stvr

30. Ma Y, Offutt J, Kwon YR. MuJava: An automated class mutation system. *Software Testing, Verification and Reliability* 2005; **15**(2):97–133.
31. Derezińska A. Quality assessment of mutation operators dedicated for C# programs. *QSIC 2006: Sixth International Conference on Quality Software*, IEEE Computer Society Press: Beijing, China, 2006; 227–234.
32. Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I. Mutation operators for WS-BPEL 2.0. *ICSSEA 2008, 21th International Conference on Software & Systems Engineering and their Applications*, 2008.
33. Boubeta-Puig J, Medina-Bulo I, García-Domínguez A. Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages. *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, IEEE Computer Society, 2011; 398–407.
34. Agrawal H, Demillo R, Hathaway R, Hsu W, Hsu W, Krauser E, Martin RJ, Mathur A, Spafford E. Design of mutant operators for the C programming language. *Technical Report SERC-TR-41-P*, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana 1989.
35. N King K, Offutt AJ. A FORTRAN language system for mutation-based software testing. *Software - Practice and Experience* 1991; **21**(7):685–718.
36. Zhang H. Fault generation tool. Master's Thesis, Kansas State University 2003. http://people.cis.ksu.edu/~hzh8888/mse_project/FGTSRS.doc.
37. Mansour N, Houri M. Testing web applications. *Information and Software Technology* 2006; **48**(1):31–42.
38. Ma Y, Offutt J, Kwon Y. MuJava: a mutation system for Java. *ICSE'06: 28th International Conference on Software Engineering*, ACM: New York, NY, USA, 2006; 827–830.
39. Madeyski L, Radyk N. Judy - a mutation testing tool for Java. *IET Software* 2010; **4**(1):32–42.
40. Ji C, Chen Z, Xu B, Wang Z. A new mutation analysis method for testing Java exception handling. *COMPSAC'09: 33rd Annual IEEE International Computer Software and Applications Conference*, IEEE Computer Society: Los Alamitos, CA, USA, 2009; 556–561.
41. Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrent Java (J2SE 5.0). *Mutation'06: Proceedings of the Second Workshop on Mutation Analysis*, IEEE Computer Society Press, 2006; 11.
42. Kim S, Clark JA, McDermid JA. Class mutation: Mutation testing for object-oriented programs. *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, 2000; 9–12.
43. Tuya J, Suárez-Cabal MJ, de la Riva C. Mutating database queries. *Information and Software Technology* 2007; **49**(4):398–417.
44. Lonetti F, Marchetti E. X-MuT: a tool for the generation of XSLT mutants. *Proceedings of the Seventh International Conference on the Quality of Information and Communications Technology*, IEEE Computer Society, 2010; 280–285.
45. ActiveVOS. Activebpel. http://sourceforge.net/projects/activebpel502/.
46. Mayer P, Lübke D. Towards a BPEL unit testing framework. *TAV-WEB'06, 2006 workshop on Testing, analysis, and verification of web services and applications*, ACM, 2006; 33–42.
47. NetBeans Enterprise. NetBeans. http://www.netbeans.org 2012.

DOI: 10.1002/stvr

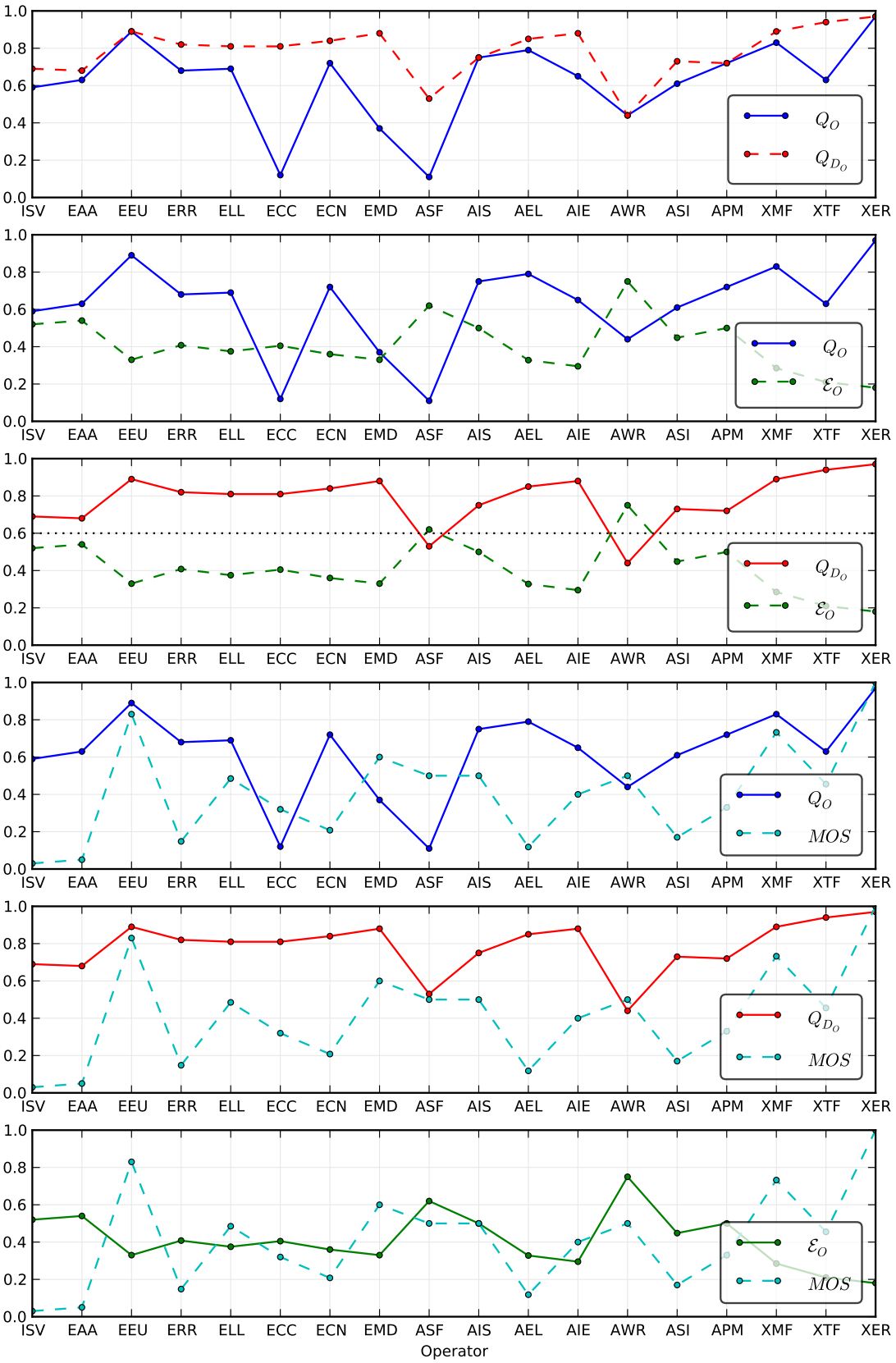


Figure 5. Comparison between quality metrics

DOI: 10.1002/stvr