Evolutionary Mutation Testing

J. J. Domínguez-Jiménez*, A. Estero-Botaro, A. García-Domínguez, I. Medina-Bulo

Dept. Computer Languages and Systems
University of Cádiz, Escuela Superior de Ingeniería
C/ Chile, nº 1, 11002, Cádiz, Spain

Abstract

Context: Mutation testing is a testing technique that has been applied successfully to several programming languages. However, it is often regarded as computationally expensive, so several refinements have been proposed to reduce its cost. Moreover, WS-BPEL compositions are being widely adopted by developers, but present new challenges for testing, since they can take much longer to run than traditional programs of the same size. Therefore, it is interesting to reduce the number of mutants required.
Objective: We present Evolutionary Mutation Testing (EMT), a novel mutant reduction technique for finding mutants that help derive new test cases that improve the quality of the initial test suite. It uses evolutionary algorithms to reduce the number of mutants that are generated and executed with respect to the exhaustive execution of all possible mutants, keeping as many difficult to kill and potentially equivalent mutants (*strong mutants*) as possible in the reduced set.
Method: To evaluate EMT we have developed GAmera, a mutation testing system powered by a co-evolutive genetic algorithm. We have applied this system to three WS-BPEL compositions to estimate its effectiveness, comparing it with random selection.
Results: The results obtained experimentally show that EMT can select all strong mutants generating 15% less mutants than random selection in

*Corresponding author. Tel. +34 956015597; fax. +34 956015139
Email addresses: `juanjose.dominguez@uca.es` (J. J. Domínguez-Jiménez), `antonia.estero@uca.es` (A. Estero-Botaro), `antonio.garciadominguez@uca.es` (A. García-Domínguez), `inmaculada.medina@uca.es` (I. Medina-Bulo)

over 20% less time for complex compositions. When generating a percentage of all mutants, EMT finds on average more strong mutants than random selection. This has been confirmed to be statistically significant within a 99.9% confidence interval.
Conclusions: EMT has reduced for the three tested compositions the number of mutants required to select those which are useful to derive new test cases that improve the quality of the test suite. The directed search performed by EMT makes it more effective than random selection, especially as compositions become more complex and the search space widens.

Keywords: Mutation Testing, Evolutionary Algorithm, Genetic Algorithm, Web Service, WS-BPEL

1. Introduction

Mutation testing [1, 2, 3] is a fault-based testing technique providing a test criterion: the *mutation score*. This criterion can be used to measure the effectiveness of a test suite in terms of its ability to detect faults. Mutation testing generates *mutants* from the program under test by applying *mutation operators* to it. These mutation operators introduce slight syntactical changes into the program that should be detected by a high-quality test suite.

Most mutant generation systems simply generate all possible mutants. Usually, they include a wide array of mutation operators. Each mutation operator generates a large number of mutants. All mutants need to be run against the test suite to determine whether they can be told apart from the original program in some of its test cases (that is, whether they are *killed* by the test suite or not). The entire process can take a long time for nontrivial programs.

Traditionally, one of the main drawbacks of mutation testing [4] has been the high computational cost involved in the execution of the large number of mutants produced for some programs against their test suites. Several strategies have been described in the literature to address this problem. One of them consists in processing only a subset of all the mutants, using *mutant reduction techniques*. *Mutant sampling* [5, 6] randomly selects a subset of the mutants. *Mutant clustering* [7] classifies mutants into different clusters according to the set of test cases that kills them. *Selective mutation* [8] applies only a subset of the available mutation operators. Recently, *higher-order mutation* [9] has been used to find rare, but valuable, higher-order mutants (HOMs) modeling subtle faults. One of these HOMs can subsume many ordinary (first-order) mutants.

We present a new mutant reduction technique, Evolutionary Mutation Testing (EMT), which tries to generate and execute only some of all the mutants, while preserving testing effectiveness. EMT generates and selects mutants in a single step, reducing the number of mutants to execute by favoring through the fitness function two kinds of mutants: surviving mutants (which have not been killed by the test suite) and difficult to kill mutants (which have been killed by one and only one test case that kills no other mutant). We call them *strong mutants*, and they can be used to improve the quality of the initial test suite. The main steps of EMT are:

1. Generate a set of mutants.
2. Execute these mutants.
3. Evaluate their fitnesses.
4. Apply the evolutionary algorithm to the set of mutants to find stronger mutants.
5. Go to step 2 until the termination condition is met.

Why are we proposing a new technique to generate less mutants? Nowadays, computers are faster than those of a decade ago, so the time needed to execute all the mutants against the test suite is now smaller. At the same time, software development is now performed at a higher level of abstraction. In particular, the WS-BPEL (Web Services Business Process Execution Language) standard [10] has been defined for "programming in the large" composing several Web Services (WS hereafter) into one.

WS-BPEL compositions are usually 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. However, this does not imply a reduction of cost at all. WS-BPEL compositions can require more resources than regular programs, so it is important to reduce the number of mutants. They are typically executed by WS-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.

For these reasons, we implemented the system GAmera to apply EMT to WS-BPEL compositions[1] [11]. GAmera can generate, execute, evaluate and classify the mutants into strong (surviving and difficult to kill mutants) and weak mutants (the rest). GAmera is based on a genetic algorithm (GA) [12, 13]. GAs have proved to be an effective heuristic optimization strategy for functions with many local optima that traditional methods find difficult to handle. WS-BPEL mutants are generated by using the 26 mutation operators originally defined for this language by Estero [14].

Experiments show that EMT can select all the strong mutants in less time than random selection, especially for complex compositions. In addition, the directed search performed by EMT improves over random selection.

The structure of the rest of the paper is as follows: Section 2 introduces mutation testing, GAs and the WS-BPEL language. Section 3 presents EMT, a new mutant reduction technique. Section 4 describes the internal structure of GAmera. Section 5 evaluates experimentally EMT using three WS-BPEL compositions. Section 6 discusses related work. Finally, Section 7 presents the conclusions and future work.

2. Background

In order to make this work self-contained, we provide some background on mutation testing (Section 2.1), some useful concepts about GAs (Section 2.2), and the main characteristics of the WS-BPEL language (Section 2.3).

2.1. Mutation Testing

Mutation testing [1, 2, 3] is a fault-based testing technique that introduces simple syntactic changes in the original program by applying mutation operators. The resulting programs are called *mutants*. Each mutation operator models a category of errors that the developer could make. Thus, if a program contains the instruction `a > 5000` and we apply the relational mutation operator (which replaces a relational operator with another), the resulting mutant could contain the instruction `a < 5000` instead, for example. If a test case is able to distinguish between the original program and the mutant, i.e. their outputs are different, it is said that this test case *kills*

[1]GAmera, like Mothra, is a character in the Japanese mutant giant monster (*daikaijuu*) movies of the 1960s.

the mutant. On the contrary, if no test case in the test suite is able to distinguish between the mutant and the original program, it is said that the mutant stays *alive*. When a mutant always produces the same output as the original program, it is said to be *equivalent*.

Mutation testing can be used in two ways. First, we can measure the quality of a test suite with its *mutation score*. It is defined as the percentage of killed mutants. More formally, we have that

$$\text{Mutation Score} = \frac{K}{M - E} \times 100 \tag{1}$$

where K is the number of killed mutants, M is the total number of mutants, and E is the number of equivalent mutants. A problem with this formula is that the value of E is in general not known, and it is necessary to manually inspect the mutants to identify those that are equivalent.

The second use case for mutation testing is to generate new test cases in order to kill the surviving mutants, and thus improve the quality of the initial test suite. Ideally, we would like the mutation score to reach 100%, indicating the test suite is *adequate* to detect all the faults modeled by the mutants.

The existence of equivalent mutants is a common problem when applying mutation testing. These mutants cannot be told apart from the original program, as they always produce the same outputs. Equivalent mutants should not be confused with *stubborn non-equivalent mutants*, which are produced because the test suite is inadequate to detect them. The general problem of determining if a mutant is equivalent to the original program is undecidable [15].

One of the main drawbacks of mutation testing is the high computational cost involved. There is usually a large number of mutation operators that generate vast numbers of mutants, each of which must be executed against the test suite. The empirical study performed by Offutt et al. [16] shows that, under certain conditions, the number of mutants has quadratic complexity in program size.

2.2. Genetic Algorithms

Genetic algorithms [12, 13] are probabilistic search techniques based on the theory of evolution and natural selection proposed by Charles Darwin, which Herbert Spencer summarized as "survival of the fittest".

GAs work with a population of solutions, known as *individuals*, and process them in parallel. Throughout successive generations of the population, GAs perform a selection process to improve the population, so they are ideal for optimization. In this sense, GAs favor the best individuals and generate new ones through the recombination and mutation of information from existing ones. The strengths of GAs are their flexibility, simplicity and ability for hybridization. Among their weaknesses are their heuristic nature and the difficulties in handling restrictions.

There is no single type of GA, but rather several families that mostly differ in how individuals are encoded (binary, floating point, permutation, ...), and how the population is renewed in each generation. Generational GAs [12] replace the entire population in each generation. However, steady-state or incremental GAs [17] replace only a few (one or two) members of the population in each generation. Finally, in parallel fine-grained GAs [18] the population has a distributed structure in nodes.

As each individual represents a solution to the problem to be solved, its *fitness* measures the quality of this solution. The average population fitness will be maximized along the different generations produced by the algorithm. The encoding scheme used and the individual fitness are highly dependent on the problem to solve, and they are the only link between the GA and the problem [19].

GAs use two types of operators: selection and reproduction. *Selection operators* select individuals in a population for reproduction. The probability of selecting an individual for reproduction can be proportional or not to its fitness. *Reproduction operators* generate the new individuals in the population. There are two types of reproduction operators. On the one hand, *crossover operators* generate two new individuals, called children, from two pre-selected individuals or parents. The children inherit part of the information stored in both parents. On the other hand, *mutation operators* aim to alter the information stored in an individual. The design of these operators heavily depends on the encoding scheme used. It is important to note that these mutation operators are related to the GA and are different from those for mutation testing.

The canonical GA [20] performs the following steps, starting with $t = 0$:

1. Randomly generates an initial population $M(0)$.
2. Computes the fitness $f(m)$ for each individual m in the current population $M(t)$.

3. Defines selection probabilities $p(m)$, proportional to $f(m)$, for each individual m in $M(t)$.
4. Generates $M(t+1)$ by probabilistic selection of individuals from $M(t)$ to produce new individuals via reproduction operators.
5. Goes to step 2 with $t' = t + 1$ until a satisfactory solution is obtained.

2.3. The WS-BPEL Language

WS-BPEL [10] is an XML-based language which implements a business process as a WS which interacts with other external WS. Standard WS-BPEL process definitions are not coupled to the implementation details of the WS-BPEL engine they run on or the WS they invoke. WS-BPEL process definitions can be divided in four sections:

1. Declarations of the relationships to the external partners. These include both the client that has invoked the business process and the external partners whose services are required to complete the request of the client.
2. Declarations of the variables used by the process and their types. Variables are used for storing both the messages received and sent from the business process and the intermediate results required by the internal logic of the composition.
3. Declarations of handlers for various situations, such as fault, compensation or event handlers.
4. Description of the business process behavior.

The major building blocks of a WS-BPEL process are *activities*, XML elements which model assignments, control structures, message passing primitives or synchronization constraints, among others. There are two types: basic and structured activities. Basic activities specify a single action, such as receiving a message from a partner or performing an assignment to a variable. Structured activities contain other activities and prescribe their execution order. Activities may have both attributes and a set of containers. These containers can also include elements with their own attributes. Here is an example:

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

WS-BPEL provides concurrency and synchronization primitives. For instance, the `flow` activity runs a set of activities in parallel. Synchronization constraints between activities can be defined. In the above example, the `flow` activity invokes three WS in parallel: `checkFlight`, `checkHotel`, and `checkRentCar`. There is another WS, `bookFlight`, that will only be invoked if `checkFlight` is completed. Activities are synchronized by linking them: the target activity of every link will only be executed if the source activity of the link has been completed successfully.

3. Evolutionary Mutation Testing

We propose a new mutant reduction technique, named Evolutionary Mutation Testing (EMT), which uses an evolutionary algorithm. Our technique generates mutants on demand, as required by the selection process. It reduces the number of mutants by favoring potentially equivalent mutants (equivalent and stubborn non-equivalent mutants) and difficult to kill mutants. We consider these to be *strong mutants*. Strong mutants are useful to improve the quality of the initial test suite.

Figure 1 shows the outline of a system based on EMT. It consists of two main components: an evolutionary algorithm, which can be the same for programs written in several languages, and an execution engine, which depends on the language being used. The algorithm generates and selects mutants using a fitness function. In order to calculate the fitness, the mutant must be executed by the execution engine against the test suite.

A significant innovation in this technique is that it does not generate all the mutants. The generation is directed by the evolutionary algorithm through the fitness function to find the strong mutants.

Optionally, a test case generator could be integrated into the system, in order to improve the quality of the initial test suite. This test case generator

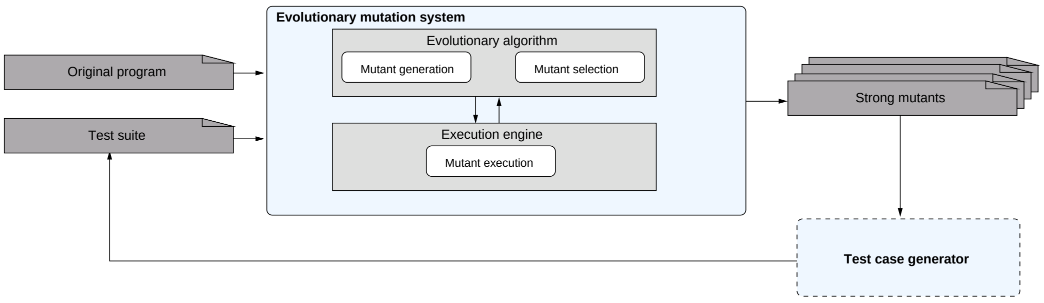


Figure 1: System based on EMT

would use the surviving strong mutants to extend the test suite with new test cases that kill them.

3.1. Mutant Encoding

A mutation on the original program is encoded using three fields (fig. 2). *Operator* contains the identifier of the mutation operator to be used, *Location* indicates where it should be applied, and *Attribute* selects the exact replacement to be made by the mutation operator, in case there are several options.

In order to automate mutant generation, an analyzer could compute the values the fields can take in a specific program. Additionally, an external converter could produce the mutated source code from the values of these fields.

Operator	**Location**	**Attribute**

Figure 2: Encoding of a mutant

3.2. Mutant Selection

We measure the usefulness of a mutant by evaluating the fitness function on it. To do so, we first need to run the mutant against the test suite.

Let M be the number of mutants in the population and T the number of test cases in the test suite. The execution results are collected into an $M \times T$ *execution matrix*, such as that in equation (2),

$$(m_{ij})_{M\times T} = \begin{pmatrix} 0 & 0 & 1 & \dots & 1 \\ 1 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 2 & 2 & 2 & \dots & 2 \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix} \tag{2}$$

where m_{ij} is 1 or 0 for most mutants, depending on whether the i-th mutant was killed or not by the j-th test case, respectively.

However, some mutants may fail to be run against the test suite, as they violate some static constraint defined by the language, i.e. they may fail to be compiled or deployed. These are commonly known as *stillborn mutants*. Their rows in the execution matrix will be set to $(2, 2, \dots 2)$. By looking for these rows in the execution matrix, we can evaluate the mutation operators themselves: if an operator produces stillborn mutants too often, its definition may have to be revised.

Stillborn mutants and their rows in the execution matrix are now discarded. Assuming that at least one mutant remains, we can establish the following constraints:

- Mutants can either be killed or survive: $\sum\limits_{j=1}^{T} m_{ij} \in \{0, \dots, T\}$, for all i.
- A test case can kill zero or more mutants: $\sum\limits_{i=1}^{M} m_{ij} \in \{0, \dots, M\}$, for all j.

The fitness function takes into account the number of test cases which kill a mutant and the number of mutants which are killed by the same test cases. Thus, the fitness for mutant I with the test suite S is given by:

$$\text{Fitness}(I, S) = M \times T - \sum_{j=1}^{T} \left(m_{Ij} \times \sum_{i=1}^{M} m_{ij} \right) \tag{3}$$

The value of the fitness function will always be comprised in the range $[0, M \times T]$. This function favors two kinds of mutants: potentially equivalent and difficult to kill mutants. A potentially equivalent mutant stays alive because one of these reasons: no test case reaches the mutation, the mutation is reached but the test cases cannot tell the mutant apart from the original, or the mutant is equivalent. Its fitness will be $M \times T$. A mutant is difficult to kill when there is one and only one test case which only kills this mutant and no other. Its fitness will be $M \times T - 1$. We consider both kinds of mutants as *strong* mutants. Therefore, the fitness of a strong mutant will always be equal or greater than $M \times T - 1$.

A non-equivalent mutant which remains alive shows our test suite is incomplete, so this mutant can be used to improve the quality of our test suite, adding a new test case that kills it. For this reason we favor potentially equivalent mutants. However, there is no automated way to know if a potentially equivalent mutant is really equivalent: the user will have to inspect it manually.

In addition, the fitness function is designed to penalize groups of mutants which are killed with the same test cases, regardless of the location, the mutation operator used or the number of mutants in the group. Their fitness will be lower than $M \times T - 1$. We consider them as *weak* mutants. These mutants neither provide new information about test cases nor can be used to improve the quality of the test suite, so they are redundant.

If most of the mutants of a mutation operator can be killed with the same test cases, these mutants will tend to have lower fitness. For this reason, after some of these mutants are generated, the fitness function will start to favor the other mutation operators. In a way, the fitness function penalizes operators which tend to produce weak mutants.

Each group of weak mutants will be penalized only after the first mutants in the group are generated. These first mutants will have a high fitness, as they are killed by a different set of test cases than the previously generated mutants. For this reason, they are more likely to be selected for mutation and crossover, allowing for quickly sampling the space of all nearby individuals. If these nearby individuals are killed by the same test cases, the fitness of the group will drop and the GA will focus on another group.

In contrast with other approaches, which try to avoid potentially equivalent mutants, the fitness function favors their selection, as a stubborn non-equivalent mutant points to a missing test case. Optionally, a test case generator could be integrated into the system, in order to improve the quality of the initial test suite. This test case generator would use the surviving strong mutants to extend the test suite with new test cases that kill them.

	T1	T2	T3	T4	T5	T6	Fitness (only Mi)	Fitness (all)
M1	1	0	1	0	0	0	18 - 2 = 16	30 - 5 = 25
M2	0	0	0	1	0	1	18 - 3 = 15	30 - 3 = 27
M3	0	1	0	1	1	0	18 - 4 = 14	30 - 6 = 24
S1	0	0	1	0	1	0		
S2	1	1	1	0	0	0		

Table 1: Fitness evaluation with and without the second population

3.3. Mutant Co-evolution

It is important that the above fitness function learns to produce better estimates from previous populations. For this reason, we use competitive co-evolution [21] with two populations: the first population changes across generations, and the second population stores all generated mutants. Each distinct mutant is only stored once in the second population, in a similar fashion to *halls of fame*. The fitness of a mutant is based on direct competition with the rest of the mutants in both populations.

By using this second population, the fitness of the mutants in a generation will depend on the mutants of the previous generations. In a way, this allows the algorithm to learn to generate better mutants.

As an example, consider Table 1, resulting from running 3 mutants of some composition (main population, Mi) against 6 test cases, having run 2 mutants in the previous generations (second population, Si). Using only the main population, M1 is the best mutant. However, using also the second population (Si), where other mutants died in T1 and T3 as well, M1 is not as good and the best mutant is now M2.

4. Evolutionary Mutation Testing for WS-BPEL Compositions: GAmera

In order to experiment with EMT, we have developed GAmera. GAmera is the first mutation generation system available for WS-BPEL compositions. We have chosen a GA as our evolutionary algorithm. This selection is motivated by the effectiveness of GAs in the resolution of optimization problems.

This section describes the mutation operators for the WS-BPEL 2.0 language used by GAmera and its three main components (Fig. 3): the analyzer, the mutant generator and the execution engine. The analyzer takes the original WS-BPEL process definition and generates the information required by the mutant generator. The execution engine runs the generated mutants and compares their outputs with those from the original process.

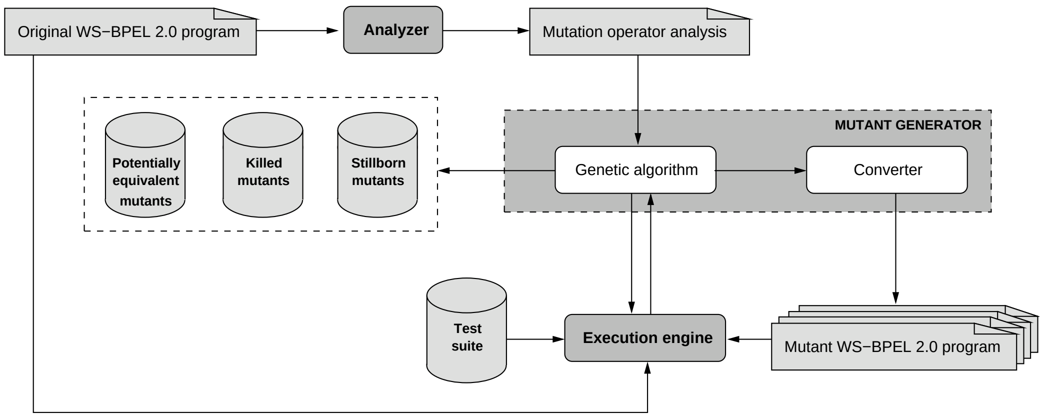

Original WS−BPEL 2.0 program
Analyzer
Mutation operator analysis
MUTANT GENERATOR
Potentially equivalent mutants
Killed mutants
Stillborn mutants
Genetic algorithm
Converter
Test suite
Execution engine
Mutant WS−BPEL 2.0 program

Figure 3: Automatic mutant generator system for WS-BPEL

GAmera can be used from the command line, or through a graphical user interface (GUI). The user can start the mutant generator and obtain the execution results of the generated mutants. Several indicators useful for measuring the quality of the test suite are produced: total number of generated mutants, killed mutants, surviving mutants and stillborn mutants. These values are also shown for each mutation operator.

This GUI allows for comparing the original program with a mutant. This is useful for checking whether an individual of the GA is an equivalent mutant. Differences between the original composition and a mutant are color-coded.

4.1. Mutation Operators for WS-BPEL

WS-BPEL process definitions are mutated by GAmera using the 26 operators defined in our previous work [14]. These operators have been classified in four categories depending on which kind of WS-BPEL syntactic element they are related to. These are identified by uppercase letters: I (*Identifier replacement operators*), E (*Expression operators*), A (*Activity operators*), and X (*eXception and event operators*).

Several mutation operators are defined within each category. Operators are uniquely identified by three uppercase letters: the first one is the category identifier, and the last two letters indicate the operator within the category. Table 2 lists their names and provides short descriptions for each of them.

These mutation operators model common mistakes that programmers can make when implementing a WS-BPEL composition. We have assumed that programmers usually do not write WS-BPEL code directly but with the help of graphical tools. Accordingly, many faults which are common in other languages when writing code by hand will not appear in WS-BPEL programs. For example, the unary negation operator is removed by the EEU operator, but there is no operator which adds the unary negation operator to an expression, as considered for other languages [22, 23, 24, 25]. This is because we assume that it is hard to add it accidentally with the graphical editors used in modern IDEs to create XPath expressions in WS-BPEL processes.

Some operators are specific for the WS-BPEL language, while others have been adapted from other languages, such as the operator ISV. The new operators specific to WS-BPEL appear marked with ☆ in Table 2.

4.2. WS-BPEL Analyzer

Before generating the mutants, it is necessary to identify the different locations of the original process definition that can be mutated. This is the role of the *analyzer*, which takes a WS-BPEL process definition and lists for each operator its name, the number of locations where it can be applied, and the number of different mutants which can be produced from each location.

As an example, we will consider a modified version of the *Loan Approval* composition [10], shown in Fig. 4. This composition receives from a client a request for a certain amount of money and aggregates results from an assessor and an approver WS. The assessor is invoked when the amount requested is less than or equal to 10000 monetary units. If the risk level of the client reported by the assessor is low the loan is granted. In any other case (i.e. the amount is greater than 10000 or the assessor reports a high risk level) the decision is left up to the approver WS (Fig. 4).

The analyzer output for this composition is shown in Fig. 5. This output shows that seven mutation operators can be applied in this composition: ERR, ECN, ASF, AEL, AIE, ASI and XMF. It shows also that the operator ERR can be applied to two relational expressions, ECN to one number constant, AIE to two `if` activities, AEL to nine activities, ASF and ASI to two `sequence` activities, and XMF to one fault handler. ERR and ECN can produce up to 4 different mutants from each location, and the rest can only produce one mutant per location.

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 `sequence` child activities
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 2: Mutation Operators for WS-BPEL 2.0

```
<process
 name="loanApprovalProcess"
 ...
 <partnerLinks>
  <partnerLink name="approver" ... />
  <partnerLink name="assessor" ... />
  <partnerLink name="customer" ... />
 </partnerLinks>
 <variables>
  <variable name="risk" ... />
  <variable name="approval" ... />
  <variable name="request" ... />
 </variables>
 <faultHandlers>
  <catch faultName="loanProcessFault" >
  <reply  faultName="unableToHandleRequest" ... />
  </catch>
 </faultHandlers>
 <sequence>
  <receive name="ReceiveRequest" ... />
  <if name="IfSmallAmount">
   <condition>
    ( $request.amount &lt;= 10000 )
   </condition>
    <sequence name="SmallAmount">
     <invoke name="AssessRiskOfSmallAmt" ... />
     <if name="IfLowRisk">
      <condition>
       ( $risk.level = 'low' )
      </condition>
      <assign name="ApproveLowRiskSmallAmtLoans">
      <copy>
       <from>true()</from>
       <to part="accept" variable="approval"/>
      </copy>
      </assign>
     <else>
      <invoke name="CheckApprover" ... />
     </else>
     </if>
    </sequence>
  <else>
   <invoke name="ApproveLargeAmt" ... />
  </else>
  </if>
  <reply name="ReportApproval" ... />
 </sequence>
</process>
```

Figure 4: The *Loan Approval* WS-BPEL composition

```
ERR 2 4
ECN 1 4
ASF 2 1
AEL 9 1
AIE 2 1
ASI 2 1
XMF 1 1
```

Figure 5: Analyzer output for *Loan Approval*

4.3. Mutant Generator

The *mutant generator* has two components: a GA, in which each individual represents a mutant, and a converter from individual to mutant. The purpose of the mutant generator is quickly generating strong mutants. The mutant generator receives the original WS-BPEL process definition, the output from the analyzer, and the GA configuration parameters. Its output consists of the set of generated mutants. After the execution of these mutants against the test suite, the GA classifies them in killed, potentially equivalent, and stillborn mutants.

4.3.1. The GA

This section describes several features of the GA which drives the overall mutant generation process.

Representation of an Individual. In Section 3.1 we explained that each mutant is encoded using three fields: *Operator*, *Location* and *Attribute*.

In our implementation *Operator* is encoded by an integer value from 1 to 26, the total number of mutation operators defined for WS-BPEL (as in Table 2).

Location is also encoded by an integer value, pointing to the *i*-th location where the selected operator is applicable.

Attribute is encoded by an integer which lies between 1 and the maximum value defined by the selected mutation operator. EAA, for instance, replaces the existing arithmetic operator from $\{+, -, \times, /, \texttt{mod}\}$ by one of the other 4 elements. For this reason, the maximum value of *Attribute* for its individuals is 4, so the operator cannot generate a mutant that is exactly the same as the original program. Suppose that the operator in the original program was `+`. *Attribute* would be interpreted as follows: if set to 1, EAA would use $-$ as replacement. Using 2 would result in $\times$, 3 in `mod` and 4 in $/$.

Operator	Value	Max.	Value of Attribute
ISV	1	N	
EAA	2	5	`+`, `−`, `*`, `div`, `mod`
EEU	3	1	
ERR	4	6	`<`, `>`, `>=`, `<=`, `=`, `!=`
ELL	5	2	`and`, `or`
ECC	6	2	`/`, `//`
ECN	7	4	+1, -1, adding, removing
EMD	8	2	0, half
EMF	9	2	0, half
ACI	10	1	
AFP	11	1	
ASF	12	1	
AIS	13	1	
AEL	14	1	
AIE	15	1	
AWR	16	1	
AJC	17	1	
ASI	18	1	
APM	19	1	
APA	20	1	
XMF	21	1	
XMC	22	1	
XMT	23	1	
XTF	24	N	
XER	25	1	
XEE	26	1	

Table 3: Values and attributes for the mutation operators

Table 3 shows for each operator the value to be placed in *Operator* and the maximum value accepted by *Attribute*. The maximum value for *Attribute* for ISV, XTF, and AEL depends on the original program and is computed by the analyzer. Other operators, like EEU, can only accept a single value in *Attribute*, as they can only produce a single mutant from a particular location. *Attribute* is not significant in these cases.

Figure 6 shows the mutant of the original *Loan Approval* WS-BPEL composition represented by the individual (15, 1, 1) in the GA, where the `else` element of the `if` activity named `IfLowRisk` has been deleted.

Normalized Representation of an Individual. The above representation has an important flaw: the range of the *Location* and *Attribute* fields changes depending on the actual value of the *Operator* field. This will pose a prob-

```
<process name="loanApprovalProcess" ...>
  <partnerLinks>
    <partnerLink name="approver" ... />
    <partnerLink name="assessor" ... />
    <partnerLink name="customer" ... />
  </partnerLinks>
  <variables>
    <variable name="risk" ... />
    <variable name="approval" ... />
    <variable name="request" ... />
  </variables>
  <faultHandlers>
    <catch faultName="loanProcessFault">
      <reply faultName="unableToHandleRequest" ... />
    </catch>
  </faultHandlers>
  <sequence>
    <receive name="ReceiveRequest" ... />
    <if name="IfSmallAmount">
      <condition>
        ( $request.amount &lt;= 10000 )
      </condition>
      <sequence name="SmallAmount">
        <invoke name="AssessRiskOfSmallAmt" ... />
        <if name="IfLowRisk">
          <condition>
            ( $risk.level = 'low' )
          </condition>
          <assign name="ApproveLowRiskSmallAmtLoans">
            <copy>
              <from>true()</from>
              <to part="accept" variable="approval"/>
            </copy>
          </assign>

          <!-- deleted <else> element -->

        </if>
      </sequence>
      <else>
        <invoke name="ApproveLargeAmt" ... />
      </else>
    </if>
    <reply name="ReportApproval" ... />
  </sequence>
</process>
```

Figure 6: Individual (15, 1, 1)

lem when applying some genetic operators. For this reason, we adopt a normalized representation with the following changes:

- The field *Location* is encoded by an integer value from 1 to L, where L is the least common multiple of the number of locations where each operator can be applied. For instance, assume only three WS-BPEL mutation operators could be applied to some composition. If the first operator could be applied to five different locations, the second one to three locations, and the third one to seven locations, the valid range of the Location field would be $[1, \mathrm{lcm}\{3, 5, 7\}] = [1, 3 \times 5 \times 7] = [1, 105]$.

- The field *Attribute* is similar to the field *Location*, and is encoded by an integer value from 1 to A, where A is the least common multiple of the number of different values it can take for each operator which can be applied to the original composition.

Before generating the mutant for an individual, the resulting *Location* L_i is denormalized to the $\lceil (L_i \times l_i)/L \rceil$-th location of the selected operator, where l_i is the number of locations for that operator in the original program. The *Attribute* field is denormalized in the same way.

For example, consider that we have two mutation operators, with 2 and 3 locations to which they can be applied, respectively. The maximum value of this field is $L = \mathrm{lcm}\{2, 3\} = 6$. If an individual that represents the first mutation operator has a value of 4 in the field *Location*, it represents a change in the location $\lceil (4 \times 2)/6 \rceil = 2$ where the mutation operator can be applied. Note that half of the cases, from 1 to 3, applies to the first location, and the other half, from 4 to 6, applies to the second location. Therefore, we can see that several individuals of the normalized representation may point to the same mutant.

Fitness of the Individuals. To compute the fitness of the individuals with the function (3), the GA will need to execute each mutant against the test suite. The execution environment is described in Section 4.4.

After executing the mutants we will produce the execution matrix (2). If a mutant has been executed in a previous generation, we avoid executing it again by reusing its row of the execution matrix.

Populations. We use competitive co-evolution, as described in Section 3.3. The first population has M individuals, where M has been set by the user. The second population stores all generated individuals.

Our algorithm, as most co-evolutionary algorithms, can present unexpected behaviors such as the Red Queen problem [26]. We have accepted this risk because we assume that a co-evolutionary approach will help the algorithm generate interesting individuals sooner, and detect in few generations if the individual represents a strong or a weak mutant.

Generations. The first population of the GA will be randomly generated. The generator is based on a generational GA where all generations following the first one consist of:

- Randomly generated individuals. The GA includes a configuration parameter (percentage of new individuals, N) which determines how many mutants will be randomly generated.

- Individuals produced from the crossover and mutation operations. The selection of the individuals involved is done through the roulette wheel method [12]. The crossover operator will be applied with the probability p_c, where p_c is a configuration parameter of the algorithm. Otherwise, the mutation operator will be applied. For this reason, the mutation operator will be applied with the probability $p_m = 1 - p_c$.

  At first sight, the roulette wheel selection may seem a poor selection mechanism because of its quick convergence. However, the algorithm actually takes advantage of this feature, since we are interested in generating as few mutants as possible to obtain the set of strong mutants. The algorithm escapes local maxima by using the second population and the fitness function to discern among similar individuals.

Genetic Operators. In order to generate new individuals, the GA can apply two genetic operators to the individuals of the previous generation. The probability of an individual being selected for applying a genetic operator is proportional to its fitness.

The crossover genetic operator exchanges fields between two individuals. The crossover point is chosen at random between the fields of the individuals. Fig. 7 shows the different possibilities for a cross between two parent individuals to generate two children, according to the selected crossover point.

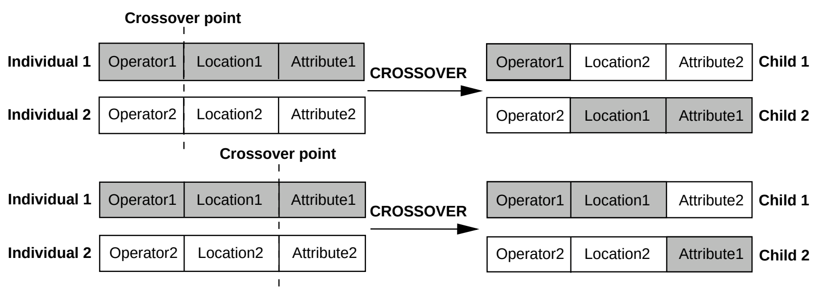


Figure 7: Crossover operator

The crossover genetic operator generates valid individuals thanks to the normalized representation described in Section 4.3.1.

The mutation genetic operator changes the value of a field of an individual. Thus, there are three mutation operators, depending on the field they modify. Since the three fields are encoded as integers, mutating them consists of adding a random integer in the range $[1, 10(1 - p_m))$ to the value of the selected individual. The maximum range for this mutation will decrease as p_m increases, reducing its impact. The operation is carried out modulo U, where U is the maximum value that the field can take. Thus, if the field to be mutated contains α, the final value β will be:

$$\beta = \alpha + random(1, 10(1 - p_m)) \pmod{U}. \tag{4}$$

The mutation genetic operator is designed to avoid generating invalid individuals, by using modular arithmetic over the valid range.

Termination. The GA stops when the size of the second population reaches the percentage of all mutants to be generated set by the user. This is a configuration parameter of the GA, P. The output of the GA will be all the individuals in the second population, divided into weak and strong subsets.

4.3.2. Individual to Mutant Converter

This component receives the three fields that each individual is encoded with and produces the mutant which will be later evaluated. To do this, it uses the *Operator* in the chosen individual to select the XSLT [27] stylesheet which will implement the transformation required. These stylesheets obviously do not need the *Operator* anymore, so they only receive the *Location* and the *Attribute*.

The stylesheets use the field *Location* to obtain a reference to the node to be transformed. The exact transformation performed depends on the field *Attribute*.

Special care has been taken in the implementation of the converter not to produce mutants that are identical to the original program. For instance, it is impossible, by design, to replace the relational operator > with itself while applying the ERR operator.

4.4. Execution Engine

Fig. 8 shows the different steps of the execution engine. The general process is divided into three main steps, which we detail below. More details about the execution engine used by GAmera are described by Palomo [28].

Packaging First, we take the mutant and prepare it so we can deploy it into an WS-BPEL engine. The process definition itself is not changed: we merely analyze it to produce the engine-specific deployment files required, which will be later sent to the deployment WS of the WS-BPEL engine.

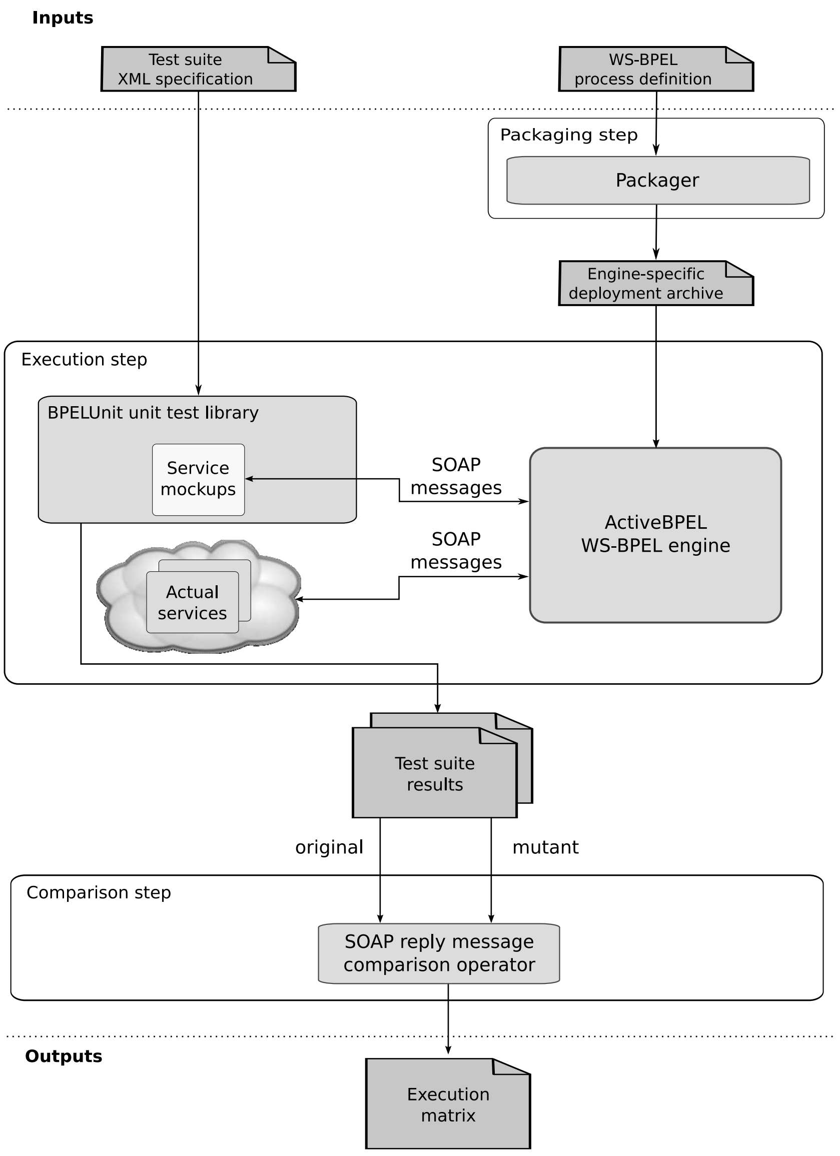
Execution Once we have deployed the composition in the WS-BPEL engine, we can run it against the test suite, collect its outputs and undeploy it.

Comparison We need to compare the collected outputs with those from the original process to decide if the mutant is killed or stays alive. The execution matrix (2) that allows us to calculate the fitness of the individuals is generated from these comparisons.

We perform a strict one-to-one comparison of their outgoing SOAP messages. These messages do not need to indicate success: if the original WS-BPEL process failed, the mutant will need to fail in the exact same way to survive. Likewise, if the WS-BPEL process was successful, the mutant will have to complete its execution successfully with the exact same results to stay alive.

5. Experimental Evaluation of EMT

In this section we describe the three WS-BPEL compositions used to evaluate EMT, the method used to estimate the optimal values for the parameters of the GA, the method used for the experimental evaluation of EMT and the evaluation results obtained by applying this method.



Inputs
Test suite XML specification
WS-BPEL process definition
Packaging step
Packager
Engine-specific deployment archive
Execution step
BPELUnit unit test library
Service mockups
SOAP messages
SOAP messages
ActiveBPEL WS-BPEL engine
Actual services
Test suite results
original
mutant
Comparison step
SOAP reply message comparison operator
Outputs
Execution matrix

Figure 8: The execution engine

All the experiments in this section have been run in an Intel Core i7 920 machine with DDR2 12GB RAM with a 400MHz FSB, running Linux 2.6.28 and OpenJDK 1.6 update 14 in an Ubuntu Server 9.04 GNU/Linux distribution.

5.1. WS-BPEL Compositions Used

We have used three WS-BPEL compositions to experimentally evaluate EMT. These compositions are: *Travel Reservation Service Extended*, *Meta Search* and *Loan Approval Extended*.

Travel Reservation Service Extended Our *Travel Reservation Service Extended* (TRSE) composition is a modified and extended version of a sample business process included in the NetBeans SOA Pack 6.1 [29]. The Travel Reservation Service acts as a logical aggregator of other services and is based on business logic typical of travel reservation systems. For this composition 13 of the 26 mutation operators are applicable and we have used a test suite of 10 test cases.

The exhaustive execution of its 210 mutants against the 10 test cases (not applying EMT) takes 2.55 hours. With this test suite, this composition presents 68 strong mutants.

Meta Search The *Meta Search* (MS) sample composition bundled with the BPELUnit [30] unit testing framework for WS-BPEL implements a meta-search engine which aggregates results from multiple Internet search engines. In particular, results from Google and MSN are interleaved, while removing duplicates. For this composition 12 of the 26 mutation operators are applicable and we have used a test suite of 9 test cases.

The exhaustive execution of its 529 mutants against the 9 test cases (not applying EMT) takes 3.35 hours. With this test suite, this composition presents 101 strong mutants.

Loan Approval Extended Our *Loan Approval Extended* (LAE) composition is a modified and extended version of the well-known sample composition included in the WS-BPEL [10] standard. For this composition 19 of the 26 mutation operators are applicable and we have used a test suite of 21 test cases.

	LOC	|NM|	|NS|	|T|	GT (h)	RT (h)
TRSE	363	210	68	10	0.04	2.51
MS	597	529	101	9	0.14	3.21
LAE	1520	3661	1362	21	1.74	94.85

Table 4: Metrics from the WS-BPEL compositions used

The exhaustive execution of its 3661 mutants against the 21 test cases (not applying EMT) takes 96.59 hours. With this test suite, this composition presents 1362 strong mutants.

The test cases used for the three compositions above have been manually written, selecting partitions from the input space of the compositions in order to maximize their branch coverage.

Table 4 collects the above information about number of mutants (|NM|), number of strong mutants (|NS|) and number of test cases (|T|). In addition, it shows the number of lines of code of each composition (LOC) after placing each XML tag in a single line and the total time required to generate (GT) and run (RT) all their mutants.

It is important to note that WS-BPEL compositions are much more concise than traditional programs, as a single line may contain multiple attributes which may dramatically change its behavior, or a complex XPath expression which may contain many operands for the WS-BPEL mutation operators. Therefore, we believe lines of code do not adequately represent the complexity of a WS-BPEL composition.

We assume the high running times are due to the client-server communication used to invoke WS-BPEL compositions. The WS-BPEL composition may not reply under the specified time limit in two cases: when execution is trapped in an endless loop (such as in other languages) and also when execution is aborted due to an internal error. The second case is not a problem for traditional languages whose mutants are directly executed by the testing tool: the tool will simply continue with the next mutant. However, WS-BPEL engines might not send any message in that case, forcing GAmera to wait until the time limit is reached.

5.2. Estimation of Optimal Values for Configuration Parameters

The GA depends on several configuration parameters, such as:

- M, population size.
- N, percentage of new individuals to be randomly generated in each generation.
- P, percentage of all mutants to be generated.
- p_c, crossover probability.

The results produced by the GA also depend on the seed used to initialize the pseudo-random number generator. Randomness is used in several important aspects: selecting the individuals for the initial population and subsequent generations, generating new individuals and applying the genetic operators. For this reason, it is important to evaluate the GA by obtaining samples from several independent runs, i.e. with different seeds.

We have carried out several experiments over the three compositions shown in Section 5.1 to determine the values for the configuration parameters producing the best results. In order to obtain the optimal values for p_c, N and M, we have adopted the following method:

1. Generate all the mutants of the composition.
2. Run all mutants sequentially against every test case. Partition them automatically into stillborn, strong and weak mutants.
3. For each combination of parameters, execute thirty independent runs of the GA.
4. Estimate the optimal values for the configuration parameters of the GA by comparing the total times needed to obtain all the strong mutants.

Table 5 shows the values used for the configuration parameters. In this table, M represents the population size as a percentage of the mutants generated from each composition. P is omitted since a special termination condition applies. When determining the optimal parameters we need to generate all the strong mutants, which cannot be ensured by selecting just a subset of all the mutants.

Tables 6, 7 and 8 present statistics of 30 different samples for each combination of parameters. Table 6 shows the results obtained for all combinations with $M = 5\%$ for the three compositions: TRSE, MS, and LAE. Similarly, Tables 7 and 8 show the results for $M = 10\%$ and $M = 20\%$. For each combination of parameters, the minimum time has been computed ("Min."),

Parameter	Values			
M	5%	10%	20%	
N	10%	20%	30%	
p_c	0.6	0.7	0.8	0.9

Table 5: Candidate values for each parameter

along with the average ("Avg."), median, maximum ("Max."), and the coefficient of variation ("CV"). Please note that the coefficient of variation is expressed as a percentage.

A first glance at these tables reveals very low values for the coefficient of variation across different samples for all parameter combinations.

Looking more closely at the results at Tables 6, 7 and 8, we can see that the times rise slightly as p_c increases. The same trend can be observed for N and M. As expected, the time increase is larger in the most complex composition, LAE, than in the others.

The TRSE composition does not provide useful information for selecting the optimal parameters: the best statistics are widely spread over the various combinations of parameters. This is because the composition is simple and the resulting search space is too small for EMT. Nevertheless, $p_c = 0.6$ still produces the best overall result for TRSE: 2.18 hours.

As the results for $p_c = 0.6$ and $p_c = 0.7$ are similar, we have calculated for each option how many mutants need to be generated to produce certain percentages of all strong mutants (SM). The results, listed in Tables 9 and 10, show that the best value for p_c is 0.7 because it obtains the best statistics (percentage of generated mutants) for the two most complex compositions, MS and LAE.

We will now estimate a recommended value of P which produces as many strong mutants as possible while generating a reduced subset of all mutants. Formally, we are looking for a value of P which maximizes the ratio SM/P, where SM is the percentage of all strong mutants found when generating P percent of all mutants. Figure 9 shows the average ratios obtained over 30 independent runs of EMT with each composition, using the previously estimated parameters: $p_c = 0.7$, $N = 10\%$ and $M = 5\%$. P ranged from 10% to 80%, with steps of 5%.

The maximum average ratio was obtained using $P = 15\%$ for the two largest compositions (MS and LAE) and taking the results of all the compo-

		$p_c = 0.6$			$p_c = 0.7$			$p_c = 0.8$			$p_c = 0.9$		
	N :	10 %	20 %	30 %	10 %	20 %	30 %	10 %	20 %	30 %	10 %	20 %	30 %
TRSE	Min.	*2.18*	2.27	2.25	2.24	2.28	2.24	2.34	2.30	2.36	2.39	2.26	2.21
	Avg.	2.51	2.48	2.49	2.50	2.50	*2.47*	2.52	*2.47*	2.50	2.50	*2.47*	*2.47*
	Median	2.54	*2.49*	2.53	2.54	2.53	*2.49*	2.54	2.50	2.52	2.51	2.50	2.50
	Max.	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56
	CV (%)	3.31	3.25	3.42	3.46	2.74	3.62	2.31	3.33	2.14	*2.00*	3.98	3.79
MS	Min.	2.95	2.80	2.98	2.91	3.07	2.94	2.96	3.09	3.05	*2.74*	3.03	3.16
	Avg.	3.22	3.23	3.23	*3.20*	3.27	3.26	3.21	3.25	3.23	3.24	3.28	3.29
	Median	*3.23*	3.26	3.25	3.24	3.29	3.29	3.24	3.25	3.24	3.27	3.29	3.30
	Max.	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.35	3.34
	CV (%)	2.89	3.53	3.17	3.75	2.21	2.52	3.87	2.30	2.50	3.89	2.26	*1.67*
LAE	Min.	*74.81*	75.13	75.32	75.35	75.15	75.35	74.82	74.97	75.32	75.15	75.13	75.63
	Avg.	*75.58*	75.82	76.02	75.68	75.80	76.02	75.74	75.91	75.99	75.93	76.14	76.23
	Median	*75.60*	75.90	76.10	75.66	75.73	76.02	75.77	75.96	76.05	75.92	76.16	76.17
	Max.	*76.09*	76.61	76.73	76.23	76.56	76.75	76.40	76.94	76.53	76.75	77.13	77.11
	CV (%)	0.43	0.49	0.49	*0.30*	0.44	0.46	0.48	0.59	0.40	0.51	0.55	0.49

Table 6: Statistics on GAmera execution times (hours, with minima emphasized) for different values of N and p_c ($M = 5\,\%$)

		$p_c = 0.6$			$p_c = 0.7$			$p_c = 0.8$			$p_c = 0.9$		
	N :	10 %	20 %	30 %	10 %	20 %	30 %	10 %	20 %	30 %	10 %	20 %	30 %
TRSE	Min.	2.25	2.21	*2.18*	2.36	2.28	2.21	2.33	2.32	2.21	2.32	2.33	2.29
	Avg.	2.50	*2.46*	*2.46*	2.51	2.48	2.49	2.51	2.47	2.48	2.49	2.49	2.49
	Median	2.52	*2.48*	2.51	2.53	2.49	2.52	2.53	2.50	2.52	2.52	2.51	2.51
	Max.	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56
	CV (%)	3.24	4.05	4.80	*2.16*	3.16	3.63	2.30	2.99	3.82	2.90	2.51	2.77
MS	Min.	2.97	3.08	2.94	2.83	2.85	3.05	*2.78*	2.99	2.96	3.06	3.06	3.08
	Avg.	*3.20*	3.24	3.23	*3.20*	3.25	3.24	3.22	3.25	3.26	3.27	3.27	3.30
	Median	*3.22*	3.27	3.28	3.25	3.29	3.25	3.27	3.27	3.29	3.28	3.29	3.31
	Max.	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34
	CV (%)	3.72	2.73	3.28	3.97	3.02	2.46	3.98	2.66	2.80	2.07	2.24	*1.84*
LAE	Min.	74.89	75.10	75.14	74.90	*74.81*	75.55	75.29	75.55	75.40	75.45	75.17	75.44
	Avg.	*75.64*	75.78	75.88	75.70	75.84	75.97	75.84	76.10	76.10	76.04	76.13	76.25
	Median	*75.64*	75.79	75.83	75.75	75.95	75.90	75.88	76.05	76.13	76.01	76.13	76.24
	Max.	76.23	*76.21*	76.57	76.35	77.06	76.59	76.64	77.34	76.79	76.95	76.85	77.26
	CV (%)	0.42	*0.34*	0.52	0.46	0.68	0.38	0.43	0.50	0.45	0.48	0.50	0.47

Table 7: Statistics on GAmera execution times (hours, with minima emphasized) for different values of N and p_c ($M = 10\,\%$)

		$p_c = 0.6$			$p_c = 0.7$			$p_c = 0.8$			$p_c = 0.9$		
	N :	10 %	20 %	30 %	10 %	20 %	30 %	10 %	20 %	30 %	10 %	20 %	30 %
TRSE	Min.	2.27	2.24	2.30	*2.20*	2.26	2.31	2.26	2.29	2.25	2.31	2.35	2.27
	Avg.	2.47	2.46	2.48	2.46	2.49	2.48	2.47	2.49	2.46	2.46	2.49	*2.45*
	Median	2.49	2.51	2.51	2.50	2.52	2.50	2.50	2.51	2.49	2.49	2.49	*2.46*
	Max.	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56	2.56
	CV (%)	3.57	4.14	3.30	4.09	3.29	3.24	3.45	3.10	3.31	3.49	*2.55*	3.38
MS	Min.	2.93	2.91	3.10	*2.89*	3.08	3.17	3.03	3.02	3.16	3.09	3.21	3.06
	Avg.	3.23	3.23	3.27	*3.19*	3.25	3.29	3.24	3.24	3.26	3.26	3.30	3.29
	Median	3.24	3.28	3.26	*3.20*	3.26	3.30	3.25	3.27	3.27	3.28	3.30	3.30
	Max.	3.34	3.34	3.34	*3.33*	3.34	3.34	3.34	3.34	3.34	3.34	3.34	3.34
	CV (%)	2.81	3.65	1.90	3.40	2.21	1.46	3.00	2.50	1.70	2.18	*1.24*	1.79
LAE	Min.	75.30	75.25	75.64	*74.76*	74.95	75.16	75.02	75.28	75.48	75.35	74.97	75.59
	Avg.	*75.78*	*75.78*	76.14	75.80	75.89	76.18	75.98	76.13	76.20	76.13	76.16	76.39
	Median	75.84	*75.75*	76.02	75.82	75.81	76.25	76.02	76.13	76.22	76.08	76.15	76.44
	Max.	*76.27*	76.30	76.91	76.33	76.78	76.72	76.54	77.02	77.02	77.15	76.93	77.01
	CV (%)	*0.34*	*0.34*	0.43	0.50	0.61	0.53	0.46	0.53	0.47	0.58	0.56	0.55

Table 8: Statistics on GAmera execution times (hours, with minima emphasized) for different values of N and p_c ($M = 20\,\%$)

		$SM = 75\,\%$	$SM = 90\,\%$
TRSE	Min.	56.67	71.90
	Avg.	64.10	79.17
	Median	64.52	79.05
	Max.	69.52	84.76
	CV (%)	4.45	3.60
MS	Min.	52.74	71.64
	Avg.	60.33	79.36
	Median	59.74	79.21
	Max.	70.32	84.69
	CV (%)	6.50	4.72
LAE	Min.	60.67	77.52
	Avg.	62.58	78.86
	Median	62.61	78.93
	Max.	63.89	79.73
	CV (%)	1.29	0.69

Table 9: Percentage of generated mutants for $p_c = 0.6$, $M = 5\,\%$, and $N = 10\,\%$

		$SM = 75\,\%$	$SM = 90\,\%$
TRSE	Min.	60.00	75.24
	Avg.	65.43	80.62
	Median	65.24	80.48
	Max.	71.43	86.19
	CV (%)	3.54	3.46
MS	Min.	51.80	68.62
	Avg.	59.25	78.88
	Median	59.36	79.11
	Max.	68.43	84.88
	CV (%)	6.60	4.48
LAE	Min.	60.53	77.19
	Avg.	62.07	78.57
	Median	61.90	78.63
	Max.	64.22	79.60
	CV (%)	1.20	0.67

Table 10: Percentage of generated mutants for $p_c = 0.7$, $M = 5\,\%$, and $N = 10\,\%$

sitions together. Based on these results, we recommend using $P = 15\%$.

Table 11 shows the selected optimal configuration parameters for the GA.

	p_c	N	M	P
VALUE	0.7	10%	5%	15%

Table 11: Optimal parameters for the GA

5.3. EMT versus Random Selection

In order to evaluate the effectiveness of EMT, we will compare it against random selection of the individuals of the GA. Our version of random selection generates and executes them in random order against all test cases, stopping when reaching a certain percentage of all mutants or all strong mutants. We have purposefully avoided the better known term *mutant sampling* as this is usually understood to stop at the first test case that kills a mutant. This would not work in our case: we need to run all test cases to find the strong mutants.

Specifically, we will compare:

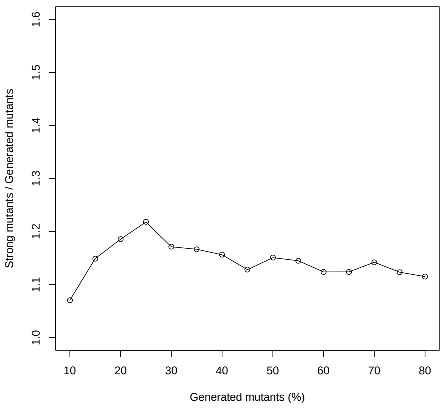
1. Required times and generated mutants to obtain all the strong mutants.
2. Strong mutants obtained when generating different percentages of all mutants (i.e. using different values of P).

5.3.1. Exhaustive Generation of Strong Mutants

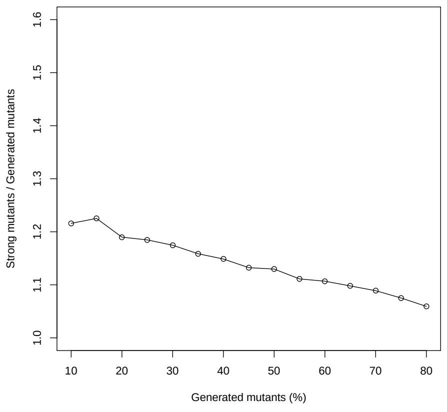
Table 12 compares the times required to generate all strong mutants for random selection and EMT. Times are only been slightly reduced for TRSE and MS, but LAE shows a noticeable decrease. We can see that EMT is only slightly faster than random selection for finding all the strong mutants in small compositions, but scales better as compositions become more complex.

With respect to the number of generated mutants, EMT generates less mutants on average for all three compositions. Again, there are only minor savings for the smaller compositions (TRSE and MS), but EMT reduces the number of mutants generated with LAE in over 16% percent on average. EMT results show little variance, with a coefficient of variation of only 0.06% for LAE.

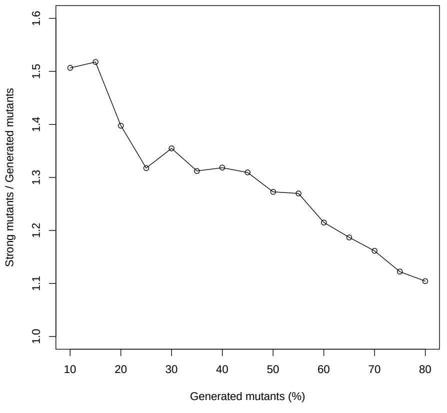
Table 13 shows generation (GT), execution (RT), GA (GAT) and total times (TT) in hours for generating all strong mutants using EMT for the



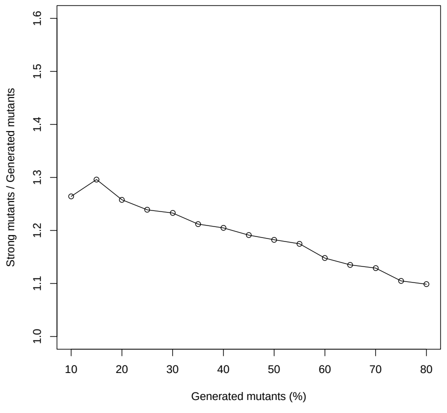
(a) TRSE



(b) LAE



(c) MS



(d) All

Figure 9: Proportion of strong mutants among generated mutants for differents values of P on TRSE, LAE, MS, and all compositions

		Random		EMT	
		T	G (%)	T	G (%)
TRSE	Min.	2.42	95.71	2.23	88.10
	Avg.	2.52	98.94	2.49	97.62
	Median	2.53	99.05	2.53	99.05
	Max.	2.55	100.00	2.55	100.00
	CV (%)	0.89	0.70	2.66	2.34
MS	Min.	3.16	96.60	2.90	90.93
	Avg.	3.32	99.38	3.20	96.48
	Median	3.34	99.62	3.23	97.16
	Max.	3.35	100.00	3.34	98.87
	CV (%)	0.93	0.49	2.90	1.53
LAE	Min.	96.21	99.73	75.35	83.39
	Avg.	96.52	99.96	75.67	83.53
	Median	96.58	99.97	75.65	83.56
	Max.	96.59	100.00	76.22	83.58
	CV (%)	0.07	0.04	0.24	0.06

Table 12: Statistics on running times (T, in hours) and percentages of generated mutants (G) for generating all strong mutants with random selection and EMT

three compositions studied. These are the mean values obtained for the 30 samples with the optimal values of the configuration parameters. From this table, we can conclude that mutant execution is by far the most expensive task. It is important to note that the GA itself is by far the most inexpensive task, as shown by the values of GAT.

5.3.2. Partial Generation of Mutants

In the previous section, we compared EMT and random selection when generating all strong mutants. However, it must be noted that most of the time, a single test case will kill several mutants, so not all strong mutants are needed to find the missing test cases. Therefore, the user will not need to produce all strong mutants, but only a subset large enough to provide information on how to improve the test suite. Once a few test cases have been added, EMT will be usually run again to find the remaining strong mutants. Therefore, each execution of EMT should find as many strong

	GT	RT	GAT	TT
TRSE	0.04	2.45	<0.01	2.49
MS	0.14	3.06	<0.01	3.20
LAE	1.40	74.26	0.01	75.67

Table 13: Average required times for generating all strong mutants in EMT (in hours)

mutants as possible in P percent of all mutants, where P is set by the user.

In order to evaluate the effectiveness of EMT in this case, we have compared the percentages of all the strong mutants generated by EMT and random selection with different values of P. We need to verify whether these differences are statistically significant or not. For this reason, we conducted Mann-Whitney U tests on the percentages of strong mutants generated by each technique.

The null hypothesis for this test was that the median percentages of strong mutants obtained by EMT and random selection were equal. The alternative hypothesis was that the median percentages of strong mutants obtained by random selection were less than the median percentages obtained by EMT. Results for the U statistic and the resulting p-values are collected in Table 14 for each composition (from 30 samples) and for all compositions (collecting all 90 samples).

The null hypothesis was rejected in all cases, except for $P = 10\%$ and $P = 15\%$ with TRSE, the simplest composition. Therefore, we can conclude that EMT generates a statistically significant (within a 99.9% confidence interval) greater percentage of strong mutants for these three compositions.

5.4. Threats to Validity

Results in this paper suggest that EMT provides savings in cost over random selection for finding strong mutants. However, the validity of these results is subject to several limitations of this study.

Only three WS-BPEL compositions have been considered: TRSE, MS, and LAE. We would need to test EMT with more compositions. However, most WS-BPEL compositions are developed in-house for internal use, as they model the business processes of the organizations. For this reason, there are few examples of WS-BPEL compositions, and the existing examples tend to be toy compositions instead of industrial cases. Ideally, a battery of

	TRSE		MS		LAE		All	
P	U	p-value	U	p-value	U	p-value	U	p-value
10%	530.5	0.1162	799	<0.01	899.5	<0.01	6366	<0.01
15%	516	0.164	881	<0.01	900	<0.01	6845	<0.01
20%	624	<0.01	855	<0.01	900	<0.01	7132	<0.01
25%	685	<0.01	818	<0.01	900	<0.01	7130	<0.01
30%	727.5	<0.01	886.5	<0.01	900	<0.01	7586	<0.01
35%	704.5	<0.01	868.5	<0.01	900	<0.01	7384	<0.01
40%	743.5	<0.01	895	<0.01	900	<0.01	7629.5	<0.01
45%	809.5	<0.01	892.5	<0.01	900	<0.01	7668	<0.01
50%	871.5	<0.01	899	<0.01	900	<0.01	7910.5	<0.01
55%	876.5	<0.01	899.5	<0.01	900	<0.01	8003	<0.01
60%	851.5	<0.01	896.5	<0.01	900	<0.01	7919	<0.01
65%	859.5	<0.01	900	<0.01	900	<0.01	8003.5	<0.01
70%	884.5	<0.01	898.5	<0.01	900	<0.01	8022	<0.01
75%	866	<0.01	878	<0.01	900	<0.01	7820	<0.01
80%	875	<0.01	884.5	<0.01	900	<0.01	7866.5	<0.01

Table 14: Mann-Whitney U tests, with alternative hypothesis “median percentage of strong mutants generated with EMT is greater than with random selection”

test programs and test cases such as the Siemens suite for C [31] should be developed.

EMT also depends on the test suite used: with different test cases, the same mutant might be classified as weak or strong. In this study we have used a single manually-designed test suite for each composition. More experiments should be performed with each composition, using several test suites with varying coverage.

We designed the fitness function to penalize groups of mutants which are killed by the same test cases, regardless of location or mutation operator. A side effect of this design is that mutation operators for which almost all mutants can be killed by the same test case are penalized after the first mutants from that operator are generated. In a way, the GA penalizes operators which tend to produce weak mutants. However, this does not mean that these mutation operators will be discarded. They can still be used by crossing over a mutant from one of those mutation operators with any other mutant, by random generation or by mutating another individual. Nevertheless, further analysis is required on the impact of this side effect.

6. Related Work

Research in mutation testing can be classified into four types of activities [32]:

1. Defining mutation operators.
2. Developing mutation systems.
3. Inventing ways to reduce the cost of mutation analysis.
4. Experimentation with mutation.

This paper focuses on the second and third activities. We present a new technique to reduce the computational cost of finding a strong mutant subset, EMT, and we develop an automatic mutant generation system based on it for WS-BPEL. This section summarizes related work about techniques to reduce the computational cost of mutation testing, mutant generation systems and usage of GAs for testing.

6.1. Cost Reduction Techniques

There are several strategies to reduce the high computational cost of mutation testing [33]:

Mutant Reduction Techniques These techniques reduce the number of generated mutants. Several techniques have been described in the literature:

- *Mutant sampling* [5, 6], where only a randomly selected subset of the mutants are run.
- *Selective mutation* [8], which reduces the number of mutants generated by applying only a subset of the mutation operators defined for the language.
- *Mutant clustering* [7], where the mutants are clustered together which are largely killed by the same set of test cases. Only one mutant is selected from each cluster.
- *High order mutation* [9], which finds those rare but valuable higher-order mutants that model subtle faults. The mutants are classified into first-order mutants (FOMs) and higher-order mutants (HOMs). HOMs are created by applying more than one mutation operator in sequence to the original program. This technique introduces the concept of a *strongly subsuming HOM* which is only killed by a subset of the intersection of test cases that kill each FOM from which it is created. The HOM would be selected over the FOMs that it subsumes.

Execution Cost Reduction Techniques These techniques optimize the mutant execution process:

- *Weak mutation* [34], which compares the internal states of the mutant and the original program immediately after the execution of the mutated portion of the program.
- *Firm mutation* [35], which compares the intermediate states after the execution of the mutated portion of the program, and the final output.
- *Runtime optimization techniques*, which generate and run each mutant as quickly as possible. Compiler-integrated program mutation [36] uses an instrumented compiler designed to generate and compile mutants. Mutant schema generation [37] encodes all mutations into a single source-level program, named *metamutant.* Bytecode translation [38] generates mutants from the compiled version of the original program, rather than from its source code.

- *Advanced platform support for mutation testing*, which distribute the computational cost over several machines, like vector processors [39], SIMD machines [40], hypercube machines [41, 42] and network computers [43].

EMT attempts to address some of the shortcomings of the existing first-order mutant reduction techniques, presenting some advantages over these techniques.

EMT only generates and executes a subset of all the mutants that can be produced, selecting them with an evolutionary algorithm. Mutant clustering needs to execute all the mutants before reducing their number.

EMT uses all the mutation operators that can be applied to the program, in contrast with selective mutation, in which some critical mutation operators may be inadvertently ignored.

Random mutant sampling is not very efficient with a large number of mutants and may miss some critical mutants altogether [44]. Our experiments (Section 5.3) show that EMT is consistently better than random selection (a version of random mutant sampling which runs all test cases) at finding the strong mutants, thanks to the directed search performed by its GA.

6.2. Mutation Testing Systems

In order to apply EMT, we have developed GAmera, a new mutant generation system based on it. This mutant generation system operates over WS-BPEL process definitions. Several mutant generation systems for various programming languages already exist:

- Mothra [23] for FORTRAN. It is likely the most widely known mutation testing system.
- MuJava [38] for Java.
- Proteum [45] for C. It implements all mutation operators designed for the ANSI C programming language.
- MiLu [46] for C. It can perform both first-order and higher-order mutation testing.
- SQLMutation [25] for database queries written in SQL.

Some of the above systems use strategies to reduce the computational cost of mutation testing. MuJava uses two execution cost reduction techniques: mutant schema generation and bytecode translation. MiLu uses high order mutation.

6.3. Genetic Algorithms and Testing

Our system, GAmera, uses EMT for mutant reduction. We have implemented this technique using a GA. This implementation proposes a novel use of GAs to mutant generation within the scope of software testing.

The application of GAs to software testing goes back to Xanthakis [47]. Since then, numerous applications have arisen, although most of them have been limited to test case generation [48, 49, 50, 51].

Only Adamopoulos et al. [44] propose a methodology for mutation testing which integrates a GA. This methodology first generates all the mutants, and then uses the GA to select difficult to kill mutants for the initial test suite, discarding potentially equivalent mutants. Instead, our system generates only a strong subset of mutants, which includes difficult to kill and potentially equivalent mutants. Potentially equivalent mutants include stubborn non-equivalent mutants and equivalent mutants. Stubborn non-equivalent mutants are useful for obtaining new test cases to enhance the quality of the initial test suite. However, our technique has a limitation: it also favors equivalent mutants. Equivalent mutants are not interesting, but our technique cannot distinguish them from stubborn non-equivalent mutants.

7. Conclusions and Future Work

We have presented a new mutant reduction technique, Evolutionary Mutation Testing (EMT). This technique is based on evolutionary algorithms and its main purpose is coping with the high computational cost of mutation testing by reducing the number of mutants.

The most significant feature of EMT is the integration of mutant generation and mutant selection to eliminate uninteresting mutants on the fly. Thus, there is no need to generate every mutant, but only a suitable subset of all the possible mutants. The whole process is guided by a specially-tailored fitness function. Our technique reduces the number of mutants while preserving testing effectiveness by favoring potentially equivalent mutants (i.e., both equivalent and stubborn non-equivalent mutants) and difficult to kill mutants, from which new test cases that improve the quality of the initial test suite can be derived. We call mutants meeting these criteria strong mutants. Additionally, the fitness function penalizes groups of mutants killed by the same test cases. These mutants are redundant in this context and we consider them weak mutants.

There are important differences with respect to other first-order mutant reduction techniques. Unlike selective mutation, EMT can use all the mutation operators, without discarding any of them a priori. Unlike clustering mutation, it only executes the subset of generated mutants. Unlike mutant sampling, EMT performs a directed search over the space of all mutants.

In order to evaluate EMT we have developed GAmera, a mutation testing system powered by a co-evolutive GA. To estimate the benefits of this new technique, we have applied it to the WS-BPEL language. The selection of WS-BPEL is not arbitrary: Web Services and their WS-BPEL compositions are becoming commonplace and bring new challenges to mutation testing. WS-BPEL compositions can require more resources than regular programs, so having a tool which implements EMT to reduce the number of mutants for WS-BPEL is important.

We have applied EMT with GAmera to three WS-BPEL compositions: *Travel Reservation Service Extended*, *Meta Search* and *Loan Approval Extended*. We have selected the optimal values for the parameters of EMT in two stages. First, we have determined all parameters except P by stopping GAmera when all strong mutants have been generated, looking for the parameters which provide the largest savings in time and number of mutants generated. Next, we have selected the value of P which finds a larger proportion of strong mutants, in relation to the number of mutants generated.

We have compared EMT against random selection when generating all the strong mutants and generating a percentage of all mutants. Results show that EMT is more effective than random selection at selecting all the strong mutants, especially with complex compositions. In addition, EMT has found more strong mutants when generating a certain percentage of all mutants (P), for values of P between 10% and 80%, in steps of 5%. We have confirmed these results within 99.9% confidence intervals.

EMT currently requires that all test cases are run against every mutant. This could be solved if the fitness function were revised to allow EMT to stop at the first test case with a different output. However, this would make EMT depend on the order in which the test cases are run. This issue could be overcome by defining a prioritization method on the test cases. The fitness function penalizes mutation operators which tend to produce weak mutants:

further studies on the impact of this feature are required.

We are interested in comparing EMT with other mutant reduction techniques. It would be also interesting to apply EMT to widely used programming languages, such as C or Java, and to adapt it for generating higher order mutants.

We plan to design and integrate an automatic test case generator with EMT. This generator will use the surviving strong mutants to obtain new test cases that enhance the quality of the initial test suite, telling apart the equivalent and stubborn non-equivalent mutants.

This paper has considered all strong mutants to be equal. We intend to perform a more in-depth study on their distribution among hard to reach, reached but hard to kill, or equivalent mutants. Likewise, this work has considered the GA as a black box: further research is required on how the strong mutants are distributed over its search space. We will also study EMT with more compositions.

Finally, we will carry an in-depth analysis of the degree in which EMT depends on the initial test suite, and how it can affect the optimal values for the GA parameters.

Acknowledgments

We are indebted to our colleague F. Palomo-Lozano for his expert help with data analysis and algorithmic issues. The anonymus reviewers provided constructive criticism, as well as valuable comments and suggestions, greatly improving the original manuscript.

References

[1] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (4) (1978) 34–41.

[2] R. G. Hamlet, Testing programs with the aid of a compiler, IEEE Transactions Software Engineering 3 (4) (1977) 279–290.

[3] M. R. Woodward, Mutation testing —its origin and evolution, Information and Software Technology 35 (3) (1993) 163–169.

[4] A. J. Offutt, R. H. Untch, Mutation testing for the new century, Kluwer Academic Publishers, Norwell, MA, USA, 2001, Ch. Mutation 2000: uniting the orthogonal, pp. 34–44.

[5] A. T. Acree, On mutation, Ph.D. thesis, Georgia Institute of Technology, Atlanta, Georgia (1980).

[6] T. A. Budd, Mutation analysis of program test data, Ph.D. thesis, Yale University, New Have, Connecticut (1980).

[7] S. Hussain, Mutation clustering, Ph.D. thesis, King's College London, Strand, London (2008).

[8] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Transactions on Software Engineering and Methodology 5 (2) (1996) 99–118.

[9] Y. Jia, M. Harman, Higher order mutation testing, Information and Software Technology 51 (10) (2009) 1379–1393.

[10] Organization for the Advancement of Structured Information Standards, Web Services Business Process Execution Language 2.0, (Last accessed: 2 March 2011) (April 2007).
URL `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`

[11] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, I. Medina-Bulo, Gamera: An automatic mutant generation system for WS-BPEL compositions, in: ECOWS '09: Proceedings of the Seventh IEEE European Conference on Web Services, 2009.

[12] D. E. Goldberg, Genetic algorithms in search, optimization and machine learning, Addison-Wesley, Reading, 1989.

[13] Holland, J. H., Adaptation in natural and artificial systems, MIT Press, Cambridge, 1992.

[14] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, Mutation operators for WS-BPEL 2.0, in: ICSSEA 2008: Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications, 2008.

[15] H. Zhu, P. Hall, J. May, Software unit test coverage and adequacy, ACM Computing Surveys 29 (4) (1997) 366–427.

[16] A. Offutt, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, Proceedings of the 15th International Conference on Software Engineering (1993) 100–107.

[17] G. Syswerda, A study of reproduction in generational and steady-state genetic algorithms, in: Foundations of genetic algorithms, Morgan Kaufmann Publishers, 1991, pp. 94–101.

[18] H. Mühlenbein, Evolution in time and space - the parallel genetic algorithm, in: Foundations of Genetic Algorithms, Morgan Kaufmann, 1991, pp. 316–337.

[19] Mitchell, M., An introduction to genetic algorithms, Massachusetts Institute of Technology, 1996.

[20] D. Whitley, A genetic algorithm tutorial, Statistics and Computing 4 (2) (1994) 65–85.

[21] C. Rosin, R. Belew, New methods for competitive coevolution, Evolutionary Computation 5 (1996) 1–29.

[22] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, E. Spafford, Design of mutant operators for the C programming language, Tech. Rep. SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana (1989).

[23] K. N. King, A. J. Offutt, A FORTRAN language system for mutation-based software testing, Software - Practice and Experience 21 (7) (1991) 685–718.

[24] A. J. Offutt, J. Voas, J. Payne, Mutation operators for Ada, Tech. Rep. ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University (1996).

[25] J. Tuya, M. J. S. Cabal, C. de la Riva, Mutating database queries, Information and Software Technology 49 (4) (2007) 398–417.

[26] R. P. Wiegand, An analysis of cooperative coevolutionary algorithms, PhD thesis, George Mason University, Fairfax, Virginia, USA (2003).

[27] XSL transformations (XSLT) version 2.0, (Last accessed: 2 March 2011) (January 2007).
URL http://www.w3.org/TR/xslt20/

[28] M. Palomo-Duarte, A. García-Domínguez, I. Medina-Bulo, Takuan: A dynamic invariant generation system for WS-BPEL compositions, in: ECOWS '08: Proceedings of the Sixth European Conference on Web Services, IEEE Computer Society, Washington, DC, USA, 2008, pp. 63–72.

[29] A. Koval, Understanding the Travel Reservation Service, (Last accessed: 2 March 2011) (Nov. 2008).
URL http://www.netbeans.org/projects/usersguide/downloads/download/NB61-SOAdocs.zip

[30] P. Mayer, Design and implementation of a framework for testing BPEL compositions, Master's thesis, Leibniz Universität Hannover (Sep. 2006).

[31] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria, in: Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society Press, Sorrento, Italy, 1994, pp. 191–200.

[32] J. Offutt, Y.-S. Ma, Y.-R. Kwon, The class-level mutants of MuJava, in: AST '06: Proceedings of the 2006 international workshop on Automation of software test, ACM, New York, NY, USA, 2006, pp. 78–84.

[33] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 99 (PrePrints).

[34] W. E. Howden, Symbolic testing and the dissect symbolic evaluation system, IEEE Transactions Software Engineering 3 (4) (1977) 266–278.

[35] M. R. Woodward, K. Halewood, From weak to strong, dead or alive? An analysis of some mutation testing issues, in: Proceedings of the 2nd workshop on Software Testing, Verification, and Analysis, IEEE Computer Society, 1988, pp. 152–158.

[36] R. DeMillo, E. Krauser, A. Mathur, Compiler-integrated program mutation, in: Proceedings of the 5th Annual Computer Software and Applications Conference, 1991, pp. 351–356.

[37] R. H. Untch, A. J. Offutt, M. J. Harrold, Mutation analysis using mutant schemata, in: International Symposium on Software Testing and Analysis, 1993, pp. 139–148.

[38] Y.-S. Ma, J. Offutt, Y. R. Kwon, MuJava: an automated class mutation system, Software Testing, Verification & Reliability 15 (2) (2005) 97–133.

[39] A. P. Mathur, E. W. Krauser, Mutant unification for improved vectorization, Tech. Rep. SERC-TR-14-P, Software Engineering Research Center, Purdue University (1988).

[40] E. W. Krauser, A. P. Mathur, V. J. Rego, High performance software testing on SIMD machines, IEEE Transaction Software Engineering 17 (5) (1991) 403–423.

[41] B. Choi, A. P. Mathur, High-performance mutation testing, Journal of Systems and Software 20 (2) (1993) 135–152.

[42] A. J. Offutt, R. P. Pargas, S. V. Fichter, P. K. Khambekar, Mutation testing of software using a MIMD computer, in: Proceedings of the International Conference on Parallel Processing, 1992, pp. 257–266.

[43] C. N. Zapf, A distributed interpreter for the Mothra mutation testing system, Ph.D. thesis, Clemson University (1993).

[44] K. Adamopoulos, M. Harman, R. M. Hierons, How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution, in: GECCO'04 (2): Proceedings of the Genetic and Evolutionary Computation Conference, 2004, pp. 1338–1349.

[45] M. Delamaro, J. Maldonado, Proteum–a tool for the assessment of test adequacy for C programs, in: Proceedings of the Conference on Performability in Computing System (PCS 96), 1996, pp. 79–95.

[46] Y. Jia, M. Harman, MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language, in: TAIC-PART '08: Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques, IEEE Computer Society, 2008, pp. 94–98.

[47] Xanthakis, S., Ellis, C., Skourlas, C., Gall, A. L., Katsikas, S., Karapoulios, K., Application of genetic algorithms to software testing (application des algorithmes génétiques au test des logiciels), in: Proceedings of the 5th International Conference on Software Engineering, 1992, pp. 625–636.

[48] T. Mantere, J. T. Alander, Evolutionary software engineering, a review, Applied Soft Computing 5 (3) (2005) 315–331.

[49] P. Mcminn, Search-based software test data generation: A survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.

[50] R. P. Pargas, M. J. Harrold, R. Peck, Test-data generation using genetic algorithms, Software Testing, Verification and Reliability 9 (4) (1999) 263–282.

[51] C. C. Michael, G. McGraw, M. Schatz, C. C. Walton, Genetic algorithms for dynamic test data generation, in: Automated Software Engineering, 1997, pp. 307–308.