# A Framework for Genetic Test-Case Generation for WS-BPEL Compositions

Antonia Estero-Botaro, Antonio García-Domínguez, Juan Domínguez-Jiménez, Francisco Palomo-Lozano, Inmaculada Medina-Bulo



# A framework for Genetic Test-Case Generation for WS-BPEL Compositions

Antonia Estero-Botaro, Antonio García-Domínguez, Juan José Domínguez-Jiménez, Francisco Palomo-Lozano, Inmaculada Medina-Bulo

Departamento de Ingeniería Informática, Universidad de Cádiz
C/Chile 1, 11002, Cádiz, España
{antonia.estero, antonio.garciadominguez}@uca.es
{juanjose.dominguez ,francisco.palomo,inmaculada.medina}@uca.es

**Abstract.** Search-based testing generates test cases by encoding an adequacy criterion as the fitness function that drives a search-based optimization algorithm. Genetic algorithms have been successfully applied in search-based testing: while most of them use adequacy criteria based on the structure of the program, some try to maximize the mutation score of the test suite.
This work presents a genetic algorithm for generating a test suite for mutation testing. The algorithm adopts several features from existing bacteriological algorithms, using single test cases as individuals and keeping generated individuals in a memory. The algorithm can optionally use automated seeding when producing the first population, by taking into account interesting constants in the source code.
We have implemented this algorithm in a framework and we have applied it to a WS-BPEL composition, measuring to which extent the genetic algorithm improves the initial random test suite. We compare our genetic algorithm, with and without automated seeding, to random testing.

## 1 Introduction

Search-based testing [10] consists of generating test data according to a certain adequacy criterion, by encoding it as the fitness function that drives a search-based optimization algorithm. Evolutionary testing is a field of search-based testing that uses evolutionary algorithms to guide the search. The global searches performed by these algorithms are usually, but not always, implemented as genetic algorithms (GAs) [9,11].

Using GAs for generating test data dates back to the work by Xanthakis et al. [22]. Since then, various alternative approaches for generating test data have been proposed, both using GAs and other evolutionary techniques. Mantere and Alander [13] published a review of the works that applied evolutionary algorithms to software testing at the time.

Search-based testing has been mostly used for structural test data generation, with branch coverage as the most common adequacy criterion. The survey by McMinn [15] classified evolutionary algorithms for structural test data generation by the type of information used by the fitness function.

Mutation testing [19] is a testing technique injecting simple faults in the program under test through the use of *mutation operators*. As a result, we obtain *mutants*: 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 for every test case in the test-suite, then the mutant remains *alive*. When this is always the case, regardless the input, the mutant is said to be *equivalent* to the original program.

Mutation testing uses the *mutation score* to measure the quality of the test-suite. The mutation score is the ratio of the number of mutants killed by the test-suite to the number of non-equivalent mutants. Obtaining test-suites with mutation scores in the 50%–70% range can be relatively easy, but increasing these scores to the 90%–100% can be very hard. For this reason, developing new techniques for generating test-suites is currently a very active field of study [4].

Several evolutionary approaches for generating test data for mutation testing already exist. Baudry et al. [4] compared genetic and bacteriological algorithms for generating test data for a mutation testing system. Likewise, May et al. [14] compared genetic and immunological algorithms for the same purpose. In both cases, the alternative worked better than the GA.

This work presents an evolutionary technique for test-case generation using a GA for a mutation testing system. Our GA adopts some features from the bacteriological algorithm proposed by Baudry et al. We built a framework to apply it to WS-BPEL (Web Service Business Process Execution Language [18]) compositions. WS-BPEL was targeted because testing service-oriented software is not an easy task, and this language is an OASIS standard and an industrial-strength alternative in its field. WS-BPEL compositions may build new WS from other WS from all around the world, while using unconventional and advanced programming concepts as well. As the relevance and economic impact of service compositions grow, the need for efficient and effective testing techniques for this kind of software increases.

This work is structured as follows. Section 2 introduces the WS-BPEL language and the basic concepts behind GAs. Section 3 discusses related work. Section 4 describes the overall design of the GA for generating test-suites. Section 5 describes how the GA can be applied to WS-BPEL compositions and shows the results obtained by the GA generating tests for a particular WS-BPEL composition. Finally, we present our conclusions and future work in Section 6.

## 2 Background

### 2.1 A Brief Introduction to WS-BPEL

WS-BPEL 2.0 is an OASIS standard [18] and an industrial-strength programming language for WS compositions. WS-BPEL is an XML-based language al-

```
<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>
```

**Fig. 1.** WS-BPEL 2.0 activity sample

lows one to specify the behavior of a business process based on its interactions with other WS. 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 may 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. We can illustrate this 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. In the aforementioned example we can see 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 after the source activity of the link has been completed.

### 2.2 Genetic Algorithms

Genetic algorithms [9,11] 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, GAs perform a selection

process to improve the population, so they are ideal for optimization purposes. 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 stochastic and heuristic nature, and the difficulties in handling restrictions.

There is no a 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 [9] replace the entire population in each generation. However, steady-state or incremental GAs [21] replace only a few (one or two) members of the population in each generation. Finally, in parallel fine-grained GAs [17] the population is distributed into different 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 and the individual fitness used are highly dependent on the problem to solve, and they are the only link between the GA and the problem [16].

GAs use two types of operators: selection and reproduction. *Selection operators* select individuals in a population for reproduction attending to the following rule: the higher the fitness of an individual, the higher the probability of being selected. *Reproduction operators* generate the new individuals in the population: *crossover operators* generate two new individuals or offspring, from two preselected individuals, their parents. The offspring inherit part of the information stored in both parents. On the other hand, *mutation operators* aim to alter the information stored in a given individual. The design of these operators heavily depends on the encoding scheme used.

Please, notice that the above mutation operators are related to the GA and are different from those for mutation testing.

## 3 Related Work

Evolutionary algorithms in general and GAs in particular have been widely used in various fields of software engineering, and especially so in software testing. A large part of the works on search-based testing used structural criteria to generate test cases.

Bottaci [5] was the first to design a fitness function for a GA that generated test data for a mutation testing system. This fitness function is based on the three conditions listed by Offutt [20]: *reachability*, *necessity* and *sufficiency*. Being the first of its kind, it has been used in many works, even with other evolutionary algorithms. As an example, Ayari [3] used it with ant colonies.

Baudry et al. [4] have compared GAs and bacteriological algorithms for generating test cases for a mutation system. The main issues of GAs were slow convergence and the need to apply the mutation genetic operator more often.

The main difference between the bacteriological algorithm and the GA is that it stores the best individuals and that it does not use the crossover operator. In addition, individuals now represent individual test cases instead of entire test suites. This alternative approach is shown to converge much faster than the previous one, producing higher mutation scores in less generations.

Our approach is based on a GA that has been modified to obtain some of the advantages of a bacteriological algorithm. The individuals of our GA are individual test cases and both crossovers and mutations are performed. We do not need to use a separate store for the best individuals, as we store all individuals in a memory to preserve the useful information that may reside in individuals with low fitness scores. Our fitness function differs from that in Baudry et al., as it takes into account both the number of mutants killed by the test-case and how many other test cases killed those mutants.

May et al. [14] presented two evolutionary approaches for obtaining test cases for a mutation testing system: one based on an immune approach, and a GA. The algorithm based on the immune approach iteratively evolved a population of antibodies (individual test cases), looking for those that killed at least one mutant not killed by any previous antibody. The selected antibodies were added to the internal set which would become the final output of the process.

## 4 Approach

We advocate an approach based on a GA whose goal is generating test cases killing the mutants generated from the program under test. Figure 2 illustrates the architecture underlying the test generation procedure. The test-case generator consists of two main components: the preprocessor and the GA.

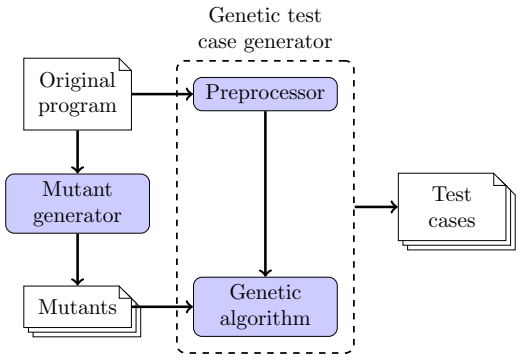


**Fig. 2.** Architecture of the test-case generator

The preprocessor can produce all the information required by the GA to generate the test cases from the original program, providing the GA with an initial

population of randomly generated test cases, completed with some test cases generated by automated seeding. The actual operations run by the preprocessor depend on the particular programming language in which the program under test is written. Please see Section 5.1 for the specific details.

The test cases produced in each generation are executed against all the mutants. The fitness of each test-case is computed by comparing the behavior of each mutant and the original program on it. This guides the search of the GA. As the GA completes its execution, a final test-suite is obtained.

The goal of a test-case generator is producing an optimal test-suite. We propose a GA with a design based on several features of bacteriological algorithms. The individuals of our GA are individual test cases. The GA will gradually improve the initial test-suite, generation by generation, using a memory similar to a *hall of fame*. The memory will store all the test cases produced so far. Therefore, the output of the GA will not be its last generation, but rather all the test cases in the memory. This memory plays the same role as the memory used in the bacteriological algorithm proposed by Baudry et al. [4].

Next, we discuss the particular choices that characterize this GA, paying particular attention to how they contribute to produce good test cases.

**Encoding of individuals** Each individual encodes one test-case. As test cases are highly dependent on the program under test, the structure of individuals should be flexible enough to adapt to any program. In order to solve this problem, each individual will contain an array of key-value pairs, where the key is the *type* of a program variable and the *value* is a particular literal of the corresponding type. Figure 3 illustrates how individuals are encoded.

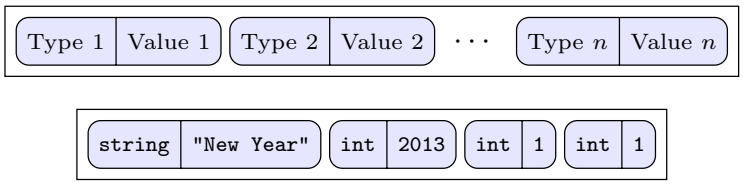


**Fig. 3.** Encoding of individuals

**Fitness of individuals** Individual fitness is a function of the number of mutants killed by the corresponding test-case. However, though tempting, just taking the number of killed mutants per test-case is not an appropriate fitness metric. Let us consider specialized test-cases able to kill just one mutant which can not be killed by any other test-case in the population. A metric based only in the number of kills would assign this important test-cases a low fitness.

For this reason, an additional variable has to be taken into account when assessing the fitness of individuals: the number of test-cases killing the same

mutant. This is key in both distinguishing individuals with the same number of kills and being fair in the evaluation of specialized test-cases.

Let $M$ be the set of mutants generated from the original program and $T$ be the set of test-cases in a given population. The fitness function is computed from the *execution matrix*, $E = [e_{ij}]$, where $e_{ij}$ is the result of comparing the execution of the original program versus the execution of mutant $m_j \in M$ on test-case $t_i \in T$: $e_{ij} = 0$ when no difference can be found, while $e_{ij} = 1$ if $m_j$ was killed by $t_i$. We define the fitness of individual $t_i$ as:

$$f(t_i) = \sum_{j=1}^{|M|} \frac{e_{ij}}{\sum_{k=1}^{|T|} e_{kj}} \tag{1}$$

**Initial population** We have implemented two ways of generating the initial population at the beginning of the execution of the GA:

1. The initial population corresponds to valid random data.
2. The initial population corresponds to valid random data with additional test-cases where some components are replaced by constants of matching types found in the source code. This approach is similar to those proposed in [1], [2], and [8]. We have named this approach *automated seeding*.

**Generations** The test-case generator is based on a generational GA in which the population of the next generation consists of offspring produced from pairs of individuals of the population of the previous generation by the genetic operations of crossover and mutation.

The selection scheme determines the way that individuals are chosen for haploid reproduction, and eventually mutation, depending on their fitness. We select individuals with the roulette wheel method designed by Goldberg [9]. Therefore, we make selection probability proportional to fitness.

Algorithm 1 illustrates how crossover and mutation operators can be applied to generate a pair of *offspring* from the current *population*. We assume that $p_c$ is the crossover probability, that $p_m$ is the probability of mutation, and that *random-uniform*$(a, b)$ produces a random number uniformly distributed in the real interval $[a, b)$. This procedure is iterated until a new population of identical size is reached. New populations will be generated until a given termination condition is met.

In a whole population of size $n$, crossover contributes a total of $np_c$ individuals. As mutation does not alter the number of individuals at all, $n(1 - p_c)$ selected individuals remain unchanged to maintain the population size constant.

**Crossover operator** The crossover genetic operator exchanges the components of parents to produce two offspring. We use one-point crossover, in which the point is randomly chosen using a uniform distribution. Figure 4 shows how crossover is performed on a pair of individuals.

```
▶ Selection with Goldberg's roulette-wheel method.
parents ← select-parents(population)
▶ Crossover with probability p_c.
if random-uniform(0, 1) < p_c then
   offspring ← do-crossover(parents)
else
   offspring ← parents
end if
▶ Mutation with probability p_m.
if random-uniform(0, 1) < p_m then
   offspring ← do-mutation(offspring)
end if
return offspring
```

**Algorithm 1: Offspring generation.**

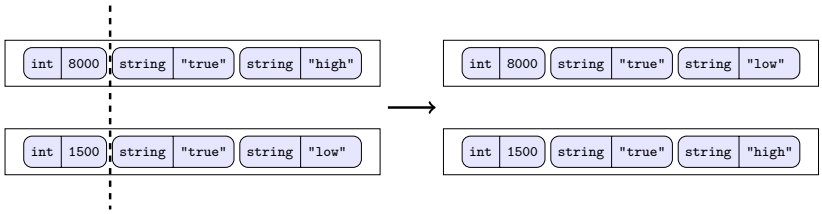


**Fig. 4.** Application of the crossover operator

**Mutation operator** The mutation genetic operator changes the *value* field of one component in an individual. The actual change will depend on its data type. For instance, a *float* value $v$ is mutated into $\hat{v}$ using the following formula:

$$\hat{v} = v + s \cdot \frac{r}{p_m} \tag{2}$$

where $s$ is randomly chosen between $-1$ or 1 and $r$ is a parameter which modulates the change of $v$. For instance, if $r = 1000$, $v$ may be incremented or decremented by up to $1000/p_m$. To summarize: $v$ undergoes a perturbation that may be positive or negative, depending on the value of $s$. This perturbation is proportional to the inverse of $p_m$: as mutations become less common, perturbations are larger.

Values of type *string* are mutated by replacing them with a new value.

When mutating *list* values, the operator randomly decides (with equal probabilities) whether to mutate its length or mutate an element picked at random. The length $l$ of the list is mutated into a new length $\hat{l}$ as follows:

$$\hat{l} = \textit{random-uniform}(0, 1) \cdot \frac{l}{p_m} \tag{3}$$

Finally, *tuple* values are mutated by picking one of its elements at random, as their lengths are fixed.

**Termination** Termination conditions are checked on each generation after computing the fitness of the individuals. Our GA implements four termination conditions that can be combined or used in isolation: maximum number of generations, percentage of mutants killed, stagnation of maximum fitness, and stagnation of average fitness.

## 5 Application to WS-BPEL

In order to evaluate our approach we have implemented a framework for WS-BPEL compositions. By design, the framework encapsulates all the language-specific (in this case, WS-BPEL-specific) details in a component known as the *Preprocessor*. Mutants are generated, executed, and evaluated using our mutation testing tool for WS-BPEL, MuBPEL [7], which incorporates the mutation operators defined by Estero et al. [6].

### 5.1 Preprocessor

A test-case for a WS-BPEL composition consists of the messages that need to be exchanged among the various partners in a business process: the client, the composition itself and the external services invoked by the composition. Generating test-cases automatically requires knowing the structure of the messages that constitute these test-cases.

In order to obtain the test-case file used by the GA as the initial population, an *Analyzer* produces a message catalog from the WSDL documents describing the public interfaces of the WS-BPEL composition. The message catalog contains a set of templates that can generate the required messages. Each template declares the variables used in it and their types. From this message catalog, a specification of the test data format is produced. This specification is used by the random test generator in the framework to produce a test data file.

We will use the *LoanApproval* composition [18] to illustrate these steps. This composition simulates a loan-approval service in which customers request a loan of a certain amount. Whether the loan is approved or denied depends on the requested amount and the risk level associated to the customer. Two external WS are used: the assessor service and the approval service. When the requested amount is modest (less than or equal to 10000 monetary units) the assessor WS is invoked to assess the risk presented by the customer. If the risk is low, the loan is immediately approved. In any other case, that is, large loans or high-risk customers, the decision is delegated to the approval WS.

The *LoanApproval* composition uses three WSDL files: one for each of the external WS and another for the composition itself. From these files, the *Analyzer* obtains the message catalogs describing the variables from which the messages exchanged between the composition and its partners can be generated. The *LoanApproval* composition uses a single input variable named *req_amount* of type *int*. The approval service is invoked by the WS-BPEL composition, and its output is controlled by the Boolean variable *ap_reply*. The assessor service

is also invoked by the composition, and its output variable *as_reply* accepts a *string* value between "`high`" or "`low`" which just represents the estimated risk.

Figure 5 shows the test data format specification that was extracted from the message catalog and that will be used to generate the test-cases. It is written in a domain-specific language used by the random test generator in the framework, which can produce files such as the one shown in Figure 6. This file shows seven test cases. Variable *req_amount* is set to 54907 in the first test case, 103324 in the second test-case and so on. Variables *ap_reply* and *as_reply* contain the replies to be sent from the mockups of the approver and the assessor services, respectively, and can take the values "`true`" or "`false`" and "`high`" or "`low`". The special value "`silent`" indicates that the mockup will not reply.

```
typedef int    (min = 0, max = 200000)                 Quantity;
typedef string (values = { "true", "false", "silent" }) ApReply;
typedef string (values = { "low",  "high",  "silent" }) AsReply;

Quantity req_amount;
ApReply  ap_reply;
AsReply  as_reply;
```

**Fig. 5.** Specification of the data used to build messages for the *LoanApproval* composition

```
#set($req_amount = [54907,   103324,   175521,  122707,  160892, 115354, 130785])
#set($ap_reply   = ["false", "silent", "false", "false", "true", "true", "silent"])
#set($as_reply   = ["high",  "silent", "low",   "high",  "low",  "low",  "high"])
```

**Fig. 6.** Test data file for the *LoanApproval* composition

Figure 7 shows a potential individual for the *LoanApproval* composition. We can see that a test-case for this composition consists of one component of type *int* and two *string* components that correspond to the variables *req_amount*, *ap_reply* and *as_reply* respectively. These were shown in Figure 6 as well.

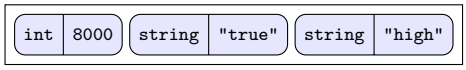


**Fig. 7.** Individual representing a test-case for the *LoanApproval* composition

### 5.2 A Case Study: The Loan Approval Composition

In this section, we will conduct several experiments to provide answers to the following research questions:

RQ1. *How much does the GA improve the quality of the initial test suite?* The initial population of the GA is randomly generated. We use three quality metrics to compare the initial population with the test-suite produced by the GA: mutation score, sentence coverage and condition/decision coverage.

RQ2. *Are the improvements just a consequence of the greater size of the final test-suite?* The GA is a test-case generator. As such, the test suite generated can be much greater than the initial test-suite. Thus, it could be more powerful just because of its size. We compare the test suites produced by the GA with randomly generated test-suites of the same size, using the same metrics employed in RQ1.

We will use the *LoanApproval* composition from Section 5.1 to answer the proposed research questions. This composition has 154 LOC, it generates 90 mutants, 2 mutants are invalid, and 9 mutants are equivalent.[1]

We can observe that 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.

Next we will describe the experimental procedure used to answer the research questions and summarize the results obtained.

Since the initial population of the GA is randomly generated, results could largely vary from one execution to another. In order to alleviate this, we decided to select as the initial population a good representative of possible initial populations through these steps:

1. Population sizes were defined.
2. Thirty random test-suites were generated, with sizes matching each population size.
3. Test-suites were run against the original composition and their mutants to measure their mutation scores, sentence coverages, and condition/decision coverages.[2]
4. Medians for the quality metrics were computed.
5. A test-suite with median mutation score was selected.

The result of this process is a typical test-suite, which has been generated at random, but it is unbiased. Thus, it is a good candidate for initial population in our experiments.

[1] Equivalent mutants have been manually detected by inspecting the source code of surviving mutants.

[2] Sentence and condition/decision coverages have been computed as the percentage of killed mutants from those generated by a sentence coverage mutation operator and a condition/decision mutation operator, respectively.

Regarding population sizes, it is well known that a GA is likely to find better solutions when the population size is greater. But bigger population sizes demand more computational resources to find those solutions. However, Krishnakumar [12] found that a population size of 10 individuals can produce similar convergence rates with a tailored GA.

Since the *LoanApproval* composition produces 90 mutants, a test suite might need up to 90 test-cases to kill them all, though much fewer test cases will usually suffice. For this reason, we defined population sizes in terms of percentages of the total number of mutants. Percentages of 15%, 20%, and 25% were selected as candidate values to produce small populations still having more than 10 individuals.

As to the number of executions, we decided to use 30 executions for each population size. Therefore, 30 different seeds were employed to generate random data for 30 different executions.

Table 1 shows the values of the three metrics for the selected test-suite (the random initial test-suite) and for the final test-suite produced by the GA, with and without automated seeding. These are the medians of all the values produced by the 30 executions for each population size. The configuration parameters for

**Table 1.** Quality metrics for the initial test-suite (random) and the test-suites generated by the GA. In columns: MS are mutation scores, SC are sentence coverages, and CDC are condition/decision coverages. Coverages are measured as percentages.

| Pop. size | Random | | | GA | | | GA (aut. seeding) | | |
|---|---|---|---|---|---|---|---|---|---|
| | MS | SC | CDC | MS | SC | CDC | MS | SC | CDC |
| 15% | 0.72 | 75.0 | 75.0 | 0.96 | 100.0 | 100.0 | 0.99 | 100.0 | 100.0 |
| 20% | 0.72 | 75.0 | 75.0 | 0.96 | 100.0 | 100.0 | 0.99 | 100.0 | 100.0 |
| 25% | 0.72 | 75.0 | 75.0 | 0.96 | 100.0 | 100.0 | 0.99 | 100.0 | 100.0 |

the GA are shown in Table 2. We set $p_m$ to 10%, as it is also suggested by Baudry et al. [4]. In agreement with Baudry's results, it was easy to obtain a mutation score within the 50%–70% range: in this particular case, the mutation score of the initial test-suite was 72%. Even without automated seeding, the three quality metrics have been considerably improved by the GA: mutation score increases from 0.72 to 0.96, and both coverage metrics increases from 75% to 100%. With automated seeding, mutation score further increases to 0.99.

**Table 2.** Configuration parameters of the GA

| Population | $p_c$ | $p_m$ | $r$ |
|---|---|---|---|
| 15% 20% 25% | 90% | 10% | 10 |

Since all the population sizes obtained the same medians in their quality metrics, we compared the results from each of their 30 executions without automated seeding and with automated seeding. The results are shown in Tables 3 and 4, respectively. Without automated seeding, the size that produces the most stable results is 25%. With automated seeding, the size that produces the best results is 20%, reaching 100% mutation score in two executions.

**Table 3.** Quality metrics obtained by the GA without automated seeding for each population size, grouped by execution results

| Population | Seeds | MS | SC (%) | CDC (%) |
|---|---|---|---|---|
| 15% | 2 | 0.72 | 75.0 | 75.0 |
| | 1 | 0.90 | 93.8 | 100.0 |
| | 1 | 0.95 | 100.0 | 100.0 |
| | 26 | 0.96 | 100.0 | 100.0 |
| 20% | 2 | 0.95 | 100.0 | 100.0 |
| | 28 | 0.96 | 100.0 | 100.0 |
| 25% | 30 | 0.96 | 100.0 | 100.0 |

**Table 4.** Quality metrics obtained by the GA with automated seeding for each population size, grouped by execution results

| Population | Seeds | MS | SC (%) | CDC (%) |
|---|---|---|---|---|
| 15% | 7 | 0.97 | 100.0 | 100.0 |
| | 23 | 0.99 | 100.0 | 100.0 |
| 20% | 3 | 0.97 | 100.0 | 100.0 |
| | 2 | 1.00 | 100.0 | 100.0 |
| | 25 | 0.99 | 100.0 | 100.0 |
| 25% | 3 | 0.97 | 100.0 | 100.0 |
| | 27 | 0.99 | 100.0 | 100.0 |

As the GA generates test-suites much bigger than the initial test-suites, the next step is comparing random generation of bigger test-suites with the results produced by our GA, with and without automated seeding. For a 20% population size, the GA executed a median of 156 distinct test-cases without automated seeding, and a median of 153 different test-cases with automated seeding.

Table 5 compares the median values of metrics corresponding to 30 random test-suites, with 156 test-cases each, with those produced by both variants of our GA. Results indicate that the metrics for random test-suites were as good as those for the GA without automated seeding.

This can be explained by the fact that small compositions usually have few execution paths and, thus, random test-suites of considerable size are likely to produce high sentence and condition/decision coverages. Moreover, at the same time, those compositions tend to generate a small number of mutants. As a consequence, their mutants might be mostly killed by a random test-suite of even modest size and high mutation scores can be expected.

This is the case, with the *LoanApproval* composition, which in fact has few mutants and execution paths. However, the GA with automated seeding obtained better mutation score (0.99) than the random test-suite (0.96).

**Table 5.** Results produced by random test generation and the two versions of our GA

| | Random | GA | GA (autom. seeding) |
|---|---|---|---|
| Size test-suite | 156 | 156 | 153 |
| MS | 0.96 | 0.96 | 0.99 |
| SC (%) | 100.0 | 100.0 | 100.0 |
| CDC (%) | 100.0 | 100.0 | 100.0 |

From these results, we can now answer the two research questions that were posed at the beginning of this section for the *LoanApproval* composition.

RQ1. *How much does the GA improve the quality of the initial test suite?* The test-suite generated by our GA improves the initial test-suite with respect to all the quality metrics measured (mutation score, sentence coverage, and condition/decision coverage). Without automated seeding, the GA increases the mutation score from 0.72 to 0.96. Sentence coverage and condition/decision coverage increase from 75% to 100%. With automated seeding, the mutation score further increases to 0.99.

RQ2. *Are the improvements just a consequence of the greater size of the final test-suite?* Once we extend the initial test-suite with sufficient random test-cases to match the size of the final test-suite, the GA shows no difference when automated seeding is disabled. However, with automated seeding, the GA obtains a higher mutation score: 0.99 instead of 0.96.

## 6 Conclusions and Future Work

We have presented a GA for generating test-cases for a mutation testing system. The initial population can be generated in two ways. The first way simply generates all individuals at random. The second way mixes in test-cases which are derived from the constants present in the source code of the program. Our GA also adopts several features of the bacteriological algorithms proposed for test-case generation by Baudry et al. [4].

We have implemented both approaches in a framework which generates test-cases for WS-BPEL compositions. The GA was applied to a standard WS-BPEL

composition. We compared the quality of the initial random test-suite against the test-suites produced by the GA. Test-suite quality was measured using mutation score, sentence coverage and condition/decision coverage. It is shown that, for this composition, the median of the quality metric values across 30 executions improves, both with and without automated seeding.

The quality of the test-suites generated by the GA has been also compared to the quality of random test-suites of the same size. In this case, random testing produces the same results as using the GA without automated seeding. However, the GA with automated seeding produces slightly better results. This has been traced to the fact that the composition under study is small and have few execution paths. Therefore, random test-suites of considerable size are likely to produce high sentence and condition/decision coverages. At the same time, its number of mutants is modest and they can be killed by a random test suite. Consequently, random test-suites enjoy higher mutation scores than usual and the margin of improvement for the GA, which starts from an initial population of random test-cases, gets drastically reduced.

These preliminary results are promising, but we acknowledge that one of the limitations of the present study is that it has only been applied to a single WS-BPEL composition so far. Future work will be devoted to validate our results with a number of increasingly complex WS-BPEL compositions. The main hindrance here is the absence of a standard set of WS-BPEL compositions suitable for functional testing.

Another limitation of the current study is that we have mainly concentrated in assessing the impact of the population size. Other configuration parameters, for which reasonable values were used, might be adjusted and hopefully improve our results. Further research and extensive experimentation could lead to better combinations of parameters for generating test-suites.

## Acknowledgments

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).

## References

1. Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. In: 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 3–12 (2011)
2. Alshraideh, M., Bottaci, L.: Search-based software test data generation for string data using program-specific search operators. Softw. Test. Verif. Reliab. 16(3), 175–203 (2006)
3. Ayari, K., Bouktif, S., Antoniol, G.: Automatic mutation test input data generation via ant colony. In: 9th Conference on Genetic and Evolutionary Computation. pp. 1074–1081. ACM (2007)

4. Baudry, B., Fleurey, F., Jézéquel, J.M., Le Traon, Y.: From genetic to bacteriological algorithms for mutation-based testing. Soft. Test. Verif. Reliab. 15(2), 73–96 (2005)
5. Bottaci, L.: A genetic algorithm fitness function for mutation testing. In: 23rd International Conference on Software Engineering using Metaheuristic Inovative Algorithms. pp. 3–7 (2001)
6. Estero-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I.: Mutation operators for WS-BPEL 2.0. In: ICSSEA 2008: 21th International Conference on Software & Systems Engineering and their Applications (2008)
7. Estero-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A.: Quality metrics for mutation testing with applications to WS-BPEL compositions. Software Testing, Verification and Reliability (2014), http://dx.doi.org/10.1002/stvr.1528
8. Fraser, G., Arcuri, A.: The seed is strong: Seeding strategies in search-based software testing. In: IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 121–130 (2012)
9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
10. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Trans. Soft. Eng. 36(2), 226–247 (2010)
11. Holland, J.: Adaptation in Natural and Artificial Systems. MIT Press, 2nd edn. (1992)
12. Krishnakumar, K.: Microgenetic algorithms for stationary and nonstationary function optimization. In: Society of Photo-Optical Instrumentation Engineers Conference Series. vol. 1196, pp. 289–296 (1990)
13. Mantere, T., Alander, J.T.: Evolutionary software engineering, a review. Applied Soft Computing 5(3), 315–331 (2005)
14. May, P., Timmis, J., Mander, K.: Immune and evolutionary approaches to software mutation testing. In: ICARIS 2007: Proceedings of the 6th International Conference on Artificial Immune systems. pp. 336–347 (2007)
15. McMinn, P.: Search-based software test data generation: A survey. Soft. Test. Verif. Reliab. 14(2), 105–156 (2004)
16. Mitchell, M.: An introduction to genetic algorithms. Massachusetts Institute of Technology (1996)
17. Mühlenbein, H.: Evolution in time and space - the parallel genetic algorithm. In: Foundations of Genetic Algorithms. pp. 316–337. Morgan Kaufmann (1991)
18. OASIS: Web Services Business Process Execution Language 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (2007)
19. Offutt, A.J., Untch, R.H.: Mutation Testing for the New Century, chap. Mutation 2000: Uniting the Orthogonal, pp. 34–44. Kluwer Academic Publishers (2001)
20. Offutt, J.: Automatic test data generation. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, USA (1988)
21. Syswerda, G.: A study of reproduction in generational and steady-state genetic algorithms. In: Foundations of genetic algorithms, pp. 94–101. Morgan Kaufmann Publishers (1991)
22. Xanthakis, S., Ellis, C., Skourlas, C., Gall, A.L., Katsikas, S., Karapoulios, K.: Application of genetic algorithms to software testing. In: Proc. of the 5th Int. Conf. on Software Engineering. pp. 625–636 (1992)