# TOWARDS HIGHER-ORDER MUTANT GENERATION FOR WS-BPEL

E. Blanco-Muñoz, A. García-Domínguez, J. J. Domínguez-Jiménez and I. Medina-Bulo
*University of Cádiz, C/ Chile, 1, Cádiz, Spain*

Keywords: Mutation testing, Genetic algorithm, Web services, Web service compositions, WS-BPEL.

Abstract: We present an architecture for automatically generating higher-order mutants for WS-BPEL compositions based on the architecture of GAmera, a first-order mutant generation system for WS-BPEL. Higher-order mutants are created by applying a sequence of first-order mutation operators to the original program. This paper also introduces the changes that GAmera has to undergo for converting the generation of first-order mutants into a process capable of higher-order mutation, while detailing the modifications carried out for adapting the crossover and mutation genetic operators to the new structure of the mutants.

## 1 INTRODUCTION

The Web Services Business Process Execution Language (WS-BPEL) (OASIS, 2007) allows us to develop new Web Services (WS) modelling more complex business processes on top of preexisting WS. The economic impact of WS-BPEL service compositions is quickly increasing (IDC, 2008), and deeper insight on how to test them effectively is therefore required.

Mutation testing (DeMillo et al., 1978; Hamlet, 1977) is a testing technique that has been applied successfully to several programming languages. Several mutant generation systems already exist (Jia and Harman, 2010), such as Mothra (King and Offutt, 1991) for FORTRAN, MuJava (Ma et al., 2005) for Java, SQLMutation (Tuya et al., 2007) for SQL, among others. In fact, we have presented GAmera (Domínguez-Jiménez et al., 2009) in previous works, an automatic mutant generation system for WS-BPEL compositions, which only deals with first-order mutants. GAmera is the first mutant generator based on a genetic algorithm (GA) (Goldberg, 1989).

However, all these tools only generate first-order mutants. In this work we present the modifications of the architecture of GAmera for automatically generating higher-order mutants for WS-BPEL compositions. A higher-order mutant is created by applying a sequence of first-order mutation operators to the original program (Jia and Harman, 2009). It has been shown empirically that about 99% of higher-order mutants are distinguished by test data that distinguish first-order mutants (known as the coupling effect) (Mathur, 1994; Wah, 2003).

This paper introduces the changes GAmera requires to convert the generation of first-order mutants into a process capable of higher-order mutation, while detailing the modifications carried out for adapting the crossover and mutation genetic operators to the new structure of the mutants.

The paper is divided into the following sections: Section 2 briefly summarizes the WS-BPEL language, mutation testing and genetic algorithms. Section 3 describes the modifications to the architecture of GAmera, as well as the new genetic operators defined. Finally, Section 5 presents the conclusions and future work.

## 2 BACKGROUND

We will first introduce the WS-BPEL language and mutation testing, and then offer some basic concepts about genetic algorithms.

### 2.1 The WS-BPEL Language

WS-BPEL (Organization for the Advancement of Structured Information Standards, 2007) 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.

## 2.2 Mutation Testing

Mutation testing (DeMillo et al., 1978; Hamlet, 1977) is a fault-based testing technique that introduces simple flaws 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. For instance, if a program contains the instruction `a > 5000` and we apply the relational mutation operator (which replaces a relational operator with another), one of the mutants produced will contain `a < 5000` instead. If a test case can tell apart the original program and the mutant, i.e. their outputs are shown to be different, it is said that this test case *kills* the mutant. Otherwise, the mutant is said to stay *alive*.

*Equivalent mutants*, which always produce the same output as the original program, are a common problem when applying mutation testing. Equivalent mutants should not be confused with *stubborn non-equivalent mutants*, which are produced because the test suite is not adequate to detect them. The general problem of determining if a mutant is equivalent to the original program is undecidable (Zhu et al., 1997).

## 2.3 Genetic Algorithms

Genetic Algorithms (Goldberg, 1989; Holland, J. H., 1992) are probabilistic search techniques based on the theory of evolution and natural selection.

GAs work with a population of solutions, known as *individuals*, and process them in parallel. Throughout the successive generations of the population, GAs perform a selection process to improve the population, and 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 their difficulties in handling restrictions.

The *fitness* of an individual measures its quality as a solution for the problem to be solved. The average fitness of the population will be maximized along the generations produced by the GA.

GAs use two types of genetic operators: selection and reproduction. *Selection operators* select individuals in a population for reproduction. The likelihood of selecting an individual may be proportional to its fitness. *Reproduction operators* generate the new individuals in the population by applying crossover and mutation operators. On the one hand, *crossover operators* generate two individuals (*children*) from two pre-selected individuals (*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.

## 3 SYSTEM ARCHITECTURE

This section describes how GAmera has changed from its previous architecture, the new genetic operators which have been adapted to the current framework and lastly, the novel genetic algorithm. The components of GAmera are still the same, although the genetic search for mutants has been modified. Figure 1 shows the core of the system: the analyzer, the mutant generator and the execution system. The analyzer takes the original program WS-BPEL process definition and produces the information required by the mutant generator. The mutant generator is divided into the GA and a converter from the individuals of the GA to the mutants of the original process definition. Finally, the execution system runs the generated mutants against the test cases and compares their outputs with those from the original process definition.

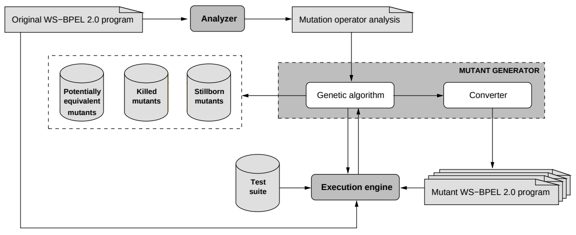
Original WS−BPEL 2.0 program
Analyzer
Mutation operator analysis
Potentially equivalent mutants
Killed mutants
Stillborn mutants
MUTANT GENERATOR
Genetic algorithm
Converter
Test suite
Execution engine
Mutant WS−BPEL 2.0 program

Figure 1: GAmera architecture.

### 3.1 Higher-order Structure

Figure 2 shows the new structure for generating higher-order mutants (HOM).

#### 3.1.1 Extending the Representation

The concept of *individual* has been changed. Individuals now encode $N$ mutations on the original program, where $N$ is the maximum order defined in the configuration. Individuals have five fields (Figure 3): the order of the individual, its fitness, the identifiers of the mutation operators to be applied, the identifiers of the locations to be mutated and the additional values which modify the behaviour of the selected mutation operator. The three lists with the mutation operators, the locations and the additional values must have the same number of elements.

With the new structure, we now define two individuals to be equal if they have the same order ($n$) and the first $n$ operator-location-attribute triplets are equal. Populations contain sequences of individuals: some of them may be equal to others.

In addition to regular populations for each generation, we keep a separate population during the execution of the GA, called the *Hall of Fame* (HOF). The HOF collects the individuals generated by the GA in previous generations. Each distinct individual is only stored once, unlike in regular populations. Individuals in the HOF also contain their *Execution Rows* with the results produced when the mutant was executed. We will detail this further in the next section.

#### 3.1.2 Fitness of the Individuals

Once the analyzer has finished, we know which mutation operators can be applied. If the value of the field *Location* is zero, it means that mutation operator cannot be used by the genetic algorithm. Therefore, it is necessary to filter the usable combinations.

To calculate the fitness of the individuals we have to convert them into mutants, by applying each mutation operator in sequence. After applying the last mutation operator mentioned in the individual, the resulting HOM can be run with the execution system.

The *execution row* of each individual is produced by running the corresponding mutant against all test cases. For each test case in the suite, its column reports if the mutant stayed alive (0), if it was killed (1) or if it could not be deployed (2).

Having set this field in the individuals of the HOF, the individual fitness of the current population can now be calculated. We use the original fitness function of GAmera. The fitness for individual $I$ is given by:

$$\text{Fitness}(I) = M \cdot T - \sum_{j=1}^{T} \left( m_{Ij} \cdot \sum_{i=1}^{M} m_{ij} \right) \qquad (1)$$

where $M$ is the number of mutants in a generation, $T$ represents the cases number of the test suite and $m$ shows the execution row of an individual $I$.

#### 3.1.3 New Genetic Operators

The first population is randomly generated. The following generations are based on a generational GA where individuals come from:

- Random generation. There is a percentage in the configuration to know how many mutants will be randomly generated.

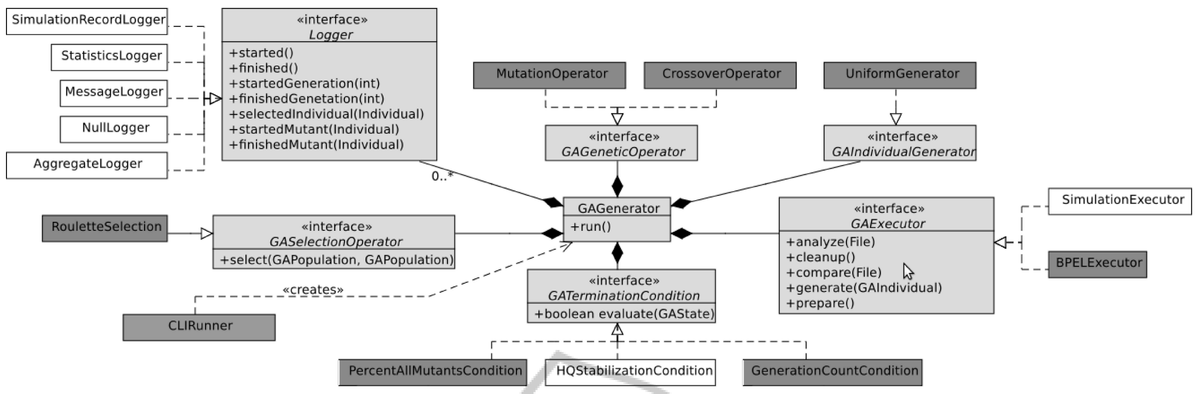

Figure 2: New higher-order structure.

- Crossover and mutation operations. The selection of individuals involved in the operations is done through the roulette wheel method. Crossovers and mutations will be done according to the probabilities specified in the configuration, $p_c$ and $p_m = 1 - p_c$, respectively. These operations have undergone several changes that are detailed below.

The *crossover operator* consists in an exchange of certain fields of the individuals involved. There are two types of crossover operators: the order and the individual crossover operator. Their probabilities are set in the configuration: $p_{oc}$ and $p_{ic}$, respectively, so $p_c = p_{oc} + p_{ic}$.

In the *order crossover operator*, a crossover point is chosen at random, between 1 to the order of the individual. Figure 5 shows how two parents generate their children, according to the selected crossover point.

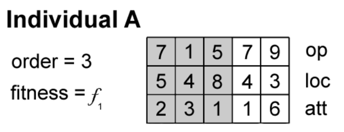

Figure 3: Representation of an individual.

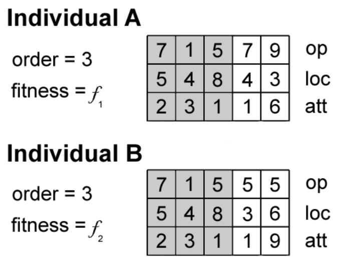

Figure 4: Equal individuals.

On the other hand, in the *individual crossover operator*, a crossover point is also chosen in the same way, but now the field which will be changed (operator, location or attribute) is randomly selected. Therefore, in this case only a value of each individual is modified. Figure 6 has an example of applying an individual crossover operator.

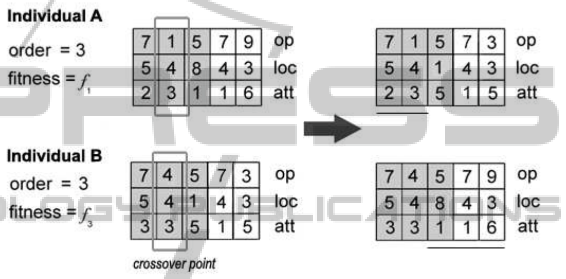

Figure 5: Order crossover operator.

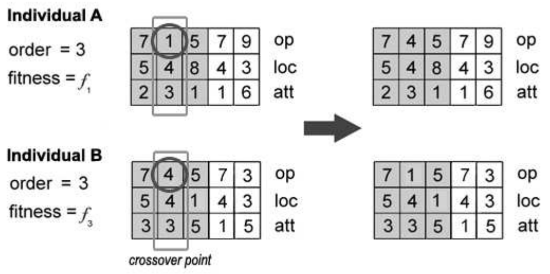

Figure 6: Individual crossover operator.

The *mutation operator* changes the value of a field of the individual. There are two types of mutation operators: the order and the individual mutation operator. Their probabilities are set by the user: $p_{om}$ and $p_{im}$, respectively, so $p_m = p_{om} + p_{im}$.

In the *order mutation operator*, the parameter which will be modified is the individual order and it will be given by:

$$\text{Order}(I) = o_{current} + rand(-1,1) \cdot (1 - p_{om}) \qquad (2)$$

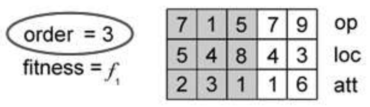

Figure 7: Order mutation operator.

In the *individual mutation operator*, a mutation point is randomly chosen and the field which will be mutated (operator, location or attribute) is selected randomly. The value which will be changed is given by:

$$\text{Value}(I) = v_{current} + rand(-1,1) \cdot (1 - p_{im}) \qquad (3)$$

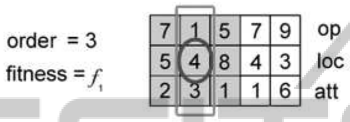

Figure 8: Individual mutation operator.

The GA will stop when the maximum percentage of generated mutants is reached.

## 4 RELATED WORKS

There are many papers on the application of mutation testing (Jia and Harman, 2010). Research in mutation testing can be classified into four types of activities (Offutt et al., 2006):

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 activity. Several mutant generation systems for various programming languages already exist:

- Mothra (King and Offutt, 1991) for FORTRAN. It is likely the most widely known mutation testing system.
- MuJava (Ma et al., 2005) for Java.
- Proteum (Delamaro and Maldonado, 1996) for C. It implements all mutation operators designed for the ANSI C programming language.
- MILU (Jia and Harman, 2008) for C. It can perform both first-order and higher-order mutation testing.
- SQLMutation (Tuya et al., 2007) for database queries written in SQL.
- GAmera (Domínguez-Jiménez et al., 2009) for WS-BPEL. This generator is the first mutant generation system based on GA.

We present a new mutant generation system based on GAmera. This mutant generation system operates over WS-BPEL process definitions, and it incorporates the generation of higher-order mutants.

## 5 CONCLUSIONS AND FUTURE WORK

This paper presents an approach towards implementing an automatic higher-order mutant generation system for WS-BPEL compositions. It improves upon our previous tool, called GAmera, which only deals with first-order mutants.

We have also described the changes needed to adapt the crossover and mutation operators to the new structure of the mutants. An advantage of this new approach is that mutants from different orders can be mixed in the same generation.

The most important contribution of this work is the proposal of the design of the first framework which generates higher-order mutants for WS-BPEL compositions.

Our future lines of work are the implementation of this framework in Java. This includes the new architecture presented in Figure 2, the extension of individual representation, and new genetic operators presented in Section 3.1.3. Finally, we will design an user-friendly graphical interface.

## REFERENCES

Delamaro, M. and Maldonado, J. (1996). Proteum–a tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing System (PCS 96)*, pages 79–95.

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

Domínguez-Jiménez, J. J., Estero-Botaro, A., García-Domínguez, A., and Medina-Bulo, I. (2009). GAmera: an automatic mutant generation system for WS-BPEL compositions. In *ECOWS'09: Proceedings of the Seventh IEEE European Conference on Web Services*.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, Reading.

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

Holland, J. H. (1992). *Adaptation in natural and artificial systems*. MIT Press, Cambridge.

IDC (2008). Research reports. http://www.idc.com.

Jia, Y. and Harman, M. (2008). 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*, pages 94–98. IEEE Computer Society.

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

Jia, Y. and Harman, M. (2010). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 99(PrePrints).

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

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

Mathur, A. (1994). Mutation testing. In Marciniak, J. J., editor, *Encyclopedia of Software Engineering*, pages 707–713. Wiley, New York, NY.

OASIS (2007). Web Services Business Process Execution Language 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Organization for the Advancement of Structured Information Standards.

Offutt, J., Ma, Y.-S., and Kwon, Y.-R. (2006). The class-level mutants of MuJava. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 78–84, New York, NY, USA. ACM.

Organization for the Advancement of Structured Information Standards (2007). Web Services Business Process Execution Language 2.0. (Last accessed: 2 March 2011).

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

Wah, K. S. H. T. (2003). An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2–3):119–161.

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