2021 18th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE). Mexico City, Mexico. November 10-12, 2021

Automatic Generation of Test Cases from Formal Specifications using Mutation Testing

Román Jaramillo Cajica
Computer science
CINVESTAV unidad Guadalajara
México
rjaramillo@cinvestav.mx

Raúl Ernesto González Torres
Computer science
CINVESTAV unidad Guadalajara
México
raul.gonzalez@cinvestav.mx

Pedro Mejía Álvarez
Computer science
CINVESTAV unidad Guadalajara
México
pedro.mejia@cinvestav.mx

***Abstract*—Testing of complex software systems often needs the execution of thousand of tests cases to find errors in the code and to ensure high integrity systems. Hence, it requires the testing tasks to be automated. Test case execution entails establishing preconditions and input data, observing output results, and comparing those results with a given oracle. This work presents two contributions: it uses a Particle Swarm Optimization (PSO) algorithm as a test case generator and also uses mutation testing on formal specifications using SOFL. Using mutation testing on formal specifications allows to find new test cases that would kill more mutants, resulting in a test suite potentially capable of finding more errors. The PSO algorithm is applied to generate the values of the input (test case). Software examples are used in this work to show the high efficiency of our testing framework.**

***Keywords*—Software testing, formal specifications, mutation testing, particle swarm optimization, test case generation**

I. Introduction

Software testing is an essential process within any software development methodology, and if not automated, it requires a large amount of time and concentration on the part of the person in charge of evaluating the behavior of the system. The essence of software testing is to identify a set of test cases for the software to be tested. One way to create this test cases is by using the system specifications. By formalizing the informal specification using a formal language, high integrity systems can be produced in a cost-effective way.

In this work we present the development of a software tool used for the generation of test sets from specifications written in the formal language SOFL. In this tool, extended functional scenarios are created from SOFL predicate specifications. For each element of the functional scenarios, the PSO algorithm is used as a generator of test cases that satisfy each predicate. From each of the test cases generated, a list of mutants is created. Finally, each mutant from the mutant list is used to generate a test case.

II. Background

A. Software testing

Software testing is the branch of software engineering that is responsible of evaluating the quality of a software in relation to its specifications. In order to do this a test suite has to be generated as well as a test oracle, which is responsible of determining whether the system's behavior is correct or not.

Software testing research aims to automatically generate test cases and test oracles. There are several research about test case generation without test oracle, other authors focus more on the quality of the test oracle and the rest of authors work in both test oracle and test suite generation.

B. Formal specifications and SOFL

SOFL uses VDM-SL notation to write specifications. It works with pre and post conditions written in First Order Logic, and also in a Disjunctive Normal Form (DNF). Each element of the predicate in DNF is known as a functional scenario. This formal method was first introduced by Liu [5] as a practical way to formalize specifications without the use of a formal model such as FSM (Finite State Machine), algebraic specification, etc. Liu [6] defines SOFL as a semi formal specification technique.

In general, SOFL specifications used in this work have the following structure [3]:

process name of the process(input variables: type of variables)
output variables: type of variables

pre predicate that is the pre condition of the process.

post predicate in DNF that is the post condition of the process.

end process.

Formally, the specification of an operation S is denoted as $S(S_{iv}, S_{ov}[S_{pre}, S_{post}]$, where S_{iv} is the set of all input variables along with its corresponding type and whose values are not changed by S. S_{ov} is the set of all output variables along with its corresponding type and whose values are generated by S. The pre condition is denoted as S_{pre}, and the post condition is defined as:

$$S_{post} \equiv (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \cdots \vee (G_n \wedge D_n),$$

where each G_i is called *Guard condition*, each D_i is called *Definition Condition*, and a ***Functional Scenario*** is the conjunction:

$$S_{pre} \wedge G_i \wedge D_i.$$

The conjunction between a single guard condition and the pre condition is called ***Test Condition*** and it is the condition used to generate values for input variables.

C. Mutation testing

Mutation testing (MT) is a popular fault-based testing technique used in software testing for test case generation and for assessing the quality of a given test suite. Mutation testing can be applied as a white-box testing technique because a code is needed in order to apply mutations to it. Also a test suite is required to execute the mutated program. MT generates a set of new programs (mutants) by changing some of the program's syntax (without a compilation error). Then each of this mutants is executed with the test suite in order to verify if the given test suite is powerful enough to discover all the errors previously added.

D. MutPy

MutPy is a tool used to apply mutation testing in the Python programming language. It takes a code (".py" file) and a test suite (also a ".py" file) and it mutates the program by applying every appropriate mutation operation. Then it executes the test suite in every mutant and if the mutant fail in at least one test case it is categorized as "killed", otherwise it is categorized as "survived". Then every mutant is shown in a list with its category and a mutation score is also shown in terminal.

E. Particle Swarm Optimization

The Particle Swarm Optimization algorithm, more commonly known as the PSO algorithm, was first introduced by Kennedy [7]. It is an optimization iterative algorithm that performs very good at minimizing and maximizing functions. A function has to be declared as the one to be minimized or maximized. In our case, a cost function was implemented accordingly to the predicate given in the specification.

The original algorithm works as follows:

Algorithm 1: Particle Swarm Optimization

```
1. Initialize a population of particles with
   random positions and velocities in D
   dimensions inside the search space.

2. Loop

3. For each particle find its fit value using
   the objective function.

4. If the new fit value is lower than the
   particle best fit value, then the new fit
   value and the position are saved.

5. Identify the particle with the best fit value
   of the population and save its position and
   fit value in a variable called G.

6. For each particle update its position and
   velocity.

7. If the best fit value is zero then the
   algorithm ends. The algorithm also ends if
   the best fit value is not zero but the limit
   of iterations has been reached. If the best
   fit value is not zero and still have not
   reached the limit of iterations then the loop
   continues.

8. End Loop
```

In order to minimize the cost function the seventh step of the algorithm has been modified:

7.If the best fit value is zero then the algorithm ends. The algorithm also ends if the rounded best fit value is not zero but the limit of resets has been reached. If the rounded best fit value is not zero and still have not reached the limit of iterations then the loop continues but if the limit of iteration is reached then the population is reinitialized. If the rounded best fit value is zero then a search of local values that minimizes the function is performed, if the result of the search is negative then the loop continues, if it succeed then the algorithm ends.

III. Related work

In this section, we show a few advanced techniques related to our methodology. Our work has a fault-based approach because it uses mutation score as metric, for this reason only fault-oriented test case generation tools that use specifications are addressed.

Model-checking is a technique for verifying whether a finite-state model of a system meets a given specification. Two of the techniques (in [10] and [11]. for instance) use model-checking for generating test cases automatically from the requirements specifications.

Software Cost Reduction (SCR) is a technique to describe the expected system behavior in a semi-formal notation. SRE syntax is used to express the specifications and generate test cases automatically [12]. Another way to describe the behavior of the system is by using a formal language, such as SOFL [14], combined with a genetic algorithm generates a high mutation scored test suite.

Larsen *et. al* [13] developed a tool called Ecdar that performs an unbounded conformance check to generate test cases automatically.

There is a small amount of related work where PSO algorithm is used in software testing in different ways (in [15] and [2]. for instance). They aim to achieve some code coverage, but none of them use formal specifications nor mutation scores.

IV. Implementation

All program specifications were in natural language, and had to be translated into SOFL language manually. SOFL specifications are files in ".txt" format. The test generation tool accept each of these files as input.

The tool creates a list of functional scenarios from the pre and post condition of the given SOFL specification. Each functional scenario is analyzed, if it has '$\geq$", '$\leq$' or '$\neq$' operators then it will be decomposed into 2 scenarios, for example, '$\geq$' operator will be divided into a predicate with "=" operator and another predicate with '$>$' operator. The result of the decomposition is a list of Extended functional scenarios, called "EFS".

A test suite is generated by creating a test case for each element of EFS. Each test case is generated using the PSO algorithm, the algorithm use the information in each functional scenario to generate values for all input and output variables. Additionally, a mutation operation is applied to each element of EFS, resulting in a list of functional scenario mutants. For each member of this new list a test case is generated (using the PSO algorithm) and added to the test suite previously generated.

The resulting test suite is a data structure containing every test case that has to be transformed into a python test file. The tool return a test suite in python language (".py" format). The test suite given by the tool can be used later by MutPy to assess its quality.

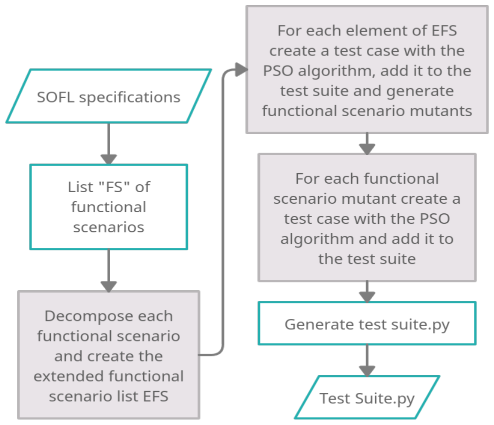
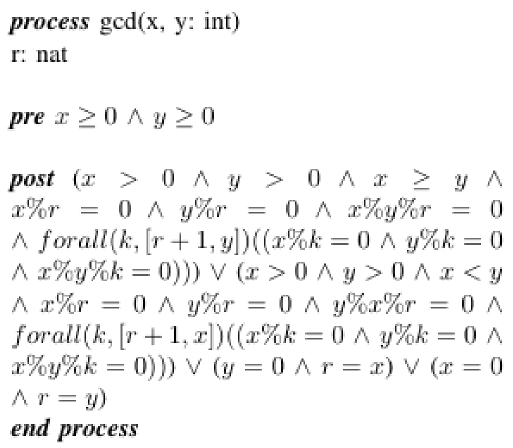

SOFL specifications
List "FS" of functional scenarios
Decompose each functional scenario and create the extended functional scenario list EFS
For each element of EFS create a test case with the PSO algorithm, add it to the test suite and generate functional scenario mutants
For each functional scenario mutant create a test case with the PSO algorithm and add it to the test suite
Generate test suite.py
Test Suite.py

Fig. 1: Tool process.

A. SOFL specifications

To illustrate the functioning of our tool, we will use the greatest common divisor (GCD) as an example. The specification of the GCD function in natural language describes that for a given pair of numbers, the function returns its greatest common divisor.

This informal specification is translated to SOFL manually by identifying those conditions that describe each possible function behavior. All possible behaviors are expressed by predicates in DFN and each element of the disjunction represent a possible behavior. As illustrated in Figure 2, each possible behavior is expressed as a conjunction of predicates involving input and output variables.



process gcd(x, y: int)
r: nat

pre $x \geq 0 \wedge y \geq 0$

post $(x > 0 \wedge y > 0 \wedge x \geq y \wedge x\%r = 0 \wedge y\%r = 0 \wedge x\%y\%r = 0 \wedge forall(k, [r+1, y])((x\%k = 0 \wedge y\%k = 0 \wedge x\%y\%k = 0))) \vee (x > 0 \wedge y > 0 \wedge x < y \wedge x\%r = 0 \wedge y\%r = 0 \wedge y\%x\%r = 0 \wedge forall(k, [r+1, x])((x\%k = 0 \wedge y\%k = 0 \wedge x\%y\%k = 0))) \vee (y = 0 \wedge r = x) \vee (x = 0 \wedge r = y)$
end process

Fig. 2: GCD function SOFL specs.

The post condition of this specification is written as a predicate in DNF. Every element of the predicate in conjunction with the pre condition represent a single functional scenario. For example, FS1 is the first functional scenario of GCD specs:

FS1: $x \geq 0 \wedge y \geq 0 \wedge x > 0 \wedge y > 0 \wedge x \geq y \wedge x\%r = 0 \wedge y\%r = 0 \wedge x\%y\%r = 0 \wedge forall(k, [r+1, y])((x\%k = 0 \wedge y\%k = 0 \wedge x\%y\%k = 0))$.

There are two key parts in FS1: test condition and definition condition. The test condition of a functional scenario is the conjunction between its guard condition and the pre condition. The guard condition is the part of FS1 without the pre condition that only involves input variables. For example the guard condition in FS1, called G1 is the following:

G1: $x > 0 \wedge y > 0 \wedge x \geq y$,

and the test condition of FS1, called T1 is the following:

T1: $x \geq 0 \wedge y \geq 0 \wedge x > 0 \wedge y > 0 \wedge x \geq y$.

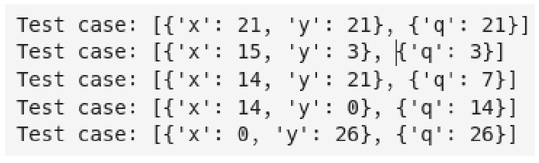
The test condition is used to generate the data for input variables. The definition condition is the part of FS1 that involves one or more output variables, and it is used to generate the expected output. For example the definition condition of FS1, called D1 is defined as follows:

D1: "$x\%r = 0 \wedge y\%r = 0 \wedge x\%y\%r = 0 \wedge forall(k, [r+1, y])((x\%k = 0 \wedge y\%k = 0 \wedge x\%y\%k = 0))$".

B. Decomposition of functional scenarios

When a functional scenario uses '$\geq$", '$\leq$' or '$\neq$' operators in the guard condition then it is divided into two scenarios. For example FS1 has in its guard condition the predicate $x \geq y$, which is decomposed into two functional scenarios with $x > y$ in one and $x = y$ in the other.

The tool decomposes each functional scenario to create a list called extended functional scenarios (EFS). Each element of EFS is used to generate a test case. The values of the input variables are assigned using the test condition with the PSO algorithm and the values for the output variables are assigned using the definition condition with the PSO algorithm. The tool return the following test suite:



```
Test case: [{'x': 21, 'y': 21}, {'q': 21}]
Test case: [{'x': 15, 'y': 3}, {'q': 3}]
Test case: [{'x': 14, 'y': 21}, {'q': 7}]
Test case: [{'x': 14, 'y': 0}, {'q': 14}]
Test case: [{'x': 0, 'y': 26}, {'q': 26}]
```

Fig. 3: First test suite generated with EFS.

C. The PSO algorithm as a test case generator

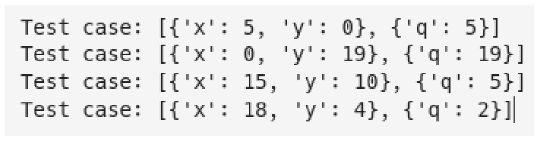
The PSO algorithm is used to generate data that satisfy a given predicate, by minimizing the value of a cost function. The algorithm is executed in two parts of the process. It is used first to generate a test case for each element of EFS, this means that for every element of EFS the PSO algorith is executed once. Then it is used to generate a test case for each predicate mutation, here it is used twice per each element of the mutated predicates, one using the test condition and the other using the definition condition.

D. Mutating a functional scenario

In order to generate a mutant, a mutation operation must be used. This mutation operation change the syntax of a predicate (or functional scenario). The tool uses a mutation operation that changes the relational operator in a predicate, for example replacing a "$>$" operator for the "$=$" operator.

Applying this mutation operation to the guard condition of a functional scenario generate a new guard condition. This guard condition is used by the PSO algorithm to generate data that satisfy it. Then this new data is used to search for the guard condition of the functional scenario that is satisfied. Then, the mutated guard condition is written in conjunction with the post condition and with the definition condition of the functional scenario that was satisfied earlier. This new conjunction is a functional scenario mutant.

Each functional scenario mutant is used with the PSO algorithm to add more test cases to the test suite previously generated with EFS. This test suite is illustrated in Figure 4.



```
Test case: [{'x': 5, 'y': 0}, {'q': 5}]
Test case: [{'x': 0, 'y': 19}, {'q': 19}]
Test case: [{'x': 15, 'y': 10}, {'q': 5}]
Test case: [{'x': 18, 'y': 4}, {'q': 2}]
```

Fig. 4: Test suite generated with functional scenarios mutants.

E. Python test file

The test suite generated with EFS and the functional scenario mutants is translated into a python test file. This python test file is used with its corresponding program (also in python) by MutPy to assess the quality of the test suite.

V. Results

The tool described in our work was proved using 5 examples implemented in Python language. Three examples are used by Jorgensen [8] (triangle, commission, and next date) and two are used by Liu [9] (mod and gcd). Table I presents the comparison between two test suites for each example: T_1 and T_2. T_1 is the test suite generated by our tool and T_2 is created by generating a test case for each functional scenario in the original formal specification. Both test suites were generated using the PSO algorithm as a test case generator. Table I shows the number of test cases of both test suites, denoted as $|T_1|$ and $|T_2|$ respectively. The mutation scores achieved by each test suite were calculated using MutPy, and they are denoted as MS(T_1) and MS(T_2) respectively. It can be seen that T_2 achieve an average mutation score of 0.96, while T_1 achieve an average mutation score of 1.

Name	MS(T_1)	$\|T_1\|$	MS(T_2)	$\|T_2\|$
Next Date	1	61	0.95	13
Mod	1	6	1	3
Triangle	1	59	0.93	8
GCD	1	9	1	4
Com	1	11	0.93	3

Table I: Mutation score results in examples.

VI. Conclusion and future work

Our work shows the effectiveness of the PSO algorithm as a test case generator, as it is the first time using the PSO algorithm with a formal language for testing purposes. The algorithm was capable of finding a test case for any functional scenario. Our tool is capable not only to generate a test suite automatically but generate it in a formal and effective fashion from SOFL specifications. The test suite can be enhanced by adding test cases generated from functional scenario mutants, generating a test suite that achieves a mutation score of 1, as shown in our results.

A comparison between our tool and other tools is considered for future work, as well as an enhancement of the tool.

References

[1] M. W. W. Matt Staats and M. P. Heimdahl. 2011. “Better testing through oracle selection,” in ICSE ’11.

[2] Rashmi Rekha Sahoo, Mitrabinda Ray. 2020. ”PSO based test case generation for critical path using improved combined fitness function,” Journal of King Saud University - Computer and Information Sciences.

[3] S. Liu and S. Nakajima. 2010. “A decompositional approach to automatic test case generation based on formal specifications,” Fourth IEEE International Conference on Secure Software Integration and Reliability Improvemet.

[4] A. P. Mathur. 2013. ”Foundations of software testing,” Pearson.

[5] Shaoying Liu and Yong Sun. 1995. ”Structured methodology+object-oriented methodology+formal methods: methodology of SOFL,” proceedings of First IEEE International Conference on Engineering of Complex Computer Systems. pp. 137-144.

[6] Cencen Li, Mo Li, Shaoying Liu, Shin Nakajima. 2012. ”Structured Object-Oriented Formal Language,” Lecture Notes in Computer Science 7787.

[7] J. Kennedy and R. Eberhart. 1995. ”Particle swarm optimization,” Proceedings of ICNN’95 - International Conference on Neural Networks, pp. 1942-1948 vol.4.

[8] Paul C. Jorgensen. 2013. ”Software Testing: a Craftman’s approach,” CRC Press, fourth edition.

[9] R. Wang, Y. Sato and S. Liu. 2019. ”Specification-based Test Case Generation with Genetic Algorithm,” IEEE Congress on Evolutionary Computation (CEC), pp. 1382-1389.

[10] Aichernig B.K., Lorber F., Ničković D.. 2013. ”Time for Mutants — Model-Based Mutation Testing with Timed Automata,” In: Veanes M., Viganò L. (eds) Tests and Proofs. Lecture Notes in Computer Science, vol. 7942. Springer, Berlin, Heidelberg.

[11] Fraser, Gordon and Wotawa, Franz. 2006. ”Using Model-Checkers for Mutation-Based Test-Case Generation, Coverage Analysis and Specification Analysis,” 16. 10.1109/ICSEA.2006.75.

[12] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, Mark Blackburn. 2014. ”NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications,” Vol. 95, pp. 275-297, ISSN 0167-6423,

[13] Larsen, K., Lorber, F., Nielsen, B., and Nyman, U. 2017. ”Mutation-Based Test-Case Generation with Ecdar,” IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp 319-328.

[14] Wang, Rong, Yuji Sato, and Shaoying Liu. 2021. ”Mutated Specification-Based Test Data Generation with a Genetic Algorithm,” Mathematics 9, no. 4: 331.

[15] Nayak, N., and Mohapatra, D. P.. 2010. ”Automatic Test Data Generation for Data Flow Testing Using Particle Swarm Optimization,” Contemporary Computing, 1–12.