Using Genetic Algorithms in Test Data Generation: A Critical Systematic Mapping

DAVI SILVA RODRIGUES, School of Arts, Sciences and Humanities, University of São Paulo
MÁRCIO EDUARDO DELAMARO, Mathematics and Computer Science Institute, University of São Paulo
CLÉBER GIMENEZ CORRÊA and FÁTIMA L. S. NUNES, School of Arts, Sciences and Humanities, University of São Paulo and School of Engineering, University of São Paulo

41

Software testing activities account for a considerable portion of systems development cost and, for this reason, many studies have sought to automate these activities. Test data generation has a high cost reduction potential (especially for complex domain systems), since it can decrease human effort. Although several studies have been published about this subject, articles of reviews covering this topic usually focus only on specific domains. This article presents a systematic mapping aiming at providing a broad, albeit critical, overview of the literature in the topic of test data generation using genetic algorithms. The selected studies were categorized by software testing technique (structural, functional, or mutation testing) for which test data were generated and according to the most significantly adapted genetic algorithms aspects. The most used evaluation metrics and software testing techniques were identified. The results showed that genetic algorithms have been successfully applied to simple test data generation, but are rarely used to generate complex test data such as images, videos, sounds, and 3D (three-dimensional) models. From these results, we discuss some challenges and opportunities for research in this area.

CCS Concepts: • **Computing methodologies → Genetic algorithms**; • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: Test data generation, test case generation, software testing, genetic algorithms, evolutionary test, scoping study, systematic mapping

ACM Reference format:
Davi Silva Rodrigues, Márcio Eduardo Delamaro, Cléber Gimenez Corrêa, and Fátima L. S. Nunes. 2018. Using Genetic Algorithms in Test Data Generation: A Critical Systematic Mapping. *ACM Comput. Surv.* 51, 2, Article 41 (May 2018), 23 pages.
https://doi.org/10.1145/3182659

Authors' addresses: D. S. Rodrigues, School of Arts, Sciences and Humanities, University of São Paulo, Av. Arlindo Béttio, 1000, São Paulo, SP, Brazil; email: davi.rodrigues@usp.br; M. E. Delamaro, Mathematics and Computer Science Institute, University of São Paulo, Av. Trabalhador São-carlense, 400, São Carlos, SP, Brazil; email: delamaro@icmc.usp.br; C. G. Corrêa, School of Arts, Sciences and Humanities, University of São Paulo, Av. Arlindo Béttio, 1000, São Paulo, SP, Brazil, School of Engineering, University of São Paulo, Av. Prof. Luciano Gualberto, 158, São Paulo, SP, Brazil; email: cleber.gimenez@usp.br; F. L. S. Nunes, School of Arts, Sciences and Humanities, University of São Paulo, Av. Arlindo Béttio, 1000, São Paulo, SP, Brazil, School of Engineering, University of São Paulo, Av. Prof. Luciano Gualberto, 158, São Paulo, SP, Brazil; email: fatima.nunes@usp.br.

https://doi.org/10.1145/3182659

1 INTRODUCTION

Information systems are present in various sectors of our society and, for this reason, system quality becomes fundamental. Software engineering enables the development of more reliable and less error-prone systems by means of verification, validation, and testing (VV&T) activities (Sommerville 2007).

The main goal of testing is to reveal software faults (Myers 1979). Ideally, software testing activity should cover the entire input domain to execute the program and to verify the outputs. However, this strategy is not feasible due to domain cardinality and makes software testing activities cost prohibitive. Therefore, to use the test data that reveal the largest number of defects within the minimum possible time and effort is essential.

Approximately, half of the entire development cost is attributed to software testing activities (Ali et al. 2010). By automating the selection of program input domain elements that meet test criteria (i.e., generating test data), it is possible to reduce costs.

Metaheuristics are search and optimization techniques capable of finding optimal solutions for complex problems (Ali et al. 2010). Since test data generation can be seen as a search problem in which the search space is the program input domain (Dave and Agrawal 2015), metaheuristics are frequently used to generate test data. Genetic algorithms (GAs) are one of the most used metaheuristics in test data generation (Bouchachia 2007).

GAs have been a trend in recent studies that aim to generate test data; Gupta and Rohil (2013) used GAs to generate test data focused on code coverage for object-oriented systems; Arora and Sinha (2014) developed a test data generation technique based on state machines, in which a GA with variable length chromosomes generates the state machines.

Some previous reviews presented generic analysis about metaheuristics or specific topics related to test data generation. Afzal et al. (2009) analyzed various studies that used numerous metaheuristics for test data generation—the test data generated is specific for non-functional criteria (execution time, quality, security, and usability), which are especially relevant for embedded systems; Ali et al. (2010) analyzed the main metrics used to evaluate the efficiency of various metaheuristics in test data generation and highlighted the possible cost reduction in software testing; Dave and Agrawal (2015) conducted a survey of the studies that use metaheuristic search algorithms and mutation analysis to generate test data.

Unlike the aforementioned reviews, this article presents a systematic mapping aiming at: (1) providing a broad overview of the use of GAs in test data generation and (2) providing a deeper analysis about some aspects of such a use. Besides providing a broad overview of the topic area discussed, this systematic mapping aims to contribute to research into test data generation using GAs, by presenting analysis of the studies selected from different points of views. The techniques proposed by authors of the selected studies were categorized according to three criteria: (1) the most significant adaptations made by the authors, (2) the evaluation metrics used, and (3) the software testing techniques (structural, functional, or mutation testing) for which test data were generated.

From the perspective of the metaheuristics analyzed, the study presented herein is more specific than systematic literature reviews (SLRs) conducted by Afzal et al. (2009) and Ali et al. (2010), and than the survey conducted by Dave and Agrawal (2015), since our research focuses on GAs. However, the SLRs and the survey previously published were limited to a subset of studies: Afzal et al. (2009) only reviewed studies about non-functional test criteria, Ali et al. (2010) only analyzed evaluation metrics, and Dave and Agrawal (2015) analyzed only studies that used mutation testing and metaheuristics to generate test data.

The overall structure of this article takes the form of seven sections, including this introduction. Section 2 presents the main test data generation techniques. Section 3 lays out the fundamental

GAs theory. Section 4 describes the review protocol used by this systematic mapping. Section 5 presents the findings of the selected studies, while Section 6 synthesizes these results and discusses gaps and opportunities for research in test data generation using GAs. Section 7 presents the final considerations.

2 TEST DATA GENERATION

Structural-, functional-, and fault-based testing techniques subdivide the program input domain according to specific criteria (Sommerville 2007). Structural testing (white-box testing) is based on the source code directly. In contrast, functional testing (black-box testing) is based on input parameters and outputs. Fault-based software testing subdivides the input domain according to the test data ability of revealing faults (Myers 1979). After determining subdomains, test data that meet test criteria are manually or automatically selected.

Test data generation using random techniques is easy to implement and has lower cost. Test data are randomly generated until test criteria are met. That said, infeasible execution paths may not be detected and there is no guarantee of generating test data that have greater probability of revealing more faults (Ali et al. 2010).

Symbolic execution can also be used to generate test data. This technique produces algebraic expressions as representations of program paths. Symbolic expressions of program output variables are obtained in terms of input variables and path execution conditions. There are limitations, though. Loops, for instance, may produce infinite execution paths. In spite of being costly compared to random generation, symbolic execution is more effective (Cadar and Sen 2013).

Dynamic execution is based on the real program execution and, therefore, does not have the limitations of symbolic execution in test data generation. Real test data are used instead of symbolic expressions: program execution flow is monitored and, then, input variables that cause errors are isolated (Xibo and Na 2011).

Test data can also be generated using fault-based test criteria such as mutation analysis (Myers 1979). In mutation analysis, faults are seeded in the program (according to mutation operators). The programs seeded with faults are called mutants (Ali et al. 2010). If the test data reveal faults, the mutant is "killed." The percentage of mutants "killed" by test data is called mutation score (Dave and Agrawal 2015).

Additionally, search and optimization techniques can be used in test data generation. These techniques consider test data generation a search in a large solution space. Each solution has a different fitness for the problem, given by the fitness function (Ali et al. 2010). In recent years, metaheuristics have been the most used search-based techniques for test data generation (Dave and Agrawal 2015).

Metaheuristics are search and optimization techniques that find optimal (or near-optimal) solutions for problems that have large complex search spaces (Ali et al. 2010). Test data generation can be seen as a search problem in which the search space is the entire program input domain. The desired solutions are those that meet test criteria and that reveal software faults (Dave and Agrawal 2015). Tabu search, simulated annealing, particle swarm optimization, ant colony optimization, memetic algorithm, artificial immune systems algorithm, bacteriological algorithms, and GAs are examples of metaheuristics (Bouchachia 2007).

3 GENETIC ALGORITHMS

GAs, proposed by Holland (1975), simulate Darwin's biological evolution process in the context of search and optimization of complex problems. Each candidate solution is considered an individual. Individuals constitute populations. In each generation, individuals reproduce and mutate. Only the fittest individuals survive. Reproduction and mutation mechanisms ensure genetic diversity

and, consequently, population diversity. In spite of the randomness of mutation, historical pieces of information are taken into consideration along the algorithm execution due to the crossover operator. This means that the crossover operator can maintain traits of previous generations in the new ones. These are the characteristics that enable GAs to efficiently find solutions in large solution spaces (Goldberg 1989).

Mitchell (1999) states that there is no precise definition of GAs, even though all implementations share the same basic components: population, selection, reproduction, and mutation operators. In GAs, a population is composed of a large number of individuals and is subject to reproduction and mutation. The following steps describe how a standard GA works (Mitchell 1999):

(1) The initial population of n chromosomes is randomly generated.
(2) A fitness function assigns a fitness value to each individual.
(3) The following steps are repeated until m new chromosomes are generated:
    (a) the selection operator chooses two chromosomes to reproduce,
    (b) the crossover operator combines genetic material from both chromosomes with probability p_c, and
    (c) the mutation operator alters the offspring with probability p_m.
(4) The replacement procedure determines the individuals that will survive to the next generation.
(5) If the termination conditions are not satisfied, go to step 2.

Compared to traditional search and optimization techniques (e.g., hill-climbing, exhaustive search), GAs present fundamental differences (Goldberg 1989):

— GAs use an encoding of parameters (i.e., the test data) instead of using the parameters directly,
— GAs search for solutions in a population of points, instead of searching isolated points,
— GAs use payoff information (i.e., fitness function), instead of derivatives or other auxiliary information, and
— GAs use probabilistic transition rules instead of deterministic ones.

These characteristics are also inherent to other metaheuristics (e.g., particle swarm optimization, simulated annealing); however, in the context of GAs they are presented as key features.

3.1 Population of Candidate Solutions

In GAs, chromosomes are candidate solutions that are usually encoded as strings composed by numbers. Each chromosome can be divided into genes, which are commonly represented by digits and located in specific positions (locus) on the chromosome. Genes encode characteristics or traits of an individual. The different possible traits are called alleles (Mitchell 1999).

There are numerous kinds of candidate solution encodings. Binary, many-character, real-valued, and tree encoding are the most commonly used encodings (Mitchell 1999).

3.2 Genetic Operators

GAs use various operators in the population evolution process. The most widely used operators (selection, crossover, and mutation) will be detailed in the next subsections.

3.2.1 Selection. Selection operator chooses individuals that will participate in reproduction, according to their fitness. A fitness function can differentiate individual ability to solve the problem and, therefore, assigns a fitness value to an individual (Mitchell 1999). The fittest individuals will be

more likely to be chosen to participate in reproduction and, consequently, more likely to produce fitter individuals (Goldberg 1989).

A rigorous selection can result in suboptimal highly fit solutions preventing optimal lower fitness solutions to survive. On the other hand, a weak selection can slow down evolution and increase execution time (Mitchell 1999).

Roulette Wheel selection (Holland (1975)), Boltzmann selection, rank selection, tournament selection, and steady-state selection are the main selection methods used in GAs (Mitchell 1999).

3.2.2 Crossover. A crossover (recombination) operator enables genetic material switching between two individuals and then generates two offspring for the next generation (Goldberg 1989).

In most GA implementations, from the defined crossover point, two individuals are divided into two parts. The first offspring will have the left-side genes of the first chromosome and the right-side genes of the second chromosome. Similarly, the second offspring will have the left-side genes of the second chromosome and the right-side genes of the first chromosome (Goldberg 1989). There are other kinds of crossover operators; e.g. multi-point crossover, uniform crossover, and crossover among more than two individuals (Mitchell 1999).

3.2.3 Mutation. Mutation is usually a secondary operator in GAs and less likely to occur. Crossover is the operator responsible for generating a fitter offspring that inherits the best characteristics from previous generations. Mutation, in turn, is the operator responsible for providing genetic diversity among individuals; this operator is one of the mechanisms that makes GAs less likely to be "stuck" on local optima (Mitchell 1999).

Crossover and mutation probabilities, p_c and p_m, respectively, can be modified during the optimization process to balance the effects of the operators in evolution (Aleti and Grunske 2015; Aleti and Moser 2016). One can thus, for example, set the crossover probability to a higher value in the beginning of the GA execution to explore the solution space. Likewise, in the end of the execution, one can set the mutation probability to a higher value to avoid local optima (Mitchell 1999).

4 SYSTEMATIC MAPPING PROCESS

Systematic mapping (also known as scoping study) is a literature review that uses well-defined methods to conduct the search and document the findings. Unlike SLRs, scoping studies aim to provide a broad overview of a topic in a research area. A systematic mapping must classify evidence presented by primary studies and answer research questions (Kitchenham and Charters 2007). Secondary studies (i.e., SLRs and systematic mappings) are excluded in scoping studies, as only primary studies can present data that help to answer the research questions.

This systematic mapping intends to provide a scenario of the state of the art research in test data generation by means of GAs. A search was conducted in December 2016 in the following databases: IEEE Xplore Digital Library (2017), ACM Digital Library (2017), Scopus (2017), and Science Direct (2017). The same search string was used in all of the databases: ("GA" OR "evolutionary test") AND ("test case generation" OR "test data generation").

The following research question guided this systematic mapping:

—What are the main test data generation techniques that use GAs?

Inclusion (I) and exclusion (E) criteria were defined to select primary studies. The selection process included studies that met at least one inclusion criterion and no exclusion criteria. The inclusion and exclusion criteria are listed below.

I1 Studies that present test data generation techniques by means of GAs.
I2 Studies that show advantages and disadvantages of using GAs in test data generation.

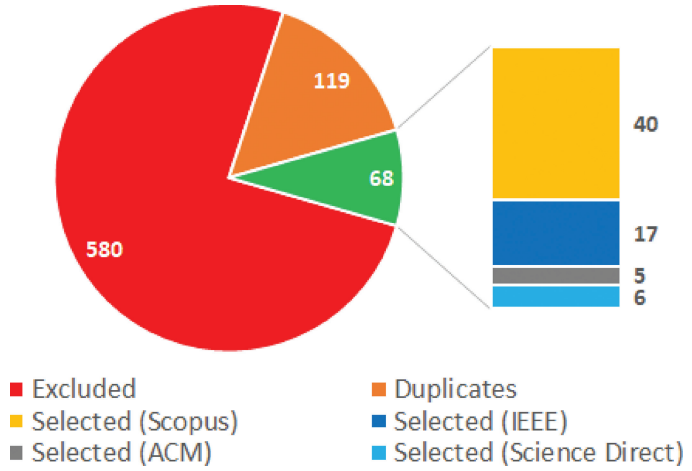

Fig. 1. Returned studies, duplicates, and selected studies.

E1 Studies that are not written in English.
E2 Studies that are not peer-reviewed (e.g., book chapters).
E3 Studies that present literature reviews.
E4 Studies that are unavailable.

Figure 1 shows the number of primary studies selected in each database. Overall, the search string returned 767 studies, of which 119 were duplicates and 580 were excluded. The selected studies are distributed among the databases in this way:

—40 studies (59%) in Scopus,
—17 studies (25%) in IEEE Xplore Digital Library,
—6 studies (9%) in Science Direct, and
—5 studies (7%) in ACM Digital Library.

The next phase of the systematic mapping, which selected 68 studies, is summarized in Figure 2. After exclusion of the duplicate studies, the remaining results were included or excluded according to the following criteria:

—Secondary studies, book chapters, studies not written in English, studies unavailable, or unrelated to the research question were excluded;
—Studies that met no exclusion criteria and presented test data generation techniques by means of GAs or showed their advantages and disadvantages were selected.

A data extraction form was designed to record relevant data from the primary studies selected: the software testing technique, the evaluation metrics used, the format used to represent the program, the country where the research was carried out, and a category related to the adaptations that are proposed.

5 RESULTS

5.1 Included Studies Overview

Tables 1, 2, and 3 present the 68 primary studies included, grouped by type of test considered in each study, as well as part of the extracted data: the corresponding category and evaluation

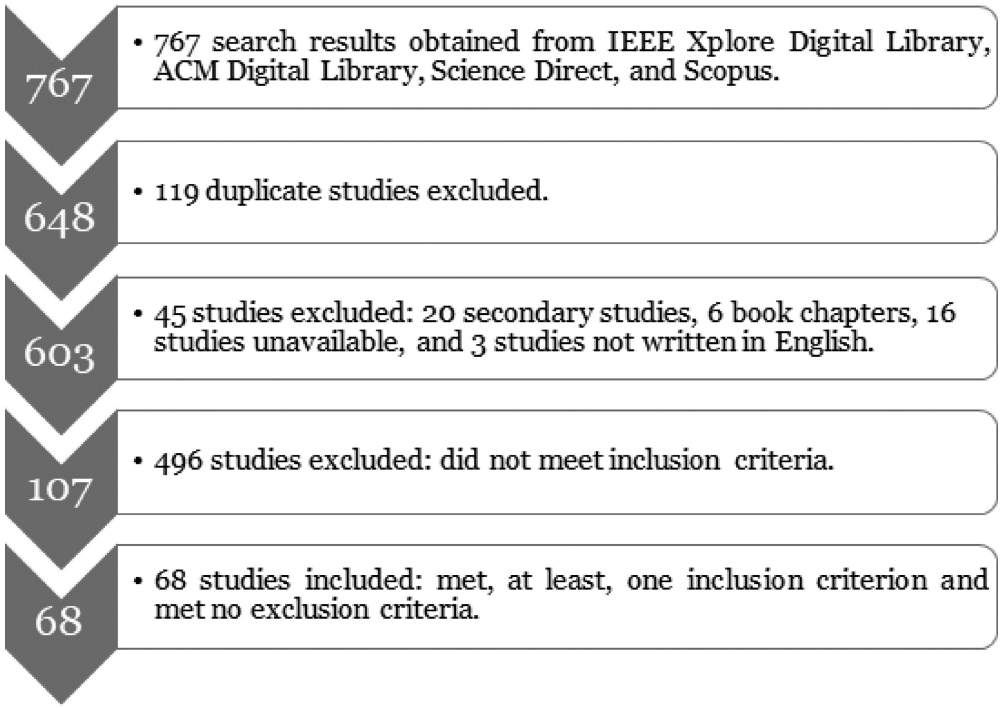

Fig. 2. Summary of the systematic mapping process.

Table 1. Primary Studies Included in the Extraction Phase of the Systematic Mapping—Functional Testing

Category	#	Study	Evaluation Metrics
Fitness function	1	(Betts and Petty 2016)	Maximum lateral deviation
	2	(Derderian et al. 2011)	State coverage, number of generations
	3	(Huang et al. 2010)	Viable test data coverage
	4	(Núñez et al. 2013)	State coverage
	5	(Sharma et al. 2012)	Number of generations, execution time
	6	(Vudatha et al. 2011)	Number of generations
Hybrid algorithm	7	(Alesio et al. 2015)	Execution time
	8	(Lin et al. 2012)	Combinations coverage
Population	9	(Fischer and Tönjes 2012)	Ratio between valid and invalid test data
	10	(Zhou et al. 2014)	Number of generations
Representation of individuals	11	(Alsmadi 2010)	Number of errors revealed, execution time
	12	(Arora and Sinha 2014)	Number of errors revealed
	13	(Lefticaru and Ipate 2007)	Path coverage
	14	(Peng and Lu 2011)	State coverage

metrics. The studies present techniques to generate test data for functional (Table 1), mutation (Table 2), and structural (Table 3) testing.

Every GA implementation found in the studies contains the basic components presented in Section 3 (i.e., population, selection, reproduction, and mutation operators). However, GA components and the problem-specific adaptations are highly related and, therefore, hard to decouple. Thus, the

Table 2. Primary Studies Included in the Extraction Phase of the Systematic Mapping—Mutation Testing

Category	#	Study	Evaluation Metrics
Fitness function	15	(Hanh et al. 2016)	Mutation score
	16	(Khan and Amjad 2015)	Mutation score
	17	(Rani and Suri 2015)	Number of mutants
	18	(Yao et al. 2015b)	Statement coverage, execution time, Mutation score
Hybrid algorithm	19	(Hanh et al. 2014)	Mutation score, execution time

Table 3. Primary Studies Included in the Extraction Phase of the Systematic Mapping—Structural Testing

Category	#	Study	Evaluation Metrics
Architecture	20	(Aleti and Grunske 2015)	Branch coverage
	21	(El-Serafy et al. 2015)	Number of generations, execution time
	22	(Manikumar et al. 2016)	Branch coverage, number of generations
	23	(Pachauri and Srivasatava 2015)	Number of generations, path coverage
Fitness function	24	(Ahmed and Ali 2015)	Path coverage, number of generations
	25	(Ahmed and Hermadi 2008)	Number of generations
	26	(Aleb and Kechid 2013)	Number of generations, number of individuals
	27	(Cao et al. 2009)	Number of generations, execution time
	28	(Chen and Zhong 2009)	Number of individuals, execution time
	29	(Ghiduk et al. 2007)	Requirements coverage, number of individuals, execution time, number of generations
	30	(Ghiduk and Girgis 2010)	Statement coverage, number of generations
	31	(Hermadi and Ahmed 2003)	Number of generations
	32	(Jin et al. 2011)	Number of generations
	33	(Khan et al. 2016)	Path coverage
	34	(Lakhotia et al. 2007)	Branch coverage
	35	(Mann et al. 2016)	Branch coverage
	36	(Parthiban and Sumalatha 2013)	Branch coverage
	37	(Pinto and Vergilio 2010)	Number of generations, execution time
	38	(Shimin and Zhangang 2011)	Execution time
	39	(Shuai et al. 2015)	Execution time
	40	(Soltani et al. 2016)	Number of reproduced bugs
	41	(Sun and Jiang 2010)	Path coverage
	42	(Varshney and Mehrotra 2016)	Number of generations, Execution time
	43	(Xibo and Na 2011)	Path coverage
	44	(Yao et al. 2015a)	Execution time
	45	(Zhang and Gong 2010)	Execution time, number of generations
Hybrid algorithm	46	(Bouchachia 2007)	Number of generations, condition coverage
	47	(Fan et al. 2015)	Number of individuals, decision coverage
	48	(Garg and Garg 2015)	Number of individuals
	49	(Ji and Sun 2012)	Number of generations, path coverage
	50	(Kifetew et al. 2013)	Branch coverage
	51	(Mayan and Ravi 2014)	Execution time, code coverage
	52	(Miller et al. 2006)	Branch coverage, statement coverage
	53	(Tan et al. 2009)	Path coverage

(Continued)

Table 3. Continued

Category	#	Study	Evaluation Metrics
Population	54	(Alshraideh et al. 2011)	Execution time, number of generations
	55	(Chen and Zhong 2008)	Path coverage, execution time
	56	(Deepak and Samuel 2009)	Number of generations, execution time, path coverage
	57	(Gong et al. 2011)	Number of fitness evaluations
	58	(Tian and Gong 2016)	Number of fitness evaluations, execution Time
	59	(Yang et al. 2016)	Branch coverage, number of generations, execution time
	60	(Yao and Gong 2014)	Path coverage, execution time
	61	(Zhang et al. 2010)	Execution time
	62	(Zhang et al. 2015)	Path coverage, execution time, number of individuals
Representation of individuals	63	(Khandelwal and Tomar 2015)	Path coverage
	64	(Koleejan et al. 2015)	Execution time, code coverage
	65	(Lijuan et al. 2012)	Viable branches coverage
	66	(Liu et al. 2013)	Path coverage
	67	(Silva and van Someren 2010)	Number of errors revealed, execution time
	68	(Tonella 2004)	Branch coverage

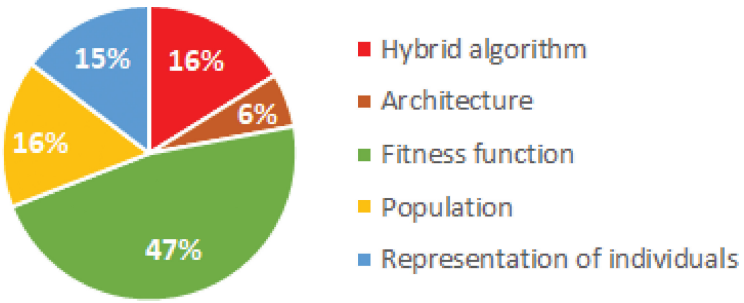

Fig. 3. Distribution of the studies included by category.

studies were categorized according to the adaptation that was highlighted by authors, considering that the components aforementioned are related both to the GA and to the problem under analysis.

To provide a better comprehension of the results we found, we divided the studies included into five categories, according to the most significant adaptation presented by the authors: fitness function (Section 5.3), population (Section 5.4), representation of individuals (Section 5.5), and architecture (Section 5.6). Lastly, hybrid algorithms are presented in Section 5.7. Figure 3 shows the distribution of the studies presented in Tables 1, 2, and 3 among the categories described. Each category was divided into up to three subcategories, according to the software testing technique for which test data were generated: structural, functional, or mutation testing.

Data extracted from the studies show that most of them were developed in China (37%) and India (22%). The 68 studies were published between 2003 and 2016 (Figure 4) and most of them were published in 2015.

5.2 Use of Genetic Algorithms in Test Data Generation

The majority of the studies included (72%) test data generation for structural testing, as fewer studies (21%) aim to generate test data for functional testing. The minority of the studies (7%) generated test data for mutation testing.

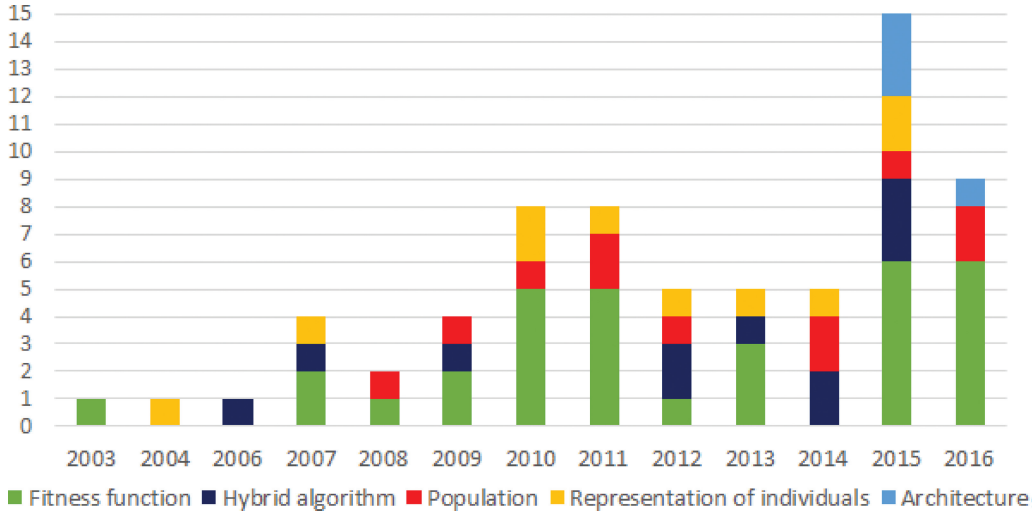

Fig. 4. Distribution of the studies included by publication year and by category.

Most test data generation techniques for structural testing extract a CFG (control-flow graph) (Aleti and Grunske 2015; Chen and Zhong 2008; Khandelwal and Tomar 2015; Lakhotia et al. 2007; Sun and Jiang 2010), a DFG (dataflow graph—a CFG extension that includes information regarding variable uses and definitions) (Khan et al. 2016; Pinto and Vergilio 2010; Varshney and Mehrotra 2016; Zhang et al. 2015), or other graphs (Jin et al. 2011; Mayan and Ravi 2014; Miller et al. 2006) from the source code and use them as a representation of the software. Many studies represented the programs under test with the source code itself (Bouchachia 2007; El-Serafy et al. 2015; Ghiduk et al. 2007; Hermadi and Ahmed 2003; Lijuan et al. 2012; Mann et al. 2016; Parthiban and Sumalatha 2013; Yang et al. 2016). Only one study proposes a technique based on a variables table (which describes all variables) and a controls table (which describes the program structure) instead of graphs or source code (Aleb and Kechid 2013). Invariably, the initial population is randomly generated (Ahmed and Ali 2015; Deepak and Samuel 2009; Kifetew et al. 2013).

Test data generation techniques for functional testing use state machines (Arora and Sinha 2014; Derderian et al. 2011; Lefticaru and Ipate 2007; Núñez et al. 2013), user interface graphs (Alsmadi 2010), and request dependence graphs of web applications (Peng and Lu 2011). Also, the initial population is randomly generated (Alsmadi 2010; Huang et al. 2010; Lefticaru and Ipate 2007) or generated upon existing test data (Peng and Lu 2011).

Techniques for test data generation for mutation testing use the source code itself instead of graphs, state machines, or other representations. The goal of these techniques is to generate test data capable of "killing" mutants of the software under test (Hanh et al. 2014, 2016; Khan and Amjad 2015; Rani and Suri 2015).

In test data generation (for structural, functional, or mutation testing), GAs (using representations of programs) generate individuals (i.e., test data) that reveal faults in programs. Chromosomal representations of test data encode known specific details of the software. Fitness functions also incorporate known details of the software and assign a fitness value to each individual. Numerous evaluation metrics are used to determine the efficiency and efficacy of test data generation techniques (Section 5.8). These metrics are mostly linked to the chosen program representation.

Figure 5 presents the relationship between program representation and software testing technique. The sizes of the bubbles indicate the number of studies in each category. As above

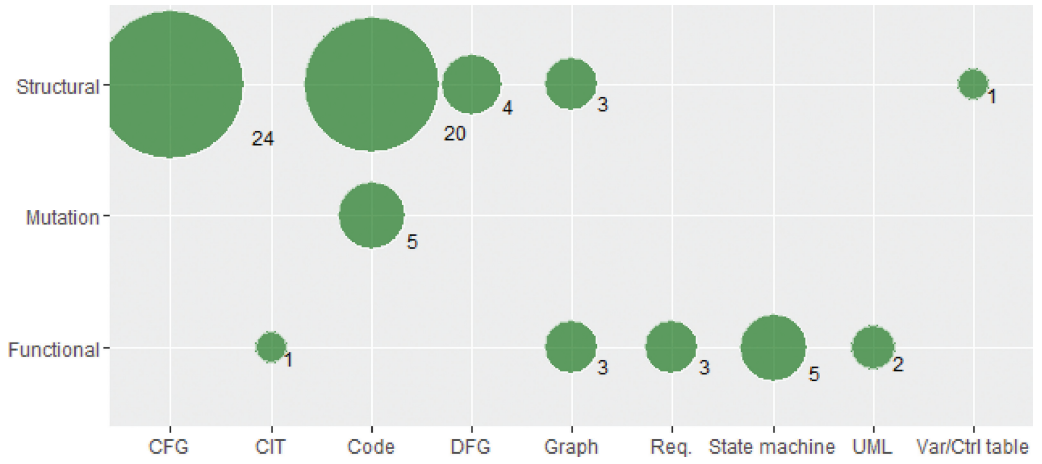

Fig. 5. Relationship between program representations and software testing techniques. Horizontal axis: CFG (control-flow graph), CIT (combination-index table), source code, DFG (dataflow graph), graph (includes CDG—control-dependence graph, EFG—event flow graph, GUI graph, PDGprogram-dependence graph, and RDG—request dependence graph), software requirements, state machine, UML (Unified Modeling Language), variable/control table.

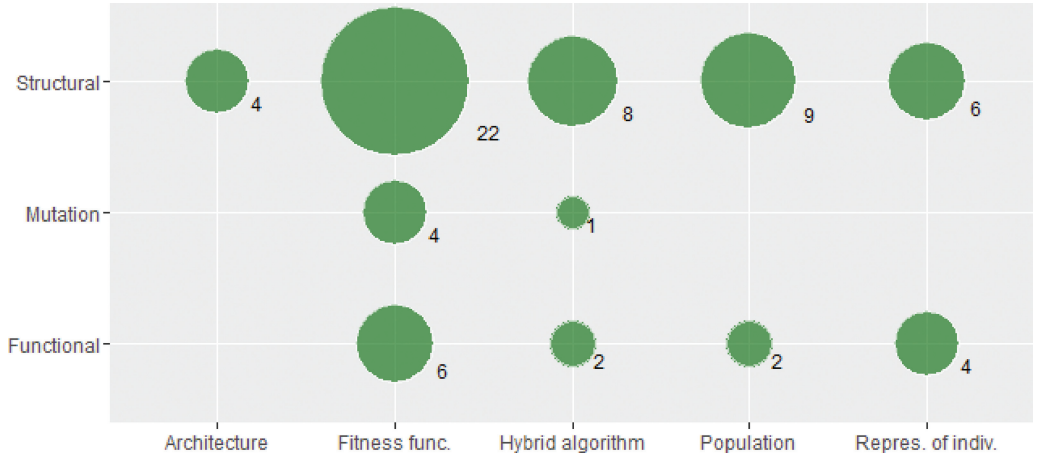

Fig. 6. Relationship between categories of studies and software testing techniques. Horizontal axis: architecture, fitness function, hybrid algorithm, population, and representation of individuals.

mentioned, studies that generate test data for structural testing use mainly CFG and DFG. In contrast, studies that generate test data for functional testing use representations that embody software specifications and requirements. Studies that proposed test data generation techniques for mutation testing use the source code directly.

Figure 6 shows the study distribution among software testing techniques and study categories. The sizes of the bubbles also indicate the number of studies in each category. Some studies proposed test data generation techniques for functional testing, but none of them focused on the architecture. In test data generation for structural testing, every adaptation was highlighted by at

least one study. All test data generation techniques for mutation testing described fitness function adaptations. Architecture was the least mentioned aspect and the fitness function was the most cited adaptation.

Adaptations to fit the problem of interest are inherent to the GA design. The authors of the studies presented in the next subsections highlighted in their articles adaptations of specific aspects that were the focus of their work and that enabled satisfactory results in test data generation for structural, functional, and mutation testing.

5.3 Fitness Function

The fitness function is the most cited adaptation in 32 studies (47%). Out of these studies, a total of 22 studies generated test data for structural testing, 6 generated test data for functional testing, and 4 generated test data for mutation testing.

5.3.1 Functional Testing. Betts and Petty (2016) proposed the use of a GA to generate test data for drone flight control software. The fitness function is based on the maximum lateral deviation from a determined trajectory. Thus, test data generated by the GA represent the valid trajectories that are harder to maintain.

Núñez et al. (2013) generated test data for real-time systems, which are represented by state machines. The fitness function penalizes test data that do not meet temporal restrictions imposed by this type of system. Derderian et al. (2011) introduced a specific fitness function for finite state machines that analyzes the probability of a state transition, as well as the complexity of applicable temporal constraints.

Vudatha et al. (2011) presented a technique that generates test data for embedded systems using their specifications. Huang et al. (2010) embedded the test data viability in the fitness function: test data, which represent actions on a graphical user interface (GUI), are executed and penalized by a factor in the fitness function if they are infeasible (i.e., if they generate impossible actions on the GUI). Sharma et al. (2012) included information related to the functional requirement constraints in the fitness function. Hence, the fitness function penalizes test data that do not meet all the constraints.

5.3.2 Mutation Testing. In these four studies, the mutation score is used as or related to the fitness function. In the study carried out by Khan and Amjad (2015), initially, test data are randomly generated. Then, the mutation score is calculated. When initial test data do not meet a minimal score, the GA execution begins and the random data are used as the initial population.

Hanh et al. (2016) introduced a fitness function specifically to assess the error propagation along program path executions and to assess the test data capability of killing mutants.

Rani and Suri (2015) replaced the fitness function by the execution of mutant programs: if the current population does not kill all mutants (does not reveal all errors), then crossover and mutation operators are used, causing population evolution.

The fitness function of the GA-based technique proposed by Yao et al. (2015b) prioritizes statement coverage and test data that are better distributed in the program input domain.

5.3.3 Structural Testing. The fitness function adaptation to generate test data for structural testing was cited in 21 studies. The authors presented five types of fitness functions: multi-objective (Lakhotia et al. 2007; Pinto and Vergilio 2010; Varshney and Mehrotra 2016; Zhang and Gong 2010), multi-path (Ahmed and Ali 2015; Ahmed and Hermadi 2008; Aleb and Kechid 2013; Cao et al. 2009; Chen and Zhong 2009; Ghiduk and Girgis 2010; Ghiduk et al. 2007; Hermadi and Ahmed 2003; Jin et al. 2011; Sun and Jiang 2010; Xibo and Na 2011; Yao et al. 2015a), single path

(Khan et al. 2016; Mann et al. 2016; Parthiban and Sumalatha 2013; Shimin and Zhangang 2011), based on code coverage (Shuai et al. 2015), and one specific to determine if the test data throws runtime exceptions (Soltani et al. 2016).

Most of these studies propose fitness functions based on the CFG (Ahmed and Ali 2015; Ahmed and Hermadi 2008; Cao et al. 2009; Chen and Zhong 2009; Ghiduk and Girgis 2010; Lakhotia et al. 2007; Pinto and Vergilio 2010; Shimin and Zhangang 2011; Sun and Jiang 2010; Yao et al. 2015a). Other techniques use different program representations, such as DFG (Khan et al. 2016; Pinto and Vergilio 2010; Varshney and Mehrotra 2016), variables/controls table (Aleb and Kechid 2013), CDG (control-dependence graph) (Jin et al. 2011), or the source code itself (Ahmed and Ali 2015; Ghiduk et al. 2007; Hermadi and Ahmed 2003; Mann et al. 2016; Parthiban and Sumalatha 2013; Shuai et al. 2015; Soltani et al. 2016; Xibo and Na 2011; Zhang and Gong 2010). Jin et al. (2011) defined CDG as a CFG extension that adds control dependency information.

Techniques that presented multi-objective fitness functions optimize test data for memory allocation, execution time, and number of errors revealed to maximize the quality of the results of each GA execution (Lakhotia et al. 2007; Pinto and Vergilio 2010). Although the goal was to cover multiple paths, the study carried out by Varshney and Mehrotra (2016) tries to optimize candidate solutions for all-uses and all-defs data flow criteria (the study uses DFG as a program representation).

Similarly, techniques that proposed multi-path coverage aim to efficiently generate test data. Ghiduk and Girgis (2010) defined a fitness function that maximizes the path coverage of a CFG, but only analyzing path dominance. Yao et al. (2015a) proposed a test data generation technique based on two fitness functions: one multi-path, which separates individuals into groups, and another specific function to assign a fitness value to each individual. Hermadi and Ahmed (2003) considered test data distance and violation relative to different paths and embedded both information in the fitness function. Zhang and Gong (2010) proposed a fitness function that assesses whether the test data cover multiple paths and reveal faults.

Mann et al. (2016) introduced a fitness function specific for a single program execution path. Mathematical functions represent conditions in the source code and indicate whether test data meet such conditions. The results of the mathematical functions are assigned to individuals as a fitness value. Khan et al. (2016) presented a technique that uses DFG as a program representation and generates test data that cover all DFG paths.

Soltani et al. (2016) used a GA to reproduce known program errors. The proposed fitness function considers if a line of code that throws runtime exceptions is covered by test data and if the generated stack trace is close to the original. Shuai et al. (2015) proposed a test data generation technique that reveals software vulnerabilities. The fitness function assigns a higher fitness value to the test data that maximizes code coverage without increasing execution costs. The authors used source code rather than CFG or DFG as program representations.

5.4 Population

Population was the adaptation highlighted by 11 studies (16% of the total). Two studies proposed techniques to generate test data for functional testing and nine studies generated test data for structural testing.

5.4.1 Functional Testing. Both techniques that generated test data for functional testing use specifications instead of graphs or source code to represent the programs under test.

Fischer and Tönjes (2012) proposed the micro genetic algorithms (μGAs), which "kill" individuals with the lower fitness value at each specific generation interval (e.g., at each 10 generations). This technique uses the software requirements as program representation.

Zhou et al. (2014) used three populations that evolve independently, but allow individuals to migrate. Two of the best individuals (according to the fitness function) of two populations replace the worst two individuals of a third population. The programs are represented by state machines.

5.4.2 Structural Testing. The authors proposed the use of multiple populations (Alshraideh et al. 2011; Chen and Zhong 2008; Deepak and Samuel 2009; Gong et al. 2011; Tian and Gong 2016; Zhang et al. 2010, 2015), population regeneration (Yang et al. 2016), and individual sharing (Yao and Gong 2014). Most of the nine techniques use CFG as program representations (Alshraideh et al. 2011; Chen and Zhong 2008; Deepak and Samuel 2009; Gong et al. 2011; Tian and Gong 2016; Yao and Gong 2014). The other studies use DFG (Zhang et al. 2015) or the source code (Yang et al. 2016).

The techniques proposed by Chen and Zhong (2008) and Zhang et al. (2015) evolve three independent populations with individuals migrating among them. Two of the best individuals (according to the fitness functions) of two populations replace the worst two individuals of a third population. The technique introduced by Yao and Gong (2014) also uses multiple populations, but they evolve in consonance. The fitness function of each population evaluates individuals of all populations and assigns a global fitness, thus increasing the chances of finding an optimal solution to the problem. Alshraideh et al. (2011) and Deepak and Samuel (2009) presented GAs with random individual migration, regardless of the fitness assigned to an individual by the fitness function.

By using current branch coverage and a population aging factor, Yang et al. (2016) defined a GA that identifies if the current population is a local optimum. If the GA population composes a local optimum, the current population is discarded and a new one is randomly generated.

Gong et al. (2011) and Zhang et al. (2010) solved the problem of multi-path test data generation by dividing the target paths among groups and processing each one as an independent subpopulation. Tian and Gong (2016) proposed a type of co-evolutionary GA that uses multiple independent sub-populations. The best candidates generated by the sub-populations constitute the initial cooperative population.

5.5 Representation of Individuals

Ten studies (15% of the total) highlighted the proposed codifications used in GAs. Four studies proposed techniques to generate test data for functional testing and six studies generated test data for structural testing.

5.5.1 Functional Testing. Lefticaru and Ipate (2007) used UML (Unified Modeling Language) state machine diagrams to abstract the program and adapted the representation of individuals. The population of candidate solutions is composed of lists of input parameters that exercise specific paths in a transitions graph. Arora and Sinha (2014) introduced a test data generation technique for structural testing also based on state machines. The authors defined variable-length chromosomes, with length linked to the number of state transitions.

Alsmadi (2010) encoded GUI elements as binary chromosomes that represent the horizontal or vertical positions of these elements. Additionally, the author proposed a data structure that links two chromosomes to each interface element (one for horizontal and another for vertical positions). Peng and Lu (2011) encoded web requests as chromosomes that have genes composed of two data structures (nodes and pages). These genes are not limited to a binary alphabet and store diverse information of interest.

5.5.2 Structural Testing. The techniques specific for structural testing represent programs with CFG (Khandelwal and Tomar 2015; Liu et al. 2013) or the source code itself (Koleejan et al. 2015; Lijuan et al. 2012; Silva and van Someren 2010; Tonella 2004).

Khandelwal and Tomar (2015) presented an approach for generating test data considering aspect-oriented programs in which the candidate solutions are defined by the exercised paths of the CFG. Liu et al. (2013) used chromosomes with real numbers to maintain the precision of the candidate solutions required by the problem.

Lijuan et al. (2012) defined a new process for generating the initial population: the input domain particularities are considered in the creation of initial solutions. For an integer parameter that usually is a negative number, for example, initial solutions are mostly created with negative values rather than positive ones. Koleejan et al. (2015) proposed a multi-vector representation of the individuals; thus, a chromosome encodes more than one candidate solution.

Silva and van Someren (2010) and Tonella (2004) proposed specific encodings to represent objects and methods: individuals have information such as method return types and valid parameter types.

5.6 Architecture of GA

The architecture was the least focused adaptation cited in the selected studies (6% of the total). All four studies proposed techniques for test data generation for structural testing: two techniques use CFG as a program representation and the other two use the source code.

By creating a new step in the execution, Aleti and Grunske (2015) proposed GAs capable of adjusting their genetic parameters during the optimization process. Manikumar et al. (2016) introduced a test data generation technique based on two GAs: the first generates specific test data for testing the nodes of a CFG and the second uses such data as the initial population. Then, a new population (optimized for the entire CFG) is generated.

El-Serafy et al. (2015) introduced a test data generation technique that uses two GAs. The first generates test data according to modified condition/decision coverage (MC/DC) structural testing criteria. The second considers existing test data as problems to be solved and, therefore, generates new test data to fit to each new problem. Pachauri and Srivasatava (2015) developed a technique based on parallel GAs. All GAs evolve the same population, but each one has a different fitness function.

5.7 Hybrid Algorithms

Eleven (16%) of the included studies combine GAs with other techniques. These studies do not highlight adaptations. Most of these studies generate test data for structural testing and use strings as input data. Studies carried out by Alesio et al. (2015) and Lin et al. (2012) present test data generation techniques for functional testing. Hanh et al. (2014) presented a technique to generate test data for mutation testing.

Some hybrid techniques added new steps to GAs (Bouchachia 2007; Ji and Sun 2012; Kifetew et al. 2013; Lin et al. 2012; Tan et al. 2009). Other studies that proposed hybrid techniques used GA as a step of a longer process (Alesio et al. 2015; Miller et al. 2006) or increased the efficiency of a genetic operator (Fan et al. 2015; Garg and Garg 2015; Mayan and Ravi 2014). All techniques were developed for general purpose systems, except for the technique proposed by Alesio et al. (2015), which was developed for real-time embedded systems.

GAs were combined with simulated annealing techniques (Hanh et al. 2014; Ji and Sun 2012; Tan et al. 2009), coverage table (Miller et al. 2006), combination-index table (CIT) (Lin et al. 2012), constraint programming (Alesio et al. 2015), artificial immune systems (Bouchachia 2007; Hanh et al. 2014; Tan et al. 2009), and singular value decomposition (Kifetew et al. 2013).

Fan et al. (2015) and Mayan and Ravi (2014) replaced the mutation operator by a Tabu search. Garg and Garg (2015) used hill-climbing in the selection operator to obtain better results. Hanh

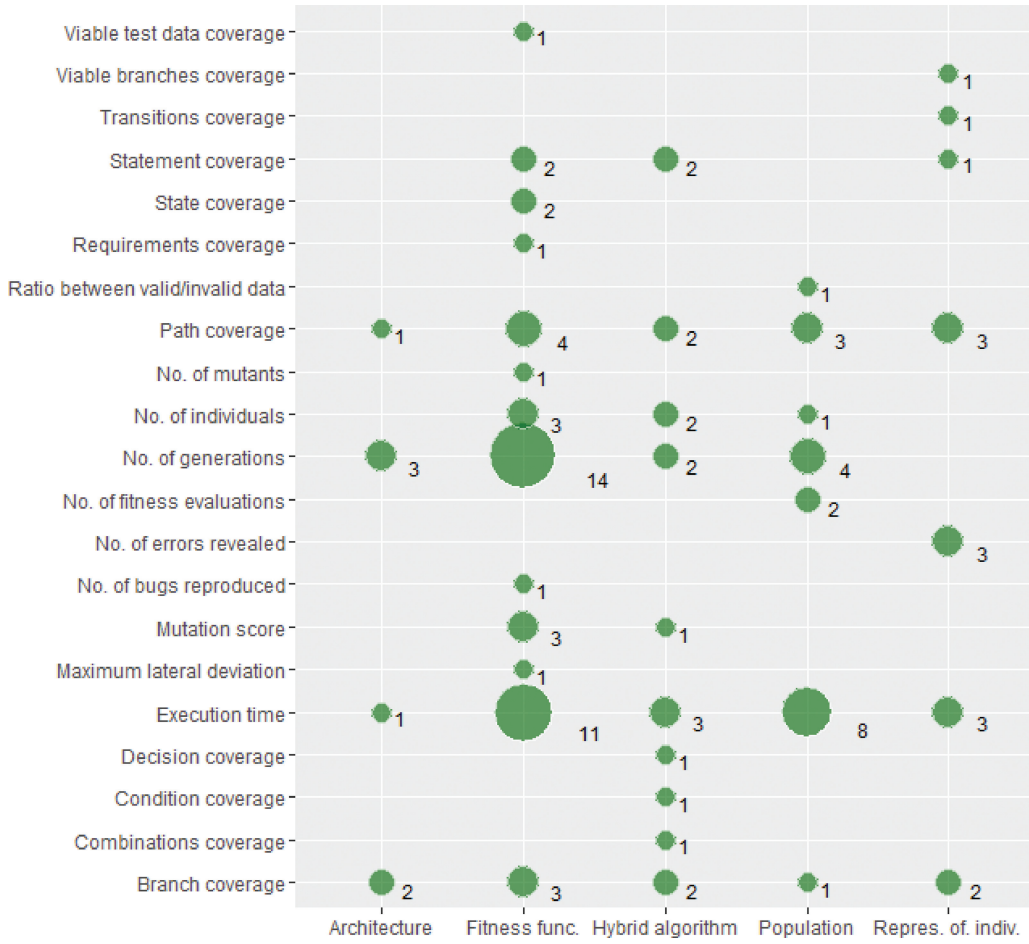

Fig. 7. Relationship between categories of studies and evaluation metrics. Horizontal axis: architecture, fitness function, hybrid algorithm, population, and representation of individuals.

et al. (2014) combined GA, simulated annealing, and artificial immune systems to generate fitter solutions in terms of mutation score.

5.8 Evaluation Metrics for Test Data Generation Techniques

All studies considered in this systematic mapping use specific metrics to evaluate the test data generated. Metrics allow assessing test data quality, feasibility of the time required to generate data, and the effectiveness and efficiency of the techniques proposed when compared to other techniques (e.g., random generation). The evaluation metrics used in each study can be seen in Tables 1, 2, and 3.

Figure 7 shows the relationship between categories of studies and evaluation metrics used in the studies. The most widely used metrics are the number of generations (GA iterations) and the execution time, especially in GAs that highlighted fitness functions adaptations. Path coverage, branch coverage, and execution time are the only evaluation metrics used in all categories.

Independently of the software testing technique for which test data are generated, evaluation metrics are based on the program under test representation and inherent characteristics of GAs (e.g., number of generations and number of individuals).

Evaluation of test data generation techniques for structural testing can be classified into three groups: metrics based on program representation, based on GAs, and based on the software under test.

- *Evaluation Metrics Based on CFG/DFG*: Branch coverage (Aleti and Grunske 2015; Kifetew et al. 2013; Lakhotia et al. 2007; Manikumar et al. 2016), viable branches coverage (Lijuan et al. 2012), path coverage (Ahmed and Ali 2015; Chen and Zhong 2008; Deepak and Samuel 2009; Ji and Sun 2012; Khan et al. 2016), statement coverage (Ghiduk and Girgis 2010; Koleejan et al. 2015; Mayan and Ravi 2014; Miller et al. 2006), decision coverage (Fan et al. 2015), and condition coverage (Bouchachia 2007).
- *Evaluation Metrics Based on GA*: Execution time (Alshraideh et al. 2011; Cao et al. 2009; El-Serafy et al. 2015; Ghiduk et al. 2007; Shimin and Zhangang 2011; Shuai et al. 2015; Zhang et al. 2010), number of individuals generated (Aleb and Kechid 2013; Garg and Garg 2015; Ghiduk et al. 2007), number of generations (Aleb and Kechid 2013; Bouchachia 2007; Cao et al. 2009; Hermadi and Ahmed 2003; Ji and Sun 2012; Zhang and Gong 2010), number of fitness function executions (Gong et al. 2011; Tian and Gong 2016), and number of errors revealed (Silva and van Someren 2010).
- *Evaluation Metrics Based on the Software under Test*: Number of reproduced bugs (Soltani et al. 2016).

The evaluation of test data generation techniques for the functional test was based on CFG path coverage (Lefticaru and Ipate 2007), viable test data (Huang et al. 2010), combinations (Lin et al. 2012), states (Derderian et al. 2011; Núñez et al. 2013), state transitions (Peng and Lu 2011), number of generations (Sharma et al. 2012; Vudatha et al. 2011; Zhou et al. 2014), number of errors revealed (Alsmadi 2010; Arora and Sinha 2014), maximum lateral deviation (Betts and Petty 2016), ratio between valid and invalid test data (Fischer and Tönjes 2012), and execution time (Alesio et al. 2015).

The test data generation techniques for mutation testing used mutation score (Hanh et al. 2014, 2016; Khan and Amjad 2015; Yao et al. 2015b), number of mutants created (Rani and Suri 2015), and execution time (Hanh et al. 2014).

6 DISCUSSION, CHALLENGES, AND OPPORTUNITIES

Over the past 15 years, the number of studies published in the topic of test data generation using GAs has increased. The studies included in this systematic mapping were published between 2003 and 2016. Mostly, the studies of the 2000s seek to affirm the efficiency of GAs to solve the problem of test data generation. The most recent studies, however, recognize GAs as powerful tools in the automation of test data generation and propose refinements, improvements, and new GA applications.

In general, test data generation techniques are most applied to structural software testing. However, independently of the software testing technique, a GA uses the program source code or a representation of the program (CFG, DFG, finite state machine, UML diagrams, and so on), which is frequently related to the technique evaluation metric. An encoding of candidate solutions must be defined (i.e., a chromosomal representation of test data). A fitness function must also be defined to assign a fitness value to each individual of the population. Genetic operators of crossover and mutation are responsible for promoting the genetic diversity (the GA "movement"

among solutions in the search space). These aspects require adaptations in GAs, and generate challenges and opportunities.

6.1 Adaptation to the Problem

GAs need to be adapted to the problem of interest to be effective. Although each of the GA's basic components (population, selection, reproduction, and mutation operators) must be adapted, most of the techniques detailed in previous sections focused on the fitness function, whose adaptation is directly related to the problem.

The need to adapt the components of a GA to the problem of interest has advantages and disadvantages. Among the disadvantages is the need of a deeper knowledge of the problem. Different problems need different adaptations, what can require constant effort of developers to reach good results for specific problems. Despite these questions, similar problems may share one or more components of a GA with a few changes. Thus, the well-defined architecture of GAs facilitates their implementation to promote reuse of components when a suitable technology—such as object-oriented languages—is used. Therefore, the main advantage of this requirement of adapting GA components is that the knowledge is embedded in the implementation (as long as the components are implemented and properly tested). With this, not only one problem, but a whole class of similar problems within a determined domain are attended, minimizing the development effort.

6.2 Complex Data and Large Systems

Information systems have incorporated multimedia data (images, videos, and sounds) and 3D information. Nonetheless, most GA applications in test data generation found in this systematic mapping involve simple input and output domains, such as characters, numbers, and symbols. None of the selected studies addressed test data generation for complex domains, such as the image domain or multimedia data. The chromosomal representations proposed were suitable for simple data, but may not be able to efficiently handle complex test data.

Development of approaches using GAs for generating test data for these complex domain requires, most of the times, serious changes in several components, since all of them have to make sense for such a specific domain. One cannot, for example, execute a crossover operator on an image if the result does not provide another valid image. Similarly, an initial random generation of images can result in invalid individuals, and maybe an initial population should be provided in this class of programs. Thus, operators, as well as fitness functions and other elements, must be projected to work with valid inputs and generate valid outputs. Therefore, systems that have complex data as inputs and outputs constitute an unexplored research opportunity.

Moreover, the studies discussed in this scoping study proposed test data generation techniques for small systems and even implementations of native methods of Java programming language. Thus, these techniques may not have acceptable performance when applied to large systems. Consequently, evaluation of the techniques when applied in large systems as well as development of more efficient techniques for this class of programs are also good opportunities of research.

Even though only small systems with simple test data were considered, the most widely used evaluation metrics in the studies were the execution time and the number of GA generations. This indicates that one of the challenges in test data generation is to automate (efficiently and with an acceptable runtime) the test of large systems that require complex test data. These are real systems present in the industry, which need the help of researchers for testing and validation.

6.3 Execution Time and Evaluation Metrics

GAs may require very long execution times. Some studies proposed parallel execution of several GAs and others presented algorithms with multiple populations. This problem gets worse if

complex domains are considered, as cited before. Future research could propose specific chromosomal representations and fitness functions to optimize execution time, since the fitness function is used several times throughout GA execution and manipulates encoded parameters. For techniques that combine GAs and mutation testing, for example, it is also possible to use mutant reduction without affecting the mutation score, but reducing the execution time (Gong et al. 2017).

Most studies evaluated test data generation techniques based on GAs using metrics such as execution time and the number of generations. As these evaluation metrics were obtained by testing simple programs, they cannot indicate whether a system with complex data can be properly tested by these techniques. The definition of a standard benchmark of programs that use complex data would make evaluation metrics more meaningful.

6.4 Three-Dimensional Virtual Environments

Virtual environments, composed of 3D models, offer the possibility of human-computer interaction in three dimensions and in real-time. Usually, virtual environments simulate real situations (e.g., training, exploration, visualization, entertainment, among others). This type of system has its presence increased in our society due to some factors such as greater access to technologies, popularization of games, and development of devices to improve interaction and provide 3D visualization.

In this way, actions performed by users to achieve a goal and system behaviors are countless. This makes generating inputs and outputs for the composition of test cases a difficult task. Three-dimensional models may have autonomous behaviors (e.g., predefined movements) and are dependent on user actions (e.g. deformation of one model after the collision with an user-manipulated model). In addition, the automation of testing activities must take into account real-time and parallel execution. GAs have the ability of searching a large complex solution space, thus they could find relevant ways of human-computer interaction in virtual environments, involving search in an universe of countless user actions and 3D models behaviors. By indicating the critical actions and behaviors, GAs could facilitate test data generation for virtual reality systems.

7 CONCLUSIONS

GAs have been used to generate test data successfully. These algorithms enable a large solution space to be explored, as they are non-deterministic. Nonetheless, the problem needs to be known in depth and this knowledge needs to be embedded in the components of the GA. It is evident from the papers analyzed in this systematic mapping that there is a deep relationship between GA components and the problem-specific adaptations. It is hard to decouple these aspects, which can make difficult the GA software reuse.

The data presented in this article show that most techniques focus on adapting the fitness function, as this function is highly related to the problem of interest. In addition, considering that GAs have a well-defined structure, it is possible to reuse components for similar problems if, for example, object-oriented languages are used. However, as already stated, reusing some components can be difficult since they are strongly related to the application.

The studies analyzed in this systematic mapping, in their majority, generate test data for structural testing, and use the source code or diagrams derived from it—such as CFGs—to generate the test data. Development of new types of representations as well as the generation of test data for other categories of testing are challenges that characterize opportunities for research.

Another point to stress is that the programs used in the studies analyzed are small, in domains with low complexity. On the other hand, the systems used in the industry are large and use complex data, such as images, videos, and sounds. Such complex data require robust and efficient implementations of GA components. Thus, new research about how to define GA components for these complex domains is necessary. Classical algorithms are not adequate for some types of data. For

example, the random generation of the initial population is enough for data such as numbers and strings, but it can generate invalid data for image domain.

Still related to complex domains, most studies considered the execution time as one of the main metrics of quality assessment of the proposed techniques. For images, videos, sounds, and other complex data, a more meaningful evaluation metric should be provided by the creation of a standard benchmark.

As shown, GA execution requires a very long execution time. Although some solutions such as the parallel execution and multiple populations were already proposed, this issue is still far from being solved. This problem gets worse with complex domains, as mentioned before. Thus, specific chromosomal representations and fitness functions to optimize execution time also constitute points that deserve deeper exploration by researchers.

Lastly, it is important to mention that systems in 3D environments with real-time interaction are gradually more present in our society, mainly due the popularization of games. Generating inputs and outputs to compose test cases is a difficult task, especially because such data are generated from actions of users. Since GAs have the ability of searching a large complex solution space, they could be used to find relevant ways of interacting within a large set of possibilities of interaction in these kinds of systems, indicating critical actions and behaviors and, then, generate test data using this information.

Generating complex test data with a feasible execution time and, at the same time, using chromosomal representations that guarantee the integrity of test data is the great challenge of research in test data generation by means of GAs.

REFERENCES

ACM Digital Library. 2017. Retrieved December 2016 from http://dl.acm.org.

Wasif Afzal, Richard Torkar, and Robert Feldt. 2009. A systematic review of search-based testing for non-functional system properties. *Inform. Softw. Technol.* 51, 6 (2009), 957–976.

Moataz A. Ahmed and Fakhreldin Ali. 2015. Multiple-path testing for cross site scripting using genetic algorithms. *J. Syst. Archit.* (2015).

Moataz A. Ahmed and Irman Hermadi. 2008. GA-based multiple paths test data generator. *Comput. Oper. Res.* 35, 10 (2008), 3107–3124. Part Special Issue: Search-based Software Engineering.

Nassima Aleb and Samir Kechid. 2013. Automatic test data generation using a genetic algorithm. In *International Conference on Computational Science and Its Applications*. Springer, 574–586.

Stefano Di Alesio, Lionel C. Briand, Shiva Nejati, and Arnaud Gotlieb. 2015. Combining genetic algorithms and constraint programming to support stress testing of task deadlines. *ACM Trans. Softw. Engi. Methodol.* 25, 1 (2015), 4.

Aldeida Aleti and Lars Grunske. 2015. Test data generation with a Kalman filter-based adaptive genetic algorithm. *J. Syst. Softw.* 103 (2015), 343–352.

Aldeida Aleti and Irene Moser. 2016. A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Comput. Surv.* 49, 3 (Oct. 2016), 56:1–56:35.

S. Ali, C. Briand, H. Hemmati, and R. K. Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. Softw. Eng.* 36, 6 (2010), 742–762.

Mohammad Alshraideh, Basel A. Mahafzah, and Saleh Al-Sharaeh. 2011. A multiple-population genetic algorithm for branch coverage test data generation. *Softw. Qual. J.* 19, 3 (2011), 489–513.

I. Alsmadi. 2010. Using genetic algorithms for test case generation and selection optimization. In *Proceedings of the 2010 23rd Canadian Conference on Electrical and Computer Engineering (CCECE'10)*. 1–4.

Anuja Arora and Madhavi Sinha. 2014. State based test case generation using VCL-GA. In *Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT'14)*. IEEE, 661–665.

K. Betts and M. Petty. 2016. Automated search-based robustness testing for autonomous vehicle software. *Model. Simul. Eng.* 2016 (2016).

A. Bouchachia. 2007. An immune genetic algorithm for software test data generation. In *Proceedings of the 7th International Conference on Hybrid Intelligent Systems, 2007. (HIS'07)*. 84–89.

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90.

Yang Cao, Chunhua Hu, and Luming Li. 2009. Search-based multi-paths test data generation for structure-oriented testing. In *Proceedings of the 1st ACM/SIGEVO Summit on Genetic and Evolutionary Computation (GEC'09)*. ACM, New York, NY, 25–32.

Y. Chen and Y. Zhong. 2008. Automatic path-oriented test data generation using a multi-population genetic algorithm. In *Proceedings of the 4th International Conference on Natural Computation, 2008 (ICNC'08)*, Vol. 1. 566–570.

Y. Chen and Y. Zhong. 2009. Experimental study on GA-based path-oriented test data generation using branch distance function. In *Proceedings of the 3rd International Symposium on Intelligent Information Technology Application, 2009. (IITA'09)*, Vol. 1. 216–219.

M. Dave and R. Agrawal. 2015. Search based techniques and mutation analysis in automatic test case generation: A survey. In *Proceedings of the 2015 IEEE International Advance Computing Conference (IACC'15)*. 795–799.

A. Deepak and P. Samuel. 2009. An evolutionary multi population approach for test data generation. In *Proceedings of the World Congress on Nature Biologically Inspired Computing, 2009 (NaBIC'09)*. 1451–1456.

Karnig Derderian, Mercedes G. Merayo, Robert M. Hierons, and Manuel Núñez. 2011. A case study on the use of genetic algorithms to generate test cases for temporal systems. In *International Work-Conference on Artificial Neural Networks*. Springer, 396–403.

A. El-Serafy, G. El-Sayed, C. Salama, and A. Wahba. 2015. Enhanced genetic algorithm for MC/DC test data generation. In *Proceedings of the 2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA'15)*. 1–8.

X. Fan, F. Y. Yang, W. Zheng, and Q. J. Liang. 2015. Test data generation with a hybrid genetic tabu search algorithm for decision coverage criteria. *Proc. Sci.* 12-13-September-2015 (2015).

Marten Fischer and Ralf Tönjes. 2012. Generating test data for black-box testing using genetic algorithms. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA'12)*. IEEE, 1–6.

D. Garg and P. Garg. 2015. Basis path testing using SGA & HGA with ExLB fitness function, J. Mathew and A.K. Singh (Eds.). *Proced. Comput. Sci.* 70 (2015), 593–602.

Ahmed S. Ghiduk and Moheb R. Girgis. 2010. Using genetic algorithms and dominance concepts for generating reduced test data. *Informatica* 34, 3 (2010).

A. S. Ghiduk, M. J. Harrold, and M. R. Girgis. 2007. Using genetic algorithms to aid test-data generation for data-flow coverage. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference, 2007 (APSEC'07)*. 41–48.

David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (13th ed.). Addison-Wesley Publishing Company.

Dunwei Gong, Gongjie Zhang, Xiangjuan Yao, and Fanlin Meng. 2017. Mutant reduction based on dominance relation for weak mutation testing. *Inform. Softw. Technol.* 81 (2017), 82–96.

Dunwei Gong, Wanqiu Zhang, and Xiangjuan Yao. 2011. Evolutionary generation of test data for many paths coverage based on grouping. *J. Syst. Softw.* 84, 12 (2011), 2222–2233.

N. Gupta and M. Rohil. 2013. Improving GA based automated test data generation technique for object oriented software. In *Proceedings of the, 2013 IEEE 3rd International Advance Computing Conference (IACC'13)*. 249–253.

L. T. M. Hanh, N. T. Binh, and K. T. Tung. 2014. Applying the meta-heuristic algorithms for mutation-based test data generation for simulink models. *ACM International Conference Proceeding Series* December 4–5, 2014, 102–109.

L. T. M. Hanh, N. T. Binh, and K. T. Tung. 2016. A novel fitness function of metaheuristic algorithms for test data generation for simulink models based on mutation analysis. *J. Syst. Softw.* 120 (2016), 17–30.

Irman Hermadi and Moataz A. Ahmed. 2003. Genetic algorithm-based test data generator. In *Proceedings of the 2003 Congress on Evolutionary Computation, 2003 (CEC'03)*, Vol. 1. IEEE, 85–91.

J. H. Holland. 1975. *Adaptation in Natural and Artificial Systems*. MIT Press.

S. Huang, M. B. Cohen, and A. M. Memon. 2010. Repairing GUI test suites using a genetic algorithm. In *Proceedings of the 2010 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*. 245–254.

IEEE Xplore Digital Library. 2017. IEEE xplore digital library. Retrieved December 2016 from http://ieeexplore.ieee.org.

H. Ji and J. Sun. 2012. Automatic test data generation based on SA-QGA. In *Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS)*. 611–615.

Rong Jin, Shujuan Jiang, and Hongchang Zhang. 2011. Generation of test data based on genetic algorithms and program dependence analysis. In *Proceedings of the 2011 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER'11)*. IEEE, 116–121.

R. Khan and M. Amjad. 2015. Automatic test case generation for unit software testing using genetic algorithm and mutation analysis. In *2015 IEEE UP Section Conference on Electrical Computer and Electronics (UPCON)*. 1–5.

R. Khan, M. Amjad, and A. K. Srivastava. 2016. Optimization of automatic generated test cases for path testing using genetic algorithm. In *Proceedings of the 2016 2nd International Conference on Computational Intelligence Communication Technology (CICT'16)*. 32–36.

Juhi Khandelwal and Pradeep Tomar. 2015. Approach for automated test data generation for path testing in aspect-oriented programs using genetic algorithm. In *Proceedings of the 2015 International Conference on Computing, Communication & Automation (ICCCA'15)*. IEEE, 854–858.

Fitsum M. Kifetew, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Paolo Tonella. 2013. Orthogonal exploration of the search space in evolutionary test case generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13)*. ACM, New York, NY, 257–267.

B. Kitchenham and S. Charters. 2007. Guidelines for performing systematic literature reviews in software engineering. Version 2.3. EBSE Technical Report EBSE-2007-01, Keele University and Durham University.

C. Koleejan, B. Xue, and M. Zhang. 2015. Code coverage optimisation in genetic algorithms and particle swarm optimisation for automatic software test data generation. *Proceedings of the 2015 IEEE Congress on Evolutionary Computation (CEC'15)*, 1204–1211.

Kiran Lakhotia, Mark Harman, and Phil McMinn. 2007. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07)*. ACM, New York, NY, 1098–1105.

Raluca Lefticaru and Florentin Ipate. 2007. Automatic state-based test generation using genetic algorithms. In *Synasc*, Vol. 7. 188–195.

Wang Lijuan, Zhai Yue, and Hou Hongfeng. 2012. Genetic algorithms and its application in software test data generation. In *Proceedings of the 2012 International Conference on Computer Science and Electronics Engineering (ICCSEE'12)*, Vol. 2. IEEE, 617–620.

Peng Lin, Xiaolu Bao, Zhiyong Shu, Xiaojuan Wang, and Jingmin Liu. 2012. Test case generation based on adaptive genetic algorithm. In *Proceedings of the 2012 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE'12)*. IEEE, 863–866.

Dan Liu, Xuejun Wang, and Jianmin Wang. 2013. Automatic test case generation based on genetic algorithm. *J. Theor. Appl. Inform. Technol.* 48, 1 (2013), 411–416.

T. Manikumar, A. John Sanjeev Kumar, and R. Maruthamuthu. 2016. Automated test data generation for branch testing using incremental genetic algorithm. *Sadhana—Acad. Proc. Eng. Sci.* 41, 9 (2016), 959–976. SAPSE

M. Mann, O. P. Sangwan, P. Tomar, and S. Singh. 2016. Automatic goal-oriented test data generation using a genetic algorithm and simulated annealing. In *Proceedings of the 2016 6th International Conference—Cloud System and Big Data Engineering (Confluence)*. 83–87.

J. Mayan and T. Ravi. 2014. Test case optimization using hybrid search technique. *ACM International Conference Proceeding Series* October 10–11, 2014.

James Miller, Marek Reformat, and Howard Zhang. 2006. Automatic test data generation using genetic algorithm and program dependence graphs. *Inform. Softw. Technol.* 48, 7 (2006), 586–605.

Melanie Mitchell. 1999. *An Introduction to Genetic Algorithms* (5th ed.). MIT Press, Cambridge, Massachusetts.

Glenford J. Myers. 1979. *The Art of Software Testing*. Wiley.

Alberto Núñez, Mercedes G. Merayo, Robert M. Hierons, and Manuel Núñez. 2013. Using genetic algorithms to generate test sequences for complex timed systems. *Soft Comput.* 17, 2 (2013), 301–315.

Ankur Pachauri and Gursaran Srivasatava. 2015. Towards a parallel approach for test data generation for branch coverage with genetic algorithm using the extended path prefix strategy. In *Proceedings of the 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom'15)*. IEEE, 1786–1792.

M. Parthiban and M. R. Sumalatha. 2013. GASE—An input domain reduction and branch coverage system based on genetic algorithm and symbolic execution. In *Proceedings of the 2013 International Conference on Information Communication and Embedded Systems (ICICES'13)*. 429–433.

Xuan Peng and Lu Lu. 2011. User-session-based automatic test case generation using GA. *Int. J. Phys. Sci.* 6, 13 (2011), 3232–3245.

G. H. L. Pinto and S. R. Vergilio. 2010. A multi-objective genetic algorithm to test data generation. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'10)*, Vol. 1. 129–134.

Shweta Rani and Bharti Suri. 2015. An approach for test data generation based on genetic algorithm and delete mutation operators. In *2015 2nd International Conference on Advances in Computing and Communication Engineering (ICACCE'15)*. IEEE, 714–718.

Science Direct. 2017. Science direct. Retrieved December 2016 from http://www.sciencedirect.com.

Scopus. 2017. Scopus. Retrieved December 2016 from http://www.scopus.com.

N. Sharma, A. Pasala, and R. Kommineni. 2012. Generation of character test input data using GA for functional testing. In *Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference (APSEC'12)*, Vol. 2. 87–94.

Liu Shimin and Wang Zhangang. 2011. Genetic algorithm and its application in the path-oriented test data automatic generation. *Proced. Eng.* 15 (2011), 1186–1190. CEIS 2011.

B. Shuai, H. Li, L. Zhang, Q. Zhang, and C. Tang. 2015. Software vulnerability detection based on code coverage and test cost. *Proceedings—2015 11th International Conference on Computational Intelligence and Security (CIS'15)*. 317–321.

Lucas Serpa Silva and Maarten van Someren. 2010. Evolutionary testing of object-oriented software. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*. ACM, New York, NY, 1126–1130.

M. Soltani, A. Panichella, and A. Van Deursen. 2016. Evolutionary testing for crash reproduction. *Proceedings—9th International Workshop on Search-Based Software Testing (SBST'16)*, 1–4.

Ian Sommerville. 2007. *Software Engineering* (8th ed.). Addison-Wesley, São Paulo.

J. H. Sun and S. J. Jiang. 2010. An approach to automatic generating test data for multi-path coverage by genetic algorithm. In *Proceedings of the 2010 6th International Conference on Natural Computation (ICNC'10)*, Vol. 3. 1533–1536.

X. Tan, Cheng Longxin, and Xu Xiumei. 2009. Test data generation using annealing immune genetic algorithm. In *Proceedings of the 5th International Joint Conference on INC, IMS and IDC, 2009 (NCM'09)*. IEEE, 344–348.

Tian Tian and Dunwei Gong. 2016. Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms. *Autom. Softw. Eng.* 23, 3 (Sept. 2016), 469–500.

Paolo Tonella. 2004. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes* 29, 4 (July 2004), 119–128.

Sapna Varshney and Monica Mehrotra. 2016. Search-based test data generator for data-flow dependencies using dominance concepts, branch distance and elitism. *Arabian J. Sci. Eng.* 41, 3 (2016), 853–881.

Chandra Prakash Vudatha, Sateesh Nalliboena, Sastry K. R. Jammalamadaka, Bala Krishna Kamesh Duvvuri, and L. S. S. Reddy. 2011. Automated generation of test cases from output domain of an embedded system using genetic algorithms. In *Proceedings of the 2011 3rd International Conference on Electronics Computer Technology (ICECT'11)*, Vol. 5. IEEE, 216–220.

Wang Xibo and Su Na. 2011. Automatic test data generation for path testing using genetic algorithms. In *Proceedings of the 2011 3rd International Conference on Measuring Technology and Mechatronics Automation*, Vol. 1. IEEE, 596–599.

S. Yang, T. Man, J. Xu, F. Zeng, and K. Li. 2016. RGA: A lightweight and effective regeneration genetic algorithm for coverage-oriented software test data generation. *Inform. Softw. Technol.* 76 (2016), 19–30.

Xiangjuan Yao and Dunwei Gong. 2014. Genetic algorithm-based test data generation for multiple paths via individual sharing. *Comput. Intell. Neurosci.* 2014 (2014), 29.

Xiangjuan Yao, Dunwei Gong, and Wenliang Wang. 2015a. Test data generation for multiple paths based on local evolution. *Chinese J. Electron.* 24, 1 (2015), 46–51.

X. Yao, D. Gong, and G. Zhang. 2015b. Constrained multi-objective test data generation based on set evolution. *IET Softw.* 9, 4 (2015), 103–108.

Na Zhang, Biao Wu, and Xiaoan Bao. 2015. Automatic generation of test cases based on multi-population genetic algorithm. *Technology* 10, 6 (2015).

Wanqiu Zhang, Dunwei Gong, Xiangjuan Yao, and Yan Zhang. 2010. Evolutionary generation of test data for many paths coverage. In *Proceedings of the 2010 Chinese Control and Decision Conference*. 230–235.

Y. Zhang and D. Gong. 2010. Evolutionary generation of test data for multiple paths coverage with faults detection. In *2010 IEEE 5th International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA'10)*. 406–410.

Xiaofei Zhou, Ruilian Zhao, and Feng You. 2014. EFSM-based test data generation with multi-population genetic algorithm. In *Proceedings of the 2014 5th IEEE International Conference on Software Engineering and Service Science (ICSESS'14)*. IEEE, 925–928.

Received January 2017; revised January 2018; accepted January 2018