Ariadne: Evolving Test Data Using Grammatical Evolution

Muhammad Sheraz Anjum[1][0000−0002−3600−8931] and Conor Ryan[1][0000−0002−7002−5815]

Department of Computer Science and Information Systems, University of Limerick, Castletroy, Limerick, Ireland {sheraz.anjum,conor.ryan}@ul.ie

Abstract. Software testing is a key component in software quality assurance; it typically involves generating test data that exercises all instructions and tested conditions in a program and, due to its complexity, can consume as much as 50% of overall software development budget. Some evolutionary computing techniques have been successfully applied to automate the process of test data generation but no existing techniques exploit variable interdependencies in the process of test data generation, even though several studies from the software testing literature suggest that the variables examined in the branching conditions of real life programs are often interdependent on each other, for example, *if (x==y)*, etc.

We propose the *Ariadne* system which uses Grammatical Evolution (GE) and a simple Attribute Grammar to exploit the variable interdependencies in the process of test data generation. Our results show that Ariadne dramatically improves both effectiveness and efficiency when compared with existing techniques based upon well-established criteria, attaining *coverage* (the standard software testing success metric for these sorts of problems) of 100% on **all** benchmarks with far fewer program evaluations (often between a third and a tenth of other systems).

Keywords: Automatic Test Case Generation · Code Coverage · Evolutionary Testing · Grammatical Evolution · Variable Interdependencies.

1 Introduction

The primary goal of software testing is to uncover as many faults as possible. In practice, a labor intensive testing is carried out in order to achieve a certain level of confidence in a software system [1]. Studies have shown that manual testing may consume as much as 50% of overall software development budget [2]. Researchers have been trying to automate software testing since as early as 1962, when [3] proposed a random test data generator for COBOL, while more recently, a variety of metaheuristic search techniques have successfully been applied to automatically generate test data as surveyed by [4,5].

The use of metaheuristic techniques to automatically generate test data is often referred as *Search Based Software Testing* (SBST). In SBST, in general,

some metaheuristic technique is applied to search through the space of all possible inputs to a particular program to find a specific test set (set of inputs) that can then be used to satisfy a particular test adequacy criterion such as branch coverage.

Genetic Algorithms (GAs) [6] are the most commonly adopted search techniques in SBST [7,5] having been used with some success [8], [9], while Memetic Algorithms (MAs), which are typically hybrids of GAs and some sort of Local Search Algorithms (LSAs), have also been shown to be useful in this case [10].

The most widely studied test adequacy criterion in literature is branch coverage [5], which aims at maximizing the number of branches executed during a test. An extended and more challenging form of branch coverage is *Condition-decision* coverage (details presented in Section 2); this paper is concerned with tackling this problem.

Variables in the branching conditions of real life programs are often interdependent on each other; for example, a branching condition may have a check if two variables have equal values. Further, there are many tests that appear in many programs, such as checking if a variable has a value equal to zero, or less than zero. This has been observed by several researchers; for example, a study of 50 COBOL programs [11] revealed that 64% of the total predicates were equality predicates and that 87% of them examined 2 or fewer variables. In [12], 120 production PL/I programs were analyzed and it was found that 98% of all expressions contained fewer than two operators while 62% of all operators were relational/comparison operators. To the best of our knowledge, to date no SBST technique has exploited these properties in the process of test data generation.

In this paper, we introduce a GE [13,14] based test data generator. GE is a grammar based genetic algorithm which enables the use of a simple grammar to exploit simple relationships between input variables, including the sorts of properties mentioned above. As condition-decision coverage involve efficiently searching the space of paths in a program to make sure that the program has been thoroughly explored, we call our system *Ariadne*, after the mythological figure who helped Theseus find his way out of the Minotaur's Labyrinth.

We apply Ariadne to the problem of condition-decision coverage, although demonstrate that the technique can be deployed for other test adequacy criteria. Our results suggest that Ariadne significantly improves both effectiveness and efficiency when compared with well-known results from the literature [8], [9] and [10], showing that across 11 popular benchmark programs, Ariadne achieved 100% coverage for all while reducing the search budget up to multiple times.

2 Background and Related Work

Software testing is a key component in software quality assurance and it can be broadly categorized as structural/white-box testing and specification-based/ black-box testing. Structural testing inspects program *structure* while specification-based testing examines *functionality*. Structural testing is labor intensive and, as a result, costs more time and money compared to other software development

activities [2]. Significant research has been conducted on ways to automate the testing process to help reduce costs.

A test adequacy criterion is a certain property that a program must satisfy to gain confidence about the absence of certain types of errors. For example, branch coverage is a structure-based adequacy criterion which requires that every branching condition must take each possible outcome at least once. If the following piece of code is under test:

If *((x < y) and (x < z))* { *some program statement(s)*}
If *(y < z)* { *some program statement(s)*}
else { *some program statement(s)*}

to achieve 100% branch coverage (often referred as complete coverage), this program must be executed with a test suite such that both if-conditions evaluate to both TRUE and FALSE outcomes at least once each. To manually achieve complete branch coverage in this case, a tester must generate a set of test inputs (test data) that execute every branch of the program under test.

Condition coverage is a more refined criterion than branch coverage. It requires that each single **condition** in the program must get both TRUE and FALSE values at least once. In case of a compound branching condition, such as in the first if-condition in the above example, all single conditions must individually get both TRUE and FALSE values. Another criterion, condition-decision coverage, is essentially a combination of branch coverage and condition coverage as it requires that all branching conditions as well as single conditions must get both TRUE and FALSE values at least once.

2.1 Related Work

The automation of test data generation has been the subject of increasing research interest [15] and it has been an area of investigation for many years [3,16,17]. One of the most straightforward methods of test data generation is to simply deploy a random search mechanism and use it to repeatedly generate input values until the required input is found. Sauder's work [3] is one of the earliest random test data generators reported in the literature, as he was the first to consider the space of all possible inputs (of COBOL programs) as a search space. *Random test data generation* can be inefficient as the test generation is not guided in any way and, in general, becomes increasingly more inefficient as the search space increases.

Another paradigm for test data generation presented in the literature is *static test data generation*. This paradigm does not require executing the program under test as a mathematical system is deployed to find the required input values. One such technique is symbolic execution, in which program variables are assigned symbolic values and the resulting mathematical expression is solved to generate test data [16], [18], [19], [20]. Major challenges associated with symbolic execution include handling complexity of constraints, procedure calls, loops and

pointers. Techniques such as domain reduction [21] and dynamic domain reduction [22] have been proposed to address some of the issues associated with symbolic execution but dealing with loops and pointers still remains a problem.

Dynamic test data generation is based on the idea of actually executing the program under test to observe its behavior. Data recorded during this observation is then used to direct the search of required test data. This idea was first presented in [17] where numerical maximization techniques were employed to generate floating point test data. This idea was later extended by various researchers [23], [24], [25], and [26]. All these dynamic test data generation techniques were based on LSAs and consequently had a risk of getting stuck in local minima.

To overcome some inherent problems associated with local search, global search techniques including simulated annealing [27], [28] and GAs [29], [30], [31], [32], [8], [9] have been employed for test data generation. MAs have also been successfully deployed for test data generation [33], [10], [34]. GA based test data generation is detailed in the next section.

The conventional test data generation techniques target one coverage goal at a time. For example, in case of branch coverage, the coverage of one branch is targeted at one time. However, Whole Test Suite Generation [35], [34] and Many-Objective Optimization [36] techniques target multiple coverage goals simultaneously.

GA Based Test Data Generation GA-based test data generation, like other SBST techniques (often referred as evolutionary testing), considers the space of all possible inputs of the program under test a search space. The individuals in the GA population are generally vectors of input parameters which serve as the test data (test cases) for the program under test. The code is usually instrumented to monitor the execution of the program under test and the fitness value is assigned according to the execution of the program. The test adequacy criterion is implemented as fitness function which, in general, measures how far or close an individual is from covering the current target. Different techniques presented in the literature use different definitions of fitness functions but, in general, fitness functions can be broadly categorized into *branch distance based* and *control flow based* fitness functions.

The concept of branch distance was introduced by [23] and it simply describes how close an individual is from satisfying the target predicate. For example in Fig. 1, if the TRUE branch from node 2 is the target branch then the predicate to be satisfied is *if (i == j)*. The fitness function in this case of equality operator will be *absolute(i - j)* and the fitness value of all the individuals reaching the predicate can be measured accordingly. If the following individuals are evaluated for the above mentioned branch: <1, 2, 5>, <0, 4, 29>, <5, 250, 251>, where in each the three values are assigned to *i, j* and *k* respectively, then the respective fitness values would be 1, 4 and 245 where lower is better. That is, their fitnesses are computed using *absolute(i - j)* in each case, giving us $|1 - 2|$, $|0 - 4|$ and $|5 - 250|$, respectively, so the individual <1, 2, 5> will get the best fitness being

closet from satisfying the target predicate. A list of fitness functions for different types of predicates is presented in [23].

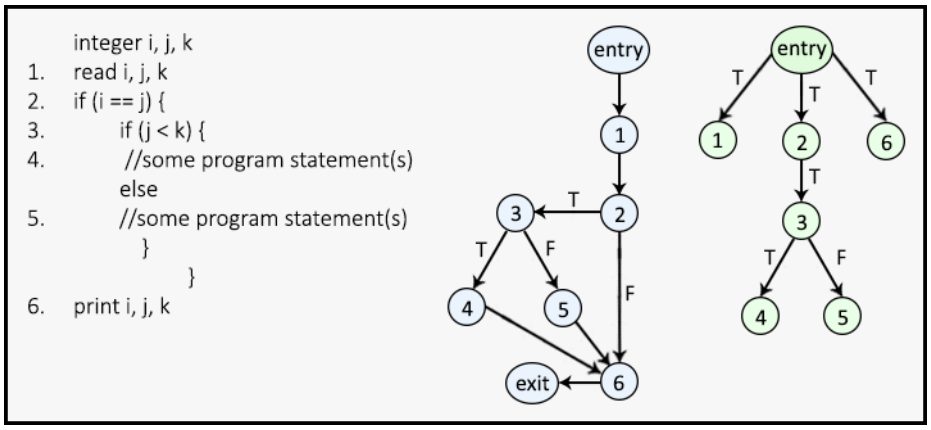

Fig. 1: An example program on the left, its Control Flow Graph in the center, and its Control Dependence Graph on the right.

GAs were first employed in SBST by [29] where a branch distance based fitness function was used. The apparent weakness of branch distance based fitness functions is that they don't guide the search towards reaching the concerned branching predicate and only effectively evaluate the individuals already reaching the concerned predicate. All the individuals missing the concerned predicate get a similar low fitness irrespective to the number of nodes they miss to reach it. This problem was partially handled by [8] where those branches were targeted first whose relevant predicates were already reached in some previous fitness evaluations. This problem can be addressed by bringing control flow information into consideration.

Control flow based fitness functions, on the other hand, solely rely on control flow information to measure the fitness. The list of nodes traversed during the execution of the program is compared with the list of critical/control-dependent branching nodes. Control-dependent nodes are the ones which must be traversed in order to reach the target. The individuals traversing more number of control dependent nodes get a better fitness value. For example if the TRUE branch from node 3, in the example presented as Fig 1, is the target branch then the list of control dependent branches are {entryT, 2T, 3T}. If the following individuals are evaluated for this branch; <6, 6, 4>, <241, 241, 12>, <5, 0, 1>, then the first two individuals will get the equal higher fitness as both of them traverse 2 control dependent branches.

The work presented in [30] primarily used a branch distance based fitness functions but it also utilized control flow information as the fitness function considered the number of required loop iterations for loop testing. Later on [31],

used the number of traversed control dependent predicates as a fitness measure for statement and branch coverage. The problem associated with control flow based approaches is that they only guide the search towards reaching the relevant predicate and do not guide the search to satisfy the predicate.

To exploit the benefits associated with both branch distance based and control flow based techniques, [32] proposed using a combination of both techniques. The work introduced the concept of *approximation level* to capture the control flow information and used it in a combination with normalized branch distance for the fitness evaluation of individuals. The approximation level simply tells that how far an individual is from reaching the target branch as it is a count of control-dependent branching nodes that are missed in the path executed by the individual. The work presented in [9] utilized the control flow information in a different way as it used the predicates of all control-dependent branching nodes in a branch distance based fitness function.

It is worth noting that it is quite common in evolutionary testing that some targets are covered accidentally while the GA is trying to cover some other targets. This phenomenon is called accidental (serendipitous) coverage and it significantly shortens the execution time as the GA does not need to be executed for the already covered targets. The effectiveness of any evolutionary testing technique is measured according to the selected adequacy criterion e.g. percentage of branches covered will be the effectiveness measure in case of branch coverage.

2.2 Grammatical Evolution

GE is a grammar-based GA that uses a grammar-based mapping process that separates search space from solution space. The evolutionary processes are applied to the genotype (search space) while the generated phenotype (solution domain) is generated by means of a problem specific grammar and is used for the purpose of fitness evaluation.

A grammar is composed of four elements i.e. terminals, non-terminals, productions rules and a start symbol. The terminals represent the constructs from the solution domain and the non-terminals are associated with a set production rule each. The production rules direct the mapping process for the expansion of non-terminals into one or more terminals and non-terminals. The start symbol is a selected non-terminal and the mapping process starts from applying an associated production rule for the start symbol. A sample grammar is shown in Fig 2.

The genotype is a binary string where each 8 bit codon represents an integer value. These integer values are consumed one at a time in the mapping process for the selection of appropriate production rules. The production rules are selected by the following formula:

Rule = (Codon Integer Value) MOD (number of total production rules for the current non-terminal)

For example, if the non-terminal <op> is to be expanded by selecting a production rule from the set of these four rules:

```
<op> ::= +    (0)
       | -    (1)
       | /    (2)
       | *    (3)
```

And the next genotype integer to be consumed is 51, then 51 MOD 4 = 3. So the # 3 (<op> ::= *) is selected. The use of the MOD operator ensures that only relevant rules will be chosen. A complete example of genotype to phenotype mapping is presented in Fig 2. While standard GE typically uses context free grammars (CFGs), it is a simple matter to change to Attribute Grammars (AGs). AGs are grammars in which some of the non-terminals have attributes that can be used to pass contextual information around a derivation tree.

3 GE based test data generation-Ariadne

In this section, we propose a GE based evolutionary test data generator named Ariadne that automatically detects and exploits variable dependencies using a simple attribute grammar designed based on the observations discussed in Section 1. We test Ariadne on a set of benchmark problems with condition-decision coverage as the test adequacy criterion, but the technique can also be used for other test adequacy criteria. Grammars allow GE to impose constraints on its individuals; typically this involves available instructions or preventing the use of certain sequences of instructions [37]. We use this power to impose dependencies between the variables being generated.

3.1 Overview

Ariadne, like other evolutionary testing techniques, considers the space of all possible inputs of the program under test as a search space. It deploys GE as a search algorithm to automate the process of test data generation, using the grammar presented below in Section 3.2, and produces a set of input variables for the program under test. We use the **same** grammar for all numeric benchmark problems.

To fulfil the adequacy criterion of condition-decision coverage, both TRUE and FALSE outcomes of all the branching nodes and condition predicates are considered as test objectives. Ariadne targets these objectives one by one and for every selected target the GE is initialized and an attempt to achieve the target, via the evolutionary process, is made. Ariadne also allows accidental/ serendipitous coverage as it records whenever a condition or decision outcome is executed for the first time, regardless of whether or not it is the current target. Once the objective at hand is achieved, the next objective is selected from the pool of currently unachieved test objectives and it continues until all the objectives are either achieved or had a failed try i.e. a run of GE. The objectives which stay unachieved can be considered either infeasible or simply unreachable by the applied technique. The success of any SBST technique can be measured in terms of percentage of test objectives achieved by that technique.

Recall from the Section 2.1 that in dynamic test data generation, the total number of Fitness Evaluations (FE) is the same as total number of target program executions (because the target program is executed once for every fitness evaluation). This count FE was used as the efficiency measures by [8], [9] and [10] and we also used the same metric measure to compare our results with the ones presented in the literature.

3.2 Grammar

We use the following simple grammar to capture observed characteristics of conditions commonly found in code. The start symbol of the grammar has only one production rule and it produces a non-terminals for each input variable in the target program.

$$< start >::=< var_1 >< var_2 >< var_3 > \cdots < var_N > \tag{1}$$

where N is the number of input variables. Each of these non-terminals is then expanded using a set of production rules of the form:

$$\begin{aligned} < var_M >::= 0|1|-1|< rand > | < dep_{var_1} > | < dep_{var_2} > \\ | \ldots | < dep_{var_{M-2}} > | < dep_{var_{M-1}} > \end{aligned} \tag{2}$$

The first three rules give grammar the ability to generate input values that can satisfy commonly used checks for zero, positive and negative integer values represented by 0, 1 and -1 respectively. The next rule, i.e. <rand>, produces a 32 bit signed random number which is generated from a seed value taken from the individual's genome as shown in genotype to phenotype mapping example in Fig 2. This ensures that not only each time the individual is evaluated it will produce the same set of random numbers, but that an offspring that inherits this seed will also generate the same set. These seeds are subject to mutation, so the sets of random numbers available can change during evolution.

The remaining non-terminals essentially simple synthesized attributes in which the value of the variable is calculated based on the value of a previously generated variable. Each variable M can use the value of any previously declared variable and, when a production rule of this form is chosen, has its value calculated as follows:

$$< dep_{var_X} >:= var_X|(var_X + 1)|(var_X - 1) \tag{3}$$

where var_X is the value of a previously generated input variable. This set of production rules is responsible for exploiting variable dependencies as it generates values which are dependent on previously generated input variables. Recall from Section 1, that it is very common for the conditions to have comparison operators (dependencies) between two variables.

The dependencies are very simple; either the new variable has the same value or is ± 1 the value. Similarly, because of the way in which the start symbol produces a single non-terminal for each input variable in a fixed order, the flow of

dependencies is always from left to right and there can be no circular dependencies. These dependencies facilitate the generation of input data that is likely to satisfy conditions with comparison operators between variables. A complete example of a grammar-based genotype to phenotype mapping for a program having three input variables is presented in Fig 2. Note that we use the **same** grammar for all experiments in this paper; the only difference is the number of input variables.

The grammar in Fig 2 doesn't have explicit attributes, rather contextual information is passed from left to right using grammar rules that can interrogate the derived value of another variable as in rule 3.

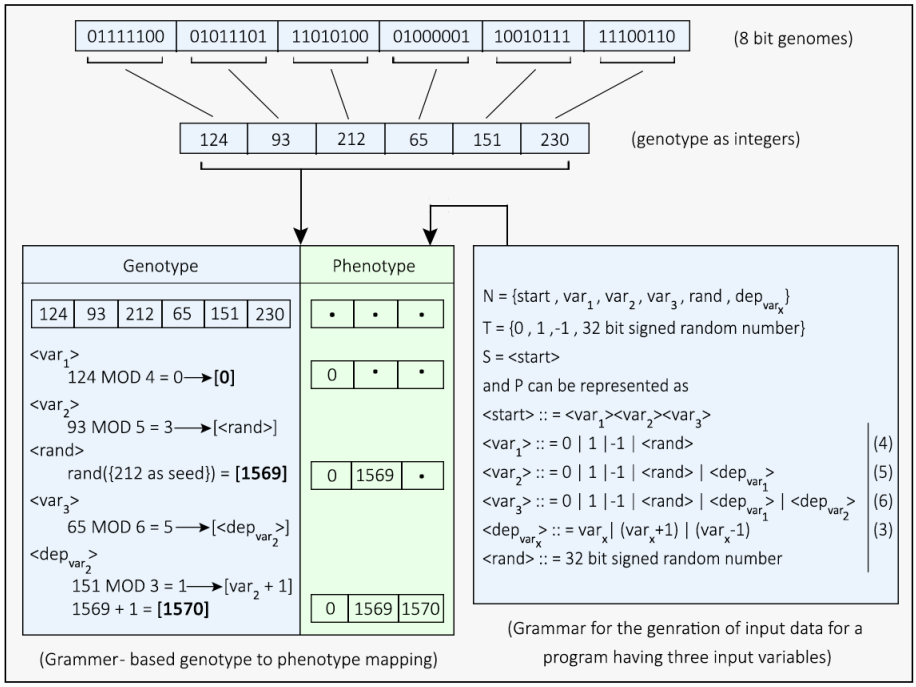

Fig. 2: An example with the genotype on the top, grammar on the right and the mapping sequence on the left. Note that the only required changes to the grammar for different problems are the number of variables and the number of dependency rules, but these are all of the same form.

3.3 Fitness Function

A fitness function, in general, measures how far or close an individual is from achieving the current target. The fitness function deployed in Ariadne is similar

to [31] and approximation level of [32] as it directs the search towards the current objective based on control dependencies. The Control Dependence Graph (CDG) of the program is used to identify the sequences of control dependent nodes for all the test objectives. Fig 1 presents an example program and its CDG.

Control dependent nodes are the nodes that must be executed to reach the current objective. For example, node 3 is control dependent for the execution of node 4, as node 4 cannot be executed without executing node 3. It is also worth noting that, for any given input the execution path of the program depends on the results of branching conditions on intermediate nodes. For example, in the program presented as Fig 1, the branching condition on node 3 must be evaluated as TRUE in order to execute node 4 whereas it must be evaluated as FALSE for the execution of node 5.

To evaluate the fitness of an individual, the target program is executed with the set of input variables (phenotype) associated with this individual. The code of target program is instrumented in order to monitor the execution of the program and the count of executed control dependent nodes is used as a fitness measure:

fitness (current_objective, phenotype_of_current_individual) = control distance (current_objective, phenotype_of_current_individual)

If an individual covers more nodes on the path to current target then it is likely that some part of it is good for satisfying the target. It is highly probable that by recombining such individuals we can get a better individual that goes even closer towards the target node. This way Ariadne reduces and eventually nullifies the number of missing control dependent nodes towards the current objective.

On all the nodes, including the final node, both TRUE and FALSE outcomes are quickly covered by the function of the grammar as explained in Section 3.2. This is one of the reasons that Ariadne does not need branch distance in its fitness function to guide the search and consequently gets the work done using a simple fitness function.

4 Experimental Results and Discussion

An empirical study was performed using two different sets of functions. The first set contains nine numeric functions which are similar[1] to those presented in [8], [9]. The second set contains two validity-check functions from a real-world program **bibclean-2.08** [38].

The validity check functions are taken from a study presented in [10]. That study generalized and validated the Holland's schema theory [39] and the *Royal*

[1] Where possible we used the same functions, but where the actual source code isn't available we have tried to reproduce the code as closely as possible and in order to facilitate future comparisons we have made available the source code at `http://bds.ul.ie/?page_id=390/`.

Road theory [40] using a set of condition coverage problems. The generalized theory essentially predicts that GAs should perform well for problems that exhibit Royal Road property. For example, a branch is said to exhibit Royal Road property if the fitter individuals for the coverage of that branch contain some building blocks which independently can participate towards improving the fitness value. In this case it will be highly probable for crossover to generate fitter offspring as compared to their parents. The validity check functions were selected for the purpose of above mentioned study because some of their branches exhibit Royal Road property.

For the purpose of this study, we used Ariadne to generate test data for the selected functions and we compared our results with the results of GA based techniques presented in [8], [9] and [10]. A brief description of the selected functions is presented below in next section.

4.1 Test Functions

The function sets we use vary in complexity. The functions in the Set 1 represent different types of test-data generation problems. These include Binary Search, Bubble Sort, *Days*, Greatest Common Divisor (GCD), Insertion Sort, Median, Quadratic Formula, Warshall's Algorithm and *Triangle Classification*.

Most of these are well known and self-explanatory, but the lesser known ones are as follows. *Days* calculates the total number of days between two dates, Warshall's Algorithm finds the shortest path in a weighted graph and Triangle Classification, which is one of the most widely studied functions in SBST [41], [26], [31], [8], [9] etc., classifies a triangle based on the lengths of three sides of the triangle.

Function Set 2 contains check_ISBN and check_ISSN functions from an open source program bibclean-2.08 [38]. Both check_ISBN and check_ISSN take a string of 30 characters as input and perform a sequential search to find valid characters for an International Standard Book Number (ISBN) and an International Standard Serial Number (ISSN) respectively. ISBN and ISSN are used to uniquely identify publications while they comprise of 10 and 8 characters respectively.

4.2 Experimental Setup

A small set of initial experiments were conducted to identify reasonable run parameters. While we assumed a population of 50 would be reasonable for the functions in Set 1, we quickly discovered that several of them were solved with very small populations of just ten individuals. These include Binary Search, Bubble Sort, GCD, Insertion Sort, Median, Quadratic Formula and Warshall's Algorithm; complete coverage was achieved on all very quickly due to a combination of the constraints imposed by the grammar and the fact that multiple targets were covered in different iterations of the loops in those functions.

For Triangle Classification and Days, both maximum number of generations and population size were left at the standard GE setting of 50, while the more

complex functions contained in Set 2 (check_ISBN and check_ISSN) were given 300 generations with a population size of 100. The probabilities of crossover and mutation were set as 0.9 and 0.05 and were kept the same during all the experiments while the methods of One Point Crossover & Flip Mutation were employed. It is worth noting that Ariadne produces similar results even when these genetic parameters are set high because the terminating criteria causes the search to stop once the target at hand is achieved, so even if we had left the population at 50 the runs would terminate early.

The automatically generated data via our system contain values including 0, 1, -1 and 32-bit signed integer values (random) in the range of -2,147,483,648 to 2,147,483,647. These generated values directly work as input values for most of the selected functions as they take integer inputs. For the functions with more complex inputs, specifically Days, check_ISBN and check_ISSN an extra mapping step is needed as the input type for Days is date and check_ISBN and check_ISSN take character strings as input. A mod based mapping function is used to convert six integer values (generated as a result of genotype to phenotype mapping) into two valid dates each consisting of a year, month and day. Similarly, for check_ISBN and check_ISSN, a mod based mapping function is used to convert thirty integer values into the ASCII codes which represent the characters of the input string.

4.3 Detailed Analysis of Experiments

In order to illustrate both the effectiveness and the efficiency of Ariadne, the results of our experiments are reported here. For all the nine functions from the first set, thirty runs were performed separately for each function and their mean performance is presented in Table 1. A comparison of our results with the best results of [8] is also presented in Table 1. Since the source code of only the Triangle Classification function was provided, we created our own versions of the other functions based on the literature. It is worth noting that [8] reported the **highest** performance among five runs in contrast to our **mean** performance over of thirty runs. It can be seen that at least one of their best performers was not able to achieve 100% coverage for each of Binary Search, Insertion Sort, Quadratic Formula and Triangle Classification, while Ariadne was able to achieve a 100% coverage in all thirty runs for all the functions.
The coverage for Binary Search, Bubble Sort, GCD, Insertion Sort, Median, Quadratic Formula and Warshall's Algorithm was immediately achieved in our experiments as either the structure of the function was simple or the condition predicates were quickly satisfied by grammar. The function Days, on average, took around 300 FE to achieve 100% coverage as the comparison conditions in its nested structure were quickly satisfied by the function of grammar. We were not able to compare these results as, among all nine functions, [8] reported the number of fitness evaluations for Triangle Classification only.

For Triangle Classification, [8] reported the best coverage of 94.29% with the search cost of about 8,000 FE as presented in Table 2. [9] reported a 100% coverage using the Program Dependence Graphs(PDG) based approach as discussed

Table 1: A comparison of Ariadne with GADGET [8] on nine benchmark functions

Program	**GADGET**[8]		**Ariadne**	
	GA	**Differential GA**	**Ariadne**	**Avg. FE**
Binary Search	70%	100%	100%	3
Bubble Sort	100%	100%	100%	1
Days	100%	100%	100%	300
GCD	100%	100%	100%	6
Insertion Sort	92.9%	100%	100%	1
Median	100%	100%	100%	4
Quadratic Formula	75%	75%	100%	15
Warshall's Algorithm	100%	100%	100%	1
Triangle Classification	94.29%	84.3%	100%	935

in Section 2.1. However, it can be clearly seen that [9] improved the coverage with a huge cost in terms of FE. On the other hand, Ariadne achieved 100% coverage on a multiple times smaller search cost. The reason Ariadne was so effective and efficient for Triangle Classification is that it was able to quickly generate input data of the form i=j=k which is extremely difficult for a GA to generate otherwise.

Table 2: Results on Triangle Classification for condition-decision coverage

Method	**Coverage**	**Avg. FE**
GADGET [8]	94%	8000
TDGen [9]	100%	97300
Ariadne	100%	935

For both of the functions from the Function Set 2, i.e. check_ISBN and check_ISSN, experiments are carried out on the same lines as that of [10] in order to have a fair comparison with their best results. [10] reported the GA-based techniques results for non-trivial branches only i.e. the branches which were not covered by random testing in their experiments. We computed our results based on sixty independent runs, for each of the non-trivial branches and compared them with the best results of [10] as Table 3.

The results are reported based on two metrics i.e. Success Rate (SR) and average number of FE where SR is defined as percentage of times that a particular branch was covered when targeted. Table 3 shows that Ariadne exhibited a 100% SR for all the branches in comparison to 95% and 98% SRs reported in [10] while the average number of FE was reduced up to an order of magnitude in many cases, but by two thirds at least.

The reason Ariadne performed even better for Royal Road functions as compared to standard GAs is that it encourages the generation of variables similar to the previously generated variables. So if the parents contain any valid characters for check_ISBN/check_ISSN, it will be highly probable that the offspring will not only retain these characters like standard GAs but also generate additional similar valid characters by the function of the grammar. The results presented in this section show that Ariadne outperforms other GA based SBST techniques and improves the effectiveness as well as efficiency by a wide margin.

Table 3: A comparison of Ariadne with [10] on non-trivial branches of check_ISBN & check_ISSN functions from **bibclean-2.08**.

Branch ID	**Harman & McMinn** [10]		**Ariadne**	
	SR	**Avg. FE**	**SR**	**Avg. FE**
B3-ISBN	95%	7986	100%	745
B4-ISBN	95%	7986	100%	747
B6-ISBN	95%	8001	100%	708
B7-ISBN	95%	9103	100%	3313
B3-ISSN	98%	5273	100%	655
B4-ISSN	98%	5273	100%	550
B6-ISSN	98%	5324	100%	542
B7-ISSN	98%	6380	100%	2662

5 Conclusion and Future Work

We have presented Ariadne, a GE-based tool to automate the process of test data generation. The work proposes the use of a simple grammar to exploit the variable interdependencies present in the branching conditions of real life programs. We have conducted our experiments using two sets of functions representing different types of test-data generation problems. Results of our experiments show that Ariadne clearly outperforms existing techniques both in terms of effectiveness and efficiency which are measured in terms of percentage (of condition-decision) coverage and number of fitness evaluations (function executions) respectively.

This paper serves an introduction to GE based test data generation and we believe that there is a lot of potential to further improve the technique. We are actively working towards improving Ariadne in a number of ways including the optimization of the grammar to make Ariadne even more efficient, as well as the extension of the grammar to accommodate more constructs such as constant values present in condition predicates.

Acknowledgments

The authors would like to thank Muhammad Hamad Khan for his help with the graphic designs. This work is supported by Lero, the Irish Software Research Centre, and the Science Foundation of Ireland.

References

1. Myers, G.J., Sandler, C., Badgett, T.: The art of software testing. John Wiley & Sons (2011)
2. Beizer, B.: Software testing techniques, van nostrand reinhold. Inc, New York NY, 2nd edition. ISBN 0-442-20672-0 (1990)
3. Sauder, R.L.: A general test data generator for cobol. In: Proceedings of the May 1-3, 1962, spring joint computer conference. pp. 317–323. ACM (1962)
4. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability 14(2), 105–156 (2004)
5. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Transactions on Software Engineering 36(6), 742–762 (2010)
6. Holland, J.H.: Genetic algorithms. Scientific american 267(1), 66–73 (1992)
7. Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I.: Software architecture optimization methods: A systematic literature review. IEEE Transactions on Software Engineering 39(5), 658–683 (2013)
8. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE transactions on software engineering (12), 1085–1110 (2001)
9. Miller, J., Reformat, M., Zhang, H.: Automatic test data generation using genetic algorithm and program dependence graphs. Information and Software Technology 48(7), 586–605 (2006)
10. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Transactions on Software Engineering 36(2), 226–247 (2010)
11. Cohen, E.I.: A finite domain-testing strategy for computer program testing. Ph.D. thesis, The Ohio State University (1978)
12. Elshoff, J.L.: An analysis of some commercial pl/i programs. IEEE Transactions on Software Engineering (2), 113–120 (1976)
13. Ryan, C., Collins, J.J., Neill, M.O.: Grammatical evolution: Evolving programs for an arbitrary language. In: European Conference on Genetic Programming. pp. 83–96. Springer (1998)
14. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation 5(4), 349–358 (2001)
15. Harman, M., Jia, Y., Zhang, Y.: Achievements, open problems and challenges for search based software testing. In: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on. pp. 1–12. IEEE (2015)
16. Boyer, R.S., Elspas, B., Levitt, K.N.: Selecta formal system for testing and debugging programs by symbolic execution. ACM SigPlan Notices 10(6), 234–245 (1975)
17. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering (3), 223–226 (1976)

18. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on software engineering (3), 215–222 (1976)
19. Ramamoorthy, C.V., Ho, S.B., Chen, W.: On the automated generation of program test data. IEEE Transactions on software engineering (4), 293–300 (1976)
20. Offutt, A.J.: An integrated automatic test data generation system. In: Case Technology, pp. 129–147. Springer (1991)
21. DeMilli, R., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), 900–910 (1991)
22. Offutt, A.J., Jin, Z., Pan, J.: The dynamic domain reduction procedure for test data generation. Software: Practice and Experience 29(2), 167–193 (1999)
23. Korel, B.: Automated software test data generation. IEEE Transactions on software engineering 16(8), 870–879 (1990)
24. Korel, B.: Dynamic method for software test data generation. Software Testing, Verification and Reliability 2(4), 203–213 (1992)
25. Korel, B.: Automated test data generation for programs with procedures. In: ACM SIGSOFT Software Engineering Notes. vol. 21, pp. 209–215. ACM (1996)
26. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology (TOSEM) 5(1), 63–86 (1996)
27. Tracey, N., Clark, J., Mander, K., McDermid, J.: An automated framework for structural test-data generation. In: ase. p. 285. IEEE (1998)
28. Tracey, N., Clark, J.A., Mander, K.: The way forward for unifying dynamic test-case generation: The optimisation-based approach. In: Proceedings of the IFIP International Workshop on Dependable Computing and Its Applications (DCIA). York (1998)
29. Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., Karapoulios, K.: Application of genetic algorithms to software testing. In: Proceedings of the 5th International Conference on Software Engineering and Applications. pp. 625–636 (1992)
30. Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Software engineering journal 11(5), 299–306 (1996)
31. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithms. Software testing, verification and reliability 9(4), 263–282 (1999)
32. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841–854 (2001)
33. Arcuri, A., Yao, X.: Search based software testing of object-oriented containers. Information Sciences 178(15), 3075–3095 (2008)
34. Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. Journal of Systems and Software 103, 311–327 (2015)
35. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering 39(2), 276–291 (2013)
36. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on. pp. 1–10. IEEE (2015)
37. Karim, M.R., Ryan, C.: Sensitive ants are sensible ants. In: Proceedings of the 14th annual conference on Genetic and evolutionary computation. pp. 775–782. ACM (2012)
38. bibclean.c. `http://www.cs.bham.ac.uk/~wbl/biblio/tools/bibclean.c` (1995 (accessed November 09, 2018))
39. Reeves, C.R., Rowe, J.E.: Genetic algorithms principles and presentation, a guide to ga theory (2002)

40. Mitchell, M., Forrest, S., Holland, J.H.: The royal road for genetic algorithms: Fitness landscapes and ga performance. In: Proceedings of the first european conference on artificial life. pp. 245–254 (1992)
41. DeMillo, R.A., Offutt, A.J.: Experimental results from an automatic test case generator. ACM Transactions on Software Engineering and Methodology (TOSEM) 2(2), 109–127 (1993)