Seeding Grammars in Grammatical Evolution to Improve Search Based Software Testing

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

Biocomputing and Developmental Systems Group, Department of Computer Science and Information Systems, Lero - The Irish Software Research Centre, University of Limerick, Ireland {sheraz.anjum,conor.ryan}@ul.ie

Abstract. Software-based optimization techniques have been increasingly used to automate code coverage analysis since the nineties. Although several studies suggest that interdependencies can exist between condition constructs in branching conditions of real life programs e.g. *(i <= 100) or (i == j)*, etc., to date, only the *Ariadne* system, a Grammatical Evolution (GE)-based Search Based Software Testing (SBST) technique, exploits interdependencies between variables to efficiently automate code coverage analysis.

Ariadne employs a simple attribute grammar to exploit these dependencies, which enables it to very efficiently evolve highly complex test cases, and has been compared favourably to other well-known techniques in the literature. However, Ariadne does not benefit from the interdependencies involving constants e.g. *(i <= 100)*, which are equally important constructs of condition predicates. Furthermore, constant creation in GE can be difficult, particularly with high precision.

We propose to *seed* the grammar with constants extracted from the source code of the program under test in order to enhance and extend Ariadne's capability to exploit richer types of dependencies (involving all combinations of both variables and constant values). We compared our results with the original system of Ariadne against a large set of benchmark problems which include 10 numeric programs in addition to the ones originally used for Ariadne. Our results demonstrate that the seeding strategy not only dramatically improves the generality of the system, as it improves the code coverage (effectiveness) by impressive margins, but it also reduces the search budgets (efficiency) often up to an order of magnitude.

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

1 Introduction

An important aspect of software quality assurance is software testing and, in practice, manual testing of a software system is laborious. It has been reported

in the various studies that manual testing can consume up to 50% of the total development budget [1,2]. In order to reduce the associated cost, many researchers [3,4,5,6] have been investigating the use of metaheuristic techniques to reduce the need of human intervention in the testing process; this field of study is referred to as *Search Based Software Testing* (SBST).

Genetic Algorithms (GAs) [7] are the most widely employed heuristic-based search techniques [8,5] in SBST and this subfield of SBST is referred to as *Evolutionary Testing* (ET). The most commonly targeted test adequacy criterion in SBST is *full branch coverage* [5], which ensures that all parts of the code are reachable. For the purpose of this paper, we have chosen *full condition-decision coverage* as the target which is an extended and thus more challenging to achieve as compared to branch coverage (detailed in Section 2).

Condition predicates of real life programs often contain interdependencies between variables and constant values, e.g. a condition to check if two variables are equal or if the value of a particular variable is between 100 and 500, as branching conditions often include the boundary values as constants. These facts are well established and have been reported in several research studies, for example, [9] studied 120 production PL/I programs and reported that 98% expressions included less than two operators while 62% of them were relational operators. In another study [10], 50 COBOL programs were analyzed and it was reported that 64% of the total predicates were equality checks and 77% of the predicates contained only a single variable; which means that majority of these predicates contained the comparison between variables and constant values.

To the best of our knowledge, *Ariadne* [11] is the only SBST technique proposed to date that exploits the interdependencies between input variables; however, it does not benefit from interdependencies involving constants which are equally important constructs of condition predicates as also apparent from the studies discussed above. Furthermore, Ariadne is a Grammatical Evolution (GE) [12,13] based system and constant creation in GE can be difficult [14,15], particularly with high precision. Therefore, it can be very difficult for Ariadne to find test data that can satisfy conditions containing any dependencies on constant values, particularly in cases where search spaces are large and complex.

GE is a grammar-based evolutionary algorithm that uses a grammar-based mapping process to separate search space from solution space. In recent years, GE has been successfully adopted to solve many software engineering problems from a wide variety of domains, including software effort estimation [16], vulnerability testing [17], integration and test order problem [18], game development [19], failure reproduction [20], software project scheduling [21] and software product line testing [22]. To the best of our knowledge, Ariadne is the only system proposed to date that targets the structural coverage testing of procedural C/C++ programs.

In this paper, we propose an improved attribute grammar for Ariadne that enhances and extends its capability to exploit interdependencies between condition constructs, by harvesting constants from the code under test and then *seeding* the grammar with them, thus making them directly available to indi-

viduals, obviating the need to evolve specific constants, and hence improving Ariadne's ability to achieve higher code coverage. The new design of grammar allows variables to take values dependent on both the previously generated variables and the extracted constant values (detailed in Section 4), which enables the system to exploit all kinds of interdependencies (involving both variables and constant values) during the whole of evolutionary process.

For the purpose of our experimentation, we employed a large set of benchmark programs which includes 10 numeric programs (that heavily rely on constant values) in addition to the ones adopted by [11]. We also created three new, extremely difficult to test programs, which contain deep levels of nesting, compound conditions and interdependencies involving both variables and constant values. We adopted condition-decision coverage as the test adequacy criterion to make a fair comparison with both original results of Ariadne and well-known results from the literature [23,24].

Our results suggest that the improved grammar dramatically improves the effectiveness of Ariadne by achieving a 100% coverage (also referred to as full coverage) on **all** benchmark programs, while the original system was not able to achieve full coverage for any of the programs that heavily used constant values. Our results also demonstrate that the improved grammar does not trade off the efficiency of the system to improve its generality as it further reduces the search budgets often up to an order of magnitude.

This paper begins with an overview of search based test data generation techniques (Section 2), followed by an introduction to Ariadne: A GE-based test data generator (Section 3). In Section 4, we present our improved grammar for Ariadne and also the philosophy behind the proposed changes in the original grammar. Finally, in Section 5, we empirically evaluate the performance of our improved system of Ariadne on a large selection of benchmark problems.

2 Background and Related Work

Structural testing inspects the code based on knowledge of its internal structure. There are multiple code *coverage criteria* which are essentially conditions with varying strictness. A coverage criterion, if met, ensures the absence of certain types of errors in the code. For example, to achieve 100% condition-decision coverage (also referred to as full condition-decision coverage), a piece of code must be executed with a test suite (set of input values) such that all of both the condition predicates and branching conditions take both possible outcomes of TRUE and FALSE at least once.

Manually achieving any type of code coverage is a laborious and difficult task as a human tester has to find a set of input values that can satisfy the respective condition(s). In order to reduce this testing cost, researchers have been trying to minimize the need for human intervention in the testing process since the 1960s [25]. It has been the subject of increasing research interest in recent years [26].

In any SBST technique, the goal is to heuristically search for a test suite that satisfies a chosen test adequacy criterion for the given program. One of the earliest SBST techniques [25] used random search for this purpose. *Random test data generation* can adequately deal with simpler problems but its scalability can be a challenge when dealing with problems having significantly complex and large search spaces.

Another SBST paradigm, known as *static test data generation*, employs some mathematical system to find the test suite. Symbolic Execution (SE) [27] is one such technique, in which a mathematical expression is formulated by placing some symbolic values at the place of program variables. The result of this expression is a set of input values that can satisfy the adequacy criterion. SE is generally supposed to resolve constraints and variable interdependencies in order to execute the required parts of the program but it has its own shortcomings which include handling procedure calls, loops, pointers and complexity of constraints. Other notable static test data generation techniques include domain reduction [28] and dynamic domain reduction [29]. These techniques address some of the inherent challenges of SE but handling of loops and pointers remains an open question.

A relatively more refined SBST approach found in the literature is *dynamic test data generation*, which essentially involves running the program under test. The execution behavior of the program is observed and this information is used to guide the search towards the required test data. This approach was first proposed by [30] and later extended/improved by various researchers [31,32]. All the above mentioned research works employed some Local Search Algorithm (LSA) and hence involve the inherent risk of getting stuck in some local minima.

To address some of the inherent challenges associated with LSAs, some global search based techniques including GA-based techniques [33,34,35,23,36] and simulated annealing-based techniques [37] have been proposed by researchers. Further, to get the benefits of both local and global search algorithms, some Memetic Algorithm (MA) based techniques [24,38] have also been investigated in the literature.

SBST techniques conventionally search for one sub-goal at a time e.g. in the case of condition coverage, the set of input values that can result into a particular outcome of a specific condition predicate is searched at one time. Some proposed techniques including *whole test suite generation* [39], [38] and *many-objective optimization* [40], search for multiple targets simultaneously.

2.1 Evolutionary Testing

In Evolutionary Testing (ET), a GA is employed to find the test suite from the domain of all possible input values for the program under test. Each individual in the population represents one possible set of input values and its fitness is calculated based on the execution of target program when run with the respective input values (test case). The code of the target program is usually instrumented to monitor its execution behavior; this instrumentation is done in conjunction

with GA's fitness function as both are designed according to the chosen test adequacy criterion.

Many variations of fitness functions can be found in the literature, but most of them rely on one or both of two measures, namely, branch distance and control flow information. The interested reader can refer to [31] and [35] for the concepts of *branch distance* and *approximation level* (control flow information) respectively.

The earliest ET technique to use a branch distance based fitness function was proposed by [41] and the earliest works that used control flow information for measuring the fitness include [33] and [34]. The fitness function deployed in [33] was primarily based on branch distance but some control flow information was also incorporated for loop testing, whereas [34] used a purely control flow based fitness function. Later, [35] proposed a hybrid fitness measures in order to attain the benefits associated with both of the measures.

2.2 SBST Techniques Benefitting From Seeding

As one of the key observations underpinning this work is the exploitation of domain knowledge in the process of test data generation, here we present some other SBST techniques that also take advantage of some related knowledge. In general, use of any previous knowledge to help solve a problem can also be referred to as *seeding*.

There are several papers in the literature on SBST that have shown that different seeding strategies can strongly influence the search process. For example, [42] proposed seeding the evolutionary algorithm with structural test data to efficiently find worst-case execution times of real-time systems. Later, [43] proposed to extract knowledge from source code, documentation and programmers and seed it to reduce qualitative human oracle costs. In another study, [44] investigated the impact of exploiting common object usage for the problem of automatic test data generation. Soon after that, seeding strategies were also explored in the domain of software product lines [45]. More recently [46,47] studied the impact of injecting knowledge, through different seeding strategies, for the problem of service composition.

Previous work has also shown that extracting and directly seeding the constant values from source code of program can significantly improve the structural coverage testing [48,49,50,51], particular for programs heavily relying on constant values. However, the impact of seeding is prominent only in earlier phases of search as the seeded values can be modified (through the genetic operators of crossover and mutation) during the evolutionary process. In this paper, we propose to inject the extracted constants values in the attribute grammar of Ariadne (described in Section 4). This will permit the system to evolve the required dependencies involving both constants and variables throughout the search process. In other words, seeding the grammar allows the system to exploit the provided knowledge (i.e. the constant values) at any stage of the evolutionary process.

3 Ariadne: GE-based test data generation

Ariadne is an SBST technique that uses GE as a search algorithm to find/evolve the required test data from the set of all possible input values for the program under test. It uses a simple attribute grammar (presented in Section 3.2) to exploit interdependencies present among input variables.

Ariadne targets full condition-decision coverage, which is an extended and thus more challenging form of branch coverage. The overall operation of Ariadne is shown in Fig 1, where o_1 to o_n represent the list of separate search objectives consisting of TRUE and FALSE outcomes of all the branching nodes (b_1 to b_l) and condition predicates (c_1 to c_m).

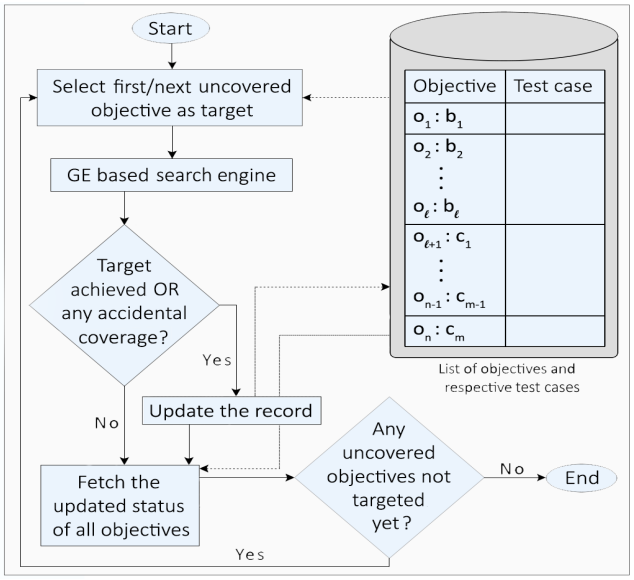

Fig. 1: System Flow Diagram of Ariadne: A GE-based test data generator.

Ariadne linearly selects its target from the list of search objectives and then performs a GE-based search to find the set of input values that can satisfy the current search objective. The GE-based search terminates as soon as the current target is achieved, otherwise, it keeps on running until the number of allowed generations are exhausted. This whole search process is repeated once for all the uncovered objectives, as some of the objectives are covered serendipitously (accidental coverage). The *efficiency* and *effectiveness* of any ET technique is measured in terms of total number of fitness evaluations and percentage of covered search objectives, respectively.

3.1 Grammatical Evolution

GE is essentially a GA that separates the search space (genotype) from solution space (phenotype) using a grammar-based mapping process. A problem-specific grammar is designed for this purpose which is comprised of four elements, i.e., terminals (T), non-terminals (N), productions rules (P) and a start symbol (S). Here, terminals are the only items that can appear in the final phenotype, while non-terminals are intermediate elements which are associated with the production rules. The mapping process always begins with the start symbol and, as it proceeds, production rules direct the mapping process.

In GE, the genotype is simply a list of integers which, in general, is represented using a binary string. GE consumes the genotype (integer-by-integer) in the process of making choices among available production rules using the following formula:

Rule = (integer value) mod (# of choices for the non-terminal at hand)

Let us consider an example where the non-terminal of <operator> is about to be expanded, while it is associated with the following four production rules:

<operator> ::= * [0]
| / [1]
| + [2]
| - [3]

Assume that the next integer to be consumed by GE engine is 62, then *62 mod 4=2*, so option #2 is selected for the further expansion of <operator> i.e. (*<operator> ::= +*). A sample grammar with a complete genotype to phenotype mapping is presented in Fig 2.

3.2 Grammar

In this section, we present the attribute grammar used in Ariadne [11] to exploit the commonly found characteristics of real life programs. The start symbol, in this case, is linked to the following production rule:

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

where N represents the total number of input variables required by the target program. Each of the above non-terminals of the form $<var_M>$ is further linked with the following set of production rules:

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

The first three choices of the above rule enable Ariadne to quickly satisfy the commonly found zero, positive and negative value checks as the values of 0, 1 and -1 represent these domains, respectively. The next production rule of <rand> is responsible for the production of 32 bit signed random numbers.

The remaining non-terminals of the form $<dep_{var_X}>$ implement the dependency rules. These dependency rules essentially enable the system to exploit variable interdependencies as they allow the input variables to take values dependent on previously generated variables. These non-terminals of the form $<dep_{var_X}>$ are expanded using the following set of production options:

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

where var_X refers to a previously generated variable. These newly generated values will be equal-to, greater-than or less-than the value of some previously generated variable; hence, the conditions involving comparisons/dependencies between the variables can be quickly satisfied.

4 Improved Grammar

A key distinguishing feature of Ariadne is its use of GE as a search algorithm (in place of conventional GAs). Design of a grammar is crucial and can have huge implications on the performance of any GE system; ideally the grammar used for test data generation should be both generic (so that it can be effectively applied to a wide range of programs) and efficient.

This section presents our newly proposed grammar design while its implications and the underpinning philosophy are detailed below in Section 4.1.

In our improved design, the non-terminal of $<var_M>$ is linked to the following set of production rules for their expansion:

$$\begin{aligned} <var_M> ::= 0 | 1 | -1 | <const> \mid <rand> \mid <dep_{var_1}> \mid <dep_{var_2}> \\ \mid \ldots \mid <dep_{var_{M-2}}> \mid <dep_{var_{M-1}}> \end{aligned} \tag{4}$$

The newly introduced production rule of $<const>$ is further associated with the following choices of production rules:

$$<const> ::= 0 | C_1 | C_2 | C_3 | \ldots | C_N \tag{5}$$

where C_1 to C_N represent the list of seeded constant values which are simply extracted from the condition predicates in the source code. This innovation allows the variables to take values directly from the pool of seeded constants by right combinations of Rule 4 and 5. Once generated, these values become part of the grammar and remain available to be exploited by the dependency rules of the form $<dep_{var_X}>$, as described in Section 3.2. Consequently, the improved Ariadne can quickly evolve test data required to satisfy complex branching conditions that contain dependencies involving both variables and constant values.

The rest of the design is kept the same as that of the original grammar (presented in Section 3.2). An example with a complete grammar and grammar-based genotype to phenotype mapping for a program with three input variables and nine seeded constants is presented as Fig. 2. Note that this same generic grammar is used for all our experiments; only the number of input variables and the list of extracted constants (seeds) were modified as per each program.

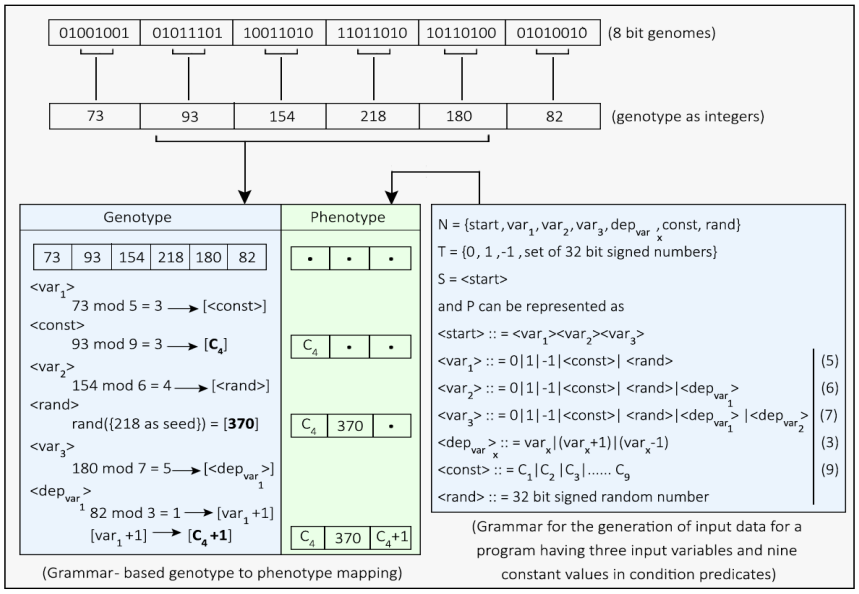

Fig. 2: An example with the genotype on the top, grammar on the right and the mapping sequence on the left.

4.1 Philosophy Behind the Proposed Changes

Ariadne, by design, does not solely rely on the evolutionary process to search for the required solution, but it also exploits variable interdependencies using its grammar, as described in Section 3. Results reported in [11] demonstrate that Ariadne clearly outperformed the well-known GA-based techniques by impressive margins. However, the original system of Ariadne is not capable of exploiting any dependencies involving constant values; furthermore, constant creation in GE with such an enormous range is a very difficult task.

Dependencies involving constant values are very common as discussed in Section 1. For example, a branching condition may contain a boundary value and look like this:

$$x > y \;\; \&\& \;\; z == 5000 \tag{6}$$

In general, it is very difficult for a conventional GA to fortuitously generate test data that can satisfy these kinds of branching condition, particularly when the search space is large. It becomes even more difficult for the original system of Ariadne as it additionally faces difficulties in the creation of constant values. To address this problem, we proposed an improved grammar for Ariadne that is capable of exploiting all kinds of interdependencies/comparisons involving both variables and constant values.

It is worth noting that the seeded constants stay available (as a part of grammar rules) throughout the search process; hence, they can also play their role

in the evolution of the values required for satisfying some deeper level nested conditions, which are only reached after some initial generations of the evolutionary process. To conclude, our novel design greatly improves Ariadne's capability to exploit interdependencies present among all kinds of condition constructs by enabling it to exploit dependencies involving constant values.

5 Experimental Results and Discussion

An empirical study was performed using three different sets of benchmark functions. The first set, **Set 1**, contains ten numeric functions[1] that heavily rely on constant values. The second, **Set 2**, includes the same well-known numeric and validity-check functions[2] that were originally adopted by [11] to compare with the earlier GA based techniques proposed in [23,36] and [24].

Set 1 contains seven real life programs and three synthetic programs of varying complexity. The real life programs include Tax Calculator, Admission Merit, Vitamin D Levels, Birth-Time Weights, HBA1c Levels (blood glucose levels), Grade Point Average (GPA) Calculator and Volume Discount. These programs are well-known and self-explanatory and their branching conditions often contain the boundary values (which are essentially constant values). The synthetic programs $S1$, $S2$ and $S3$, are artificially created to be difficult coverage targets of varying complexity as they contain deep nesting (up to four levels), compound conditions and interdependencies among the condition constructs (involving both variables and constant values).

We employed Set 2 to make a fair comparison with the original system of Ariadne and also with earlier well-known results from the literature [23,24]. For the purpose of this paper, we adopted only those numeric functions that had average search costs of at least 10 fitness evaluations in the previously reported results [11] as the rest of the benchmark functions proved trivial for grammar-based approach. The adopted numeric function include Days, Quadratic Formula (QCF) and *Triangle Classification* which is one of the most commonly adopted functions in SBST [32,34,23,36] etc. While two validity-check functions named check ISBN and check ISSN are a part of an open source program, **bibclean-2.08** [52]. These all are among popular benchmark functions in SBST and their short descriptions as well as justifications for their selection can be found in [11].

5.1 Experimental Setup

We first conducted some initial experiments to identify reasonable settings for GE run. We noticed that the maximum of 200 generations with a population size of 50 were found appropriate for all but some synthetic functions. So the synthetic functions, being more complex, were run with a population size of 200

[1] In order to facilitate future comparisons we have made available the source code at `http://bds.ul.ie/?page_id=390/`.

[2] The source code of these benchmark functions was made available by [11] at `http://bds.ul.ie/?page_id=390/`.

and maximum number of generations was kept at 500. For a fair comparison with [11], the crossover and mutation operators (i.e. One Point Crossover & Bit Mutation) and their probabilities (crossover: 0.9, mutation: 0.05) were kept the same as that of original system of Ariadne.

The input values generated by our improved grammar lie in the same range as that of the original system (i.e. from the range of $-2,147,483,648$ to $2,147,483,647$) as both systems generate 32-bit signed integers. These integer values directly serve as input values for most of the benchmark functions; for the functions of Days, check_ISBN and check_ISSN, an extra mod based step was deployed by [11] to convert these integer values into valid input formats as per the respective functions. For the sake of our experiments, we also used a similar mapping step for both input and seeding in order to have a fair comparison with the original system.

5.2 Detailed Analysis of Experiments

We performed **200 independent runs** for all the benchmarks and present their **mean** performance. We also repeated the same set of experiments using the original grammar in order to have a better statistical comparison with [11]; our results were very similar to the originally reported results.

We report our results in terms of three metrics, i.e., Maximum Coverage (MC), Success Rate (SR) and average number of fitness evaluations (AE). MC is the best performance (maximum achieved coverage) of all 200 runs. SR for each coverage target is the percentage of 200 runs/times that the target was successfully covered. AE is the average number of benchmark function executions that were performed in each run.

It can be clearly seen in Table 1 that the original system was not able to achieve a full coverage for any of the benchmark programs from Set 1 (which requires the generation of specific constant values). Despite being given a decent search budget, the maximum coverages achieved by the original system remain in the range of 25% to 75%. On the other hand, our improved system exhibited a full coverage (i.e a 100% coverage) in all of its runs; hence achieving a 100% SR for all the benchmark functions from Set 1.

The original system was never able to attain a full coverage as all these benchmark functions contain boundary values in their branching conditions. In other words, they contain interdependencies involving constant values. The original system is neither able to exploit these interdependencies nor able to successfully evolve constant values, therefore, it could never generate the test data required to satisfy these branching conditions. On the other hand, our improved system was able to exploit the presence of these boundary values as they were directly seeded in the grammar, and hence it was able to quickly evolve the test data containing all the dependencies (involving both variables and constant values) needed to satisfy these branching conditions.

For all the benchmark functions from Set 2, both the original system and our improved system were able to exhibit a 100% SR as presented in Table 2. As it

Table 1: A comparison of our improved Ariadne with the original system of Ariadne [11] on ten benchmark functions in Set 1. MC, SR and AE are maximum coverage, success rate and average number of fitness evaluations, respectively.

Branch ID	Original Ariadne [11]			Improved Ariadne		
	MC	SR	AE	MC	SR	AE
Tax Calculator	67%	0%	20108	100%	100%	27
Admission Merit	25%	0%	150759	100%	100%	827
Vitamin D Levels	63%	0%	30157	100%	100%	34
Birth-time Weights	67%	0%	20108	100%	100%	20
HBA1c Levels	75%	0%	10058	100%	100%	11
GPA Calculator	56%	0%	70359	100%	100%	96
Volume Discount	58%	0%	50259	100%	100%	57
S1	38%	0%	501003	100%	100%	134
S2	70%	0%	601223	100%	100%	2608
S3	56%	0%	801606	100%	100%	11202

was also reported in [11] that the original system was already able to achieve a full coverage for these programs, the purpose of adopting these benchmarks here was to study if our improved system was also able to retain similar good results (both in terms of effectiveness and efficiency) for these well-known benchmarks in SBST. In order to have a fair comparison with [24,11], the experiments for the validity check functions were performed on the same lines and the results were separately reported for all the non-trivial branches.

Table 2 shows that our improved system retained a 100% SR while consuming significantly smaller search budgets, particularly for the validity-check functions where the AE was reduced to anything just from 9% to 14% of that of the original system. The reason behind this dramatic improvement in efficiency is the presence of interdependencies involving constant values, which were successfully exploited by our improved system via seeding strategy. For example, the validity-check functions contained many constants in the condition predicates, which were made a part of the grammar using Rule 5. The conditions containing comparisons/dependencies involving these (seeded) constant were quickly satisfied by the function of dependency rules as described in Section 3.2. In can also be clearly seen that these improvements are even more impressive when compared to other GA-based techniques.

To conclude, the results presented in this section demonstrate that the grammar is made more generic without compromising on its efficiency as our improved system clearly outperforms the original system of Ariadne as well as the other GA based SBST techniques (both in terms of effectiveness and efficiency) by wide margins.

Table 2: A comparison of our improved Ariadne with the original system of Ariadne [11] and with earlier GA-based techniques [23,24]. MC, SR and AE are maximum coverage, success rate and average number of fitness evaluations, respectively.

Branch ID	**Conventional GAs**			**Original Ariadne [11]**			**Improved Ariadne**		
	MC	**SR**	**AE**	**MC**	**SR**	**AE**	**MC**	**SR**	**AE**
GADGET[23]									
Tri	94%	N/A	8000	100%	100%	958	100%	100%	355
Days	100%	N/A	N/A	100%	100%	288	100%	100%	218
QCF	75%	N/A	N/A	100%	100%	16	100%	100%	13
Harman & McMinn [24]									
B3-ISBN	100%	95%	7986	100%	100%	591	100%	100%	69
B4-ISBN	100%	95%	7986	100%	100%	581	100%	100%	69
B6-ISBN	100%	95%	8001	100%	100%	718	100%	100%	70
B7-ISBN	100%	95%	9103	100%	100%	4215	100%	100%	594
B3-ISSN	100%	98%	5273	100%	100%	525	100%	100%	47
B4-ISSN	100%	98%	5273	100%	100%	522	100%	100%	50
B6-ISSN	100%	98%	5324	100%	100%	584	100%	100%	53
B7-ISSN	100%	98%	6380	100%	100%	3755	100%	100%	344

6 Conclusion and Future Work

We have proposed to seed the grammar with constants extracted from source code in order to improve its effectiveness/generality; this improved grammar is capable of exploiting a richer class of dependencies (involving both variables and constant values). We compared our results with the original system of Ariadne against the same sets of benchmark functions that were originally used as well as against an additional set of 10 numeric programs. The results of our experiments show that the seeding strategy improves the effectiveness/generality of the system by impressive margins without compromising on its efficiency as it further reduces the search budgets often up to an order of magnitude. In other words, the improved system clearly outperforms both the original system of Ariadne as well as the other GA based SBST techniques both in terms of effectiveness and efficiency.

We believe that there is much potential to further improve this GE based SBST technique. For example, the seeding strategy can be further improved by adding support for numeric values observed at run time (dynamic seeding) and/or by exploring the possibility of accommodating other data types such as strings, as currently only numeric values are seeded in the grammar. The grammar can also be improved by systematically adding additional domain knowledge. Further, we are also conducting a rigorous study to investigate the scalability of GE-based test data generation.

This paper is the first to propose, investigate and discuss the implications of seeding the grammars in GE. Although we have used the seeding strategy in the area of SBST, we believe that there is huge potential to benefit from this

strategy in other GE-based systems from different domains in which constants and other low level structures are present in the problem description.

Acknowledgments

The authors would like to thank Aidan Murphy, Muhammad Hamad Khan and Sehrish Saeed for their help with conceptualization of the idea, graphic designs and benchmark functions, respectively. This work is supported by the Science Foundation of Ireland (SFI) Grant Number 16/IA/4605.

References

1. Beizer, B.: Software testing techniques, van nostrand reinhold. Inc, New York NY, 2nd edition. ISBN 0-442-20672-0 (1990)
2. Myers, G.J., Badgett, T., Thomas, T.M., Sandler, C.: The art of software testing, vol. 2. Wiley Online Library (2004)
3. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability 14(2), 105–156 (2004)
4. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology 51(6), 957–976 (2009)
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. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software 86(8), 1978–2001 (2013)
7. Holland, J.H.: Genetic algorithms. Scientific american 267(1), 66–73 (1992)
8. 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)
9. Elshoff, J.L.: An analysis of some commercial pl/i programs. IEEE Transactions on Software Engineering (2), 113–120 (1976)
10. Cohen, E.I.: A finite domain-testing strategy for computer program testing. Ph.D. thesis, The Ohio State University (1978)
11. Anjum, M.S., Ryan, C.: Ariadne: Evolving test data using grammatical evolution. In: European Conference on Genetic Programming. Springer (2019)
12. 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)
13. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Transactions on Evolutionary Computation 5(4), 349–358 (2001)
14. Dempsey, I., O'Neill, M., Brabazon, A.: Constant creation in grammatical evolution. International Journal of Innovative Computing and Applications 1(1), 23–38 (2007)
15. Azad, R.M.A., Ryan, C.: The best things dont always come in small packages: Constant creation in grammatical evolution. In: European Conference on Genetic Programming. pp. 186–197. Springer (2014)

16. Barros, R.C., Basgalupp, M.P., Cerri, R., da Silva, T.S., de Carvalho, A.C.: A grammatical evolution approach for software effort estimation. In: Proceedings of the 15th annual conference on Genetic and evolutionary computation. pp. 1413–1420. ACM (2013)
17. Sparks, S., Embleton, S., Cunningham, R., Zou, C.: Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). pp. 477–486. IEEE (2007)
18. Mariani, T., Guizzo, G., Vergilio, S.R., Pozo, A.T.: Grammatical evolution for the multi-objective integration and test order problem. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016. pp. 1069–1076. ACM (2016)
19. Patten, J.V., Ryan, C.: Procedural content generation for games using grammatical evolution and attribute grammars (2014)
20. Kifetew, F.M., Jin, W., Tiella, R., Orso, A., Tonella, P.: Reproducing field failures for programs with complex grammar-based input. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 163–172. IEEE (2014)
21. de Andrade, J., Silva, L., Britto, A., Amaral, R.: Solving the software project scheduling problem with hyper-heuristics. In: International Conference on Artificial Intelligence and Soft Computing. pp. 399–411. Springer (2019)
22. Lima, J.A.P., Vergilio, S.R., et al.: Automatic generation of search-based algorithms applied to the feature testing of software product lines. In: Proceedings of the 31st Brazilian Symposium on Software Engineering. pp. 114–123. ACM (2017)
23. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE transactions on software engineering (12), 1085–1110 (2001)
24. 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)
25. 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)
26. 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)
27. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on software engineering (3), 215–222 (1976)
28. DeMilli, R., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), 900–910 (1991)
29. 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)
30. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering (3), 223–226 (1976)
31. Korel, B.: Automated software test data generation. IEEE Transactions on software engineering 16(8), 870–879 (1990)
32. 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)
33. Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Software engineering journal 11(5), 299–306 (1996)
34. 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)

35. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841–854 (2001)
36. 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)
37. Tracey, N., Clark, J., Mander, K., McDermid, J.: An automated framework for structural test-data generation. In: ase. p. 285. IEEE (1998)
38. Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. Journal of Systems and Software 103, 311–327 (2015)
39. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering 39(2), 276–291 (2013)
40. 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)
41. 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)
42. Tlili, M., Wappler, S., Sthamer, H.: Improving evolutionary real-time testing. In: Proceedings of the 8th annual conference on Genetic and evolutionary computation. pp. 1917–1924. ACM (2006)
43. McMinn, P., Stevenson, M., Harman, M.: Reducing qualitative human oracle costs associated with automatically generated test data. In: Proceedings of the First International Workshop on Software Test Output Validation. pp. 1–4. ACM (2010)
44. Fraser, G., Zeller, A.: Exploiting common object usage in test case generation. In: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. pp. 80–89. IEEE (2011)
45. Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Egyed, A., Alba, E.: Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In: 2014 IEEE Congress on Evolutionary Computation (CEC). pp. 387–396. IEEE (2014)
46. Chen, T., Li, M., Yao, X.: On the effects of seeding strategies: a case for search-based multi-objective service composition. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 1419–1426. ACM (2018)
47. Chen, T., Li, M., Yao, X.: Standing on the shoulders of giants: Seeding search-based multi-objective optimization with prior knowledge for software service composition. Information and Software Technology (2019)
48. Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 3–12. IEEE Computer Society (2011)
49. Fraser, G., Arcuri, A.: The seed is strong: Seeding strategies in search-based software testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 121–130. IEEE (2012)
50. Rojas, J.M., Fraser, G., Arcuri, A.: Seeding strategies in search-based unit test generation. Software Testing, Verification and Reliability 26(5), 366–401 (2016)
51. Bidgoli, A.M., Haghighi, H.: A new approach for search space reduction and seeding by analysis of the clauses. In: International Symposium on Search Based Software Engineering. pp. 343–348. Springer (2018)
52. bibclean.c. `http://www.cs.bham.ac.uk/~wbl/biblio/tools/bibclean.c` (1995 (accessed September 15, 2019))