Seeding Grammars in Grammatical Evolution to Improve Search-Based Software Testing

Muhammad Sheraz Anjum¹ · Conor Ryan¹

Received: 29 July 2020 / Accepted: 1 April 2021 / Published online: 19 May 2021
© The Author(s) 2021

Abstract

Heuristic-based optimization techniques have been increasingly used to automate different types of code coverage analysis. Several studies suggest that interdependencies (in the form of comparisons) may exist between the condition constructs, of variables and constant values, in the branching conditions of real-world programs, e.g. $(i \leq 100)$ or $(i == j)$, etc. In this work, by interdependencies we refer to the situations where, to satisfy a branching condition, there must be a certain relationship between the values of some specific condition constructs (which may or may not be a part of the respective condition predicates). For example, the values of variables i and j must be equal to satisfy the condition of $(i == j)$, and the value of variable k must be equal to 100 for the satisfaction of the condition of $(k == 100)$. To date, only the *Ariadne*, a Grammatical Evolution (GE)-based system, exploits these interdependencies between input variables (e.g. of the form $(i \leq j)$ or $(i == j)$, etc.) to efficiently generate test data. *Ariadne* employs a simple attribute grammar to exploit these dependencies, which enables it to evolve complex test data, and has been compared favourably to other well-known techniques in the literature. However, *Ariadne* does not benefit from interdependencies involving constants, e.g. $(i \leq 100)$ or $(j == 500)$, etc., due to the difficulty in evolving precise values, and these 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 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. Moreover, we also performed a rigorous analysis to investigate the *scalability* of our improved *Ariadne*, showing that it stays highly scalable when compared to both the original system of *Ariadne* and GA-based test data generation approach.

Keywords Automatic test data generation · Code coverage · Evolutionary testing · Grammatical evolution

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 [12, 46]. To reduce this cost, many researchers [1, 3, 5, 41] have investigated 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) [31] are the most widely employed metaheuristic search techniques [2, 3] in SBST

This article is part of the topical collection “Evolution, the New AI Revolution” guest edited by Anikó Ekárt and Anna Isabel Esparcia-Alcázar.

✉ Muhammad Sheraz Anjum
sheraz.anjum@ul.ie

Conor Ryan
conor.ryan@ul.ie

¹ Biocomputing and Developmental Systems Group,
Department of Computer Science and Information Systems,
Lero – Science Foundation Ireland Research Centre
for Software, University of Limerick, Limerick, Ireland

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* [3], 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 extended and thus more challenging to achieve than to branch coverage (detailed in "[Background and Related Work](#)").

Condition predicates of real-world programs often contain *interdependencies*, through relational operators, between variables and constant values. These dependencies may result in the need for highly precise data values (such as of the form $i == j == k$) for code coverage analysis. For example, for the satisfaction of conditions like $(i \leq j)$ or $(i == j)$, there must be a particular relationship between the values of variables i and j . Similarly, the value of variable i must be in some way dependent on the constant values of 100 and 500 to satisfy the conditions of $(i \leq 100)$ and $(j == 500)$, respectively. In more complex cases, all the relevant condition constructs might not directly be available in the respective condition predicates, as the nested conditions may create a chain of interdependencies.

The presence of such dependencies is a well-established fact as several research studies have reported similar findings, for example, one such study [22] examined 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 [18], 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 (and hence dependencies) between variables and constant values.

To the best of our knowledge, *Ariadne* [6] is the only SBST technique proposed to date that exploits the interdependencies between input variables and, as reported in our earlier works [6, 7], it significantly outperforms the existing GA-based techniques of [43, 44] and [30] both in terms of effectiveness (i.e. coverage percentage) and efficiency (i.e. associated costs). However, *Ariadne* 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) [48, 53]-based system and constant creation/optimization in GE can be difficult [9, 21], particularly with high precision, although, as demonstrated in "[Detailed Analysis and Discussion](#)", all GA-based SBST systems suffer from this problem. Thus, it can be difficult for *Ariadne* to find test data that can satisfy conditions containing dependencies that rely 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 [11], vulnerability testing [55], integration and test order problem [39], game development [51], failure reproduction [35], software project scheduling [19] and software product line testing [37]. To the best of our knowledge, *Ariadne* is the only GE-based system proposed to date that targets the structural coverage testing of procedural C/C++ programs.

The present paper complements and extends our previous work [8], where we proposed an improved attribute grammar for *Ariadne* to enhance and extend its capability of exploiting interdependencies between condition constructs. The new design works by harvesting constants from the code under test and then *seeding* the grammar with them, thus making them directly available to individuals, obviating the need to evolve specific constants, and hence improving *Ariadne*'s ability to achieve higher code coverage. This enhancement in the grammar allows variables to take values dependent on both the previously generated variables and the extracted constant values (detailed in "[Empirical Evaluation](#)"), which enables the system to exploit all kinds of interdependencies (involving both variables and constant values) during the whole of evolutionary process. It is also worth mentioning that our novel seeding strategy is more advantageous as compared to the conventional seeding approach (detailed in "[Philosophy Behind the Proposed Changes](#)").

In addition to that, as an in-depth analysis of any SBST technique is required to verify its *scalability*, we also present here the results of a rigorous study carried out to examine how our improved *Ariadne* performs/scales for increasingly complex testing problems. The inspiration for performing this analysis is taken from [43] and [7]; Michael et al. [43] studied the scalability of both GA-based and random test data generation approaches, while in [7], we previously investigated the scalability of the original system of *Ariadne* [6] in a comparison with GA-based test data generation approach. For the sake of this paper, we performed a detailed study carried out to examine the scalability of our improved *Ariadne* in a comparison with **both** the original system of *Ariadne* [6] and GA-based test data generation approach.

To empirically evaluate the performance of our improved *Ariadne*, 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 [6]. We also created three new programs which are extremely challenging to test as they contain deep levels of nesting, compound conditions and interdependencies involving both variables and constant values. Moreover, a large set of 18 highly tunable and increasingly complex numeric benchmark programs (which also contain interdependencies involving both variables and constant values) is designed for the sake of our

scalability analysis. These benchmark programs are formulated while taking into account the recommendations made in [40] and they comply with many of their recommended criteria of an *ideal benchmark suite*. The list includes, but is not limited to, precisely defined, tunably difficult, easy to implement and reproduce, fast (in regard to fitness evaluation), relevant and accessible to other implementers.

The results of our empirical evaluation suggest that the improved grammar improves the effectiveness of Ariadne by achieving a 100% coverage (also referred to as full coverage) on **all** testing problems, 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. Furthermore, the results of our detailed scalability analysis suggest that the improved system of Ariadne stays scalable as it also demonstrated a 100% coverage across **all** benchmark programs of increasing complexity. Our results show that the improved Ariadne exhibited this high coverage while consuming significantly smaller search budgets as compared to both the original Ariadne and the GA-based test data generation approach. In contrast, all other approaches showed poor performance and failed to achieve a 100% coverage by wide margins even after consuming huge search budgets.

The current scope of Ariadne is to automatically generate test cases (input data) for procedural programs, but it potentially can also be extended to automatically generate test cases (test programs) for object-oriented programs. For the purpose of this research, we have used C/C++ programs as benchmark problems; however, the same test data generation approach can also be employed for programs written in other procedural programming languages. Moreover, Ariadne primarily benefits from the presence of interdependencies among the numeric input variables, but it also provides some level of support for other data types as well as for implicit data dependencies, i.e., situations where condition predicates are defined on derived local variables rather than the input variables themselves, as described in "[Empirical Evaluation](#)". Furthermore, the proposed strategy of seeding the constant numeric values can also be extended in many ways, e.g. to add support for dynamic seeding or additional data types, etc., as discussed in "[Improved Grammar](#)".

This paper begins with an overview of search-based test data generation techniques ("[Background and Related Work](#)"), followed by an introduction to Ariadne ("[Ariadne: GE-Based Test Data Generation](#)"). Then we present our improved grammar for Ariadne and also the philosophy behind the proposed changes in the original grammar. Finally, we empirically evaluate the performance of our improved system of Ariadne on a large selection of benchmark problems ("[Empirical Evaluation](#)") followed

by a detailed scalability analysis of our improved system ("[Scalability Analysis of the Improved GE-Based Test Data Generation](#)").

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 set of input values (test data) 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). To reduce this testing cost, researchers have been trying to minimize the need for human intervention in the testing process since the 1960s [54]. It has been the subject of increasing research interest in recent years [29].

In any SBST technique, the goal is to heuristically search for a set of test data that satisfies a chosen test adequacy criterion for the given program. One of the earliest SBST techniques [54] 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 a mathematical system to find the test data. Symbolic Execution (SE) [17] 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 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 approaches include domain reduction [20] and dynamic domain reduction [47]. 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 [45] and later extended/

improved by various researchers [23, 36]. All the above-mentioned research works employed some Local Search Algorithm (LSA) and hence involve the inherent risk of getting stuck in local minima.

To address some of the inherent challenges associated with LSAs, global search-based approaches including GA-based techniques [34, 43, 44, 50, 58] and simulated annealing-based approaches [57] have been proposed by researchers. GA-based test data generation approach (i.e. Evolutionary Testing) is detailed in "Evolutionary Testing". Further, to get the benefits of both local and global search algorithms, Memetic Algorithm (MA) based techniques [26, 30] 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 approaches including *whole test suite generation* [25], [26] and *many-objective optimization* [49], search for multiple targets simultaneously.

Ariadne is different from the above-mentioned approaches with its primary focus on exploiting variable interdependencies. Moreover, Ariadne is capable of actually evolving the required set of interdependencies in a true sense by virtue of its grammar-based GA. Ariadne, like other GA-based test data generators, enjoys the perks of global search and dynamicity; however, unlike the conventional GA-based approaches, it does not need any external (and computationally expensive) support to satisfy complex numerical constraints. For example, [10, 28, 32, 60] and [33, 44] supplemented the GA with SE and data dependence analysis, respectively.

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 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, Tlili et al. [56] proposed seeding the evolutionary algorithm with structural test data to efficiently find worst-case execution times of real-time systems. Later, McMinn et al. [42] proposed to extract knowledge from source code, documentation and programmers and seed it to reduce qualitative human oracle costs. In another study, Fraser and Zeller [27] 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 [38]. More recently [15, 16] 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 [4, 14, 24, 52], particular for programs heavily relying on constant values. However, due to the nested nature of target programs, the impact of the conventional seeding approach is prominent only in the earlier phases of search process. For example, in case of branch coverage, some specific constant value(s) might be needed to satisfy a deeply nested branching condition. The evolutionary search process keeps on finding solutions that can satisfy more and more branching conditions towards the target branch, as it proceeds generation by generation (detailed in "Evolutionary Testing"). The required constant values, even if seeded in the beginning of the evolutionary process (i.e. in the initial population), can be modified (and hence lost) through the genetic operators of crossover and mutation as the deeply nested branching conditions are usually reached after a considerable number of generations.

In this research work, by **seeding** we refer to our novel strategy of injecting the extracted constant values in the attribute grammar of Ariadne (detailed description is provided in "Improved Grammar"), which enables Ariadne to evolve the required dependencies involving both constants and variables throughout the search process. In other words, the proposed seeding strategy allows the system to exploit the provided knowledge (i.e. the constant values) at any stage of the evolutionary process. Moreover, unlike the conventional seeding approach, our newly proposed seeding strategy is not limited to be only capable of satisfying equality comparisons (as described in "Philosophy Behind the Proposed Changes").

To the best of our knowledge, 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 for the automatic generation of test data, 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.

Evolutionary Testing

In Evolutionary Testing (ET), a GA is employed to find the test data 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 data). The code of the target program is usually instrumented to monitor its

Fig. 1 System flow diagram of Ariadne: a GE-based test data generator

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. *Branch distance* simply indicates that how close an individual is from **satisfying** the target condition. The measure of *control flow information*, on the other hand, describes how far an individual is from **reaching** the target. In other words, it represents the number of branching conditions that are satisfied on the way towards the target branch by the respective individual. As the evolutionary process keeps on finding increasingly better solutions (i.e. fitter individuals) over the course of generations, the fitness measures of branch distance and control flow information can direct the search towards satisfying the target condition and reaching the target branch, respectively. The interested reader can refer to [36] and [58] for the detailed description 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 [59] and the earliest works that used control flow information for measuring the fitness include [34, 50]. The fitness function deployed in [34]

was primarily based on branch distance but some control flow information was also incorporated for loop testing, whereas [50] used a purely control flow based fitness function. Later, [58] proposed a hybrid fitness measures to attain the benefits associated with both of the measures.

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 "Grammar") to exploit interdependencies present among input variables.

Ariadne targets full condition-decision coverage (detailed in "Background and Related Work"), 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 distinct *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). Here, l represents the total number of branching nodes and m represents the total number of individual condition predicates. Thus, the total number of search objectives, i.e. n , will be twice of the sum of l and m , as both TRUE and

Fig. 2 An overview of steps involved in grammatical evolution

FALSE outcomes of all the condition predicates and decisions (branching nodes) are considered as separate coverage objectives in the case of condition-decision coverage.

It is worth noting that, for the simple cases, where there is just a single condition predicate in a branching node, there is no need to separately consider the condition predicate as a search objective; however, in the case of a compound branching condition, all the individual condition predicates are separately added in the list of search objectives. In addition to that, the termination conditions of loop statements (i.e. *for*, *while* and *do-while*) and the cases of *switch* statements are also considered as branching nodes.

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. It is worth mentioning that the *accidental coverage* is also a common phenomenon in SBST as some of the objectives are covered serendipitously while the GA is targeting some other objectives. The *efficiency* and *effectiveness* of any ET technique is often measured in terms of total number of fitness evaluations and percentage of covered search objectives, respectively.

Grammatical Evolution

GE is a grammar-based GA that separates the search space (genotype) from solution space (phenotype) using a grammar-based mapping process. This mapping process is the only difference/addition in the mechanical process of GE, when compared to a conventional GA, as shown in Fig. 2. The genetic operations of selection, crossover and mutation are performed on the genotype while the fitness evaluation of every individual is performed using its phenotype. In other words, in GE, the genotype is a generic representation of individuals which is evolved independent of the problem representation; whereas, the phenotype is a problem-specific representation of individuals which is used for the purpose of fitness/suitability evaluation.

A problem-specific grammar is designed (for the genotype-phenotype mapping process) 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 binary string where each 8-bit codon corresponds to an integer value; these integers are used by the genotype-to-phenotype mapper to

Fig. 3 An example with the genotype on the top, grammar on the right and the mapping sequence on the left

make 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 $\langle operator \rangle$ is about to be expanded, while it is associated with the following four production rules:

$\langle operator \rangle ::= = * [0]$
 $\quad \quad \quad | / [1]$
 $\quad \quad \quad | + [2]$
 $\quad \quad \quad | - [3]$

Assume that the next integer to be used by GE engine is 62, then $62 \bmod 4 = 2$, so option #2 is selected for the further expansion of $\langle operator \rangle$ i.e. ($\langle operator \rangle ::= +$). A sample grammar with a complete genotype to phenotype mapping is presented in Fig. 3.

Grammar

In this section, we present the attribute grammar used in Ariadne [6] to exploit the commonly found characteristics of real-world programs. The start symbol, in this case, is linked to the following production rule:

$\langle start \rangle ::= \langle var_1 \rangle \langle var_2 \rangle \langle var_3 \rangle \dots \langle var_Z \rangle$, (1)

where Z represents the total number of input variables required by the target program. For example, the grammar shown as part of Fig. 3 is for a program with three input variables, hence the value of Z is 3 in this case. Each of the above non-terminals of the form $\langle var_Y \rangle$, where Y represents the number of variable for which the value is to be generated, is further linked with the following set of production rules:

$\langle var_Y \rangle ::= 0|1|-1|\langle rand \rangle|\langle dep_{var_1} \rangle|\langle dep_{var_2} \rangle|\dots|\langle dep_{var_{Y-2}} \rangle|\langle dep_{var_{Y-1}} \rangle$. (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 $\langle rand \rangle$ is responsible for the production of 32-bit signed random numbers.

The remaining non-terminals of the form $\langle dep_{var_x} \rangle$ to implement the dependency rules. Here, X represents the number of one of the previously generated variables, such as $1 \leq X < Y$. 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. For example, in the grammar shown as part of Fig. 3, variable number 3 can take a value which is dependent on the values of variable number 1 or 2 (as the set of available production rules associated with the non-terminal of $\langle \text{var}_3 \rangle$ also includes $\langle \text{dep}_{\text{var}_1} \rangle$ and $\langle \text{dep}_{\text{var}_2} \rangle$, along-with other options). The non-terminals of the form $\langle \text{dep}_{\text{var}_x} \rangle$ are further expanded using the following set of production options:

$$\langle \text{dep}_{\text{var}_x} \rangle ::= \text{var}_x | (\text{var}_x + 1) | (\text{var}_x - 1), \quad (3)$$

where var_x refers to the value of a previously generated (X th) input variable. The values generated by this rule will be equal-to, greater-than (made by adding 1) or less-than (made by subtracting 1) the value of some previously generated variable; hence, the conditions involving comparisons/dependencies between the variables can be quickly satisfied.

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 in "Philosophy Behind the Proposed Changes".

In our improved design, the non-terminals of the form $\langle \text{var}_y \rangle$ are linked to the following set of production rules for their expansion:

$$\begin{aligned} \langle \text{var}_y \rangle ::= & 0 | 1 | -1 | \langle \text{const} \rangle | \langle \text{rand} \rangle | \langle \text{dep}_{\text{var}_1} \rangle | \langle \text{dep}_{\text{var}_2} \rangle \\ & | \dots | \langle \text{dep}_{\text{var}_{y-2}} \rangle | \langle \text{dep}_{\text{var}_{y-1}} \rangle. \end{aligned} \quad (4)$$

The newly introduced production rule of $\langle \text{const} \rangle$ is further associated with the following choices of production rules:

$$\langle \text{const} \rangle ::= 0 | C_1 | C_2 | C_3 | \dots | C_S, \quad (5)$$

where C_1 to C_S represent the list of seeded constant values which are simply extracted from the source code. For the purpose of this paper, we extracted constant numeric values from the condition predicates of *if-else* statements, case values of *switch* statements and termination conditions of loop statements (i.e. *for*, *while* and *do-while*). However, the same seeding strategy can be adopted to inject any additional knowledge including, but not limited to, constants extracted from any other parts of the programs or even from other sources such as documentation, numeric values observed at

run time (often referred to as dynamic seeding), and values of other data types such as strings.

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 $\langle \text{dep}_{\text{var}_x} \rangle$, as described in "Grammar". 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 "Grammar"). 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. 3. 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.

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 "Ariadne: GE-Based Test Data Generation". Results reported in [6, 7] demonstrate that Ariadne clearly outperformed the well-known GA-based techniques. However, the original system of Ariadne is not capable of exploiting any dependencies involving constant values; furthermore, constant creation/optimization in GE with such an enormous range is a very difficult task.

Dependencies involving constant values are very common

as discussed in "Introduction". For example, a branching condition may contain a boundary value and look like this:

$$x > y \ \&\& \ z == 5000. \quad (6)$$

In general, it is very difficult for a conventional GA to fortuitously generate test data that can satisfy these kinds of branching conditions, particularly when the search space is large. It becomes even more difficult for the original Ariadne system as it additionally faces difficulties in the creation of constant values. To address this problem, we propose to harvest boundary values (constants) from the condition predicates in the source code and then injecting them in the attribute grammar of Ariadne (as described in "Improved Grammar"), which consequently leads to the

quick generation of the specific (constant) values needed to satisfy condition predicates.

Moreover, the condition predicates in the program under test may also include other relational dependencies (of $<$, $\leq$, $>$, $\geq$, $\neq$) in addition to the equality comparisons. For example, a branching condition may apply a range check and look like this:

$$z > 100 \ \&\& \ z < 150. \quad (7)$$

The values of 100 and 150, even if conventionally seeded in the evolutionary process, cannot satisfy these condition predicates. On the other hand, our improved system of Ariadne can also quickly generate the values needed to satisfy these kinds of conditions. For example, in this case, the values from the required range (i.e. 101 and 149) can be generated by the dependency rules of the form $< \text{dep}_{\text{var}_x} >$, as described in "Improved Grammar".

It is worth noting that, unlike the conventional seeding approach, the constants seeded in the grammar stay available (by virtue of Rule 5) 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 usually reached after a considerable number of initial generations (as described in "SBST Techniques Benefitting From Seeding" and "Evolutionary Testing"). For example, the value of input variable f should be equal to -80 to satisfy the condition predicate of $(f == -80)$, which lies at third level of nesting depth in the example program shown in Fig. 4. The required value can be generated by the right combination of Rules 4 and 5 at any stage (i.e. in any generation) of the evolutionary process.

To conclude, our novel design greatly improves Ariadne's ability to exploit interdependencies/comparisons present among all kinds of condition constructs by enabling it to exploit dependencies involving constant values.

Empirical Evaluation

To evaluate the performance of our improved Ariadne, we performed an empirical study using two different sets of benchmark functions. The first set, **Set 1**, contains ten numeric functions¹ that heavily rely on constant values. Numeric functions are the functions that manipulate numeric data. The second, **Set 2**, includes the same well-known numeric and validity-check functions² that were originally adopted by [6] to compare with the earlier GA

based techniques proposed in [43, 44] and [30]. We also performed a detailed scalability analysis of our improved system of Ariadne using a different set of benchmark functions (detailed in "Test Functions and Experimental Setup") and presented the results separately in "Scalability Analysis of the Improved GE-Based Test Data Generation".

Set 1 contains seven real-world programs and three synthetic programs of varying complexity. The real-world 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 [30, 43]. 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 [6] 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 [23, 43, 44, 50], etc. One of the features that make Triangle Classification a challenging test data generation problem is that it also contains implicit data dependencies as some of its branching conditions are defined on derived (local) variables rather than the input variables themselves. Set 2 also contains two validity-check functions, named check ISBN and check ISSN, which are a part of an open source program, **bibclean-2.08** [13]. These all are among popular benchmark functions in SBST and their short descriptions as well as justifications for their selection can be found in [6].

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 and maximum number of generations was kept at 500. For a fair comparison with [6], the crossover and mutation operators (i.e. One Point Crossover and Bit Mutation) and their probabilities (i.e. crossover: 0.9, mutation: 0.05) were kept the same as that of original system of Ariadne.

¹ 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 [6] at http://bds.ul.ie/?page_id=390/.

Table 1 A comparison of our improved Ariadne with the original system of Ariadne [6] on ten benchmark functions in Set 1

Branch ID	Original Ariadne [6]			Improved Ariadne		
	MC (%)	SR (%)	AE	MC (%)	SR (%)	AE
Tax calculator	67	0	20,108	100	100	27
Admission merit	25	0	150,759	100	100	827
Vitamin D levels	63	0	30,,157	100	100	34
Birth-time weights	67	0	20,108	100	100	20
HBA1c levels	75	0	10,058	100	100	11
GPA calculator	56	0	70,359	100	100	96
Volume discount	58	0	50,,259	100	100	57
S1	38	0	501,003	100	100	134
S2	70	0	601,223	100	100	2608
S3	56	0	801,606	100	100	11,202

MC, SR and AE are maximum coverage, success rate and average number of fitness evaluations, respectively

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 [6] to convert these integer values into valid input formats (of date and character string) as per the respective functions. For the sake of our experiments, we also used a similar mapping step (to convert the generated integers into respective input formats) to have a fair comparison with the original system.

Results and Discussion

We here present the results of **200 independent runs**, performed separately for all the benchmark programs. We also repeated the same set of experiments using the original grammar to have a better statistical comparison with [6]; 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 25–75%. 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 was also reported in [6] 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. To have a fair comparison with [6, 30], 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

Table 2 A comparison of our improved Ariadne with the original system of Ariadne [6] and with earlier GA-based techniques [30, 43]

Branch ID	Conventional GAs			Original Ariadne [6]			Improved Ariadne		
	MC (%)	SR (%)	AE	MC (%)	SR (%)	AE	MC (%)	SR (%)	AE
GADGET [43]									
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 and McMinn [30]									
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

MC, SR and AE are maximum coverage, success rate and average number of fitness evaluations, respectively

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. It can also be clearly seen that these improvements are even more impressive when compared to other GA-based techniques.

It is also worth noting that, in case of our improved Ariadne, the average number of fitness evaluations for some benchmark problems stayed considerably small (at times even smaller than the population size) as presented in Tables 1 and 2. In other words, these testing problems were proven to be trivial for our improved grammar and, at least for some problems, the grammar alone with randomly generated genotypes can result into a 100% coverage. Further, to better understand the roles of both the grammar and the evolutionary process in the working of Ariadne, we are also actively conducting a rigorous study aimed at *quantifying* the role of grammar in the process of test data generation but that is outside the scope of this paper.

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.

Scalability Analysis of the Improved GE-Based Test Data Generation

In this section, we present the results of a rigorous study performed to investigate how our improved GE-based test data generation approach (i.e. Ariadne) performs/scales for increasingly complex testing problems in a comparison with both the original system of Ariadne [6] and the GA-based test data generation approach. For the sake of our experiments, we designed and employed a large set of highly tunable benchmark programs of varying complexity. Our synthetic benchmark programs represent a wide variety of testing problems ranging from very basic to very complex. It is worth noting that our design is scalable, as the complexity of the benchmark programs can be varied by tuning some complexity-decisive features (detailed in Section 6.1). The inspiration for creating these synthetic programs is taken from [43] and [7] as discussed in Section 1.

Test Functions and Experimental Setup

We designed a set of **18** numeric benchmark programs³ with increasing complexity. These testing problems are formulated while keeping in mind the desired features of an ideal benchmark suite [40] as detailed in Section 1. The complexity of these synthetic benchmark programs was gradually increased by tuning the complexity-decisive features of *condition complexity* and *nesting complexity*. Condition complexity here refers to the number of individual

³ To facilitate future comparisons we have made available the source code at http://bds.ul.ie/?page_id=390/.

Fig. 4 The code of *comp(3,2)*. It contains a total of 24 coverage objectives, i.e. both TRUE and FALSE outcomes of 4 branching nodes and 8 individual condition predicates

```
void comp_32(int a, int b, int c, int d, int e, int f, int g,
int h, int i, int j) {
    if (c > 100 || a == d) {
        /* TRUE outcome of branching node # 1 */
        if (e == 110 || c == j) {
            /* TRUE outcome of branching node # 2 */
            if (h < j && i >= g) {
                /* TRUE outcome of branching node # 3 */
                if (e >= c || f == -80) {
                    /* TRUE outcome of branching node # 4 */
                } else {
                    /* FALSE outcome of branching node # 4 */
                }
            } else {
                /* FALSE outcome of branching node # 3 */
            }
        } else {
            /* FALSE outcome of branching node # 2 */
        }
    } else {
        /* FALSE outcome of branching node # 1 */
    }
}
```

conditions in each branching node of the program while the nesting complexity refers to the maximum nesting depth of branching nodes in that program.

For the sake of our detailed analysis, we manually formulated a total of 18 testing problems, i.e., one for each possible combination of both complexities, where *condition complexity* $\in \{1, 2, 3\}$ and *nesting complexity* $\in \{0, 1, 2, 3, 4, 5\}$. We kept the common parts intact in all these programs of increasing complexity to better study the scalability, as it is otherwise possible that a smaller program, containing relatively hard-to-satisfy branching conditions, can be a more difficult testing problem when compared to a larger program. To achieve this, we started by creating the simplest program of our study which was then gradually extended to build the programs of varying complexity. For the sake of consistency, the number of input variables was kept as 10 across all these programs. From now onwards, we will use the expression *comp(nesting_complexity, condition_complexity)* to refer to the benchmark programs of respective complexities.

An example synthetic program *comp(3,2)* is shown in Fig. 4. It can be seen that this program has a maximum nesting depth of 3 and there are 2 condition predicates in each of its branching nodes. It is worth noting that the branching conditions in this program contain a rich set of interdependencies (involving both variables and constant values) by virtue of relational operators ($<$, $\leq$, $>$, $\geq$, $\neq$, $=$) and logical operators ($\&\&$, $\|$), which makes it a very difficult (coverage) testing problem.

For the sake of this scalability analysis, we again conducted a small set of preliminary experiments to identify some reasonable settings for our GE/GA runs. We noted that the maximum of 500 generations with a population size of 300 were suitable for all our experiments. Other settings, including the type of crossover and mutation operators (i.e. One Point Crossover and Bit Mutation) as well as their probabilities (i.e. crossover: 0.9, mutation: 0.05), were kept the same as that of our earlier set of experiments presented in Section 5.

Detailed Analysis and Discussion

We performed **200 independent runs** for all the synthetic benchmark functions, separately with different settings, and presented their **mean** performance. We first performed these experiments using our improved system of Ariadne (i.e. GE-based test data generation approach) while the input variables were set to be **32-bit** signed integers. We then repeated the same set of experiments, separately, for both the original Ariadne [6] and the GA-based test data generation approach (i.e. without the added grammar-based mapping step of GE as explained in Section 3.1).

Unsurprisingly, the GA-based test data generation approach performed very poorly in our preliminary experiments as the search space for each of the input variables is huge (i.e. from $-2, 147, 483, 648$ to $2, 147, 483, 647$). These results were very similar to the ones presented in earlier studies of [43] and [7]. In our previous work [7], we made

the GA-based approach more comparable with the GE-based approach by constraining the search spaces of input values (and hence making the testing problems relatively easier) for the GA-based approach. We adopted a similar strategy here and additionally performed the complete set of our experiments with each of **16-bit** (i.e. from $-32,768$ to $32,767$) and **8-bit** (i.e. from -128 to 127) input variables for the GA-based approach.

In addition to that, in this study, we also implemented a similar seeding strategy for the GA-based approach to have a better view of how it performs and compares to the performance of the seeding strategy of Ariadne. The results of our preliminary experiments for the GA-based approach with 32-bit input variables still remained very poor even after the seeding. To gain better insights about the impact of seeding, we implemented the seeding strategy on the GA-based approach with the reduced search space (i.e. 16-bit input variables). Moreover, to also study the impact of different seeding probabilities, we repeated the same set of experiments with two different seeding probabilities i.e. 0.1 and 0.2. A 0.1 seeding probability means that, in the initial population, every individual input variable was given a 10% chance to take a value from the same pool of seeds that was injected in the attribute grammar of Ariadne (as described in Section 4).

The mean performances over 200 runs for all the benchmark programs with condition complexity of 1 and nesting complexities of 0 to 5 are shown in Fig. 5. The consumed search budgets (in terms of the number of fitness evaluations) are shown on the horizontal axis and the percentage of achieved (condition-decision) coverage is shown on the vertical axis. As the search budgets consumed by other approaches were often in orders of magnitude larger than that of our improved Ariadne, a **logarithmic scale** is used on the horizontal axis to better visualize the results. Similarly, the results for the programs with condition complexities of 2 and 3 are shown in Figs. 6 and 7 respectively.

It can be seen in Fig. 5 that $comp(0,1)$, being a very easy testing problem, quickly achieved a full coverage (i.e. a 100% coverage) under all the settings. In fact, both variants of Ariadne (i.e. the GE-based approaches) took longer when compared to the GA-based approach as the problem was too simple for a sophisticated solution; however, the search costs always stayed trivial under all the settings. The next two benchmark functions of $comp(1,1)$ and $comp(2,1)$ were again fully covered by both variants of Ariadne (on similar small search costs) while the GA-based approach was only able to achieve a 100% coverage with 8-bit input variables. It is worth noting that the GA-based approach achieved this high coverage on significantly larger search costs, even though the search spaces were substantially reduced. Lastly, the benchmark functions of $comp(3,1)$, $comp(4,1)$ and $comp(5,1)$

were successfully covered **only** by our improved system of Ariadne.

It is also worth noting that the conventional seeding strategy for the GA-based approach resulted into a notable improvement, in terms of coverage speed, only for the benchmark functions of $comp(2,1)$ and $comp(5,1)$. While the coverage percentage did not show any significant improvement for any of these benchmark programs with the condition complexity of 1. In contrast, the seeding strategy in our improved Ariadne resulted into a significant improvement in the coverage percentage for the benchmark functions of $comp(3,1)$, $comp(4,1)$ and $comp(5,1)$ as it exhibited a 100% coverage.

Figure 5 shows that where other approaches were not able to achieve a 100% coverage, despite consuming huge search budgets of hundreds of thousands of fitness evaluations, our improved Ariadne continued to exhibit a 100% coverage (with 32-bit input variables) on relatively tiny search costs. Our improved system performed so well as it was able to exploit the presence of all kinds of interdependencies (involving both variables and constant values) by the function of our improved grammar.

Another interesting trend that can also be observed in the coverage plots is that the original Ariadne always started by performing very similarly to that of our improved Ariadne but, in some cases, it stopped achieving any further coverage and flattened in the coverage plots. The only other comparable poor performance was shown by the GA-based approach with an equisized search space (i.e. with 32-bit input variables) for which the coverage flattened even more frequently and on further lower levels. This happened mainly because when any of these systems fail to evolve the required set of input values that can satisfy a particular branching condition, no further nested conditions under that branching node can be accessed. The reason behind the early blockage of the GA-based approach (with 32-bit input variables) is that it solely relies on the evolutionary process to find the required test data from the vast search space. The original Ariadne, on the other hand, is additionally capable of benefitting from the presence of variable interdependencies (but not the dependencies involving constant values) by virtue of its original grammar.

In other words, the original Ariadne quickly achieved coverage (similar to that of our improved Ariadne) for the conditions involving variable interdependencies; however, for those conditions containing dependencies involving constant values, it was neither able to exploit the presence of these interdependencies nor able to evolve the specific values required to satisfy the respective conditions. Hence, in the latter case, the original system of Ariadne performed similarly to (as bad as) the GA-based approach with an identical search space (i.e. with 32-bit input variables).

Fig. 5 Coverage plots comparing the performance of our improved Ariadne i.e. GE-based approach (with 32-bit signed integers as input values) with the original system of Ariadne [6] (with 32-bit signed integers as input values), the GA-based approach (with each of 8, 16

and 32 bit signed integers as input values) and the seeded GA-based approach (with the seeding probabilities of 0.1 and 0.2) on the benchmark programs with nesting complexities of 0–5 and condition complexity of 1

Fig. 6 Coverage plots comparing the performance of our improved Ariadne i.e. GE-based approach (with 32-bit signed integers as input values) with the original system of Ariadne [6] (with 32-bit signed integers as input values), the GA-based approach (with each of 8, 16

and 32 bit signed integers as input values) and the seeded GA-based approach (with the seeding probabilities of 0.1 and 0.2) on the benchmark programs with nesting complexities of 0–5 and condition complexity of 2

Fig. 7 Coverage plots comparing the performance of our improved Ariadne, i.e. GE-based approach (with 32-bit signed integers as input values) with the original system of Ariadne [6] (with 32-bit signed integers as input values), the GA-based approach (with each of 8, 16 and 32 bit signed integers as input values) and the seeded GA-based approach (with the seeding probabilities of 0.1 and 0.2) on the benchmark programs with nesting complexities of 0–5 and condition complexity of 3

The results of our experiments for the programs with condition complexities of 2 and 3 also show very similar trends, as shown in Figs. 6 and 7, respectively. The original Ariadne was able to achieve a full coverage only for the functions with nesting complexity of 0 while the GA-based approaches, both with the reduced search space (i.e. with 8-bit input variables) and the seeded variant (with 16-bit input variables) performed slightly better and managed to achieve a 100% coverage for the functions with nesting complexities of up to 2. On the other hand, our improved system of Ariadne successfully achieved a 100% coverage in **all** 200 runs performed independently for each of these benchmark functions.

The conventional seeding in the GA-based approach resulted in a notably improved performance (in terms of one or both of the coverage percentage and coverage speed) as compared to the plain GA-based approach. However, it still stays unmatchable to the performance of our improved Ariadne, as our improved system is not only able to take advantage of the variable interdependencies but our seeding strategy is also more advantageous as compared to the conventional seeding approach (as described in Section 4.1). As far as the impact of seeding probability is concerned, the more generous seeding (i.e. the seeding probability of 0.2) resulted in a relatively better performance, as it not only improved the availability of the seeded values but also potentially increased the chances of the satisfaction of equality predicates.

It is also worth noting that the **highest mean** search budget consumed by our improved Ariadne stayed just around fifty thousand fitness evaluations (i.e. for *comp(5,3)* which is the most difficult testing problems used in this study); the other approaches were not able to achieve a similar performance even after consuming millions of fitness evaluations.

Another interesting and common pattern that can also be observed is the discontinuous and immediate jumps in the percentage coverage, particularly at the beginning of the coverage plots. The initial jumps come about because when the program under test is executed (for the first time) with some random test data at the very beginning of the evolutionary process, it takes TRUE or FALSE outcomes for all the encountered conditions and decisions. As all of these outcomes are registered for the first time in the evolutionary search process, they all are recorded as the newly

covered targets. The later instantaneous jumps also appear for a similar reason. That is, when a new side of a branching node is executed, it exposes underneath (if any) branching nodes and conditions for the coverage, some of which are instantaneously covered.

To conclude, the results of this rigorous analysis demonstrate that our improved Ariadne stays scalable when compared to both the original Ariadne and the GA-based test data generation approach as it not only continued to scale up for the testing problems of increasing complexity (containing interdependencies involving both variables and constant values) but it also managed to do so while consuming budget multiple times smaller.

Conclusion and Future Work

We have proposed to seed the grammar with constants extracted from source code 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 demonstrate that the seeding strategy improves the effectiveness/generality of the system by impressive margins without compromising its efficiency as it further reduces the search budgets often up to an order of magnitude. Moreover, the results of our detailed scalability analysis show that the improved Ariadne is highly scalable as it retained a 100% coverage (across all the synthetic benchmark programs of increasing complexity) with significantly small search costs. In contrast, both the original Ariadne and the GA-based approach continued to fail by wide margins even after consuming massive search budgets.

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. For example, the integration of mathematical operators in the design of grammar can enable Ariadne to also evolve the required dependencies involving mathematical operations that can consequently satisfy the branching conditions of the form $(a == b + 10)$ or $(x == \text{square}(y))$, etc.

Moreover, we are currently using Ariadne for procedural C/C++ programs (i.e. to automatically generate input data) but it would be an interesting research topic to study that how it can be used for other programming paradigms, in

particular, for object-oriented languages (i.e. to automatically generate test programs). Further, we are also conducting a rigorous study aimed at measuring the effective improvement achieved by Ariadne and quantifying the role of grammar in the working of Ariadne.

To the best of our knowledge, 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.

Acknowledgements The authors would like to thank Aidan Murphy, Atif 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.

Funding Open Access funding provided by the IReL Consortium. This study was funded by the Science Foundation of Ireland (SFI) Grant number 16/IA/4605.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Informed consent Not applicable.

Rights retention This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 16/IA/4605. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Afzal W, Torkar R, Feldt R. A systematic review of search-based testing for non-functional system properties. *Inf Softw Technol.* 2009;51(6):957–76.
2. Aleti A, Buhnova B, Grunske L, Koziolok A, Meedeniya I. Software architecture optimization methods: a systematic literature review. *IEEE Trans Softw Eng.* 2013;39(5):658–83.
3. Ali S, Briand LC, Hemmati H, Panesar-Walawege RK. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans Softw Eng.* 2010;36(6):742–62.
4. 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.* IEEE Computer Society; 2011. p. 3–12.
5. Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, McMinn P, Bertolino A, et al. An orchestrated survey of methodologies for automated software test case generation. *J Syst Softw.* 2013;86(8):1978–2001.
6. Anjum MS, Ryan C. Ariadne: Evolving test data using grammatical evolution. In: *European Conference on Genetic Programming.* Springer; 2019.
7. Anjum MS, Ryan C. Scalability analysis of grammatical evolution based test data generation. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference.* p. 1213–21, 2020.
8. Anjum MS, Ryan C. Seeding grammars in grammatical evolution to improve search based software testing. In: *European Conference on Genetic Programming (Part of EvoStar).* Springer; 2020. p. 18–34.
9. Azad RMA, Ryan C. The best things don't always come in small packages: constant creation in grammatical evolution. In: *European Conference on Genetic Programming.* Springer; 2014. p. 186–97.
10. Baars A, Harman M, Hassoun Y, Lakhota K, McMinn P, Tonella P, Vos T. Symbolic search-based testing. In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society; 2011. p. 53–62.
11. Barros RC, Basgalupp MP, Cerri R, da Silva TS, de Carvalho ACPLF. A grammatical evolution approach for software effort estimation. In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation.* ACM; 2013. p. 1413–20.
12. Beizer B. *Software testing techniques*, van nostrand reinhold, 2nd edn. New York: Inc; 1990. ISBN 0-442-20672-0
13. bibclean.c. 1995. <http://www.cs.bham.ac.uk/~wbl/biblio/tools/bibclean.c>. Accessed 15 Sept 2019.
14. Bidgoli AM, Haghighi H. A new approach for search space reduction and seeding by analysis of the clauses. In: *International Symposium on Search Based Software Engineering.* Springer; 2018. p. 343–8.
15. Chen T, Li M, Yao . On the effects of seeding strategies: a case for search-based multi-objective service composition. In: *Proceedings of the Genetic and Evolutionary Computation Conference.* ACM; 2018. p. 1419–26.
16. 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. *Inf Softw Technol.* 2019;114:155–75.
17. Clarke LA. A system to generate test data and symbolically execute programs. *IEEE Trans Softw Eng.* 1976;3:215–22.
18. Cohen EI. A finite domain-testing strategy for computer program testing. PhD thesis, The Ohio State University, 1978.
19. 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.* Springer; 2019. p. 399–411.
20. DeMilli RA, Offutt AJ. Constraint-based automatic test data generation. *IEEE Trans Softw Eng.* 1991;17(9):900–10.
21. Dempsey I, O'Neill M, Brabazon A. Constant creation in grammatical evolution. *Int J Innov Comput Appl.* 2007;1(1):23–38.

22. Elshoff JL. An analysis of some commercial PL/I programs. *IEEE Trans Softw Eng.* 1976;2:113–20.
23. Ferguson R, Korel B. The chaining approach for software test data generation. *ACM Trans Softw Eng Methodol (TOSEM).* 1996;5(1):63–86.
24. 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. IEEE; 2012. p. 121–30.
25. Fraser G, Arcuri A. Whole test suite generation. *IEEE Trans Softw Eng.* 2013;39(2):276–91.
26. Fraser G, Arcuri A, McMinn P. A memetic algorithm for whole test suite generation. *J Syst Softw.* 2015;103:311–27.
27. Fraser G, Zeller A. Exploiting common object usage in test case generation. In: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. IEEE; 2011. p. 80–9.
28. Galeotti JP, Fraser G, Arcuri A. Improving search-based test suite generation with dynamic symbolic execution. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). IEEE; 2013. p. 360–9.
29. Harman M, Jia Y, Zhang Y. Achievements, open problems and challenges for search based software testing. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE; 2015. p. 1–12.
30. Harman M, McMinn P. A theoretical and empirical study of search-based testing: local, global, and hybrid search. *IEEE Trans Softw Eng.* 2010;36(2):226–47.
31. Holland JH. Genetic algorithms. *Sci Am.* 1992;267(1):66–73.
32. Inkumsah K, Xie T. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. ACM; 2007. p. 425–8.
33. Jin R, Jiang S, Zhang H. Generation of test data based on genetic algorithms and program dependence analysis. In: 2011 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems. IEEE; 2011. p. 116–21.
34. Jones BF, Sthamer H-H, Eyres DE. Automatic structural testing using genetic algorithms. *Softw Eng J.* 1996;11(5):299–306.
35. Kifetew FM, 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. IEEE; 2014. p. 163–72.
36. Korel B. Automated software test data generation. *IEEE Trans Softw Eng.* 1990;16(8):870–9.
37. Lima JAP, Vergilio SR, 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. ACM; 2017. p. 114–23.
38. Lopez-Herrejon RE, 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). IEEE; 2014. p. 387–96.
39. Mariani T, Guizzo G, Vergilio SR, Pozo ATR. Grammatical evolution for the multi-objective integration and test order problem. In: Proceedings of the Genetic and Evolutionary Computation Conference 2016. ACM; 2016. p. 1069–76.
40. McDermott J, White DR, Luke S, Manzoni L, Castelli M, Vanneschi L, Jaskowski W, Krawiec K, Harper R, De Jong K, et al. Genetic programming needs better benchmarks. In: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, p. 791–8, 2012.
41. McMinn P. Search-based software test data generation: a survey. *Softw Test Verif Reliab.* 2004;14(2):105–56.
42. 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. ACM; 2010. p. 1–4.
43. Michael CC, McGraw G, Schatz MA. Generating software test data by evolution. *IEEE Trans Softw Eng.* 2001;12:1085–110.
44. Miller J, Reformat M, Zhang H. Automatic test data generation using genetic algorithm and program dependence graphs. *Inf Softw Technol.* 2006;48(7):586–605.
45. Miller W, Spooner DL. Automatic generation of floating-point test data. *IEEE Trans Softw Eng.* 1976;3:223–6.
46. Myers GJ, Badgett T, Thomas TM, Sandler C. The art of software testing, vol. 2. New York: Wiley Online Library; 2004.
47. Offutt AJ, Jin Z, Pan J. The dynamic domain reduction procedure for test data generation. *Softw Pract Exp.* 1999;29(2):167–93.
48. O'Neill M, Ryan C. Grammatical evolution. *IEEE Trans Evol Comput.* 2001;5(4):349–58.
49. Panichella A, Kifetew FM, Tonella P. Reformulating branch coverage as a many-objective optimization problem. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE; 2015. p. 1–10.
50. Pargas RP, Harrold MJ, Peck RR. Test-data generation using genetic algorithms. *Softw Test Verif Reliab.* 1999;9(4):263–82.
51. Patten JV, Ryan C. Procedural content generation for games using grammatical evolution and attribute grammars. 2014.
52. Rojas JM, Fraser G, Arcuri A. Seeding strategies in search-based unit test generation. *Softw Test Verif Reliab.* 2016;26(5):366–401.
53. Ryan C, Collins JJ, Neill MO. Grammatical evolution: Evolving programs for an arbitrary language. In: European Conference on Genetic Programming. Springer; 1998. p. 83–96.
54. Sauder RL. A general test data generator for cobol. In: Proceedings of the May 1–3, 1962, Spring Joint Computer Conference, p. 317–23, 1962.
55. 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). IEEE; 2007. p. 477–86.
56. Tlili M, Wappler S, Sthamer H. Improving evolutionary real-time testing. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation. ACM; 2006. p. 1917–24.
57. Tracey N, Clark J, Mander K, McDermid J. An automated framework for structural test-data generation. In: ASE. IEEE; 1998. p. 285.
58. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. *Inf Softw Technol.* 2001;43(14):841–54.
59. Xanthakis S, Ellis C, Skourlas C, Le Gall A, Katsikas S, Karapoulos K. Application of genetic algorithms to software testing. In: Proceedings of the 5th International Conference on Software Engineering and Applications, p. 625–36, 1992.
60. Zhu Z, Jiao L, Xu X. Combining search-based testing and dynamic symbolic execution by evolvability metric. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE; 2018. p. 59–68.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.