The case for Grammatical Evolution in test generation

Aidan Murphy
aidan.murphy@ucd.ie
School of Computer Science,
University College Dublin
Dublin, Ireland

Thomas Laurent
thomas.laurent@ucd.ie
SFI Lero & School of Computer
Science, University College Dublin
Dublin, Ireland

Anthony Ventresque
anthony.ventresque@ucd.ie
SFI Lero & School of Computer
Science, University College Dublin
Dublin, Ireland

ABSTRACT

Generating tests for software is an important, but difficult, task. Search-based test generation is promising, as it reduces the time required from human experts, but suffers from many problems and limitations. Namely, the inability to fully incorporate a tester's domain knowledge into the search, its difficulty in creating very complex objects, and the problems associated with variable length tests. This paper illustrates how Grammatical Evolution could address and provide a possible solution to each of these concerns.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software testing and debugging**.

KEYWORDS

Automatic Test Generation, Search Based Software Testing, Grammatical Evolution

ACM Reference Format:
Aidan Murphy, Thomas Laurent, and Anthony Ventresque. 2022. The case for Grammatical Evolution in test generation. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3520304.3534042

1 EVOLUTION OF TEST CASES

Software requires testing in order to identify bugs present in the code and gain confidence in its correct behaviour. The creation of these tests is a manual task which is both time consuming and requires expertise of the System Under Test (SUT) and in test case construction. It is also possible that the tests are poorly written and do not fully test all the functionalities of the SUT, meaning that buggy software gets released.

Search Based Software Testing (SBST) techniques have emerged to lower the effort required by human testers. This area investigates the use of heuristic-based optimisation techniques to automatically generate tests, leading to a faster and more systematic testing process. Evolutionary Testing is one such technique that uses evolutionary computation as its heuristic.

Search based test generation has shown promising results in different contexts. A prominent example is Evosuite [4], a powerful, open source tool which generates and optimises tests for Java programs. It follows a hybrid approach which uses evolutionary computation, specifically a Genetic Algorithm (GA), combined with knowledge extracted from the classes under test. Its performance has lead to its wide adoption, both in academia and in industry [1], as well as to awards in search based software testing competitions [8]. Evosuite has been extended with many search algorithms and strategies, including multi objective search, and allows diverse coverage criteria to be used to guide the search. Consequently, it remains the state of the art for Java program unit test generation.

2 PROBLEMS OF SEARCHED BASED TESTING

Despite these successes and wide adoption, SBST still shows limitations. Among them are:

Incorporating Human Expertise. Current SBST techniques, particularly evolutionary methods, generally do not allow any human expertise to be easily incorporated into the test generation process. It is possible a tester may want to generate many tests containing a specific value or fragment of code. This would greatly speed up the search, as it will only focus on areas the tester deems fruitful, and may aid in the interpretability of the final tests found.

Creating Complex Objects. SBST can struggle to evolve complex objects, the structure of which may or may not be known to the tester. Shamshiri et al. [7] encountered this problem when generating regression tests for the *Closure* project in the Defects4J benchmark [5]. The tests they generated could not detect many of the bugs in the dataset for this project. Detecting these bugs requires tests that create a control-flow graph, an object which can be easily created by a human tester. However, the task has proven difficult for search based techniques, Evosuite included, as the data format needed to generate a valid object (and even more so one that is interesting for testing) is very precise and introduces many dependencies [1].

Domain Flexibility. SBST tools are often domain specific. To move from one domain to another may require augmenting of the existing tools' code. Custom functionality or operators must be written in order to produce the desired tests.

3 GRAMMATICAL EVOLUTION

Grammatical Evolution (GE) [6] is an evolutionary computation technique. It uses a grammar, often a context free grammar, to create syntactically correct objects in any arbitrary language. GE's flexibility has seen it achieve great success in a variety of domains.

[1]See examples here

```
<cfg>           ::= <try> | <try_catch>
<try>           ::= <try_statement> | <try_recursion>
<try_statement> ::= try{<cond>
                        } finally {}
<try_recursion> ::= try{<cond>
                        <cfg>
                        } finally {}
<try_catch>     ::=  .....
```

Figure 1: Example grammar used to evolve a Control Flow Graph, i.e. generate valid test cases for *Closure-14* [5]

GE is often thought of as a variant of Genetic Programming (GP). The key difference, however, lies in how the search takes place. GE uses bit-strings, genotypes, and maps them onto computer programs via the grammar. The search operators are performed on the strings, as in the regular GA, and not the actual structures which are examined and given a fitness. If a GA is used as the search technique, as in EvoSuite, crossover and mutation would occur on bit-strings as it does in the regular GA and these newly evolved bit stings would be mapped to executable programs. This separation between the search space, at the genome level, and the program space, on the phenotype level, is seen as one of GE's many advantages over regular GP as it greatly simplifies the search operation and guarantees closure, among other characteristics. While guaranteeing semantically correct objects will be created, this standard GE approach does lead to poor locality.

As GE traditionally uses a GA as its search engine, the same as many SBST tools and techniques such as Evosuite, it means GE is compatible with their architecture. The remainder of this section explains how integrating GE in this architecture could help alleviate the limitations of SBST highlighted in Section 2.

3.1 Incorporating Human Expertise into the Search

By specifying a grammar, a user defines the search space they want their tests to be created from. Grammars can be general, create a wide variety of tests, or specific to a particular problem. A In addition to this, a grammar allows for edge cases to be seeded into the search without editing the search algorithms code, a difficult task given the complexity of test generation algorithms. If utilising GE, all that is required is a simple grammar modification. A GE implementation to generate test cases for procedure programs has recently been introduced, achieving vastly improved results in both fitness and computational cost to earlier GA based techniques [2]. It allowed the user to provide seeded constants and exploited variable interdependencies to efficiently generate test data. Extending GE use for Object Oriented programs with more complicated interdependencies is the next step for such a system.

3.2 Creating Complex Objects in Tests

Creating complex objects is a difficult task for SBST, and evolutionary computation in general. It is often necessary to create these objects as the SUT may exhibit subtle, complex or unique behaviour which cannot be fully assessed by simple tests.

GE can alleviate this concern by allowing the tester to specify the structure of the desired solution in the grammar, or to list key modules a solution must contain. This is done by altering the grammar meaning no altering of the code base is necessary. Rather than spending computation time searching for the structure of solutions, the search can instead focus on optimising the content contained within that structure. The search space can be dramatically reduced and needless exploration of fruitless areas avoided. The tester has more control over the types of tests the search creates.

Figure 1 shows an example of a grammar which can be used to create a Control Flow Graph. Constructing a Control Flow Graph with two consecutive connected *finally* blocks on its edge is required to cover the *Closure-14* fault. While this grammar is specialised to this particular fault, extending it to other faults or generalising it to create a wider variety of Control Flow Graphs is straightforward. There is no necessity to change the underlying search code.

3.3 Flexibility of Grammars

The separation between GE's search space and program space offers great flexibility. To move from one domain to another requires a simple modification to, or replacement, of the grammar. A GE based SBST tool would not need to be extended with custom functions or operators to deal with modified tests. For context aware tests, which depend on previous actions undertaken, an attribute grammar could be specified. Indeed, a recent work for evolving test suites for Scratch programs used GE as the engine to create the tests for this reason [3].

4 CONCLUSION

This work highlights current limitations of automated search based test generation and puts Grammatical Evolution forward as a tool to address them. Namely, it contends that GE can help incorporate testers' expertise into the search; can enable search based techniques to create tests that instantiate complex objects requiring precise, dependency-laden call sequences; and can be flexible enough to be easily applied to different contexts.

ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grants 20/FFP-P/8818 and 13/RC/2094_P2.

REFERENCES

[1] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *ICSE-SEIP*. IEEE.
[2] Muhammad Sheraz Anjum and Conor Ryan. 2021. Seeding Grammars in Grammatical Evolution to Improve Search-Based Software Testing. *SN Computer Science* 2, 4 (2021).
[3] Adina Deiner, Christoph Frädrich, Gordon Fraser, Sophia Geserer, and Niklas Zantner. 2020. Search-Based Testing for Scratch Programs. In *SSBSE*. Springer.
[4] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *ESEC/FSE*.
[5] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*.
[6] Conor Ryan, John James Collins, and Michael O Neill. 1998. Grammatical Evolution: Evolving programs for an arbitrary language. In *EuroGP*. Springer.
[7] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *ASE*. IEEE, 201–211.
[8] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. 2021. EVOSUITE at the SBST 2021 Tool Competition. In *SBST*. IEEE.