Automatic Synthesis of Code Using Genetic Programming

Doron Peled(✉)

Department of Computer Science, Bar Ilan University, 52900 Ramat Gan, Israel
doron.peled@gmail.com

Abstract. Correct-by-design automatic system construction can relieve both programmers and quality engineers from part of their tasks. Classical program synthesis involves a series of transformations, starting with the given formal specification. However, this approach is often prohibitively intractable, and in some cases undecidable. Model-checking-based genetic programming provides a method for software synthesis; it uses randomization, together with model checking, to heuristically search for code that satisfies the given specification. We present model checking based genetic programming as an alternative to classical transformational synthesis and study its weakness and strengths.

1 Introduction

Automatic synthesis of correct-by-design code is a very appealing approach. It can assist programmers in producing the hard-to-code parts of systems. Further, the code is already correct with respect to the specification. We are still quite far from achieving this situation. For one, it is not always clear that writing correct and complete specification is easier than programming. Moreover, classical approaches for software synthesis is proved to be doubly exponential for interactive systems [16], and undecidable for concurrent systems [17].

Genetic programming (GP) [12] is a search based software engineering approach [4], i.e., an evolutionary based heuristic search methodology for finding computer programs that perform user defined tasks. In GP, programs are generated and evolved by applying biologically inspired ideas, such as reproduction, mutations, and natural selection. GP uses a fitness function that measures the quality of the candidate solutions generated during the search. GP can also be used to improve programs, e.g., to speeding up the performance of systems [13] or correct erroneous programs [10].

Model-checking based genetic programming (MCGP) [6–10], is basically a search technique that uses model checking as its fitness function (heuristic measure). In [1], model checking was used within a generate-and-test feedback loop in order to construct correct-by-design solutions for the mutual exclusion problem. It exhaustively passes throughout the possible candidates (given some limit on

D. Peled—The research was supported in part by ISF grant 126/12 “Efficient Synthesis Method of Control for Concurrent Systems”.

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 182–187, 2016.
DOI: 10.1007/978-3-319-47166-2_12

the resources), revealing the correct solutions. MCGP offers a heuristic, rather than exhaustive, search through candidates. It utilizes randomness in initially generating the candidates and in progressing between them.

One of the main obstacles in using MCGP is that the fitness function of GP requires a good separation between different candidates. The fitness function provides a measure for how far the candidate program is from completely satisfying the complete specification, and needs to separate between stronger and weaker candidates. However, it is hard to attain this goal based on model checking, as there are often a very limited number of specification properties. This makes the landscape of the fitness function discrete rather than smooth. We discuss how this problem can be alleviated.

The traditional use of a large test suite can provide a smother fitness function. The test suite can use standard manual testing techniques to generate a test suite that captures a large set of expected problems. However, it does not guarantee the correctness of the constructed code, and the set of test cases may prove itself to be biased.

In contrast, in transformational synthesis of reactive systems [16], even a single specification property is sufficient. This consists of translating the specification(s) into an automaton determinizing it, finding a game strategy such that the system will be able to make good choices in response to the choices of the environment.

2 Preliminaries

Model Checking of Temporal Properties

Model checking [2] is an automatic method for verifying the correctness of a finite state software or hardware system against its formal specification. It is often used to verify models of concurrent algorithms, protocols and reactive systems. Such models usually have many possible executions, due to concurrency and nondeterministic choices made by scheduling or interacting with the environment.

A finite state system can be modeled by an automaton. Each *state* of the automaton corresponds to an evaluation of the variables, programs counters, communication buffers of the system. An *execution* is then a maximal sequence of *states*, starting from some *initial* state; *transitions* between subsequent states represent the effect of atomic actions of the system. Propositions are used to represent properties of states, e.g., p may hold in states where $x > 0$ and q in states where the program is at the beginning of its first loop. The specification can be written as a set of properties in a logic such as *Linear Temporal Logic* (LTL), which combines propositional variables and logic operators with temporal operators. For example, $\Box p$ stands for 'p holds in every state' (in the execution) and $\Diamond q$ stands for 'q holds in some future state'.

A standard model checking procedure checks whether a system M satisfies a specification φ. The specification φ is often converted into automata A_φ over infinite words [3]. The simplest kind of such automata is called Büchi automata [19];

an infinite word (representing in our context an execution) is *accepted* if in a run of the automaton over that word, at least one of a set of states that are distinguished as *accepting* occurs infinitely many times. For some LTL specifications such as $\Diamond\Box p$ ('p holds for some state forever'), the translation necessarily results in a nondeterministic Büchi automaton [19]. In transformational synthesis, this nondeterminism needs to be removed by a further transformation into another kind of automata [18].

The specification automaton represents all of the executions (abstracted as sequences of propositional values) *allowed* by the specification properties. The model checking algorithm then checks whether the language of the model automaton is *contained* in the language of the specification automaton. If this holds, then the checked property is satisfied by the model. Otherwise, there are executions of the model that violate the specification.

Genetic Programming

During the 1970s, Holland established the field known as *Genetic Algorithms* [5]. According to this methodology, individual candidate solutions are represented as fixed length strings of bits, and are manipulated mainly by the *crossover* and *mutation* genetic operations. The *crossover* operation takes parts of strings from two parent solutions; it combines them into a new solution, which potentially inherit useful attributes from his parents, and become fitter. The *mutation* operation randomly alters the content of small number of bits in the string, thus allowing the insertion of new building blocks (or genes) into the population.

Genetic programming [12] is a direct successor of genetic algorithms. In GP, each individual "organism" represents a computer program. Thus, instead of fixed length strings, programs are represented by variable length structures, such as trees, linear lists or graphs. Each individual solution is built from a set of functions and terminals, and corresponds to a program or an expression in a programming language that can be executed. In tree-based genetic programming, crossover is performed by selecting type compatible subtrees on each of the parents, and then swapping between them. Mutation can be carried out by choosing a subtree and replacing it by another randomly generated subtree of the same type. The fitness is calculated by directly running the generated programs on a large set of test cases and evaluating the results.

3 Software Synthesis Using Genetic Programming Based on Model Checking

In [7–10], we present a framework combining genetic programming and model checking that allows to automatically synthesize software code for given problems. The user provides the formal specification of the problem, as well as additional constraints on the structure of the desired solutions.

The synthesis process generally goes through the following steps:

1. The user provides a *configuration*, which is a set of structural constraints on the programs that are allowed to be generated (thus, defining the space of candidate programs).

2. The user provides a formal specification for the problem. This can be a set of LTL properties, as well as additional requirements on the program behavior.
3. The GP engine randomly generates an initial population of programs based on the configuration.
4. The model checking based verifier analyzes the behavior of the generated programs against the specification properties, and provides fitness measures based on the amount of satisfaction.
5. Based on the verification results, the GP engine then creates new programs by applying genetic operations of crossover and mutation. The next iteration contains the newly generated candidates, and also some of the old candidates that were chosen using a random selection: the probability to remain in the next iteration is based on the relative fitness value. The number of candidate solutions remains invariant between the different iterations.
   Steps 4 and 5 are repeated until either a perfect program is found (fully satisfying the specification), or until the maximal number of iterations is reached.
6. The results are sent back to the user. This includes a program that satisfies all the specification properties, or a failure report.

4 Fitness Functions Based on Model Checking

The shortcomings of transformational synthesis and of testing based GP motivates the MCGP approach. However, in order to make MCGP practical, we need a way of smoothening the fitness function. The result of model checking is binary: yes or no (providing also a counterexample in the latter case). Naively counting the number of properties that are satisfied does not provide a good fitness function, and it will often fail to stir the genetic process towards convergence. It is also not clear that e.g., satisfying the first two properties is better than satisfying the third one. In many cases in fact, the number of properties given is rather small. We present several possibilities to provide more meaningful fitness values for MCGP.

Quantitative levels. The fitness function is made proportional also to the fraction of executions that are correct. For properties based on finite executions or their approximations, one can generate many levels by applying statistical model checking [14]. This approach also helps alleviating the intractability of model checking.

Qualitative levels. One can define meaningful levels of satisfaction of properties that can be verified using variants of model checking. One such level represent the fact that *some* (but not *all*) executions satisfy some property. Another level confines the bad executions to those that are highly improbable, e.g., where the same nondeterministic choices made all the time.

Co-evolution. We can develop test cases along with the genetic process. The fitness of a test case can grow up with the number of candidates that it manages to fail.

5 Experience and Further Work

We discussed MCGP approach. This has been implemented as a prototype tool [11]. In particular, it was used successfully in different cases:

- Finding existing and new solutions for mutual exclusion.
- Finding a solution to the leader election problem in a ring.
- Correcting the α-core algorithm [15].

Implementing MCGP is a comprehensive effort: it consists of the following components:

- Translation between code and syntax tree representation.
- Implementation of model checking and its derivatives (probabilistic model checking, statistical model checking).
- The search engine, including the fitness calculation and the genetic operations of mutation and crossover.

Because the synthesis problem is in general undecidable (in particular, for concurrent systems with LTL specifications), MCGP cannot always guarantee to terminate successfully. Often, after a number of iterations, the user would stop the running of the tool and would restart it either with a new random seed, or by changing parameters. The latter can involve giving different weights for the different properties when calculating the fitness functions. Indeed, while the success cases reported here would for in a few minutes using the tool, tuning the parameters until this has started to happen often took days or weeks.

In [10] a broad approach to co-evolution is presented. There, the goal is to use MCGP to correct a large, parametric, communication protocol [15]. While model checking is undecidable for parametric (e.g., in the communication architecture) programs, it can be seen as a generalization of testing: each particular communication architecture forms an instance of model checking; hence model checking is exhaustive against the *particular chosen architectures*. The different architectures are also generated using the genetic programming techniques (e.g., using mutation). The more useful architectures (based on causing candidate programs to fail) are kept from generation to generation for checking against further candidates.

References

1. Bar-David, Y., Taubenfeld, G.: Automatic discovery of mutual exclusion algorithms. In PODC, p. 305 (2003)
2. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000)
3. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembiński, P., Średniawa, M. (eds.) IFIP WG6.1. IFIP, pp. 3–18. Springer, Heidelberg (1995)
4. Harman, M., Jones, B.F.: Software engineering using metaheuristic innovative algorithms: workshop report. Inf. Softw. Technol. **43**(14), 905–907 (2001)

5. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, Cambridge (1992)
6. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 114–124. Springer, Heidelberg (2007)
7. Katz, G., Peled, D.A.: Genetic Programming and model checking: synthesizing new mutual exclusion algorithms. In: Cha, S.S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 33–47. Springer, Heidelberg (2008)
8. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 141–156. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_11
9. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 117–132. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19237-1_13
10. Katz, G., Peled, D.: Code mutation in verification and automatic code correction. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 435–450. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12002-2_36
11. Katz, G., Peled, D.: MCGP: a software synthesis tool based on model checking and genetic programming. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 359–364. Springer, Heidelberg (2010)
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
13. Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. IEEE Trans. Evol. Comput. **19**(1), 118–135 (2015)
14. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N., Barringer, H. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010)
15. Perez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurr. Pract. Exp. **16**(12), 1173–1206 (2004)
16. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
17. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: FOCS, pp. 746–757 (1990)
18. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988, pp. 319–327 (1988)
19. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 133–192 (1990)