MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming*

Gal Katz and Doron Peled

Department of Computer Science, Bar Ilan University
Ramat Gan 52900, Israel

Abstract. We present MCGP - a tool for generating and correcting code, based on our synthesis approach combining deep **M**odel **C**hecking and **G**enetic **P**rogramming. Given an LTL specification, genetic programming is used for generating new candidate solutions, while deep model checking is used for calculating to what extent (i.e., not only whether) a candidate solution program satisfies a property. The main challenge is to construct from the result of the deep model checking a fitness function that has a good correlation with the distance of the candidate program from a correct solution. The tool allows the user to control various parameters, such as the syntactic building blocks, the structure of the programs, and the fitness function, and to follow their effect on the convergence of the synthesis process.

1 Introduction

With the growing success of model checking for finding bugs in hardware and software, a natural challenge is to generate automatically correct-by-design programs. This is in particular useful in intricate protocols, which even skillful programmers may find difficult to implement. Automatically constructing a reactive system from LTL specification was shown to be an intractable problem (in 2EXPTIME) in [12]. For concurrent systems, the situation is even worse, as synthesizing a concurrent system with two processes from LTL specification is already undecidable [13]. A related (and similarly difficult) problem is correcting a piece of code that fails to satisfy its specification.

Genetic programming [1] is a method for automatically constructing programs. This is a directed search on the space of syntactically limited programs. Mutating and combining candidate solutions is used for generating new candidates. The search progresses towards a correct solution using a fitness function that is calculated for the newly generated candidates. Traditionally, the fitness is based on testing. Recently, we studied the use of model checking as a basis for providing fitness values to candidates [7]. Experimentally, we learned that the success of the model checking based genetic programming to synthesize correct programs is highly dependent on using fitness functions that are designed for the specific programming goal. One of the main features of our tool is the ability

* This research was supported in part by ISF Science Foundation grant 1262/09.

A. Bouajjani and W.-N. Chin (Eds.): ATVA 2010, LNCS 6252, pp. 359–364, 2010.

to provide the user with flexible ways for constructing the fitness function based on model checking and various parameters of the desired code.

A central building block of our tool is using *deep model checking*, which does not only checks whether a candidate program satisfies a property or not, but *to what extent* it does so. This provides a finer analysis, and consequently helps the convergence of the genetic search. Several levels of satisfactions that we use are described later. Furthermore, one may want to assign different priorities to properties (e.g., if a basic safety property does not hold, there is no gain in considering other satisfied liveness properties), and include other code parameters (e.g., decreasing fitness according to the length of the code). Our experience with the method shows that experimenting with fine-tuning the fitness function is a subtle task; its goal is to anticipate a good correlation between the fitness of a candidate program and its distance from a correct program, thus helping to steer the search in the right direction.

In previous work, we used our tool for the synthesis of various protocols. However, this was done in an ad hoc manner, requiring to dynamically change the tool for every new synthesis problem. The current presented tool is a generalization of our previous experience. It allows the user to automatically synthesize new protocols by only providing and tuning a set of definitions, without a need to change the tool itself.

2 Tool Architecture and Configuration

The tool is composed of a server side (written in C++) responsible for the synthesis process, and a user interface used for specifying the requirements, and interactively observing the synthesis results. The tool architecture and information flow are depicted in Fig. 1. The server side combines a deep model checker and a genetic programming engine, both enhanced in order to effectively integrate with each other. This part can be used directly from the command line, or by a more friendly Windows based user interface. In order to synthesize programs, the user first has to fill in several definitions described throughout this section, and then initiate the synthesis process described in section 3.

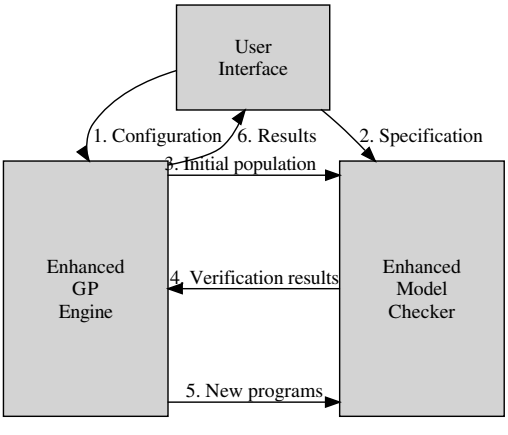

Fig. 1. Tool architecture

Genetic Programming Definitions. The first step in using the tool is to define the various process types that will comprise the synthesized programs.

A program consists of one or more process types containing the execution code itself, and an *init* process responsible for creating instances of the other process types, and initializing global variables. For each process type, the user then has to choose the appropriate building blocks from which its code can be generated. The tool comes with a library of building blocks used by common protocols and algorithms, such as variables, control structures, inter-process communication primitives, and more. Furthermore, the user can define new building blocks by combining existing building blocks, and optionally by writing additional C++ code. These building blocks can vary from simple expressions, macros and functions to complex statements that will be later translated into atomic transitions.

Each process type can contain both static parts set by the user, and dynamic parts, which the tool is required to generate and evolve. Depending on these settings, the tool can be used for several purposes:

- Setting all parts as *static* will cause the tool to simply run the deep model checking algorithm on the user-defined program, and provide its detailed results, including the fitness score assigned to each LTL property.
- Setting the *init* process as *static*, and all or some of the other processes as *dynamic*, will order the tool to synthesize code according to the specified architecture. This can be used for synthesizing programs from scratch, synthesizing only some missing parts of a given partial program, or trying to correct or improve a complete given program.
- Setting the *init* process as *dynamic*, and all other processes as *static*, is used when trying to falsify a given parametric program. In this case the tool will reverse its goal and automatically search for configurations that violate the specification (see [9]).
- Setting both the *init* and the program processes as *dynamic*, is used for synthesizing parametric programs. It causes the tool to alternatively evolve various programs, and configurations under which these programs have to be satisfied [9].

Specification Definitions. At this stage the user should provide a full system specification from which the tool can later derive the fitness function assigned to each generated program. The specification is mainly based on a list of LTL properties. The atomic propositions used by these properties are defined as C expressions representing Boolean conditions. The properties and the atomic propositions are then compiled into executable code that is later used by the model checking process (this is inspired by the way the Spin model checker [4] handles LTL properties). The user can set up a hierarchy between the checked properties, instructing the tool to start with some basic properties, and only then gradually check more advanced properties [7]. In addition to the LTL properties, other quantitative goals can be added, such as minimizing the size of generated programs. Further advanced model checking options (such as the DFS search depth, and the use of partial order reduction) can be tuned as well.

3 The Synthesis Process

After completing the definitions step, the synthesis process can be initiated. First, using the skeletons and building blocks provided by the user for each process type, an initial set P of programs is randomly generated. The code of each process type is stored as a syntactic tree whose nodes are instances of the various building blocks, representing variables, statements, functions, constants, etc.

Next, an iterative process of improvements begins. At each iteration, a small subset of μ programs is selected randomly from P. Then, various genetic operations are performed on the selected programs, leading to the generation of new λ modified programs. The main operation in use is *mutation*, which basically adds, removes or changes the program code, by manipulating its syntactic tree [7]. Deep model checking is then performed on each new program, in order to measure the degree on which it satisfies the specification.

A probabilistic selection mechanism is then used in order to select new μ programs that will replace the originally selected μ programs, where a program has a chance to survive proportionally to its fitness score. The iterative process proceeds until either a perfect solution is found, or after the maximal allowed iterations number is reached.

During the process, the user can watch the gradual generation of solutions, by following the best generated programs, and by navigating through the chain of improvements. For each selected program, the generated code is displayed, as well as the deep model checking results, including the fitness score assigned to each specification property. Often, watching these results leads to some insights regarding tuning required to the specification, the building blocks, or other parameters, and the fitness function based on the above. Fig. 2 shows an example of the tool's screen during the synthesis of a mutual exclusion protocol described in [7].

Deep Model Checking. The main challenge in making the model checking based genetic approach work in practice is to obtain a fitness function that

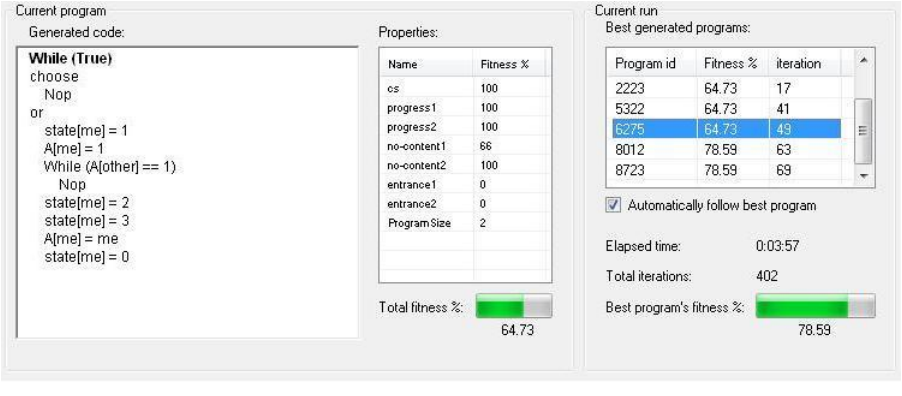

Fig. 2. The user interface during synthesis of a mutual exclusion algorithm

correlates well with the potential of the candidate programs. In order to achieve this, it is not enough just to count the number of LTL properties that hold, as done in [5]; this gives a coarse indication that behaves poorly under experimentations. With this tool, we introduce different levels or *modes* of satisfaction for each LTL property; a property does not necessarily have to be satisfied by all the executions in order to contribute to the fitness value of the checked candidate.

We consider the following modes of satisfaction:

- None of the program's executions satisfy the property (level 0),
- some of the program's executions satisfy the property (level 1),
- each prefix of a program execution can be extended into an execution satisfying the property (level 2), and
- all of the program's executions satisfy the property (level 3).

Model checking algorithms usually only check membership for levels 3. By using also the formula itself, and not only its negation, we can also check membership for level 0, leading to a third possible level, where neither level 0 nor level 3 holds. Further distinction between levels 1 and 2 requires a different kind of analysis, and in fact, the complexity of checking it is higher than simple model checking: a reduction in [10] can be used to show that it is in EXPTIME-Complete. One can use a logic that permits specifying, separately, the LTL properties, and the modes of satisfaction [11] or develop separate optimized algorithms for each level as we did [7].

A main concern is the model checking efficiency: while we often try to synthesize just basic concurrent programs, with small as possible requirements, they can still have a large number of states (e.g., tens of thousands). Moreover, model checking is performed here a large number of times: we may have thousands of candidates before the synthesis process terminates or fails. Accordingly, in the latest version of the tool, we decided to slightly change the definition of the intermediate fitness levels, by adopting a technique similar to probabilistic qualitative LTL model checking. We treat the nondeterministic choices as having some probabilities, and base the distinction between the new levels on the probability p that the program satisfies the *negation* of the checked property ($p > 0$ implies new level 1, and $p = 0$ implies new level 2). While this new algorithm occasionally shifts the boundary between the two levels (compared to the original ones), it has the advantage of having PSPACE complexity [2].

In order to compute the fitness function, the following is applied to each LTL property. The property, and its negation are first translated into standard Büchi Automata, by running the LTL2BA code [3]. Then these automata are further extended (in order to fit into our specialized algorithm), and combined with the program's automaton, yielding the fitness levels mentioned above. Additional factors such as implication properties, and deadlocks can affect the fitness scoring as well [7]. The scoring for all of the LTL properties are finally summed up, possibly in conjunction with other quantitative measures (such as the program size) into a final fitness score assigned to the program.

Tool Evaluation. Using our tool, we successfully synthesized correct solutions to a series of problems, including known [7], and novel [6] two-process mutual

exclusion algorithms, and parametrized leader election protocols [8]. Recently we used the tool's ability of synthesizing architectures, for the automatic discovery and correction of a subtle bug in the complicated α-core protocol [9]. The synthesis duration usually varies from seconds to hours, depending on many aspects of the problems, such as the number of processes, the solutions size, and the model checking time. The main synthesis algorithm has a large amount of parallelism, and thus execution time can be greatly improved when running on multi-core servers. A weakness of the genetic programming based approach is its probabilistic nature, which does not guarantee convergence into perfect solutions. The duration and success of the synthesis depend on the choices made by the user for the various building blocks and parameters. Additional information about the tool, including its freely available latest version, and some running examples, can be found at: http://sites.google.com/site/galkatzzz/mcgp-tool

References

1. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications, 3rd edn. Morgan Kaufmann/Dpunkt Verlag (2001)
2. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM 42(4), 857–907 (1995)
3. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
4. Holzmann, G.J.: The SPIN Model Checker. Pearson Education, London (2003)
5. 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)
6. Katz, G., Peled, D.: 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)
7. 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)
8. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: HVC (2009)
9. 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)
10. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design 19(3), 291–314 (2001)
11. Niebert, P., Peled, D., Pnueli, A.: Discriminative model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 504–516. Springer, Heidelberg (2008)
12. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
13. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: FOCS, pp. 746–757 (1990)