Synthesizing, Correcting and Improving Code, Using Model Checking-Based Genetic Programming

Gal Katz and Doron Peled

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

Abstract. The use of genetic programming, in combination of model checking and testing, provides a powerful way to synthesize programs. Whereas classical algorithmic synthesis provides alarming high complexity and undecidability results, the genetic approach provides a surprisingly successful heuristics. We describe several versions of a method for synthesizing sequential and concurrent systems. To cope with the constraints of model checking and of theorem proving, we combine such exhaustive verification methods with testing. We show several examples where we used our approach to synthesize, improve and correct code.

1 Introduction

Software development is a relatively simple activity: there is no need to solve complicated equations, to involve chemical materials or to use mechanical tools; a programmer can write tens of lines of code per hour, several hours a day. However, the number of possible combinations of machine states in even a very simple program can be enormous, producing frequently unexpected interactions between tasks and features. Quite early in the history of software development, it was identified that the rate in which errors are introduced into the development code is rather high. While some simple errors can be observed and corrected by the programmer, many design and programming errors survive shallow debugging attempts and find themselves in the deployed product, sometimes causing hazardous behavior of the system, injuries, time loss, confusion, bad service, or massive loss of money.

A collection of *formal methods* [20] were developed to assist the software developers, including testing, verification and model checking. While these methods were shown to be effective in the software development process, they also suffer from severe limitations. Testing is not exhaustive, and frequently, a certain percentage of the errors survive even thorough testing efforts. Formal verification, using logic proof rules, is comprehensive, but extremely tedious; it requires the careful work of logicians or mathematicians for a long time, even for a small piece of code. Model checking is an automatic method; it suffers from high complexity, where memory and time required to complete the task are sometimes prohibitively high.

V. Bertacco and A. Legay (Eds.): HVC 2013, LNCS 8244, pp. 246–261, 2013.

It is only natural that researchers are interested in methods to automatically convert the system specification into software; assuming that formal specification indeed represents the needed requirements fully and correctly (already, a difficult task to achieve), a reliable automatic process would create correct-by-design code. Not surprisingly, efficient and effective automatic synthesis methods are inherently difficult. Problems of complexity and decidability quickly appear. Unless the specification is already close in form to the required system (e.g., one is an automaton, and the other is an implementation of this automaton), this is hardly surprising. One can show examples where the required number of states of a reactive system that is described by a simple temporal specification (using Linear Temporal Logic) grows doubly exponential with the specification [16]. This still leaves open the question of whether there is always a more compact representation for such a specification, a problem that is shown [6] to be as hard as proving open problems about the equivalence of certain complexity classes.

Software synthesis is a relatively new research direction. The classical Hoare proof system for sequential programs [7] can be seen not only as a verification system, but also as an axiomatic semantics for programs, and also as a set of rules that can be used to preserve correctness while manually refining the requirement from a sequential system into correct code. The process is manual, requiring the human intuition of where to split the problem into several subparts, deciding on where a sequential, conditional or iterating construct needs to be used, and providing the intermediate assertions. Manna and Wolper [17] suggested the transformation of temporal logic into automata, and thence to concurrent code with a centralized control. A translation to an automaton (on infinite sequences) provides an operational description of these sequences. Then, the operations that belong, conceptually, to different processes, are projected out on these processes, while a centralized control enforces globally the communication to occur in an order that is consistent with the specification.

More recent research on synthesis is focused on the interaction between a system and its environment, or the decomposition of the specified task into different concurrent components, each having limited visibility and control on the behavior of the other components. The principle in translating the given temporal specification into such systems is based on the fact that the components need to guarantee that the overall behavior will comply with the specification while it can control only its own behavior. This calls for the use of some intermediate automata form. In this case, it is often automata on trees, which include the possible interactions with the environment. In addition, synthesis includes some game theoretical algorithms that refine the behaviors, i.e., the possible branches of the tree, so that the overall behavior satisfy the specification. The seminal work of Pnueli and Rosner [22] shows that the synthesis of an open (interactive) system that satisfies LTL properties can be performed using some game theoretical principles. On the other hand, Pnueli and Rosner show [23], that synthesis of concurrent systems that need to behave according to given distributed architecture (as opposed to centralized control) is undecidable.

The approach presented here is quite different. Instead of using a direct algorithmic translation, we perform a generate-and-check kind of synthesis. This brings back to the playground the use of verification methods, such as model checking or SAT solving, on given instances. An extreme approach would be to generate all possibilities (if they can be effectively enumerated) and check them, e.g., by using model checking, one by one. In the work of Bar-David and Taubenfeld [3], mutual exclusion algorithms are synthesized by enumerating the possible solutions and checking them. We focus on a directed search that is based on *genetic programming*. In a nutshell, genetic programming allows us to generate multiple candidate solutions at random and to mutate them, again as a stochastic process. We employ enhanced model checking (model checking that does not only produce an affirmation to the checked properties or a counterexample, but distinguishes also some finer level of correctness) to provide the *fitness* level; this is used by genetic programming to increase or decrease the chance of candidate programs to survive. Our synthesis method can be seen as a heuristic search in the space of syntactically fitting programs.

2 Genetic Programming Based on Model Checking

We present a framework combining genetic programming and model checking, which allows to automatically synthesize software code for given problems. The framework we suggest is depicted at Figure 1, and is composed of the following parts:

- A *user* that provides a formal specification of the problem, as well as additional constraints on the structure of the desired solutions,
- an *enhanced GP engine* that can generate random programs and them evolves them, and
- a *verifier* that analyzes the generated programs, and provides useful information about their correctness.

The synthesis process generally goes through the following steps:

1. The user feeds the GP engine with a set of constraints regarding the programs that are allowed to be generated (thus, defining the space of candidate programs). This includes:
   (a) a set of functions, literals and instructions, used as building blocks for the generated programs,
   (b) the number of concurrent processes and the methods for process communication (in case of concurrent programs), and
   (c) limitations on the size and structure of the generated programs, and the maximal number of permitted iterations.
2. The user provides a formal specification for the problem. This can include, for instance, a set of temporal logic properties, as well as additional requirements on the program behavior.
3. The GP engine randomly generates an initial population of programs based on the fed building blocks and constraints.

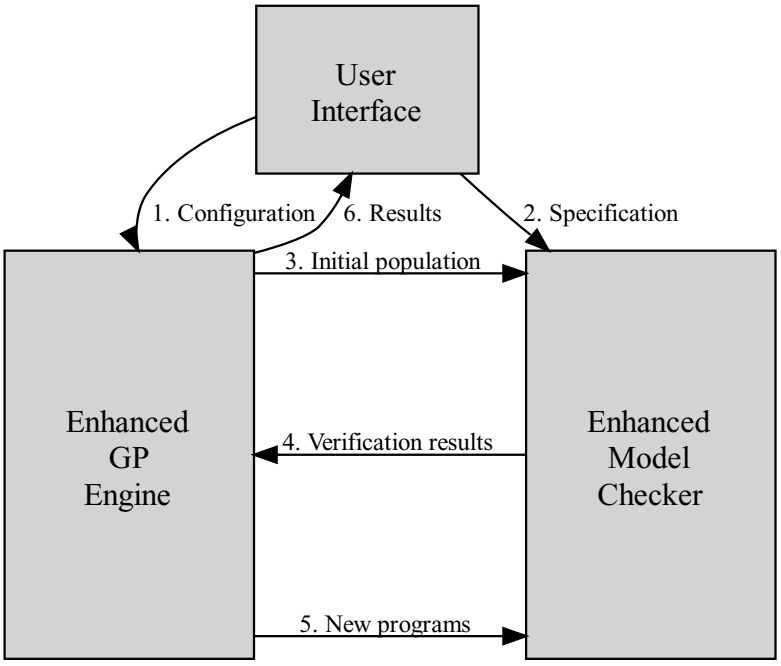

Fig. 1. The Suggested Framework

4. The 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 such as mutation, which performs small changes to the code, and crossover, which cuts two candidate solutions and glues them together, to the existing programs population. Steps 4 and 5 are then 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, if one exists, or the best partially correct programs that was found, along with its verification results.

For steps 4 and 5 above we use the following selection method, which is similar to the Evolutionary Strategies [24] $\mu + \lambda$ style:

- Randomly choose at set of μ candidate solutions.
- Create λ new candidates by applying mutation (and optionally crossover) operations (as explained below) to the above μ candidates.
- Calculate the fitness function for each of the new candidates based on "deep model checking".

– Based on the calculated fitness, choose μ individuals from the obtained set of size $\mu + \lambda$ candidates, and use them to replace the old μ individuals selected at step 2.

Programs Representation. Programs are represented as trees, where an instruction or an expression is represented by a single node, having its parameters as its offspring, and terminal nodes represent constants. Examples of the instructions we use are *assignment, while* (with or without a body), *if* and *block*. The latter is a special node that takes two instructions as its parameters, and runs them sequentially.

A strongly-typed GP [18] is used, which means that every node has a type, and also enforces the type of its offspring.

Initial Population Creation. At the first step, an initial population of candidate programs is generated. Each program is generated recursively, starting from the root, and adding nodes until the tree is completed. The root node is chosen randomly from the set of instruction nodes, and each child node is chosen randomly from the set of nodes allowed by its parent type, and its place in the parameter list. A "grow" method [15] is used, meaning that either terminal or non-terminal nodes can be chosen, unless the maximum tree depths is reached, which enforces the choice of terminals. This method can create trees with various forms. Figure 2(i) shows an example of a randomly created program tree. The tree represents the following program:

```
while (A[2] != 0)
    A[me] = 1
```

Nodes in bold belong to instructions, while the other nodes are the parameters of those instructions.

Mutation. Mutation is the main operation we use. It allows making small changes on existing program trees. The mutation includes the following steps:

1. Randomly choose a node (internal or leaf) from the program tree.
2. Apply one of the following operations to the tree with respect to the chosen node:
   (a) Replace the subtree rooted by the node with a new randomly generated subtree.
   (b) Add an immediate parent to the node. Randomly create other offspring to the new parent, if needed.
   (c) Replace the node by one of its offspring. Delete the remaining offspring of that node.
   (d) Delete the subtree rooted by the node. The node ancestors should be updated recursively (possible only for instruction nodes).

Mutation of type (a) can replace either a single terminal or an entire subtree. For example, the terminal "1" in the tree of Fig. 2(i), is replaced by the grayed subtree in 2(ii), changing the assignment instruction into `A[me] = A[0]`. Mutations of type (b) can extend programs in several ways, depending on the new parent node type. In case a "block" type is chosen, a new instruction(s) will be

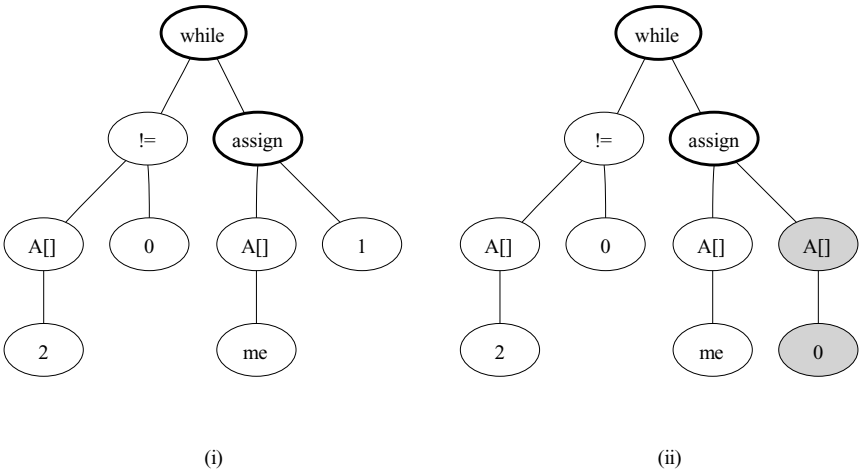

Fig. 2. (i) Randomly created program tree, (ii) the result of a replacement mutation

inserted before or after the mutation node. For instance, the grayed part of Fig. 3 represents a second assignment instruction inserted into the original program. Similarly, choosing a parent node of type "while" will have the effect of wrapping the mutation node with a while loop. Another situation occurs when the mutation node is a simple condition which can be extended into a complex one, extending, for example, the simple condition in Fig. 2 into the complex condition: `A[2] != 0 and A[other] == me`. Mutation type (c) has the opposite effect, and can convert the tree in Fig. 3 back into the original tree of Fig. 2(i). Mutation of type (d) allows the deletion of one or more instructions. It can recursively change the type, or even cause the deletion of ancestor nodes.

The type of mutation applied on candidate programs is randomly selected, but all mutations must obey strongly typing rules of nodes. This affects the possible mutation type for the chosen node, and the type of new generated nodes.

Crossover. The crossover operation creates new individuals by merging building blocks of two existing programs. The crossover steps are:

1. Randomly choose a node from the first program.
2. Randomly choose a node from the second program that has the same type as the first node.
3. Exchange between the subtrees rooted by the two nodes, and use the two new programs created by this method.

While traditional GP is heavily based on crossover, it is quite a controversial operation (see [2], for example), and may cause more damage than benefit in the evolutionary process, especially in the case of small and sensitive programs that we investigate. Thus, crossover is barely used in our work.

The Fitness Function. *Fitness* is used by GP in order to choose which programs have a higher probability to survive and participate in the genetic

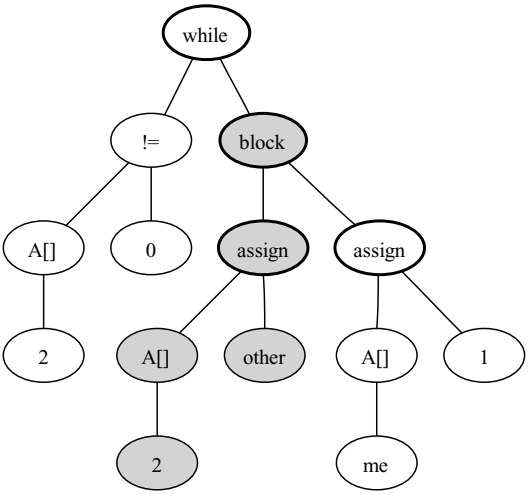

Fig. 3. Tree after insertion mutation

operations. In addition, the success termination criterion of the GP algorithm is based on the fitness value of the most fitted individual. Traditionally, the fitness function is calculated by running the program on some set of inputs (a training set) which suppose to represent all of the possible inputs. This can lead to programs that work only for the selected inputs (overfitting), or to programs that may fail for some inputs, which might be unacceptable in some domains. In contrast, our fitness function is not based on running the programs on sample data, but on an enhanced model checking procedure. While the classical model checking provides a yes/no answer to the satisfiability of the specification (thus yielding a two-valued fitness function), our *deep model checking* algorithm generated a smoother function by providing several levels of correctness. In fact, we have four levels of correctness, per each specification property, written in Linear Temporal Logic:

1. None of the executions of the program satisfy the property.
2. Some, but not all the executions of the program satisfy the property.
3. The only executions that do not satisfy the property must have infinitely many decisions that avoid a path that does satisfy the property.
4. All the executions satisfy the property.

We provided several methods for generating the various fitness levels:

- Using Streett Automata, and a strongly component analysis of the program graph [10].
- A general deep model checking logic and algorithm. [9,19].
- A technique inspired by probabilistic qualitative LTL model checking [13].

We use a fitness-proportional selection [8] that gives each program a probability of being chosen that is proportional to its fitness value. In traditional GP,

after the μ programs are randomly chosen, the selection method is applied in order to decide which of them will participate in the genetic operations. The selected programs are then used in order to create a new set of μ programs that will replace the original ones.

3 Finding New Mutual Exclusion Algorithms

The first problem for which we wanted to synthesize solutions was the classical *mutual exclusion* problem [4]. The temporal specification (in Linear Temporal Logic) for the problem are given in Table 1.

Table 1. Mutual Exclusion Specification

No.	Type	Definition	Description
1	Safety	$\Box\neg(p_0$ in `CS` $\wedge\ p_1$ in `CS`$)$	Mutual Exclusion
2,3	Liveness	$\Box(p_{me}$ in `Post` $\rightarrow \Diamond(p_{me}$ in `NonCS`$))$	Progress
4,5		$\Box(p_{me}$ in `Pre` $\wedge\ \Box(p_{other}$ in `NonCS`$)) \rightarrow \Diamond(p_{me}$ in `CS`$))$	No Contest
6		$\Box((p_0$ in `Pre` $\wedge\ p_1$ in `Pre`$) \rightarrow \Diamond(p_0$ in `CS` $\vee\ p_1$ in `CS`$))$	Deadlock Freedom
7,8		$\Box(p_{me}$ in `Pre` $\rightarrow \Diamond(p_{me}$ in `CS`$))$	Starvation Freedom
9	Safety	$\Box\neg($`remote writing`$)$	Single-Writer
10	Special	Bounded number of remote operations	Local-Spinning

Initially, we tried to rediscover three of the classical mutual exclusion algorithms - the *one bit* protocol (a deadlock-free algorithm for which we used properties $1-6$ from the above specification), and two starvation-free algorithms (satisfying also properties $7-8$) - *Dekker's* and *Peterson's* algorithms. Our framework (and tool) successfully discovered all of these algorithms [9], and even some interesting variants of them.

Inspired by algorithms developed by Tsay [25] and by Kessels [14], our next goal was to start from an existing algorithm, and by adding more constraints and building blocks, try to evolve into more advanced algorithms.

First, we allowed a minor asymmetry between the two processes. This is done by the operators *not0* and *not1*, which act only on one of the processes. Thus, for process 0, $not0(x) = \neg x$ while for process 1, $not0(x) = x$. This is reversed for $not1(x)$, which negates its bit operand x only in process 1, and do nothing on process 0.

As a result, the tool found two algorithms which may be considered simpler than Peterson's. The first one has only one condition in the *wait* statement, written here using the syntax of a *while* loop, although a more complicated atomic comparison, between two bits. Note that the variable turn is in fact A[2] and is renamed here turn to accord with classical presentation of the extra global bit that does not belong to a specific process.

```
Pre CS
A[me] = 1
turn = me
While (A[other] != not1(turn));
Critical Section
A[me] = 0
```

The second algorithm discovered the idea of setting the turn bit one more time after leaving the critical section. This allows the while condition to be even simpler. Tsay [25] used a similar refinement, but his algorithm needs an additional if statement, which is not used in our algorithm.

```
Pre CS
A[me] = 1
turn = not0(A[other])
While (A[2] != me);
Critical Section
A[me] = 0
turn = other
```

Next, we aimed at finding more advanced algorithms satisfying additional properties. The configuration was extended into four shared bits and two private bits (one for each process). The first requirement was that each process can change only its 2 local bits, but can read all of the 4 shared bits (the new constraint was specified as the safety property 9 in the table above). This yielded the following algorithm.

```
Pre CS
A[me] = 1
B[me] = not1(B[other])
While (A[other] == 1 and B[0] == not1(B[1]));
Critical Section
A[me] = 0
```

As can be seen, the algorithm has found the idea of using 2 bits as the "turn", were each process changes only its bit to set its turn, but compares both of them on the while loop. Finally, we added the requirement for busy waiting only on local bits (i.e. using local spins). The following algorithm (similar to Kessels') was generated, satisfying all properties from the table above.

```
Non Critical Section
A[other] = 1
B[other] = not1(B[0])
T[me] = not1(B[other])
While (A[me] == 1 and B[me] == T[me]);
Critical Section
A[other] = 0
```

4 Synthesizing Parametric Programs

Our experience with genetic program synthesis quickly hits a difficulty that stems from the limited power of model checking: there are few interesting fixed finite state programs that can also be completely specified using pure temporal logic. Most programming problems are, in fact, parametric. Model checking is undecidable even for parametric families of programs (say, with n processes, each with the same code, initialized with different parameters) even for a fixed property [1]. One may look at mutual exclusion for a parametric number of processes. Examples are, sorting, where the number of processes and the values to be sorted are the parameters, network algorithms, such as finding the leader in a set of processes (in order to reinitialize some mutual task), etc. In order to synthesize parametric concurrent programs, in particular those that have a parametric number of processes, and even a parametric architecture, we use a different genetic programming strategy.

First, we assume that a solution that is checked for a large number of instances/parameters is acceptable. This is not a guarantee of correctness, but under the prohibitive undecidability of model checking for parametric programs, at least we have a strong evidence that the solution may generalize to an arbitrary configuration. In fact, there are several works on particular cases where one can calculate the parameter size that guarantees that if all the smaller instances are correct, then any instance is correct [5]. Unfortunately, this is not a rule that can be applied to any arbitrary parametric problem. We apply a *co-evolution* based synthesis algorithm: we collect the cases that fail as counterexamples, and when suggesting a new solution, check it against the collected counterexamples. We can view this process as a genetic search for both correct programs and counterexamples. The fitness is different, of course, for both tasks: a program gets higher fitness by being close to satisfying the full set of properties, while a counterexample is obtaining a high fitness if it fails the program.

One way to observe this kind of co-evolution also as using model checking for instance of the parameters. For example, consider seeking a solution for the classical *leader election in a ring* problem, where processes initially have their own values that they can transfer around, with the goal of finding a process that has the highest value. Then, the parameters include the size of the ring, and the initial assignment of values to processes. While we can check solutions up to a certain size, and in addition, check all possible initial values, the time and state explosion is huge, for both size and permutation of initial values. We can then store each set of instances of the parameters that failed some solution, and, when checking a new candidate solution, check it against the failed instances.

In this sense, the model checking of a particular set of instances can be considered as a generalized *testing* for these values: each set of instances of the parameters provides a single finite state systems that is itself comprehensively tested using model checking. This idea can be used, independently, for model checking: for example, consider a sorting program with a parametric set of values and initial values to be sorted; for a particular size and set of values, the model

checking provides automatic and exhaustive test, but the check is not exhaustive for all the array sizes or array values, but rather samples them.

5 Correcting Erroneous Program

Our method is not limited to finding new program that satisfy the given specification. In fact, we can start with the code of an existing program and try to improve or correct it. When our initial population consists of a given program, which is either non optimal, or faulty, we can start our genetic programming process with it, instead of with a completely random population. If our fitness measure includes some quantitative evaluation, the initial program may be found inferior to some new candidates that are generated. If the program is erroneous, then it would not get a very high fitness value by failing to satisfy some of the properties.

In [12] we approached the ambitious problem of correcting a known protocol for obtaining interprocess interaction called α-core [21]. The algorithm allows multiparty synchronization of several processes. It needs to function in a system that allows nondeterministic choices, which makes it challenging, as processes that may consider one possible interaction may also decide to be engaged in another interaction. The algorithm uses asynchronous message passing in order to enforce live selection of the interactions by the involved processes. This nontrivial algorithm, which is used in practice for distributed systems, contains an error. The challenges in correcting this algorithm are the following:

Size. The protocol is quite big, involving sending different messages between the controlled processes, and new processes, one per each possible multiparty interaction. These messages include announcing the willingness to be engaged in an interaction, committing an interaction, cancelling an interaction, request for commit from the interaction manager processes, as well as announcement that the interaction is now going on, or is cancelled due to the departure of at least one participant. In addition to the size of the code, the state space of such a protocol is obviously high.

Varying architecture. The protocol can run on any number of processes, each process with arbitrary number of choices to be involves in interactions, and each interaction includes any number of processes.

These difficulties make also the model checking itself undecidable [1] in general, and the model checking of a single instance, with fixed architecture, hard. In fact, we use our genetic programming approach first to find the error, and then to correct it. We use two important ideas:

1. Use the genetic engine not only to generate programs, but also to evolve different architectures on which programs can run.
2. Apply a co-evolution process, where candidate programs, and test cases (architectures) that may fail these programs, are evolved in parallel.

Specifically, the architecture for the candidate programs is also represented as code (or, equivalently, a syntactic tree) for spanning processes and their interactions, which can be subjected to genetic mutations. The fitness function directs the search into program that may falsify the specification for the current program. After finding a "bad" architecture for a program, one that causes the program to fail its specification, our next goal is to reverse the genetic programming direction, and try to automatically correct the program, where a "correct" program at this step, is one that has passed model checking against the architecture. Yet, correcting the program for the first found wrong architecture only, does not guarantee its correctness under different architectures. Therefore, we introduce a new algorithm (see Algorithm 1) which co-evolves both the candidate solution programs, and the architectures that might serve as counterexamples for those programs.

Algorithm 1: Model checking based co-evolution

```
MC-CoEvolution(initialProg, spec, maxArchs)
(1)     prog := initialProg
(2)     InstantList := ∅
(3)       while |archList| <maxArchs
(4)         arch := EvolveArch(prog, spec)
(5)         if arch = null
(6)           return true // prog stores a "good" program
(7)         else
(8)           add arch to archlist
(9)         prog := EvolveProg(archlist, spec)
(10)        if prog is null
(11)          return false // no "good" program was found
(12)      return false // can't add more architectures
```

The algorithm starts with an initial program *initProg*. This can be the existing program that needs to be corrected, or, in case that we want to synthesize some code, a randomly generated program. It is also given a specification *spec* which the program to be corrected or generated should satisfy. The algorithm then proceeds in two steps. First (lines $(4) - (8)$), the *EvolveArch* function is called. The goal of this function is to generate an architecture on which the specification *spec* will not hold. If no such architecture is found, the *EvolveArch* procedure returns *null*, and we assume (though we cannot guarantee) that the program is correct, and the algorithm terminates. Otherwise, the found architecture *arch* is added to the architecture list *archList*, and the algorithm proceeds to the second step (lines $(9) - (11)$).

In this step, the architecture list and the specification are sent to the *EvolveProg* function which tries to generate programs which satisfy the specification under *all* of the architectures on the list. If the function fails, then the algorithm terminates without success. Since the above function runs a Genetic Programming process which is probabilistic, instead of terminating the algorithm, it is possible to increase the number of iterations, or to re-run the function so a new

search is initiated. If a correct program is found, the algorithm returns to the first step at line (4), on which the newly generated program is tested. At each iteration of the *while* loop, a new architecture is added to the list. This method serves two purposes. First, once a program was suggested, and refuted by a new architecture, it will not be suggested again. Second, architectures that were complex enough to fail programs at previous iterations, are good candidates to do so on future iterations as well. The allowed size of the list is limited in order to bound the running time of the algorithm.

Both *EvolveProg* and *EvolveArch* functions use genetic programming and model checking for the evolution of candidate solutions (each of them is equipped with relevant building blocks and syntactic rules), while the fitness function varies. For the evolution of programs, a combination of the methods proposed in [10,11] is used: for each LTL property, an initial fitness level is obtained by performing a deep model checking analysis. This is repeated for all the architectures in *archList*, which determines the final fitness value. For the evolution of the architectures, we reverse the goal of the fitness function, and give higher score for architectures that are having a better chances to falsify the program. At the end, the smallest architecture that manifested the failure included two processes, with two alternative communication between both of them.

6 A Tool for Genetic Programming Based on Model Checking

We constructed a tool, MCGP [13], that implements the our ideas about model checking based genetic programming. Depending on these setting, the tool can be used for several purposes:

- Setting all parts as *static* will cause the tool to just run the deep model checking algorithm on the user-defined program, and provide its detailed results.
- 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 by searching for a configuration that violates its specification (see [12]).
- Setting both the *init* and the program processes as *dynamic*, is used for synthesizing parametric programs, where the tool alternatively evolves various programs, and configurations under which the programs have to be satisfied.

7 Conclusions

We suggested the use of a methodology and a tool that perform a genetic programming search among versions of a program by code mutation, guided by

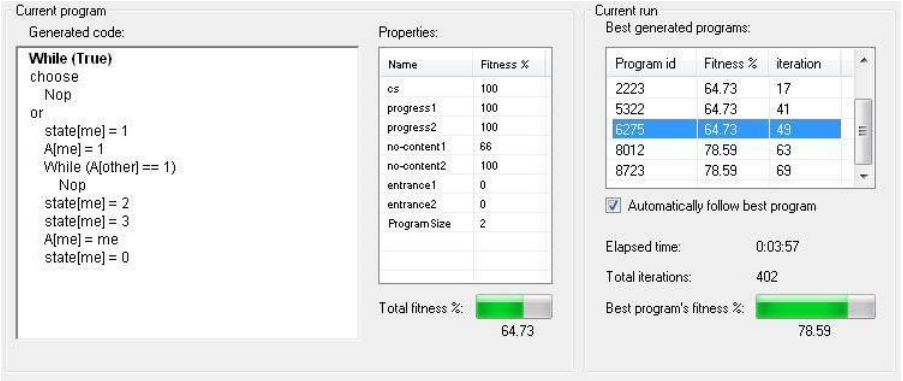

Fig. 4. User interface during synthesis of a mutual exclusion algorithm

model checking results. Code mutation is at the kernel of genetic programming (crossover is also extensively used, but we did not implement it). Our method can be used for

- synthesizing correct-by-design programs,
- finding an error in protocol with complicated architecture (where the architecture can also undergo genetic mutation),
- automatically correcting erroneous code with respect to a given specification, and
- improve code, e.g., to perform more efficiently.

We demonstrated our method on the classical mutual exclusion problem, and were able to find existing solutions, as well as new solutions.

In general, the verification of parametric systems is undecidable, and in the few methods that promise termination of the verification, quite severe restrictions are required. The same apply to code synthesis. Nevertheless, we provide a co-evolution method for synthesize parametric systems based on accumulating cases to be checked: architectures on which the synthesis failed before, or test cases based on previous counterexamples are accumulated to be checked later with new candidate solutions. As the model checking itself is undecidable, we finish if we obtain a strong enough evidence that the solution is correct on the accumulated cases.

Although our method does not guarantee termination, neither for finding the error, nor for finding a correct version of the algorithm, it is quite general and can be fine tuned through provided heuristics in a convenient human-assisted process of code correction.

An important strength of the work that is presented here is that it was implemented and applied on a complicated published protocol to find and correct an actual error.

References

1. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction. In: On the Automatic Evolution of Computer Programs and its Applications, 3rd edn. Morgan Kaufmann, dpunkt.verlag (2001)
3. Bar-David, Y., Taubenfeld, G.: Automatic discovery of mutual exclusion algorithms. In: PODC, p. 305 (2003)
4. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM 8(9), 569 (1965)
5. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL, pp. 85–94 (1995)
6. Fearnley, J., Peled, D., Schewe, S.: Synthesis of succinct systems. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 208–222. Springer, Heidelberg (2012)
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
8. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, Cambridge (1992)
9. 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. Katz, G., Peled, D, vol. 5311, pp. 33–47. Springer, Heidelberg (2008)
10. 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)
11. 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)
12. 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)
13. 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)
14. Kessels, J.L.W.: Arbitration without common modifiable variables. Acta Inf. 17, 135–141 (1982)
15. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
16. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design 19(3), 291–314 (2001)
17. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Trans. Program. Lang. Syst. 6(1), 68–93 (1984)
18. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation 3(2), 199–230 (1995)
19. Niebert, P., Peled, D., Pnueli, A.: Discriminative model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. Niebert, P., Peled, D., Pnueli, A, vol. 5123, pp. 504–516. Springer, Heidelberg (2008)
20. Peled, D.: Software Reliability Methods. Springer (2001)

21. Perez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency - Practice and Experience 16(12), 1173–1206 (2004)
22. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
23. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: FOCS, pp. 746–757 (1990)
24. Schwefel, H.-P.P.: Evolution and Optimum Seeking: The Sixth Generation. John Wiley & Sons, Inc., New York (1993)
25. Tsay, Y.-K.: Deriving a scalable algorithm for mutual exclusion. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 393–407. Springer, Heidelberg (1998)