Int J Softw Tools Technol Transfer (2017) 19:449–464
DOI 10.1007/s10009-016-0418-1

REGULAR PAPER

Synthesizing, correcting and improving code, using model checking-based genetic programming

Gal Katz[1] · Doron Peled[1]

Published online: 23 March 2016

Abstract We show here how the use of genetic programming in combination of model checking 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. We show several examples where we used our approach to synthesize, improve and correct code.

Keywords Genetic programming · Model checking · Synthesis

1 Introduction

Formal methods, e.g., testing, verification and model checking, can provide the means for manually or automatically enhancing the reliability of programs and algorithms. In particular, *model checking* can be practically used for automatic verification of finite state concurrent systems, such as hardware circuits and communication protocols, against their specification. Model checking tools can be used to prove the correctness of models of such systems, or provide counterexamples, in case the specification is violated.

A more ambitious goal is the synthesis of systems directly from their specification [44]. Such an ability would allow skipping the stages of development and verification, since the synthesized code is guaranteed to be correct by design. While there are various theoretical results related to system synthesis, practical implementations of synthesis are quite rare. First, for concurrent systems there are undecidability results [45], while positive decidability results are quite restrictive [32]. Furthermore, even for the decidable cases, synthesis algorithms are usually complicated, having high computational complexity [16,44], and the generated code is not guaranteed to be efficient.

While there are various methodologies aiding with the development of large software projects, the essence of solving new algorithmic problems still requires programmers to exhibit creativity and ingenuity. Even when initial solutions are found, it may a very long time until further improvements are achieved. This difficulty, along with the impressive success of evolution, has encouraged computer scientists to develop techniques that mimic some aspects of the evolutionary process, in order to automatically generate and improve computer programs. Genetic programming (GP) [2,30,46] is a search based software engineering approach [18], 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., Test-based genetic programming is used in [34] to speed up the performance of systems.

Our goal is to combine ideas from both the *genetic programming* and *formal verification methods* to develop a synthesis method that would benefit from the synergy between these domains and hopefully provide better results than those achieved when using the techniques separately. We

The research of the 2nd author was supported in part by ISF grant 1262/09 “Synthesis of programs using combination of verification and genetic programming.

✉ Doron Peled
doron.peled@gmail.com

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

present a framework that allows starting with a formal specification for a given problem and automatically synthesizing code that satisfies the specification. While genetic programming fitness function is based oftentimes on test cases, in our approach it is based on the more comprehensive (yet more complex) model checking.

Our motivation for using genetic programming for program synthesis stems from the mentioned undecidability of synthesis of concurrent programs. Genetic programming is a heuristic search, which means it can succeed (but may also sometimes fail) in the task of finding code for a given concurrent programming task even though this task was shown to be in general undecidable. Our method allows the user to fine-tune the search, e.g., by giving additional specifications, and by changing the way that model checking results are affecting the fitness. The particular kind of programming challenges that we aim at are small but intricate. These are not big programs for user applications, but rather the kind of problems where new solutions are published in research papers. Model checking itself has rather high complexity (PSPACE complete in both the size of the system and the specification), but already possesses many efficient heuristics. Hence our method, which applies model checking repeatedly on candidate programs, is best suited for the cases where the aimed generated code is not large: tens to a couple of hundreds lines of code.

Through the use of genetic programming, instead of applying a direct algorithmic translation as in classical synthesis, we perform a generate-and-check kind of synthesis. This brings back to the playground the use of verification methods such as model checking 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.

The use of model checking to provide fitness instead of the traditional use of test cases has several aspects. On the one side, passing a large number of test cases does not guarantee correctness. On the other hand, using a large test suite allows providing a large number of fitness levels, while the number of properties against which the system is verified is relatively small, providing quite a discrete fitness function. Practice shows that it is essential for the success of the GP process to make the fitness function more smooth (or, as in terms of Harman and Jones [18], avoiding the fitness landscape from being flat). In order to make model checking-based genetic programming practical, we introduced "enhanced model checking" [23]; accordingly, our model checking does not only produce a check of correctness of properties, but in order to provide a smoother fitness function, distinguishes also finer levels of correctness, e.g., whether some or most executions satisfy a given specification. Indeed, we demonstrate success in synthesizing some challenging algorithms.

As far as we are aware, the first paper suggesting that genetic programming fitness is based on model checking is by Johnson [22], predating our first own work by a year. There, fitness is obtained by counting the number of temporal properties that hold in generated candidates. One of the main differences between our work and that of Johnson is in providing more fitness values through deep model checking, which distinguishes multiple levels of satisfaction for a given property. Other differences include the use of the logic LTL (Linear Temporal Logic) for specification instead of the logic CTL (Computational Tree Logic), the synthesis of parametric programs and the scale in which we managed to apply our method.

We start with synthesizing solutions for classical concurrent algorithms, such as those used for *mutual exclusion*. This work is composed of both theoretical and practical ideas. Already in the early stages of the research, a prototype tool[1] was developed, implementing the framework described in this paper and the ideas suggested during our work.

We next extend our framework to deal with parametric problems. In this case, the sought programs have to be correct for any instantiation of the parameters. The parameters can include the number of processes, initial values of variables and the communication channels connecting between the different processes. Since parametric verification is in general undecidable [1], we can only verify the correctness of programs for some instances. In order to deal with this limitation, we used our genetic programming method in the co-evolution of both incorrect instances and candidate solutions; new candidates are checked against instances of the parameters, e.g., initial values or communication architectures that already failed some of the previous attempts. Because of the undecidability of model checking for these cases, we stop the synthesis process when some substantial evidence of correctness is accumulated (say, when verification was successful up to some certain number of processes). This use of mutations of parameters and model checking of *instances* of parametric algorithms makes model checking, in some sense, a comprehensive method of testing, when strict model checking is insufficient.

Based on these co-evolution principles, we used our genetic synthesis method to find a solution to the known *leader election* problem [6,14,43]. Furthermore, we used our technique to find an error in a complicated synchronization algorithm, called α-core [41], thus overcoming the theoretical limitation of plain model checking. Subsequently, our prototype tool discovered automatically a correction to this algorithm.

[1] After some years, the tool MCGP is not currently available for free download. We hope to be able to reinstall it in the future.

The rest of the paper is organized as follows: In Sect. 2 we present background on model checking, genetic programming and automatic program synthesis. Section 3 describes our approach of model checking-based genetic programming. Section 4 provides insight into the method through the synthesis of solutions for the mutual exclusion problem. This includes both rediscovering the classical solutions and finding new solutions under more realistic and practical requirements. Section 5 presents the synthesis of a solution for the parametric problem of leader election in a unidirectional ring. In Sect. 6 we present our experience while automatically correcting a complicated communication protocol, called α-core. Here, not only the number of processes is parametric, but also the protocol works with varying communication architectures. GP was used both to find the architecture that manifests the error, as well as to automatically correct the protocol. Section 7 concludes the paper.

2 Preliminaries

2.1 Model checking of temporal properties

Model checking [10] 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 execution paths, due to concurrency and nondeterministic choices made by scheduling or interacting with the environment.

A finite system can be modeled by an automaton. Each *state* of the automaton corresponds to an evaluation of the variables, program 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 identify information essential for the checked property. For example, proposition p may be defined to hold in a state where $x > 0$, while q may be defined to hold when the message buffer between two processes is empty. The same propositions are used by the specification. The formal 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. LTL has the following syntax:

$$\begin{aligned}\varphi \ ::= & \ \mathit{true} \mid \mathit{false} \mid p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid \\ & (\varphi \rightarrow \varphi) \mid \Box\varphi \mid \Diamond\varphi \mid \bigcirc\varphi \mid (\varphi\,\mathcal{U}\varphi),\end{aligned}$$

where $p \in AP$ is a set of atomic propositions. LTL formulas are interpreted over an infinite sequence of states $\xi = s_0 s_1 s_2 \ldots$, where we denote by ξ_i the suffix $s_i s_{i+1} s_{i+2} \ldots$ of ξ (hence $\xi = \xi_0$). For a suffix ξ_i of ξ, the LTL semantics is defined as follows:

- $\xi_i \models p$ iff $s_i \in p$.
- $\xi_i \models \neg\varphi$ iff not $\xi_i \models \varphi$.
- $\xi_i \models (\varphi \vee \psi)$ iff $\xi_i \models \varphi$ or $\xi_i \models \phi$.
- $\xi_i \models \bigcirc\varphi$ iff $\xi_{i+1} \models \varphi$.
- $\xi_i \models (\varphi\,\mathcal{U}\psi)$ iff for some $j \geq i$, $\xi_j \models \psi$, and for all $i \leq k < j$, $\xi_k \models \varphi$.

The rest of connectives can be defined using the following identities: $\mathit{true} = (p \vee \neg p)$, $\mathit{false} = \neg\mathit{true}$, $(\varphi \wedge \psi) = \neg(\neg\varphi \vee \neg\psi)$, $(\varphi \rightarrow \psi) = (\neg\varphi \vee \psi)$, $\Diamond\varphi = (\mathit{true}\,\mathcal{U}\varphi)$, $\Box\varphi = \neg\Diamond\neg\varphi$. For an automaton M representing a system, $M \models \varphi$ if for every execution ξ of M we have $\xi \models \varphi$.

A standard model checking procedure checks whether a system M satisfies a specification φ [52]. The specification φ is often converted into automata A_φ over infinite words [17]. The simplest kind of such automata is called Büchi automata [49]; 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 $\Diamond\Box p$, the translation will necessarily result in a nondeterministic Büchi automaton.

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. These executions can be provided to the user as counterexamples that may help in finding the cause of the violation.

Formally, let M be the model automaton and A_φ the automaton that accepts the executions of the specification φ. We need to check whether $L(M) \subseteq L(\varphi)$. Since

$$\begin{aligned}L(M) \subseteq L(\varphi) \leftrightarrow L(M) \cap \overline{L(\varphi)} &= \emptyset \leftrightarrow L(M) \cap L(\neg\varphi) \\ &= \emptyset \leftrightarrow L(M) \cap L(A_{\neg\varphi}) \\ &= \emptyset\end{aligned}$$

Thus, we can negate φ, building $A_{\neg\varphi}$ rather than building A_φ and then complementing. Then we check whether the language $L(M) \cap L(A_{\neg\varphi})$ is empty.

While for some LTL properties the translation results in rather small Büchi automata [17], there are families of LTL properties for which their corresponding Büchi automata grow exponentially in the size of the properties. The size of the model automaton may grow exponentially with the number of variables and processes. This phenomenon, known as the "state-space explosion", implies that direct verification of certain types of models may be impractical in terms of time and space. There are various techniques, e.g., partial order reduction [9] and abstraction [8] that combat this problem. Many practical problems are parametric, where their specification need to hold for every number of processes. Though parametrized model checking is in general undecidable [1], there are several techniques that can deal with limited cases of parametric problems [53].

While model checking usually returns a yes/no answer, there are some algorithms that can give quantitative and qualitative results. One example is probabilistic model checking [12], which can be used where it is possible to assign probabilities to the transitions in the model.

2.2 Genetic programming

During the 1970s, Holland established the field known as *Genetic Algorithms* (GA) [21]. 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 and combines them into a new solution, which potentially inherits its useful attributes from its parents. 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 algorithms were successfully applied in a large variety of domains, and many variants were suggested for their solutions representation and genetic operations. These algorithms share the following general steps:

1. Randomly generate initial solutions.
2. Evaluate the fitness of each of the solutions.
3. If a satisfactory solution is found, terminate.
4. Otherwise, apply genetic operations on the solutions, and generate new ones.
5. Go to step 2.

Genetic programming [30] 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. The genetic operations were customized to match the flexible structure of individuals. For instance, in tree-based genetic programming, crossover is performed by selecting subtrees on each of the parents and then swapping between them; This forms two new programs, each having parts from both of its parents. Mutation can be carried out by choosing a subtree and replacing it by another randomly generated subtree. The fitness is calculated by directly running the generated programs on a large set of test cases and evaluating the results.

Genetic programming may be viewed as a beam search in the space of computer programs. Mutations allow the advance from one point in the space to another, and crossover provides the ability of "jumping" to new points by merging several previously explored points. In Koza's work [30], crossover is the main genetic operation, and mutations are negligible. On the other hand, there is an ongoing debate about the actual role and importance of crossover. The main question is whether it indeed combines building blocks into larger blocks of code, or just acts as a macro mutation. Hence, there were various suggestions of improving crossover, while other researches focused on the mutation operation [7]. GP has successfully generated complex solutions to problems in a broad range of domains, and it constantly yields human-competitive results [31]. Herman and Jones [18] subscribed genetic programming to a class of heuristic search methods that they termed *search-based software engineering*. These fitness guided search methods, which include also simulated annealing, are aimed at constructing, improving and correcting software artifacts.

2.3 Synthesis

Software synthesis is a relatively new research direction. The classical Hoare proof system for [20] can be seen as an axiomatic semantics for sequential programs. It provides a set of rules that can be used to gradually transform the formal specification into sequential system, while preserving its correctness. 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. Synthesis of infinite state programs is undecidable. As in verification, substantial progress was made once research started to focus on finite state systems.

Manna and Wolper [38] suggested a transformation of temporal logic specification into automata. This translation to an automaton (on infinite sequences) provides an operational description of these sequences. Then, the operations that belong to different processes are projected out on these processes, while a centralized scheduler enforces globally the communication to occur in an order that is consistent with the specification. The main disadvantage of this approach is that due to the centralized scheduler, concurrency is lost.

Reactive systems are required to alternate their internal behavior with inputs provided by an environment. The system does not have control on the inputs from the environment; still, one needs to guarantee that the overall behavior will satisfy the required specification. Research on synthesis of reactive systems is focused on modeling the interaction between the system and the environment as a kind of a two-player game between the system and the environment. The realization of the synthesis problem is in fact the obtained winning strategy for the system. Thus, a classical solution for the synthesis problem [44] consists of translating the specification into an automaton and finding a two-player strategy that would guarantee a correct behavior. The standard translation of an LTL property into a Büchi automaton on infinite words may result in a nondeterministic automaton [49]. While model checking works well with nondeterministic automata [52], finding a winning strategy for the system requires using a deterministic automaton (otherwise, the environment may exploit nondeterminism to win). Thus, a determinization is required [47]. (In fact, this requires using a more structured type of automata on infinite words so that each LTL property has a corresponding deterministic representation.) The translation and the subsequent determinization entrails a doubly exponential explosion. Indeed, one can show a family of LTL specification whose realization requires a state space that is doubly exponential in the size of the specification [32].

Concurrent systems are even more complicated to synthesize: the specified task needs to be decomposed into different components, where each having limited visibility and control on the behavior of the other components. Pnueli and Rosner [45] showed that synthesis of concurrent systems is, in general, undecidable. Decidable cases are quite restrictive, see, e.g., [33].

3 Software synthesis using genetic programming based on model checking

We present a framework combining genetic programming and model checking that allows to automatically synthesize software code for given problems. The framework is depicted in Fig. 1 and is composed of the following parts:

- A *user* 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 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 *configuration*, which is 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),
   (c) Limitations on the size and structure of the generated programs, and the maximal number of permitted iterations.

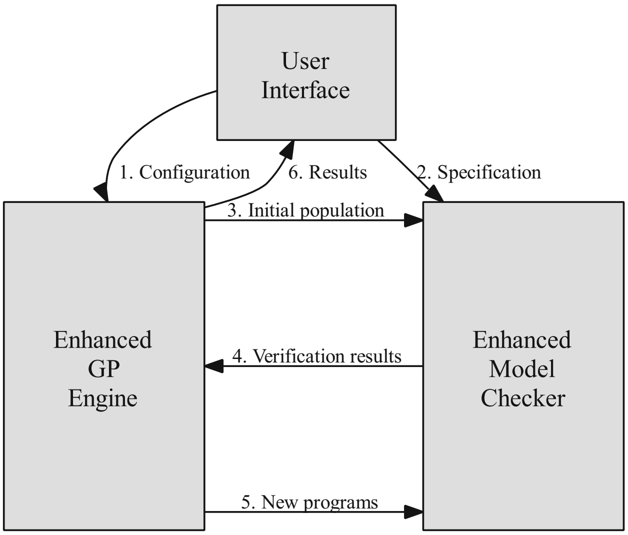


Fig. 1 The suggested framework

2. The user provides a formal specification for the problem. This can include, for instance, 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 building blocks and constraints.
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 such as mutation, which perform small changes to the code, and crossover, which cuts two candidate solutions and glues them together.
   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 [48]:

- Randomly choose a 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 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.

3.1 Programs representation

Programs are represented as trees, where an instruction or an expression is represented as a single node having its parameters as its offspring. Terminal (leaf) nodes represent constants or variables. 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 [39] is used, which means that every node has a type, and also enforces the type of its offspring.

3.2 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 [30] 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. 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 boldface fonts belong to instructions, while the other nodes are the parameters of those instructions.

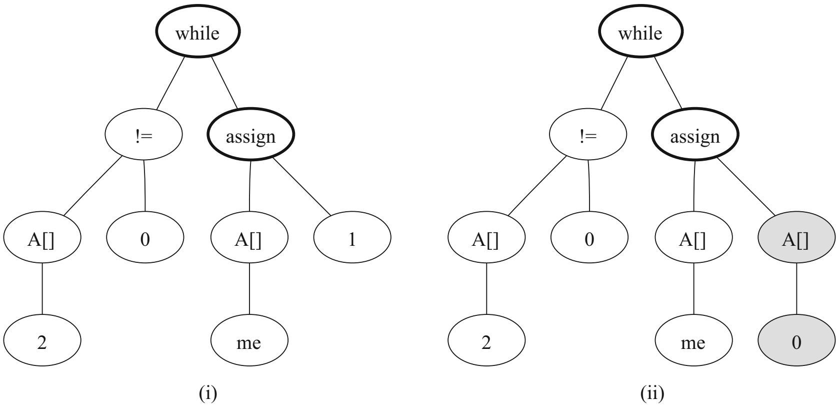
3.3 Mutation

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 subtree consisting of internal node `A[]` and descendant leaf `0`. In Fig. 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 inserted before or after the mutation node. For instance, the tree in Fig. 3 is obtained from the one in Fig. 2(1) by inserting a `block` between the `while` node and its subtree; then a second assignment instruction is inserted as an additional (left) subtree of the `block` node. 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 sim-

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



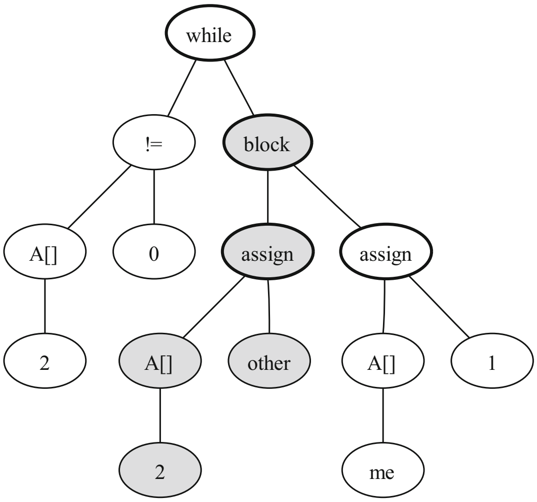


Fig. 3 Tree after insertion mutation

ple condition that can be extended into a complex one, For example, the simple condition `A[2] != 0` in Figure 2 can be converted into the complex condition `A[2] != 0 and A[other] == me` by first inserting an `and` node, above the condition and then complementing its right-hand subtree to correspond to `A[other] == me`.

Mutation type (c) has the opposite effect to mutation type (b) 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 instruction nodes.

The type of mutation applied to 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 newly generated nodes.

3.3.1 Crossover

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

1. Randomly choose a node from the tree representing the first program.
2. Randomly choose a node from the tree representing 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 was not implemented in our work.

3.4 The fitness function

Fitness is an objective function, used by GP to approximate, by means of a single number (say, between 0 and 100), how close is a given candidate programs to satisfy its goal (which is, in our case, satisfying the set of LTL specifications). As with many heuristic measures, the fitness is only an *estimation* of the quality of the candidate solutions: the total order between fitness values of different candidates does not reflect in absolute terms the distance (say in number of mutations needed) between the candidate and an acceptable solution.

Programs with higher probability have a better chance to survive and participate in the genetic 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 or test cases (a training set) which is supposed to represent the possible inputs. This can lead to programs that work only for the selected inputs (over fitting) or to programs that may fail for some inputs. 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.

We use a fitness-proportional selection [21] that gives each program a probability of being chosen that is proportional to its fitness value. Thus, the preference of candidates with higher fitness value is only probabilistic and not absolute. As in traditional GP, after the μ programs are randomly chosen, the selection method is applied to decide which of them will participate in the genetic operations. The selected programs are then used to create a new set of μ programs that will replace the original ones.

3.5 Deep model checking

In this work, we use model checking for constructing the genetic programming fitness function. A fitness function that just sums up the number of specification properties that hold for a candidate solution is quite flat and has a very little chance of leading the genetic process towards convergence. Deep model checking allows us to provide intermediate correctness levels. One possible way to assign further meaningful fitness levels based on model checking is according to the following possibilities, listed according to an increasing order of fitness:

1. φ is not satisfied by any of the executions of M.
2. φ is satisfied with probability 0. This means that φ may still hold for some executions, but the fragment of exe-

cutions that satisfies it is smaller than any fixed rational number.

3. φ is satisfied in M with some positive probability that is neither 0 nor 1.
4. φ is satisfied with probability 1 (i.e., $\neg\varphi$ is satisfied with probability 0).
5. φ is satisfied by all the executions of M.

In traditional model checking, one does not separate between the levels 1–4; they all represent the fact that the property does not hold for at least some executions, which is unacceptable as a correctness criterion. However, for our purpose of providing fitness of candidate solutions, the multiple levels provide a finer way of distinguishing between the imperfect solutions that are generated along the way. Thus, we are not only interested in finding whether a property is satisfied or not, but also in some meaningful intermediate levels. We call this *deep model checking*. The use of deep model checking to provide multiple fitness values is one of the main differences between this work and a prior work that used the binary possibilities of model checking for providing fitness [22]. This allows the genetic programming to scale up and converge into solutions for complicated specifications.

Performing model checking according to level 1 (level 5, respectively) can be done by translating both the system and the property $\varphi(\neg\varphi)$, respectively) into a automata on infinite words (usually, a Büchi automaton) and checking emptiness of the intersection between these two automata. The complexity to do this is PSPACE, (based on binary search). However, most practical model checkers do it with exponential time and space (which is much faster than the merely theoretical PSPACE algorithm) [52]. To calculate the probabilistic levels, we need to apply other algorithms. A simple algorithm translates the property φ (or $\neg\varphi$) to a deterministic automaton on infinite words. For this, a Büchi automata would not be sufficient (not every LTL property can be expressed using a deterministic Büchi automaton), and a more structured acceptance condition, such as Rabin or Street [49], can be used. The translation from φ to such an automaton is doubly exponential [47]. There are also alternative decision procedures that check these levels at the same complexity as checking LTL correctness, e.g., [11,12]. Note that the probabilistic levels (2 to 4) are independent of the actual distribution of the choice of transitions. Further probabilistic levels can be obtained by assigning a probability distribution on choices made from each state. As a default, probabilistic model checking tools [19] often assume uniform distribution for all possible choices.

Alternatively, we can define other levels:

A. No execution satisfies the property.
B. Some executions satisfy the property (excluding levels C and D below).
C. Every finite prefix of an execution can be extended into an execution that satisfies φ.
D. All the executions satisfy the property.

Levels A and D are the same as levels 1 and 5 above, respectively. Levels A, B and D can be checked using standard model checking algorithms. Level C can be checked according to an algorithm in [40], whose complexity is doubly exponential. Level C cannot be ordered with the probabilistic levels above; hence we cannot add it to the previous list of levels.

Overall, given several properties, the fitness is not necessarily the sum of fitness values of the different properties. We sometimes decide to give some properties higher weights over others. Tuning these weights is part of the tools available for the person applying the genetic programming. It helps in obtaining convergence into a correct implementation when the genetic process intermediately fails.

In case of concurrent programs, one may need to require that the program would work under certain "fairness" conditions [35] disallowing an execution where a transition or a process is neglected forever when it can be executed continuously or infinitely many times from some point. Some properties cannot be guaranteed without fairness assumptions, which represent realistic physical restrictions on the execution model. The probabilistic based levels 2, 3 and 4, in fact, include some implied assumption about the executions:

Strong Transition Fairness: Every transition that has infinitely many points in the execution where it can be executed, will execute infinitely many times.

This is because with probability 1, any infinite execution will include any possible choice from each state that occur infinitely often on it [51]. A property φ holds with probability 1 (level 4) exactly when all the executions that are fair under this definition satisfy φ. It holds with probability 0 (level 0) if only unfair executions satisfy φ. This fairness assumption is stronger than some common fairness assumptions [37] that are used for verification, e.g.,

Weak Process Fairness. Every process that can execute some transition from some point in the execution and onwards will eventually execute some transition.
Weak Transition Fairness. Every transition that can be executed from some point onwards will eventually be executed.
Strong Process Fairness. Every process that has infinitely many points in the execution where it can execute a transition will execute infinitely many transitions.

Note that when one fairness assumption is stronger than another, it is satisfied by the same or *less* executions. Consequently, the same or *more* properties are satisfied under the stronger fairness assumption. For details of model checking under fairness algorithms, see [36].

A typical phenomenon of genetic algorithms, called *parsimony*, is that the code can easily bloat by gain unnecessary components, e.g., `x:=y; y:=x`. We apply *parsimony pressure* [4], meaning that we provide some negative fitness value, depending on the length of the generated code. As a consequence, even a perfect solution would not have a clean 100 fitness value. Thus, the condition for successfully finishing the genetic search is not based on this value, but rather that all the checked properties are verified (model checked) to hold.

4 Synthesizing solutions for the mutual exclusion problem

As an example, we have used our method to automatically generate solutions to several variants of the Mutual Exclusion Problem.

4.1 The classical mutual exclusion problem

In this problem, described by Dijkstra [13], two or more processes are repeatedly running critical and non-critical sections of a program. The goal is to avoid the simultaneous execution of the critical section by more than one process. We limit our search for solutions to the case of two processes. The configuration of problem requires the following program parts, executed in an infinite loop:

```
Non Critical Section
Pre Protocol
Critical Section
Post Protocol
```

These four program parts are denoted by `NonCS`, `Pre`, `CS` and `Post`, respectively.

The `Non Critical Section` represents the process part on which it does not require an access to the shared resource. A process can make a nondeterministic choice whether to stay in that part, or to move into the `Pre Protocol`. From the `Critical Section`, a process always has to move into the `Post Protocol`. The `Non Critical Section` and `Critical Section` are fixed, while our goal is to automatically generate code for the `Pre Protocol` and `Post Protocol`, such that the entire program will fully satisfy the specification.

We use a restricted high-level language based on the C language. Each process has access to its id (0 or 1) using the `me` literal and to the other process' id using the `other` literal. The processes can use an array of shared bits. The number of allowed shared bits is specified as part of the configuration. The two processes run the same code. The available node types are *assignment, if, while, empty-while, block, and ,or* and *array*. Terminals include the constants: *0, 1, 2, me* and *other*.

Table 1 describes the properties that define the problem specification. Property 1 is the basic safety property requiring the mutual exclusion. Properties displayed in pairs are symmetrically defined for the two processes. Properties 2 and 3 guarantee that the processes are not hung in the `Post Protocol`. Similar properties for the `Critical Section` are not needed since it is a fixed part without an evolved code. Properties 4 and 5 require that a process can enter the critical section if it is the only process trying to enter it. Property 4 requires that if both processes are trying to enter the critical section, at least one of them will eventually succeed. This property can be replaced by the stronger requirements 7 and 8 that guarantee that no process will starve.

There are several known solutions to the Mutual Exclusion problem, depending on the number of shared bits in use, the type of conditions allowed (simple/complex) and whether starvation-freedom is required. The variants of the problem we wish to solve are shown in Table 2.

4.2 Experimental results

We used a specially designed model checker and GP engine, which implement the methods described earlier. This prototype tool was called MCGP and is described in [28]. It allows selecting the set of allowed commands and the number of variables and modes of communication (synchronous, asynchronous). Synthesis starts with a given program structure or

Table 1 Mutual exclusion specification

No.	Type	Definition	Description	Level
1	Safety	$\Box\neg(p_0 \text{ in } \mathtt{CS} \wedge p_1 \text{ in } \mathtt{CS})$	Mutual exclusion	1
2,3	Liveness	$\Box(p_{me} \text{ in } \mathtt{Post} \rightarrow \Diamond(p_{me} \text{ in } \mathtt{NonCS}))$	Progress	2
4,5		$\Box(p_{me} \text{ in } \mathtt{Pre} \wedge \Box(p_{other} \text{ in } \mathtt{NonCS})) \rightarrow \Diamond(p_{me} \text{ in } \mathtt{CS}))$	No contest	3
6		$\Box((p_0 \text{ in } \mathtt{Pre} \wedge p_1 \text{ in } \mathtt{Pre}) \rightarrow \Diamond(p_0 \text{ in } \mathtt{CS} \vee p_1 \text{ in } \mathtt{CS}))$	Deadlock freedom	4
7,8		$\Box(p_{me} \text{ in } \mathtt{Pre} \rightarrow \Diamond(p_{me} \text{ in } \mathtt{CS}))$	Starvation freedom	4

Table 2 Mutual exclusion variants

Variant no.	Number of bits	Conditions	Requirement	Relevant properties	Known algorithm
1	2	Simple	Deadlock freedom	1,2,3,4,5,6	One bit protocol [5]
2	3	Simple	Starvation freedom	1,2,3,4,5,7,8	Dekker [13]
3	3	Complex	Starvation freedom	1,2,3,4,5,7,8	Peterson [42]

Table 3 Test results

Variant no.	Successful runs (%)	Avg. run duration (s)	Avg. no. of tested programs per run
1	40	128	156,600
2	6	397	282,300
3	7	363	271,950

architecture, where some parts are fixed. The architecture can also include the number of processes and the communication links between them. For mutual exclusion, the configuration must dictate a loop that contains the critical section. Providing initial code forces the genetic search to start from a given solution that we may need to improve or correct. These modes were not used for the synthesis of simple mutual exclusion solutions. Thus, we return to the modes of execution in the tool at the end of the last synthesis example, in Sect. 6.

Three different configurations were used to search for solutions to the variants described in Table 2. Each run included the creation of 150 initial programs by the GP engine, and the iterative creation of new programs until a perfect solution was found, or until a maximum of 2000 iterations. In each iteration, five programs were randomly selected, mutated and replaced using mutation (and crossover) operations, as described in Sect. 3. The values $\mu = 5, \lambda = 150$ where chosen. The tests were performed on a 2.6-GHz Pentium Xeon Processor. For each configuration, multiple runs were performed. Some of the runs converged into perfect solutions, while others found only partial solutions. The results are summarized in Table 3.

Test 1

At the first test, we tried to find a deadlock-free algorithm solving the mutual exclusion problem. The configuration in this case allows the use of two shared bits and only simple conditions. Following is the analysis of a successful run. The numbers in the square brackets under each program below represent the program fitness scores.

The initial population contained 150 randomly generated programs with various fitness scores. Many programs did not satisfy even the basic mutual exclusion safety property 1 and thus achieved a fitness score of zero.

The programs were gradually improved by the genetic operations, until program (a) was created. This program fully satisfies all of the properties, which makes it a correct solution. At this stage, we could end the run; however, we kept it for some more iterations. Due to parsimony pressure, the program is finally evolved by a series of deletion and replacement mutations into program (b). This program is a perfect solution to the requirements, which is actually the known one-bit protocol [5].

```
Non Critical Section
A[me] = 1
While (A[other] != 0)
    A[me] = me
    While (A[other] != A[0])
        While (A[1] != 0)
    A[me] = 1
Critical Section
A[me] = 0
```

(a) [96.50]

```
Non Critical Section
A[me] = 1
While (A[other] != 0)
    A[me] = me
    While (A[other] == 1)
    A[me] = 1
  Critical Section
A[me] = 0
```

(b) [97.10]

Test 2

At the second test we changed the configuration to support three shared bits. This allowed the creation of algorithms like Dekker's [13] which uses the third bit to set turns between the two processes. Since the requirements were similar to those of the previous test (accept the change of property 6 by 7 and 8), many runs initially converged into deadlock-free algorithms using only two bits. queryPlease consider rephrasing the following sentence: Those algorithms have execution paths at which one of the processes starve, hence only partially satisfying properties 7 or 8. Program (c) shows one of those algorithms, which later evolved into program (d).Those algorithms have execution paths at which one of the processes starve, hence only partially satisfying properties 7 or 8. Program (c) shows one of those algorithms, which later evolved into program (d). The evolution first included the addition of the second line to the *post protocol* section (which only slightly decreased its fitness level due to the parsimony pressure). A replacement mutation then changed the inner while loop condition, leading to a perfect solution similar to Dekker's algorithm.

Another interesting algorithm generated by one of the runs is program (e). This algorithm (also reported at [3]) is a perfect solution too, but it is shorter than Dekker's algorithm.

```
Non Critical Section
A[me] = 1
While (A[other] == 1)
   While (A[0] != other)
      A[me] = 0
   A[me] = 1
Critical Section
A[me] = 0
```

(c) [94.34]

```
Non Critical Section
A[me] = 1
While (A[other] == 1)
   While (A[2] == me)
      A[me] = 0
A[me] = 1
Critical Section
A[2] = me
A[me] = 0
```

(d) [96.70]

```
Non Critical Section
A[other] = other
if (A[2] == other)
  A[2] = me
  While (A[me] == A[2])
Critical Section
A[other] = me
```

(e) [97.50]

Test 3

At this test, we added the *and* and *or* operators to the function set, allowing the creation of complex conditions. Some of the runs evolved into program (f) which is the known Peterson's algorithm [42].

```
Non Critical Section
A[me] = 1
A[2] = me
While (A[other] == 1 and A[2] != other)
Critical Section
A[me] = 0
```

(f) [97.60]

4.3 Finding new mutual exclusion algorithms

Inspired by algorithms developed by Tsay [50] and by Kessels [29], our next goal was to start from an existing algorithm and, by adding more constraints and building blocks, try to evolve into advanced mutual exclusion 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 does nothing on process 0.

As a result, the tool found two algorithms that 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 using 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 [50] 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 two local bits, but can read all of the 4 shared bits. 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 GP algorithm discovered the idea of using two bits as the "turn", where 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' [29]) 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
```

5 Synthesizing parametric programs

Our experience with genetic program synthesis has quickly hit a difficulty: there are few interesting fixed finite state programs that can be completely specified using pure temporal logic. Most programming problems are, in fact, parametric. Model checking is undecidable 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 example of a parametrized problem is mutual exclusion for an arbitrary (i.e., parametric) number of processes. Another one is sorting, where the number of processes and the values to be sorted are the parameters.

We chose to look at a programming problem, called *leader election* [6,14,43]. In the version of the problem we considered [25], there is a unidirectional ring of processes, each having a unique value. We want to select a process, the one that has the largest value, to be a *leader*. The processes are symmetric in the sense that the perform exactly the same code and are not aware of the size of the ring. The selection of the leader will allow breaking the symmetry for the benefit of other network algorithms.

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 cases where one can calculate the parameter size that guarantees that if all the smaller instances are correct, then any instance is correct [15]. 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 and keep them as counterexamples. When suggesting a new solution, it is checked 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 the two tasks: a proposed program gets higher fitness by being close to satisfying the full set of properties, while a counterexample is obtaining a high fitness value if it fails the program.

During the co-evolution process, we use model checking for verifying instance of the parameters. In the leader election problem, the parameters include the size of the ring and the initial assignment of values to processes. For simplicity, we can assume that the values assigned to processes in a ring of size n are just a permutation of the values $\{1, 2, \ldots, n\}$. In light of the undecidability of the parametric model checking problem, one possibility to gain confidence in our solution is to check solutions up to a certain size and in addition check all possible initial permutations. However, model checking is intractable because of the number of concurrent processes and the number of permutations of initial values. Instead, during the co-evolution we store each set of instances of the parameters that failed some solution and when checking a new candidate solution, check it against the collected failed instances. In this sense, the model checking of a particular set of instances can be considered as a generalized *testing* for these values. Rather than an execution, in our case, each test case is a finite state system that is comprehensively explored using model checking.

We evaluated our experiments with respect to existing solutions. The Chang and Roberts algorithms [6] required $O(n^2)$ messages for n processes. The algorithms of Peterson [43] and that of Dolev, Klawe and Rodeh [14] require the asymptotically more modest $O(n \times \log n)$ messages. We managed to generate via GP several solutions, including the Chang and Roberts algorithm. However, we did not manage to generate any algorithm of complexity $O(n \times \log n)$.

6 Correcting erroneous programs

Our method is not limited to finding new programs 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 qualitative 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 [27] we approached the ambitious problem of correcting a known protocol for obtaining interprocess interaction, called α-core [41]. The algorithm allows multiparty synchronization of several processes. Besides the processes that perform local and synchronized transitions there are several supervisory processes, each responsible for a fixed type of interaction between multiple processes. It needs to function

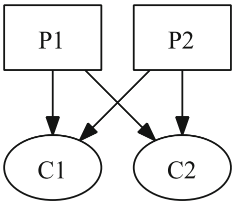


Fig. 4 An architecture with two processes and two supervisory managers

in a system that allows nondeterministic choices: processes that may consider one possible interaction may also decide to be engaged in another interaction. The algorithm uses asynchronous message passing between the processors and the synchronization supervisors to enforce a selection of the interactions by the involved processes without deadlock. In Fig. 4 we demonstrate two processes P_1 and P_2 that are involved in two types of interactions with each other, through the supervision of managers C_1 and C_2. This nontrivial algorithm, which is used in practice in distributed systems, contains an error in its published version. 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, canceling an interaction, requesting for commit from the interaction manager processes, as well as announcing that the interaction is now going on, or is canceled due to the departure of at least one participant. The state space of such a concurrent protocol is obviously high.
> *Varying architecture.* The protocol can run on any number of processes, each process with arbitrary number of choices to be involved in interactions and with each interaction involving any number of processes.

The parametric nature of the problem makes the model checking itself undecidable [1] in general, and even model checking of a particular instance, with fixed architecture, is hard. In fact, we used our genetic programming approach first to find the error, including architecture instance that manifests the problem, and then to correct it. We used two important ideas:

1. We employ the genetic engine not only to generate programs, but also to evolve different architectures on which programs can run.
2. We apply a co-evolution process, where evolution of candidate programs and of architectures that fail these candidates is intermixed.

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 for a 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. Then we try to automatically correct the program. 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 for 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 these steps, the architecture list and the specification are sent to the *EvolveProg* function, which tries to generate programs that 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 rerun 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 against different architectures. 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 failed programs at previous iterations are good candidates to do so on future iterations as well. The allowed size of the list is limited 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 [24,26] 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.

For the α-core algorithm, the smallest architecture that manifested the failure included two processes, with two alternative communications between both of them. The architecture that was found to produce the error in the original α-core algorithm is the one appearing in Figure 4. A message sequence chart in Fig. 5 demonstrate the bad scenario that was found. While we did not describe in this paper the α-core algorithm, the scenario demonstrates the intricacy of the algorithm. The correction consisted of changing the following line of code

```
if n > 0 then n := n − 1
```

into

```
if sender ∈ shared then n := n − 1
```

We are not aware of any correction of the α-core algorithm that results in this particular change.

In order to support the different capabilities of synthesizing and correcting code for fixed and varying architectures, the prototype tool MCGP [28] we constructed for carrying out the experiments can be used in different modes:

- 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 an *init* process that defines the architecture of processes and the interaction between them 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 [27]).
- 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.

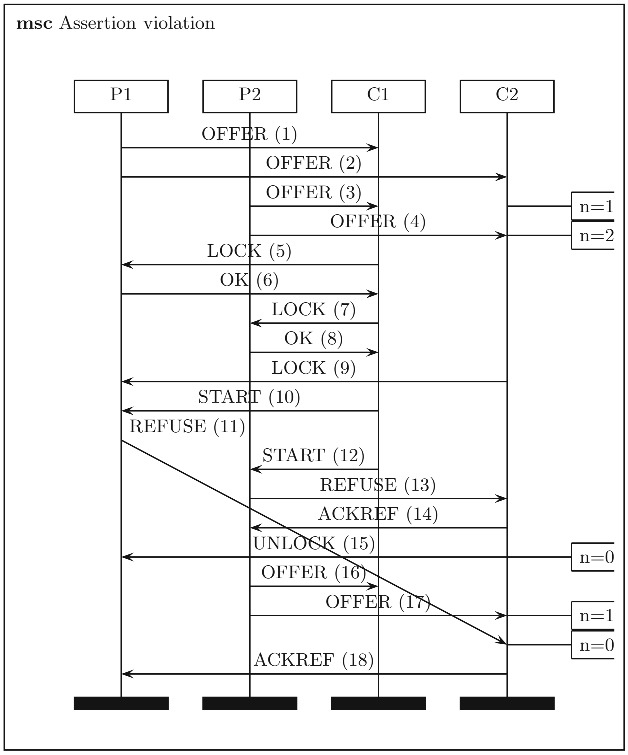


Fig. 5 A message sequence chart showing the counterexample for the α-core protocol

7 Conclusions

We studied the use of a methodology that performs a genetic programming search guided by model checking results. Our method was used for the following:

- 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
- improving 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. An important factor for convergence of the genetic programming algorithm is providing a fitness function with many levels. While the use of model checking instead of the traditional use of test cases provides a more reliable correctness criterion for the resulted code, the number of specification properties is typically small. We tackle this problem using deep model checking, which provides further meaningful levels for the benefit of the fitness function.

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 applies to code synthesis. We provided a co-evolution method for synthesizing 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, either for finding the error or a correct version of the algorithm, it can be fine-tuned through a convenient human-assisted process. An important strength of the work that is presented here is that it was implemented and applied to a complicated published protocol to find and correct an actual error.

Further research directions include adding more probabilistic levels for refining the fitness levels. One can also use statistical model checking instead or in addition to the deep model checking. In the current framework, the mutations are decoupled from the model checking results: the fitness affects the chance of a candidate to be mutated. Instead, causality analysis can be used to try and locate the source of the error. This can provide a probabilistic distribution not only on candidates for mutation but also on where within the tree representing the code we would like to apply the mutation.

References

1. Apt, K.R., Kozen, D.C.: 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; 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. Burke, D.S., Jong, K.A.D., Grefenstette, J.J., Ramsey, C.L., Wu, A.S.: Putting more genetics into genetic algorithms. Evolut. Comput. **6**(4), 387–410 (1998)
5. Burns, J.E., Lynch, N.A.: Bounds on shared memory for mutual exclusion. Inform. Comput. **107**(2), 171–184 (1993)
6. Chang, E.J.H., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (1979)
7. Chellapilla, K.: Evolving computer programs without subtree crossover. IEEE Trans. Evol. Comput. **1**(3), 209–216 (1997)
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
9. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State space reduction using partial order techniques. STTT **2**(3), 279–287 (1999)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, New York (2000)
11. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
12. Couvreur, J.M., Saheb, N., Sutre, G.: An optimal automata approach to LTL model checking of probabilistic systems. In: LPAR. pp. 361–375 (2003)
13. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569 (1965)
14. Dolev, D., Klawe, M.M., Rodeh, M.: An o(n log n) unidirectional distributed algorithm for extrema finding in a circle. J. Algorithms **3**(3), 245–260 (1982)
15. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL, pp. 85–94 (1995)
16. Fearnley, J., Peled, D., Schewe, S.: Synthesis of succinct systems. In: ATVA, pp. 208–222 (2012)
17. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, pp. 3–18 (1995)
18. Harman, M., Jones, B.F.: Software engineering using metaheuristic innovative algorithms: workshop report. Inform. Softw. Technol. **43**(14), 905–907 (2001)
19. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: a tool for automatic verification of probabilistic systems. In Tools and Algorithms for the Construction and Analysis of Systems TACAS **2006**, 441–444 (2006)
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
21. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, Cambridge (1992)
22. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11–13, 2007, Proceedings, pp. 114–124 (2007)
23. Katz, G., Peled, D.: Genetic programming and model checking: synthesizing new mutual exclusion algorithms. In: ATVA, vol. 5311 of LNCS, pp. 33–47 (2008)
24. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: TACAS, vol. 4963 of LNCS, pp. 141–156 (2008)
25. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: HVC, vol. 6405 of LNCS, pp. 117–132 (2009)
26. Katz, G., Peled, D.: Synthesizing solutions to the leader election problem using model checking and genetic programming. In: HVC (2009)

27. Katz, G., Peled, D.: Code mutation in verification and automatic code correction. In: TACAS, pp. 435–450 (2010)
28. Katz, G., Peled, D.: Mcgp: a software synthesis tool based on model checking and genetic programming. In: ATVA, pp. 359–364 (2010)
29. Kessels, J.L.W.: Arbitration without common modifiable variables. Acta Inf. **17**, 135–141 (1982)
30. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
31. Koza, J.R.: Human-competitive results produced by genetic programming. Genet. Program. Evol. Mach. **11**(3–4), 251–284 (2010)
32. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Design **19**(3), 291–314 (2001)
33. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16–19, 2001, Proceedings, pp. 389–398 (2001)
34. Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. IEEE Trans. Evol. Comput. **19**(1), 118–135 (2015)
35. Lehmann, D.J., Pnueli, A., Stavi, J.: Impartiality, justice and fairness: The ethics of concurrent termination. In: Automata, Languages and Programming, 8th Colloquium, Acre (Akko), Israel, Proceedings, pp. 264–277 (1981)
36. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, pp. 97–107 (1985)
37. Manna, Z., Pnueli, A.: How to cook a temporal proof system for your pet language. In: Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983, pp. 141–154 (1983)
38. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Trans. Program. Lang. Syst. **6**(1), 68–93 (1984)
39. Montana, D.J.: Strongly typed genetic programming. Evol. Comput. **3**(2), 199–230 (1995)
40. Niebert, P., Peled, D., Pnueli, A.: Discriminative model checking. In: CAV, vol. 5123 of LNCS, Springer, pp. 504–516 (2008)
41. Perez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurr. Pract. Exp. **16**(12), 1173–1206 (2004)
42. Peterson, F.: Economical solutions to the critical section problem in a distributed system. In: STOC: ACM Symposium on Theory of Computing (STOC) (1977)
43. Peterson, G.L.: An o(n log n) unidirectional algorithm for the circular extrema problem. ACM Trans. Program. Lang. Syst. **4**(4), 758–762 (1982)
44. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
45. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: FOCS, pp. 746–757 (1990)
46. Poli, R., Langdon, W.W.B., McPhee, N.F., Koza, J.R.: A field guide to genetic programming. Lulu. com (2008)
47. 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)
48. Schwefel, H.-P.P.: Evolution and Optimum Seeking: The Sixth Generation. Wiley, New York (1993)
49. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 133–192 (1990)
50. Tsay, Y.K.: Deriving a scalable algorithm for mutual exclusion. In: DISC, pp. 393–407 (1998)
51. Vardi, M.Y.: Probabilistic linear-time model checking: An overview of the automata-theoretic approach. In: Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, pp. 265–276 (1999)
52. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of IEEE Symposium on Logic in Computer Science, Boston. pp. 332–344 (1986)
53. Zuck, L.D., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems (a survey). Comp. Lang. Syst. Struct. **30**(3–4), 139–169 (2004)