Model Checking-Based Genetic Programming with an Application to Mutual Exclusion

Gal Katz and Doron Peled

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

Abstract. Two approaches for achieving correctness of code are verification and synthesis from specification. Evidently, it is easier to check a given program for correctness (although not a trivial task by itself) than to generate algorithmically correct-by-construction code. However, formal verification may give quite limited information about how to correct the code. Genetic programming repeatedly generates mutations of code, and then selects the mutations that remain for the next stage based on a fitness function, which assists in converging into a correct program. We use a model checking procedure to provide the fitness value in every stage. As an example, we generate algorithms for mutual exclusion, using this combination of genetic programming and model checking. The main challenge is to select a fitness function that will allow constructing correct solutions with minimal effort. We present our considerations behind the selection of a fitness function based not only on the classical outcome of model checking, i.e., the existence of an error trace, but on the complete graph constructed during the model checking process.

1 Introduction

The challenge in automatic programming is synthesizing programs automatically from their set of requirements. Genetic programming (GP) is an automatic program generation methodology; a population of programs is randomly created, and evolves by a biologically inspired process; the fitness of each program is usually calculated by running the program on some test cases, and evaluating its performance. Orthogonally, model checking [1] can be used to analyze a given program, verifying that it satisfies its specification, or providing a counterexample of that fact.

One of the possibilities of program synthesis is based on a brute force generation and analysis of the possible solutions. For example, Perrig and Song [2] successfully synthesized security protocols, while Bar-David and Taubenfeld [3] used a similar approach for the generation of mutual exclusion algorithms. Checking all the possibilities one by one guarantees that given enough time, a correct solution will be found. However, synthesis is an intractable problem in nature, which becomes quite quickly prohibitively expensive. This is also the case for synthesis algorithms that are not based on enumeration.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 141–156, 2008.

Genetic programming is often appropriate for solving sequential optimization related problems, where providing better solutions for some inputs than for others is acceptable. GP is less used traditionally for designing communication protocols, concurrent or reactive systems, where there is a specification on the behavior of the system over time, which must not be violated. Recently, Johnson [4] experimented with using the results of model checking for providing the fitness function for synthesizing a reactive system with GP. Using model checking for providing the fitness function has the advantage over testing that *all* the executions of the generated code are checked, rather than sampled. However, finding a fitness function with enough values to allow the gradual improvements typical to GP is not an easy task.

We provide a framework for genetic programming that is based on intensive analysis of a model checking procedure. The main challenge is to provide a fitness function that will improve the chances and speed of convergence towards a correct solution. For that, model checking is not only used to decide whether some properties of the specification hold or not (as in [4]), but the graph generated during model checking is analyzed, in order to extract more information. Finally, we provide experimental results of applying our framework to the mutual exclusion algorithm generation problem.

We experimented with several alternative analysis methods based on the result of state based model checking, such as probabilistic [5] and quantitative [6] model checking. In our experiments, these methods have not led to convergence toward solutions. Our method for calculating a fitness function is based on an analysis of the strongly connected components in the model checking graph. We show experimental results where this analysis leads to the generation of mutual exclusion algorithms. While the stochastic nature of GP eliminates the ability of finding all solutions, or proving that no such exists, it drastically reduces the average time and search steps required for finding a solution, compared to the enumeration methods. This is especially true when the solution programs are long, resulting in an extremely large search space.

In particular, our analysis gives a lower fitness weight to the case where an error can occur after a finite number of steps and a higher fitness weight to the case where infinite number of choices is needed in order for it to occur. The intuition behind this distinction is that the first kind of errors is more basic, whereas the second kind of errors is due to subtle scheduling.

The rest of the paper is organized as follows: Section 2 gives background on Genetic Programming and Model Checking. A description of the combined approach is given in Sect. 3. The mutual exclusion example is described in Sect. 4, followed by conclusions and future work in Sect. 5.

2 Background

2.1 Genetic Programming

Genetic Programming [7] is a method for automatic synthesis of computer programs by an evolutionary process. GP is derived from the field of Genetic

Algorithms (GA) [8], and uses a similar methodology. An initial population of candidate solutions is randomly generated and gradually improved by various biologically inspired operations. The main advantages of GP over GA are the explicit use of computer programs in the solution space, and the flexible structure of the solutions, usually represented as trees (although other representation are possible as well). The GP algorithm we use in this work goes through the following steps (described in details below):

1. Create initial population of candidate solutions.
2. Randomly choose μ candidate solutions.
3. Create λ new candidates by applying mutation (and optionally crossover) operations (as explained below) to the above μ candidates.
4. Calculate the fitness function for each of the new candidates.
5. 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.
6. Repeat steps 2-5 until a perfect candidate is found, or until the maximal permitted number of iterations is reached.

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 [9] is used, which means that every node has a type, and also enforces the type of its offspring. It also affects the genetic operations which must preserve the program typing rules.

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 [7] 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 1(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 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:

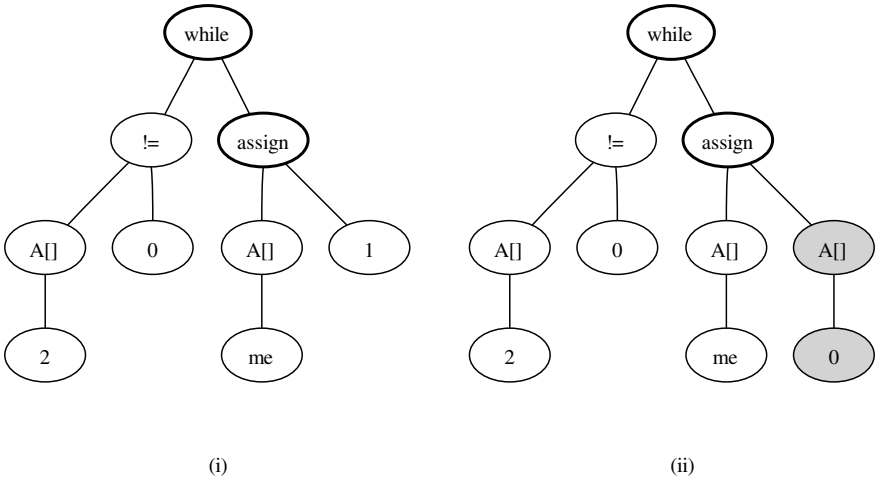

Fig. 1. (i) Randomly created program tree, (ii) the result of a replacement mutation

(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. 1(i), is replaced by the grayed subtree in (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 grayed part of Fig. 2 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. 1 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. 2 back into the original tree of Fig. 1(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.

Mutation type 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.

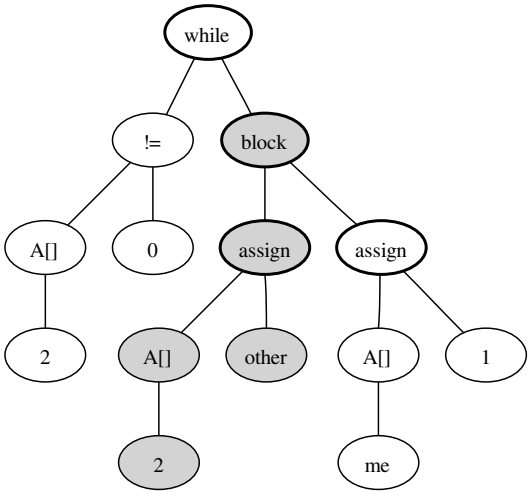

Fig. 2. Tree after insertion mutation

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 [10], 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.

Fitness is used by GP in order to choose which programs have a higher probability 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 (a training set) which suppose to represent all of the possible inputs. This can lead to programs which 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, as described later.

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.

We use another method, which is more similar to the Evolutionary Strategies [11] $\mu + \lambda$ style. In this method, genetic operations are applied to all of the μ programs, in order to produce a much larger set of λ offspring. The fitness proportional selection is then used in order to chooses μ programs from the set of parents and offspring that will replace the original μ parents.

2.2 Model Checking

A *finite ω-automaton* is a tuple $A = (\Sigma, S, S_0, \Delta, L, \Omega)$ where:

- Σ is a finite alphabet.
- S is a finite set of states.
- $S_0 \subseteq S$ is a set of initial states.
- $\Delta \subseteq S \times S$ is a transition relation.
- $L : S \rightarrow \Sigma$ is a labeling function of the states.
- Ω is the acceptance condition (defined later).

Note that in the automaton type defined here, the labels are on the states instead of on the arcs. Nevertheless, it is easy to transform one type of the automaton into another. An automaton can be treated as a directed graph $G_A = (V, E)$ by setting $V = S$ and $E = \Delta$.

A *run p of A* over an infinite word $w = w_0 w_1 w_2 ... \in \Sigma^\omega$ is the sequence $s = s_0 s_1 s_2 ... \in S^\omega$ such that: $s_0 \in S_0$, for all $i \geq 0$, $(s_i, s_{i+1}) \in \Delta$, and $w_i = L(s_i)$. We denote by *Inf(p)* the set of states appearing infinitely on the run p.

A *maximal strongly connected component (SCC)* is a maximal set of nodes $C \subseteq S$ such that for each $s, t \in C$ there is a path from s to t. An SCC is *non-trivial* if it has at least one edge. A graph can be decomposed into SCCs by a linear time algorithm, such as Tarjan's. The SCCs in a graph can be converted into a directed acyclic graph (DAG) where each SCC is represented by a simple node, and edges between these latter nodes represent paths from one corresponding SCC to another. A *bottom SCC (BSCC)* is an SCC that has no paths to other SCCs, i.e. it is associated with a leaf in the SCCs DAG defined in the previous paragraph. A *Büchi automaton* is an ω-automaton with the acceptance condition defined as a set of states $F \subseteq S$ where a run p over a word w is accepted if $\mathrm{Inf}(p) \cap F \neq \emptyset$.

A *Streett automaton* is an ω-automaton with the acceptance condition defined as a set of k ordered pairs (E_i, F_i), where $E_i, F_i \subseteq S, 1 \leq i \leq k$. A run p over a word w is accepted if $\mathrm{Inf}(p) \cap E_i \neq \emptyset \rightarrow \mathrm{Inf}(p) \cap F_i \neq \emptyset$ for all pairs. Every Streett automaton can be converted into a Büchi automaton accepting the same language, and vice versa [12]. Streett automata are closed under determinization [13], but this is not the case for Büchi automata where their nondeterministic version is more expressive than their deterministic one. The *language* of A, denoted by *L(A)* is defined as the set of all words accepted by A. A is *nonempty* if $L(A) \neq \emptyset$, i.e. A accepts at least one word. An *accepting SCC* is defined as an SCC C, for which there exists an accepting run p such that $\mathrm{Inf}(p) \subseteq C$. We say that this SCC is *not empty*.

Algorithm 1. Checking non-emptiness of a Streett automaton or SCC

```
IsNonempty(A, IsEntireAutomaton)
(1)   if IsEntireAutomaton
(2)       Decompose A into SCCs reachable from S_0
(3)   repeat
(4)       changed = FALSE
(5)       for each pair (E_i, F_i) ∈ Ω
(6)           for each nontrivial SCC C
(7)               if C ∩ E_i ≠ ∅ and C ∩ F_i = ∅
(8)                   remove from C all states {e | e ∈ E_i}
(9)                   rebuild the decomposition of C into SCCs
(10)                  changed = TRUE
(11)  until changed = FALSE
(12)  if A contains nontrivial components return TRUE
(13)  else return FALSE
```

Algorithm 1 can check in polynomial time the non-emptiness of an entire Streett automaton or a single SCC [14]. The second parameter should be set to TRUE of FALSE respectively.

Street automata are closed under intersection, i.e. if

$$A = (\Sigma, S_A, S_A^0, \Delta_A, L_A, \Omega_A) \quad , \quad B = (\Sigma, S_B, S_B^0, \Delta_B, L_B, \Omega_B)$$

are two Street automata using the same alphabet, then there exists a Street automaton C such that $L(C) = L(A) \cap L(B)$. C is constructed as follows:

$$C = (\Sigma, S_A \times S_B, S_A^0 \times S_B^0, \Delta_C, L_C, \Omega_C) \text{ where}$$

- $\Delta_C = \{(s, s'), (t, t') \mid (s, t) \in \Delta_A$ and $(s', t') \in \Delta_B$ and $L(s) = L(s')$ and $L(t) = L(t')\}$.
- For each state (s, s') of C, $L(s, s') = L(s)$.
- $\Omega_C = \{(E \times S_B, F \times S_B) \mid (E, F) \in \Omega_A\} \cup \{(S_A \times E, S_A \times F) \mid (E, F) \in \Omega_B\}$.

Büchi automata are closed under intersection as well.

A *finite-state system* M can be represented as an ω-automaton A_M by using the following settings:

- $\Sigma = 2^{AP}$ where AP denote a set of atomic propositions that may hold on the system states.
- S is a set of the system states, where a system state consists of the values of its variables, program counter, buffers, etc. at a specific time.
- S_0 is the initial state of the system.
- $\Delta \subseteq S \times S$ is a set of state pairs (s, r) such that r can be obtained from s by an atomic transition of M.
- $L : S \to 2^{AP}$ assigns to each state a set of propositions that hold in that state.
- Ω is set to accept all runs. This can be done on Büchi automata by setting $\Omega = S$, and on Street automata by setting $\Omega = \emptyset$. Another option is to set Ω to some fairness conditions that will allow only fair runs of the system.

In order to define the specification properties we use *Linear Temporal Logic (LTL)*. LTL is a modal logic having the following syntax:

$$\varphi ::= p \mid (\varphi) \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi$$

where $p \in AP$, 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. For a suffix ξ_k 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 by using the following identities:

$$true = p \vee \neg p,\ \varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi),\ \varphi \rightarrow \psi = \neg\varphi \vee \psi,\ \Diamond\varphi = true \mathcal{U} \varphi,\ \Box\varphi = \neg\Diamond\neg\varphi$$

We say that $\xi \models \varphi$ if $\xi_0 \models \varphi$, i.e. the complete sequence ξ satisfies the LTL formula φ. For a finite-state system M, $M \models \varphi$ if for every fair run ξ on M, $\xi \models \varphi$.

Specification properties defined as LTL formulas can be converted into ω-automata. For nondeterministic Büchi automata, this can be done by a direct construction algorithm in time exponential to the formula size [15]. For deterministic Streett automata, the process may involve a construction of a Büchi automaton, and a determinization process that ends with a deterministic Streett automaton [13]. The translation may result in a doubly exponential blowup [16].

A standard Model Checking procedure checks whether a system M satisfies a specification φ [17]. This is done by building the automata A_M and A_φ of M and φ respectively, and checking whether $L(M) \subseteq L(\varphi)$. Since

$$L(M) \subseteq L(\varphi) \leftrightarrow L(M) \cap \overline{L(\varphi)} = \emptyset \leftrightarrow L(M) \cap L(\neg\varphi) = \emptyset$$

it is possible to use the negation of φ, build $A_{\neg\varphi}$, and check whether the language $L(M) \cap L(\neg\varphi)$ is empty. The Model Checking process is usually performed using Büchi automata, since the conversion from LTL, and the language emptiness checking are easier with these automata. However, for our purposes we will use Streett automata, as explained in the next section.

3 Combining GP and Model Checking

The standard model checking procedure gives a yes/no answer to the satisfaction of a system M by a specification formula φ (in addition to a counterexample, when φ is not satisfied). As we wish to base the fitness function on the model checking results, using a function with just two values may give a limited ability to gradually improve the programs ([4], for instance). Therefore, we try to quantify the level of satisfaction, by comparing the system computations on which

the specification is satisfied, with those on which it is not. This can be done by checking the intersection graph of the system and the specification, for accepting and non-accepting paths.

However, when using a nondeterministic (such as Büchi) automaton, finding a non-accepting path of a computation does not necessarily mean that the computation itself is not accepted, since there might be another accepting path for the same computation. For this reason, we use deterministic Streett automata, on which each computation is represented by a single path. While this choice has an additional cost on the size of the specification automata, and on the model checking procedure performance, the symmetry between accepting and non-accepting paths is crucial for our analysis.

Having a system M and a specification formula φ, we perform the enhanced model checking procedure showed below.

Algorithm 2. Enhanced model checking

EnhancedMC(M, φ)

(1) univ := FALSE
(2) Construct Streett automata A_M and A_φ for M and φ respectively.
(3) Create the intersection automaton $A_{pos} = A_M \cap A_\varphi$.
(4) Decompose A_{pos} into SCCs reachable from its initial states
(5) For each SCC $C \in A_{pos}$:
(6) acc(C) := IsNonempty(C, FALSE)
(7) if for each SCC C, acc(C) = TRUE
(8) Construct Streett automaton $A_{\neg\varphi}$ for the negation of φ
(9) Create the intersection automaton $A_{neg} = A_M \cap A_{\neg\varphi}$
(10) univ := ¬ IsNonempty(A_{neg}, TRUE)

The algorithm first checks the non-emptiness of every SCC of the graph and stores the result in the SCC's *acc* variable (lines (1) - (6)). In the case that all of the SCCs are accepting, an additional step is performed in order to check the automaton for universality (stored in the *univ* variable) (lines (7) - (10)).

The results of the algorithm are used for setting the value of the fitness function, as detailed in Table 1. In order to assign a fitness level, we assume that the choice of program transitions is made by a hostile scheduler (or environment) that tries to cause the violation of the checked property. The amount of choices the scheduler has to make during an execution determines the fitness level (for a related approach for analyzing counterexamples, see [18]). The lowest fitness level of 0 is given when the checked property is never satisfied (thus, no hostile scheduling choices are needed to violate the property). Level 1 is assigned when the graph contains a non-accepting bottom SCC. In this case, a finite number of choices can lead to that BSCC, from which the failure is unavoidable. A higher level of 2 is assigned where all BSCCs are accepting, hence, a violation can be caused only by an infinite scheduler choices that will consistently skip the accepting paths. The highest level of 3 indicates that the property is always satisfied (thus, even a hostile scheduler cannot cause a violation of the property).

Table 1. Fitness Levels

Fitness level	Condition	Description	Hostile scheduler choices	Score
0	For all SCCs, acc(C)=FALSE	The property is not satisfied on any execution	None	0
1	At least one SCC with acc(C)=TRUE. At least one BSCC with acc(C)=FALSE	The program can reach a state from which the violation of the property is unavoidable	Finite	70
2	For all BSCCs, acc(C)=TRUE, univ=FALSE	Property violation is always avoidable. It can be violated only by an infinite scheduler choices	Infinite	80
3	univ=TRUE	The property is always satisfied	Impossible	100

The scores assigned to each fitness level are intended to encourage the gradual evolution of programs among the various levels. The specific score values were chosen so that two or more partially satisfied properties will have a higher score than a single fully satisfied property. This gives a better chance to a more parallel and smooth evolution.

A class of properties that need a special treatment are properties of the form $\Box(P \rightarrow \Diamond Q)$. These properties can be vacuously satisfied by a run on which P never happens (see [19]). When comparing the accepting and non-accepting paths of the program, we wish to ignore those vacuous runs, and concentrate only on runs where P eventually happens.

Consider for instance the property $\Box$ (p *in* `Pre` $\rightarrow \Diamond$ (p *in* `CS`)) (defined at Sect. 4), requiring that a process trying to enter the critical section will eventually enter it. We do not wish to give extra score for program runs that stay infinitely on the `Non Critical Section`. Neither can we just add "$\wedge \Diamond$ (p *in* `Pre`)" to the property, since this will treat the above runs as violating the property. Instead, we wish to evaluate only runs where the process enters the `Pre Protocol` section, and ignore other runs and their related SCCs.

In order to achieve that, a prior step is added before line (3) of algorithm 2 . In this step A_M is intersected with the automaton of the property $\Diamond P$, and the intersection is used instead of A_M. In the special case when the intersection is empty, all runs are vacuously satisfied, and a predefined fitness level (usually 0) is assigned to the program, without running the rest of the algorithm.

Another reason for restricting program runs is related to fairness. In this work we use weak process fairness by adding Streett conditions to the program automaton (strong fairness can be applied as well with Streett conditions).

On cases when not all of the program runs are accepted by the program automaton A_M (such as the two cases above), algorithm 2 needs a refinement. Prior to the non-emptiness check at lines (5) - (6), we run a similar check, but only with the Streett conditions derived from A_M. Empty SCCs found at this stage are considered *not-relevant*, and are not used by further steps of the algorithm and the scoring analysis. After this stage, the standard check of lines

(5) - (6) is performed on the *relevant* SCCs, including all of the Streett conditions (derived from both the program and the specification).

Deadlocks in programs are considered a fundamental failure, and are usually tested at early stages along with other safety properties. In our case, however, programs can evolve to a certain fitness score, even if they contain deadlocks. This is a result of the fact that as shown on the previous section regarding vacuity, some liveness properties can be relevant only in some parts of the state space, and may not be affected by other parts with deadlocks.

For this reason, we do not check for deadlocks explicitly. Instead, when the graph of a specific liveness property contains a deadlock, it will be detected by the above algorithm as a non-accepting BSCC, and will be assigned fitness level 1. In order to distinguish this case from the usual cases of level 1, we slightly decrease the score given to the property if the BSCC contains only a single node (which is the case when a deadlock occurs). While programs may simply raise their score by adding lines that turn the deadlock into a livelock, adding these lines increases the probability of a mutation in the program lines related to the deadlock, which may help in eliminating it. This was the case on the run described later.

The procedure above is performed for each of the properties included in the specification, resulting in multiple fitness functions. These values have to be merged into a single fitness score. This can be done by summing up the values of all functions (as done in [4]). However, our experience shows that some properties may depend on other more basic properties, and may become trivially satisfied where those basic properties are not satisfied. In order to prevent this biased fitness calculation, we assign a level to each property, starting from 1 for the most basic properties, and checking properties by the order of levels. If at level i, not all of the properties are fully satisfied, the checking process is stopped, and all properties at levels greater than i receive a score of 0. This also saves time by not checking unneeded properties. The total fitness value is then divided by the total number of properties (including those that were not checked) in order to get an average fitness value.

GP programs tend to grow in size over time until they reach the maximum allowed tree depth. This phenomena [20,21], known as "bloating" is caused by non-relevant portions of code which are called "introns" after the analogous biological term for non-relevant DNA. While there is evidence that introns may help the evolutionary process [10], they may hurt performance and even prevent the convergence, especially in areas such as protocols and concurrent programs, where programs has to be small and accurate. One of the ways of preventing introns from appearing is giving a penalty to large programs by decreasing their fitness.

We use parsimony as a secondary fitness measure by subtracting from the total score the number of program nodes multiplied by a small constant (0.1). The constant is chosen to be small enough, so that programs with various length can evolve, and the distinction between fitness levels is preserved, but still large enough to affect programs at the same level to have a reduced size. Note that this means programs cannot get a perfect score of 100, but only get closer to

it. Instead, we mark programs as perfect when all properties are fully satisfied. Shorter programs can be created as a result of the genetic operations, as well as by removing dead-code detected by the model checking process.

4 Example - The Mutual Exclusion Problem

As an example, we use our method in order to automatically generate solutions to several variants of the Mutual Exclusion Problem. In this problem, first described and solved by Dijkstra [22], 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 only two processes. The problem is modeled using the following program parts that are executed in an infinite loop:

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

The `Non Critical Section` part 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` part. From the `Critical Section` part, a process always has to move into the `Post Protocol` part. The `Non Critical Section` and `Critical Section` parts are fixed, while our goal is to automatically generate code for the `Pre Protocol` and `Post Protocol` parts, such that the entire program will fully satisfy the problem's specification.

We use a restricted high level language based on the C language. Each process has access to its id (0 or 1) by the `me` literal, and to the other process' id by the `other` literal. The processes can use an array of shared bits with a size depended on the exact variant of the problem we wish to solve. 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 2 describes the properties which define the problem specification. The four program parts are denoted by `NonCS`, `Pre`, `CS` and `Post` respectively. Property 1 is the basic safety property requiring the mutual exclusion. Properties displayed

Table 2. Mutual Exclusion Specification

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

in pairs are symmetrically defined for the two processes. Properties 2 and 3 guarantee that the processes are not hung in the `Post Protocol` part. 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 showed in Table 3.

Table 3. 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 [23]
2	3	Simple	Starvation Freedom	1,2,3,4,5,7,8	Dekker [22]
3	3	Complex	Starvation Freedom	1,2,3,4,5,7,8	Peterson [24]

4.1 Experimental Results

We used a specially designed model check and GP engine which implements the methods described earlier. Three different configurations where used, in order to search for solutions to the variants described in Table 3. 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. At each iteration, 5 programs were randomly selected, bred, and replaced using mutation and crossover operations, as described on Sect. 2.1. 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 on Table 4.

Table 4. Test results

Variant No.	Successful runs (%)	Avg. run durtaion (sec)	Avg. no. of tested programs per run
1	40	128	156600
2	6	397	282300
3	7	363	271950

Test 1. At the first test, we tried to find a deadlock-free algorithm solving the mutual exclusion problem. The programming language in this case was set to allow the use of two shared bits, and only simple conditions. Followed is an

analysis of one of the successful runs. 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 the parsimony pressure caused by the secondary fitness measure, 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 [23].

```
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 [22] 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. 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 a second line to the *post protocol* section (which only slightly decreased its fitness level due to the parsimony measure). 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
```

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

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

```
A[me] = 0
```

(c) [94.34]

```
A[2] = me
A[me] = 0
```

(d) [96.70]

(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 [24].

```
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]

5 Conclusions and Future Work

One of the main features of our scoring system is that a failure of a property to hold in a program is considered to be more severe where there is a finite prefix from which the property would not be fixable. On the other hand, when failure involves infinitely many choices that would steer away the execution from a correct one, the failure is considered to be "weaker". This provides an interesting dichotomy in analysis of correctness, and can be further refined (e.g., by recognizing the case when *all* of the executions are of the same severe failure, rather than that there exists at least one such).

Experimentally, this distinction turned out to provide good scoring results with a high probability of convergence. On the other hand, the disadvantage of using GP for generating solutions for problems like mutual exclusion is that because GP involves probabilistic decisions, one does not know when and whether the search space is exhausted.

Further work includes refining the scoring system, and making experiments with other concurrent or distributed algorithms.

References

1. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
2. Perrig, A., Song, D.: Looking for diamonds in the desert - extending automatic protocol generation to three-party authentication and key agreement protocols. In: CSFW, pp. 64–76 (2000)
3. Bar-David, Y., Taubenfeld, G.: Automatic discovery of mutual exclusion algorithms. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 136–150. Springer, Heidelberg (2003)

4. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 114–124. Springer, Heidelberg (2007)
5. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
6. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
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. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation 3(2), 199–230 (1995)
10. 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, San Francisco (2001)
11. Schwefel, H.P.P.: Evolution and Optimum Seeking: The Sixth Generation. John Wiley & Sons, Inc. New York (1993)
12. Safra, S., Vardi, M.Y.: On ω automata and temporal logic. In: 21th Annual Symp. on Theory of Computing, pp. 127–137 (1989)
13. Safra, S.: Complexity of automata on infinite objects. PhD thesis, Rehovot, Israel (1989)
14. Emerson, E.A.: Automata, tableaux and temporal logics. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 79–88. Springer, Heidelberg (1985)
15. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)
16. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design 19(3), 291–314 (2001)
17. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. IEEE Symp. on Logic in Computer Science, Boston, July 1986, pp. 332–344 (1986)
18. Jin, H., Ravi, K., Somenzi, F.: Fate and free will in error traces. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 445–459. Springer, Heidelberg (2002)
19. Beatty, D.L., Bryant, R.E.: Formally verifying a microprocessor using a simulation methodology. In: DAC, pp. 596–602 (1994)
20. Angeline, P.J.: Genetic programming and emergent intelligence. In: Advances in Genetic Programming, pp. 75–98. MIT Press, Cambridge (1994)
21. Tackett, W.A.: Recombination, selection, and the genetic construction of computer programs. PhD thesis, Los Angeles, CA, USA (1994)
22. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM 8(9), 569 (1965)
23. Burns, J.E., Lynch, N.A.: Bounds on shared memory for mutual exclusion. Information and Computation 107(2), 171–184 (1993)
24. Peterson, G.L., Fischer, M.J.: Economical solutions to the critical section problem in a distributed system. In: ACM Symposium on Theory of Computing (STOC), pp. 91–97 (1977)