Synthesis of Parametric Programs using Genetic Programming and Model Checking

Gal Katz Doron Peled
Department of Computer Science, Bar Ilan University
Ramat Gan 52900, Israel

Formal methods apply algorithms based on mathematical principles to enhance the reliability of systems. It would only be natural to try to progress from verification, model checking or testing a system against its formal specification into constructing it automatically. Classical algorithmic synthesis theory provides interesting algorithms but also alarming high complexity and undecidability results. The use of genetic programming, in combination of model checking and testing, provides a powerful heuristic to synthesize programs. The method is not completely automatic, as it is fine tuned by a user that sets up the specification and parameters. It also does not guarantee to always succeed and converge towards a solution that satisfies all the required properties. However, we applied it successfully on quite nontrivial examples and managed to find solutions to hard programming challenges, as well as to improve and to correct code. We describe here several versions of our method for synthesizing sequential and concurrent systems.

1 Introduction

Formal methods [16] assist software and hardware developers in enhancing the reliability of systems. They provide methods and tools to search for design and programming errors. While these methods are effective in the software development process, they also suffer from severe limitations: testing is not exhaustive, formal verification is extremely tedious and model checking is limited to particular domains (usually, finite state systems) and suffers from high complexity, where memory and time requirements are sometimes prohibitively high.

A natural progress from formal methods are algorithms for automatically converting the formal specification into code or a description of hardware. Such algorithms would create correct-by-design code or piece of hardware. However, high complexity [19] and even undecidability [20] appear in some main classical automatic synthesis problems.

The approach presented here is quite different from algorithmic synthesis. We perform a generate-and-check kind of synthesis and use model checking or SAT solving to evaluate the generated candidates. An extreme approach would be to enumerate the possible programs (say, up to a certain size) and use model checking to find the correct solution(s). This was applied in Taubenfeld [3] to find mutual exclusion algorithms. Our synthesis method is based on *genetic programming*. It allows us to generate multiple candidate solutions at random and to mutate them, as a stochastic process. We employ enhanced model checking (model checking that does not only produce an affirmation to the checked properties or a counterexample, but distinguishes also some finer level of correctness) to provide *fitness* levels that are used to direct the search towards solutions that satisfy the given specification. Our synthesis method can be seen as a heuristic search in the space of syntactically fitting programs. It is not completely automatic, in the sense that the user can refine the specification and change the way the fitness is evaluated when the formal properties are satisfied. Our method is not guaranteed to terminate with a correct solution; we might give up after some time and can restart the search from a new random seed or with a refinement of the way the method assigns fitness.

L. Clemente and L. Holik (Eds.): 15th International Workshop on
Verification of Infinite-State Systems (INFINITY'13)
EPTCS 140, 2014, pp. 70–84, doi:10.4204/EPTCS.140.5

Although this marriage between genetic programming and model checking is quite promising, it suffers from some limitations of model checking. First, model checking is primarily designed for finite state systems. Although some extensions of it exist (e.g., to programs with a single stack), model checking does not work in general for parametric systems. Unfortunately, most systems that we would like to synthesize are parametric in nature: almost every abstract algorithm on data structures, be it queue, tree, graph, is parametric, where the size of the structure, is not fixed. It is easy to demonstrate model checking on a sorting program with a fixed vector of numbers and some fixed initial assignment of values. However, when the length of the vector is parametric, and we need to prove correctness with respect to arbitrary set of values, existing model checking techniques often fail.

For this reason, we use model checking in our approach for synthesizing parametric systems not as a comprehensive method for finding correctness, but as a generalized testing tool, which can make exhaustive checks for fixed parameters. Under this setting, we accept candidate programs when there is ample evidence that they are correct, specifically, when they passed enough checks, rather than when we establish comprehensive correctness.

Our genetic programming synthesis approach allows us not only to generate code that satisfies a given temporal specification but also to improve and correct code. We can start with an existing solution for a specification, and use the genetic process to improve it. We can also start with some flawed version of the code and use our method to correct it.

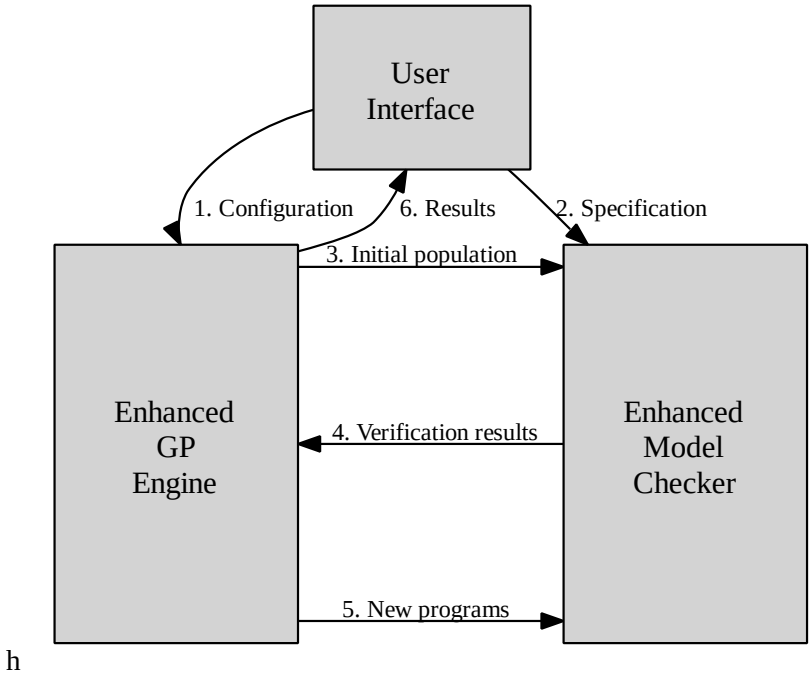
2 Genetic Programming Based on Model Checking

We present in [8, 9, 10, 11, 12] a framework combining genetic programming and model checking, which allows to automatically synthesize code for given problems. The framework we suggest is depicted in Figure 1.

- The *formal specification* of the problem, as well as the required architecture and constraints on the structure of the desired solutions is provided by the user. This may also include some initial versions of the desired code that either need correction or improvement.
- An *enhanced GP engine* that generates random programs and then evolves them using mutation operations that allow to change the code randomly.
- A *verifier* that analyzes the generated programs, and provides useful information about their correctness. This can be a model checker, often enhanced to provide more information than yes/no (and counterexample), or a SAT solver.

The synthesis process goes through the following steps:

1. The user feeds the GP engine with the desired architecture and a set of constraints regarding the programs that are allowed to be generated. This includes:
   (a) a set of functions literals and instructions used as building blocks for the generated programs,
   (b) the number of concurrent processes, the methods of communication between processes (in case of concurrent programs),
   (c) limitations on the size and structure of the generated programs, and the maximal number of permitted iterations.
   (d) The user may also provide some initial versions of the code that may be either incorrect or suboptimal. The genetic process can exploit these versions to evolve into better (correct or optimized) code.



h

Figure 1: The Suggested Framework

2. The user provides a formal specification for the problem. This consists, in our case, of a set of linear temporal logic properties, as well as additional quantitative requirements on the program behavior.
3. The GP engine randomly generates an *initial population* of candidate programs based on the provided building blocks and constraints.
4. The verifier analyzes the behavior of the generated candidates against the specification properties, and provides *fitness measures* based on the amount of satisfaction.
5. The GP engine creates new programs by applying the genetic operations of *mutation*, which performs small random changes to the code, and *crossover*, which glues together parts of different candidate solutions. Steps 4 and 5 are repeated until either a perfect program is found (fully satisfying the specification), or until the maximal number of iterations is reached.
6. The results are sent back to the user. This includes programs that satisfy 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:

- Randomly select μ candidate programs.
- Create λ new candidates by applying mutation (and optionally crossover) operations (as explained below) to the above μ candidates. We now have $\mu + \lambda$ candidates.
- Calculate the fitness function for each of the new candidates based on "enhanced model checking".
- Based on the calculated fitness, choose new μ candidates from the set of $\mu + \lambda$ candidates. Candidates with higher fitness values are selected with a higher probability than others. Replace the originally selected μ with the ones selected at this step.

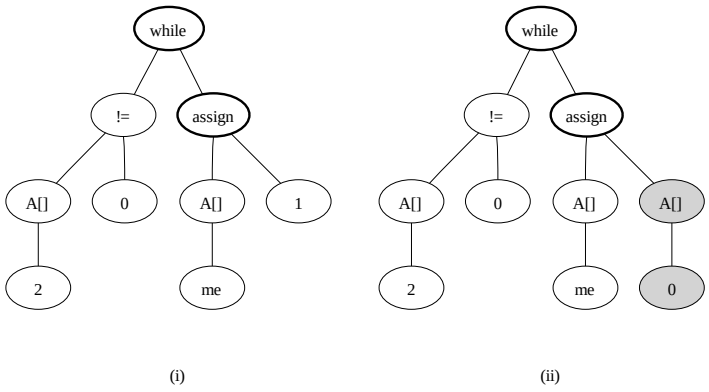


Figure 2: (i) Randomly created program tree, (ii) the result of a replacement mutation

We represent programs as trees, where an instruction or an expression is represented by a single node, having its parameters as its offspring. Terminal nodes represent constants. Examples of the instructions we use are *assignment, while , if* and *block.* The latter construct is a sequential composition of a pair of instructions.

At the first step, an initial population of candidate programs is generated. Each program is generated recursively, starting from the root, 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. Figure 2(i) shows an example of a randomly created tree that represents the following program:

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

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

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

Mutation of type (a) can replace either a single terminal or an entire subtree. For example, the terminal "1" in the tree of Figure 2(i), is replaced by the grayed subtree in 2(ii), changing the assignment instruction into `A[me] = A[0]`. Mutations of type (b) can extend programs in several ways, depending on the new parent node type. In case a "block" type is chosen, a new instruction(s) will be inserted before or after the mutation node. For instance, the grayed part of Figure 3 represents a second assignment instruction inserted into the original program. Similarly, choosing a parent node of type "while" will have the effect of wrapping the mutation node with a while loop. The type of mutation applied to candidate programs is randomly selected. All mutations must of course produce legal code. This affects the possible mutation type for the chosen node, and the type of new generated nodes.

Another operation that is frequently used in genetic programming is *crossover*. The crossover operation creates new candidates by merging building blocks of two existing programs. The crossover steps are:

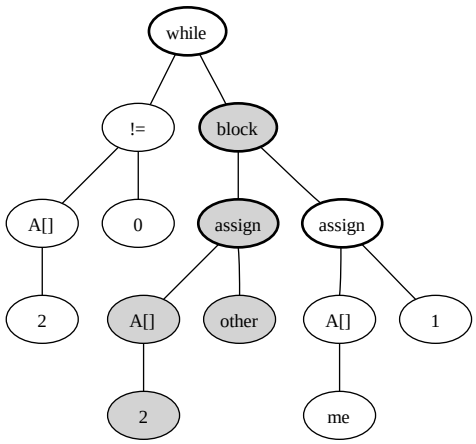


Figure 3: Tree after insertion mutation

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

While traditional GP is heavily based on crossover, it is quite a controversial operation (see [2], for example). Crossover is not 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 candidate. Traditionally, the fitness function is calculated by running the program on some set of inputs (a training set), which represent the possible 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 satisfiablity of the specification (thus yielding a two-valued fitness function), our model checking algorithm generates a smoother function by providing several levels of correctness. Often, we have the following four levels of correctness, per each linear temporal logic property:

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

We provided several methods for generating the various fitness levels:

- Using Streett Automata, and a strongly-connected component analysis of the program graph [9].
- Enhanced model checking logic and algorithm [8, 15].
- Probabilistic model checking.

There are several other considerations in setting up the calculation of the fitness. First, priority between the properties is used to suppress assigning fitness value due to the satisfaction of a liveness property (e.g., “when a process wants to enter its critical section, it would eventually be able to do

so") when the safety property does not hold (e.g., "the two processes cannot enter their critical sections simultaneously"). Another consideration is to prevent needless growth of the program by useless code. To alleviate this, we use some negative fitness value related to the program's length. This entails that a solution that satisfies all the specification is accepted even if it does not have perfect fitness value (due to length).

3 Example: Mutual Exclusion Algorithms

As an example, we used 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 [5], 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
```

These parts are fixed, and, together with the number of processes involved (two) and the number of variables allowed, consist of the architecture provided to our genetic programming tool, together with the temporal specification.

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 1 describes the properties that 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

Table 1: 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

mutual exclusion. Properties displayed in pairs are symmetrically defined for the two processes. Prop-

erties 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 6 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 2.

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 [4]
2	3	Simple	Starvation Freedom	1,2,3,4,5,7,8	Dekker [5]
3	3	Complex	Starvation Freedom	1,2,3,4,5,7,8	Peterson [18]

Three different configurations where used, in order 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. At each iteration, 5 programs were randomly selected, bred, and replaced using mutation. The values $\mu = 5, \lambda = 150$ where chosen.

In addition to the temporal specification of mutual exclusion, our configuration allows three shared bits. The famous Dekker's algorithm [5] uses two bits to announce that they want to enter the critical section, and the third bit is used to set turns between the two processes. Many runs initially converged into deadlock-free algorithms using only two bits. Those algorithms have executions in which one of the processes starve, hence only partially satisfying properties 7 or 8. Program (a) shows one of those algorithms, which later evolved into program (b). 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 measure). A replacement mutation then changed the inner while loop condition, leading to a perfect solution similar to 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
```

(a) [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
```

(b) [96.70]

Inspired by algorithms developed by Tsay [21] and by Kessels [13], our next goal was to start from an existing algorithm, and by adding more constraints and building blocks, try to evolve into more advanced algorithms.

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

As a result, the tool found two algorithms 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 with 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 uses 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 [21] used a similar refinement, but his algorithm needs an additional if statement, which is not used in our algorithm.

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

Next, we aimed at finding more advanced algorithms satisfying additional properties. The configuration was extended into four shared bits and two private bits (one for each process). The first requirement was that each process can change only its 2 local bits, but can read all of the 4 shared bits 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
```

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

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

4 Synthesizing Parametric Programs

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

First, we assume that a solution that is checked for a large number of instances/parameters is acceptable. This is not a guarantee of correctness, but under the prohibitive undecidability of model checking for parametric programs, at least we have a strong evidence that the solution may generalize to an arbitrary configuration. In fact, there are several works on particular cases where one can calculate the parameter size that guarantees that if all the smaller instances are correct, then any instance is correct [6]. 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 parameters from failed checked cases and keep them as counterexamples. When suggesting a new solution, we check it against the collected counterexamples. We can view this process as a genetic search for both correct programs and counterexamples. The fitness is different, of course, for both tasks: a program gets higher fitness by being close to satisfying the full set of properties, while a counterexample is obtaining a high fitness if it fails the program.

In this sense, the model checking of a particular set of instances can be considered as a generalized *testing* for these values: each set of instances of the parameters provides a single finite state system that is itself comprehensively tested using model checking. This idea can be also used, independently, for model checking parametric systems. For example, consider a concurrent sorting program consisting of a parametric array of processes, each containing some initial value. Adjacent processes may exchange values during the algorithm. For any particular size and set of values, the model checking provides automatic and exhaustive test for a particular set of values, but the check is not exhaustive for all the array sizes or array values, but rather samples them.

In the classical *leader election in a ring* problem, the processes initially have their own values that they can transfer around, with the goal of finding a process that has the highest value. Then, the parameters include the size of the ring, and the initial assignment of values to processes. While we can check solutions up to a certain size, and in addition, check all possible initial values, the time and state explosion is huge. Instead, we can then store each set of instances of the parameters that failed for some candidate solution, and, when checking a new candidate solution, check it against the failed instances. A solution for the leader election, albeit not the most optimal one, was obtained using our genetic programming methods [10].

5 Correcting Erroneous Program

Our method is not limited to finding new program that satisfy the given specification. In fact, we can start with the code of an existing program instead of a completely random population and try to improve or correct it. In order to *improve code*, our fitness measure may include some quantitative evaluation; then the initial program may be found inferior to some later generated candidates. If the program we start with is *erroneous*, then it would not get a very high fitness value by failing to satisfy some of the properties.

In [11] we approached the ambitious problem of correcting a known protocol for obtaining inter-process interaction called α-core [17]. The algorithm allows multiparty synchronization of several processes. It needs to function in a system that allows nondeterministic choices, which makes it challenging, as processes that may consider one possible interaction may also decide to be engaged in another interaction. The algorithm uses asynchronous message passing in order to enforce selection of the interactions by the involved processes. This nontrivial algorithm, which is used in practice for distributed systems, contains an error.

The protocol is quite big, involving sending different messages between the controlled processes, and the controlling 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, request for commit from the interaction manager processes, as well as announcement that the interaction can start, or is canceled due to the departure of at least one participant. The state space of such a protocol is obviously high. In addition, the protocol can run on any number of processes, each process with arbitrary number of choices to be involves in interactions, and each interaction includes any number of processes.

Recall that model checking of parametric programs is undecidable in general [1]. In fact, we use our genetic programming approach first to find the error, and then to correct it. We use two important ideas:

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

Specifically, the architecture for the candidate programs is also represented as code (or, equivalently, a syntactic tree) for spanning processes and their interactions, which can be subjected to genetic mutations. The fitness function directs the search for a program that may falsify the specification for the given erroneous program. After finding a "bad" architecture for a program, one that causes the program to fail its specification, our next goal is to reverse the genetic programming direction, and try to automatically correct the program, where a "correct" program at this step, is one that has passed model checking against the architecture. Yet, correcting the program for the first found wrong architecture only, does not guarantee its correctness under different architectures, hence more architectures that fail candidate solutions are collected. Note that we use for the co-evolution two separate fitness functions: one for searching for "bad" architectures, and one for searching for a correct solution.

In Figure 4 we show the architecture that was found to produce the error in the original α-core algorithm. A message sequence chart in Figure 5 demonstrate the found bad scenario. The correction consisted of changing the line of code

$$\mathsf{if\ n{>}0\ then\ n{:=}n\text{-}1}$$

into

$$\mathsf{if\ sender{\in}shared\ then\ n{:=}n\text{-}1}$$

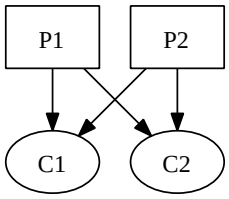


Figure 4: An architecture violating the assertion

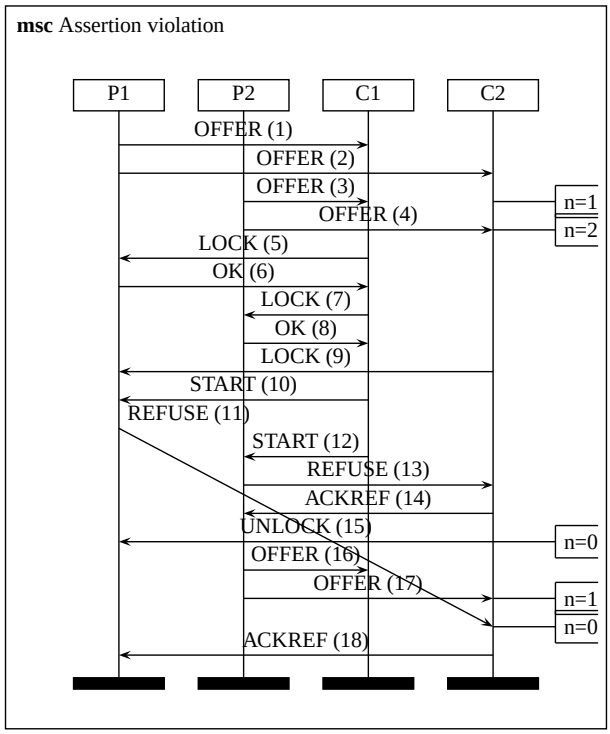


Figure 5: A Message Sequence Chart showing the counterexample for the α-core protocol

6 A Tool for Genetic Programming Based on Model Checking

We constructed a tool, MCGP [12], that implements our ideas about model checking based genetic programming. Depending on these settings, the tool can be used for several purposes:

- Setting all parts as *static* will cause the tool to just run the enhanced model checking algorithm on the user-defined program, and provide its detailed results.
- Setting the *init* process as *static* and all or some of the other processes as *dynamic* will order the tool to synthesize code according to the specified architecture. This can be used for synthesizing programs from scratch, synthesizing only some missing parts of a given partial program, or trying to correct or improve a complete given program.
- Setting the *init* process as *dynamic* and all other processes as static, is used when trying to falsify a given parametric program by searching for a configuration that violates its specification (see [11]).

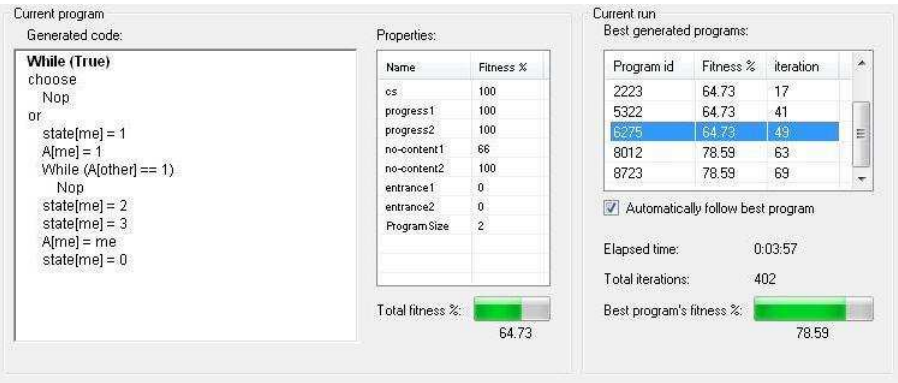


Figure 6: MCGP screen shot during synthesis of a mutual exclusion algorithm

- Setting both the *init* and the program processes as *dynamic* is used for synthesizing parametric programs, where the tool alternatively evolves various programs and configurations under which the programs have to be satisfied.

7 Replacing Model Checking by SAT Solving

Our approach can use automated deductive techniques instead of model checking in order to prove the correctness of the synthesized algorithms. However, it requires the verification procedures to be both fully automatic, and quite fast, so it can be repeated a large number of times. Obviously, most theorem provers that require some user interaction during the proof process cannot be used along with our framework. Furthermore, verification in this case is in general undecidable, so fast and complete procedure is not achievable.

Recently, there is a growing use of *SAT* and *SMT solvers* for verification purposes. These tools can function as high performance, and light-weight theorem provers for a broad range of decidable theories over first order logic, such as those of equalities with uninterpreted functions, bit-vectors and arrays. If we restrict our domain and structure of the synthesized programs as shown later, we can successfully (and quite quickly) verify their correctness for all inputs in the related domains. For some theories, variables are theoretically unbounded, while for other theories, we must limit their width.

Our work is inspired by [7], in which a set of short but ingenious and nontrivial programs, selected from the book *Hacker's Delight* [22], were successfully synthesized. These programs are loop-free, and use expressions over the decidable theory of bit-vectors. Thus, they can be easily converted into first order formulas which can then be verified by an SMT solver. The theory of bit-vectors is decidable only when limiting the width of its related variables. From a practical point of view, this does not impose a real constraint, since we can easily check the correctness of programs even with 128-bit variables. Unlike [7], we do not use the SMT solver for the direct synthesis of programs. Instead, we generate and evolve programs using our GP engine, and integrate the SMT solver into our verification component.

We modify our original framework in order to adopt it to the synthesis of sequential programs. In this new framework, the configuration provided by the user to the GP engine includes a set of building blocks, such as variables and functions that are related to the theory in use. Only loop-free programs are generated. The specification provided by the user consists of first order logic formulas describing pre and post-conditions over the above variables. A new verification component is built for dealing with

sequential programs, including two modules. A *Prover* module is able to get programs from the GP engine, and transfer them into logical formulas that are then checked for correctness by the SMT solver against the specification. The results received from the SMT solver are then used for calculating the fitness function, and for generating counterexamples. The core of this module is based on the *Microsoft Z3* SMT Solver [14]. A *Runner* module is able to run programs directly, and check their correctness for specific given test cases.

For sequential programs, we can use the Hoare notation $\{\varphi\}P\{\psi\}$ to denote the requirement that if the execution of the program starts with a state satisfying the (first order formula) φ, upon termination, it satisfies ψ. The formula φ is over the input variables. Assume they do not change. Otherwise, we can use an additional copy of them; a fixed part of the code copies them to the changeable copy. The formula ψ represents the connection between the input and output variables upon termination. Termination is not an issue here, as our generated loop-free programs must always terminate by construction (they contain no loops). Let φ be the common precondition and $\psi_1..\psi_n$ be a set of post-conditions. We want to check for each ψ_i whether $\{\varphi\}P\{\psi_i\}$ holds. Using standard construction, we obtain a formula η_P that represents the relationship between the input and output variables.

For each postcondition ψ_i we define

$$F_i := \varphi \wedge \eta_P \wedge \neg\psi_i$$

and

$$F_i' := \varphi \wedge \eta_P \wedge \psi_i$$

We can define the following three fitness levels in order of increasing value:

1. F_i' *is not satisfiable.* Then, the program P is incorrect (w.r.t. ψ_i) for all possible inputs.
2. *both* F_i *and* F_i' *are satisfiable.* There exists an input for which P satisfies ψ_i.
3. F_i *is unsatisfiable.* P is correct (for all inputs).

As an example for using our basic method, we tried first to synthesize one of the simplest programs from [22], which is required to output 0 in the variable R if and only if its input X equals $2^n - 1$ for some non-negative n. The GP engine was allowed to generate straight line programs, using only *assignment* instructions, bit-vectors related operators (such as *and*, *or* and *xor*) constants (*0* and *1*), and variables. Within a few seconds, the following correct program was generated.

```
T = X + 1
R = T and X
```

Solutions found by the method described above are guaranteed to be correct for every possible input (in the domain of the variables, such as bit-vectors with a specific width). This is a major advantage over solutions generated by traditional GP, which can usually guarantee correctness only for the set of test cases. However, using test cases can help in building a smoother fitness function that can direct the generated programs into gradual improvements. Hence, we used for calculating the fitness function, in addition to the above satisfiability based levels, a collection of test cases that failed on previous selected candidates. Each test case is obtained using the SAT solver when checking satisfiability of F_i on a previous candidate program. It consists of initial values from which we can run new checked candidates. Note that running the code on test cases, using the runner module, is faster than applying SAT solving using prover module.

After adding the ability to use test cases, we tried to synthesize a more advanced program that is required to compute the floor of the average of its inputs X and Y without overflowing (which may be

```
R = X + Y
R = R >> 1
```

(a)

```
T = X >> 1
R = Y >> 1
R = T + R
```

(b)

```
R = X and Y
T = X xor Y
T = T >> 1
R = R + T
```

(c)

Figure 7: Synthesized Programs for Computing *avg(X,Y)*

caused by simply summing the inputs before dividing by two). Figure 7 shows some of the programs generated during the synthesis process (the logical shift right operator is denoted by ">>").

Program *(a)* is the naive way for computing the average. However, the addition may cause an overflow, and indeed the program was refuted by the SMT solver, yielding a counterexample with big inputs. At the next iteration, program *(b)* was generated. While not overflowing, the program is still incorrect if both of its inputs are odd, which was reflected by a second counterexample. Finally, the more ingenious program *(c)* was generated, and verified to be a correct solution (identical to the one presented in [22]).

8 Conclusions

We suggested the use of a methodology and a tool that perform synthesis of programs based on genetic programming guided by model checking. Code mutation is at the kernel of genetic programming (crossover is also extensively used, but we did not implement it). Our method can be used for

- synthesizing correct-by-design programs,
- finding errors in protocols with complicated architectures,
- 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.

In general, the verification of parametric systems is undecidable, and in the few methods that promise termination, quite severe restrictions are required. The same apply to code synthesis. Nevertheless, we provide a co-evolution method for synthesizing parametric systems based on accumulating cases to be checked. Parameters or 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.

We allowed constructing the architecture (processes and the channels between them) as part of the code that can be mutated. Then the genetic mutation operation can be used in finding architectures in which given algorithms fail. This can be used to model check code with varying architecture, and furthermore, to correct it.

We started recently to look at replacing model checking by SAT and SMT tools. This provides an efficient alternative for some synthesis problem. In particular, SMT solvers may succeed in some parametric cases where model checking fails.

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

References

[1] Krzysztof R. Apt & Dexter Kozen (1986): *Limits for Automatic Verification of Finite-State Concurrent Systems*. *Inf. Process. Lett.* 22(6), pp. 307–309, doi:10.1016/0020-0190(86)90071-2.

[2] W. Banzhaf, P. Nordin, R. E. Keller & F. D. Francone (2001): *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications (3rd edition)*. Morgan Kaufmann, dpunkt.verlag.

[3] Yoah Bar-David & Gadi Taubenfeld (2003): *Automatic discovery of mutual exclusion algorithms*. In: *PODC*, p. 305, doi:10.1145/872035.872080.

[4] James E. Burns & Nancy A. Lynch (1993): *Bounds on Shared Memory for Mutual Exclusion*. *Information and Computation* 107(2), pp. 171–184, doi:10.1006/inco.1993.1065.

[5] Edsger W. Dijkstra (1965): *Solution of a problem in concurrent programming control*. *Commun. ACM* 8(9), p. 569, doi:10.1145/365559.365617.

[6] E. Allen Emerson & Kedar S. Namjoshi (1995): *Reasoning about Rings*. In: *POPL*, pp. 85–94, doi:10.1145/199448.199468.

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari & Ramarathnam Venkatesan (2011): *Synthesis of loop-free programs*. In: *PLDI*, pp. 62–73, doi:10.1145/1993498.1993506.

[8] Gal Katz & Doron Peled (2008): *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*. In: *ATVA*, *LNCS* 5311, pp. 33–47, doi:10.1007/978-3-540-88387-6_5.

[9] Gal Katz & Doron Peled (2008): *Model Checking-Based Genetic Programming with an Application to Mutual Exclusion*. In: *TACAS*, *LNCS* 4963, pp. 141–156, doi:10.1007/978-3-540-78800-3_11.

[10] Gal Katz & Doron Peled (2009): *Synthesizing Solutions to the Leader Election Problem using Model Checking and Genetic Programming*. In: *HVC*, *LNCS* 6405, pp. 117–132, doi:10.1007/978-3-642-19237-1_13.

[11] Gal Katz & Doron Peled (2010): *Code Mutation in Verification and Automatic Code Correction*. In: *TACAS*, LNCS, pp. 435–450, doi:10.1007/978-3-642-12002-2_36.

[12] Gal Katz & Doron Peled (2010): *MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming*. In: *ATVA*, pp. 359–364, doi:10.1007/978-3-642-15643-4_28.

[13] Joep L. W. Kessels (1982): *Arbitration Without Common Modifiable Variables*. *Acta Inf.* 17, pp. 135–141, doi:10.1007/BF00288966.

[14] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS*, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[15] Peter Niebert, Doron Peled & Amir Pnueli (2008): *Discriminative Model Checking*. In: *CAV*, *LNCS* 5123, Springer, pp. 504–516, doi:10.1007/978-3-540-70545-1_48.

[16] Doron Peled (2001): *Software Reliability Methods*. Springer, doi:10.1007/978-1-4757-3540-6.

[17] Jose Antonio Perez, Rafael Corchuelo & Miguel Toro (2004): *An order-based algorithm for implementing multiparty synchronization*. *Concurrency - Practice and Experience* 16(12), pp. 1173–1206, doi:10.1002/cpe.903.

[18] Peterson & Fischer (1977): *Economical Solutions to the Critical Section Problem in a Distributed System*. In: *STOC: ACM Symposium on Theory of Computing (STOC)*, pp. 91–97, doi:10.1145/800105.803398.

[19] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of a Reactive Module*. In: *POPL*, pp. 179–190, doi:10.1145/75277.75293.

[20] Amir Pnueli & Roni Rosner (1990): *Distributed Reactive Systems Are Hard to Synthesize*. In: *FOCS*, pp. 746–757, doi:10.1109/FSCS.1990.89597.

[21] Yih-Kuen Tsay (1998): *Deriving a Scalable Algorithm for Mutual Exclusion*. In: *DISC*, pp. 393–407, doi:10.1007/BFb0056497.

[22] Henry S. Warren (2002): *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.