Program Generation using Simulated Annealing and Model Checking*

Idress Husien and Sven Schewe

Department of Computer Science, University of Liverpool, UK

Abstract Program synthesis can be viewed as an exploration of the search space of candidate programs in pursuit of an implementation that satisfies a given property. Classic synthesis techniques facilitate exhaustive search, while genetic programming has recently proven the potential of generic search techniques. But is genetic programming the right search technique for the synthesis problem? In this paper we challenge this belief and argue in favor of simulated annealing, a different class of general search techniques. We show that, in hindsight, the success of genetic programming has drawn from what is arguably a hybrid between simulated annealing and genetic programming, and compare the fitness of classic genetic programming, the hybrid form, and pure simulated annealing. Our experimental evaluation suggests that pure simulated annealing offers better results for automated programming than techniques based on genetic programming.

1 Introduction

The development of correct code can be quite challenging, especially for concurrent systems. Classical software engineering methods, where the validation is based on testing, do not seem to provide the right way to approach this type of involved problems, as bugs easily elude predefined tests. Guaranteeing correctness for such programs is also not trivial. Manual proof methods for verifying the correctness of the code against a given formal specification were suggested in the late 60s. The next step for achieving more reliable software has been to offer an automatic verification procedure through model checking [6,2,18,1,26,3,27,15].

The holy grail of such techniques would be synthesis: the automated construction of programs that are correct by construction. Such synthesis techniques have long been held to be impossible for reactive systems due to the complexity of synthesis, which ranges from EXPTIME for CTL synthesis [5,25] to undecidable for distributed systems [33,30,13,34].

This line of thought has come under attack on many fronts. On the theoretical side, bounded [14] and succinct [11] synthesis techniques have levelled the playing field between the verification and synthesis of reactive systems by shifting the focus from the input complexity to the cost measured in the minimal explicit

* This work was supported by the Ministry of Higher Education in Iraq through the University of Kirkuk and by the EPSRC through grant EP/M027287/1.

and symbolic solution, respectively. One could argue that this is the theoretical underpinning of successful approaches, including implementations of bounded synthesis [12,10] and methods based on genetic programming [20,21,22,23].

The success of genetic programming is also based on the observation that the neighborhood of good solutions are often 'not bad', and would often still display many sought after properties, such as satisfying a number of sub-specifications fully, and others partially. Such properties are translated to a high fitness of the candidate solution. Vice versa, the higher the fitness of a candidate, the more likely is it to find a full solution in its proximity. This observation is also at the heart of traditional engineering techniques: usually the elimination of a bug does not cause errors in other places. It is also the assumption used when applying program repair [19,36] techniques. The successive development into correct programs is also distantly related to counter-example-guided inductive synthesis [35] for inductive programs, where a genetic approach has also been discussed [7].

Our work is at the same time inspired by the success of genetic programming and driven by the doubt if genetic programming is the right generic search technique to use. The success of genetic programming for synthesis is thoroughly documented by a series of papers by Katz and Peled [21,22,23]. The doubts, on the other hand, are fueled by the general observation that genetic programming is often outperformed by simulated annealing [8,28,31].

On a conceptual level, the difference between simulated annealing and genetic programming techniques are rather minor. These difference are threefold. The first difference is in the number of candidates considered in each iteration. In genetic programming, these are many. In the Katz and Peled papers [21,22,23], for example, these are typically 150, 5 from the previous cycle and 145 mutated programmes—numbers we have copied for our own experiments with genetic programming. In simulated annealing, there is typically one new implementation in each iteration. The second difference is that genetic approaches may use crossovers, a proper mix of two candidate solutions, in addition to mutations, whereas simulated annealing only uses mutations[1]. The third difference is the way the selection takes place. The rules for selection is typically static for genetic programming, while the entropy falls over time in simulated annealing.

It is important to note that crossovers are not always used in genetic programming, and we are not aware of any genetic programming approach that has tried to exploit crossovers for synthesis. Personal communication with the authors of [21,22,23] showed that they did not believe that crossover would be useful in the context of synthesis. Simulated annealing has been reported [8,28,31] to outperform genetic programming when crossovers do not provide an advantage or are not used. Broadly speaking, this is because keeping only a single instance increases the update speed (where the factor is roughly the number of instances), whereas many instances reduce the search depth or increase the likelihood of suc-

[1] The changes are usually not referred to as mutations, but the rules of obtaining them are the same. We use the term mutations for simulated annealing, too, in order to ease the comparison between simulated annealing and genetic programming.

cess in a bounded search with a fixed number of iterations. Overall, the speed-up of the update tends to outweigh the increase in depth, or the reduction in the success rate, of a bounded search. This led us to the hypothesis that the same holds when these techniques are used in synthesis.

Finally, the paper series on genetic programming by Katz and Peled [21,22,23] has used a layered approach, where the weighing of the search function differs over time, starting with establishing the safety properties. The effect of this difference is comparable to the effect of cooling when a stable level of quality is reached. We took this as another hint that simulated annealing is the more appropriate technique when implementing synthesis based on general search with model checking as a fitness measure. In this work we suggest to use simulated annealing for program synthesis and compare it to similar approaches based on genetic programming. We use a formal verification technique, model checking, as a way of assessing its fitness in an inductive automatic programming system. We have implemented a synthesis tool, which uses multiple calls to the model checker NuSMV [3] to determine the fitness for a candidate program. The candidate programs exist in two forms. The main form is a simple imperative language. This form is subject to mutation, but it is translated to a secondary form, the modeling language of NuSMV, for evaluating its fitness. All choices of how exactly a program is represented and how exactly the fitness is evaluated are disputable. Generic search techniques are, however, usually rather robust against changes in such details. While there has been further research on how to measure partial satisfaction [17], we believe that the best choice for us is to keep to the choices made for promoting genetic programming [21,22,23], as this is the only choice that is completely free of suspicion of being selected for being more suitable for simulated annealing than for genetic programming. A second motivation for this selection is that it results in very simple specifications and, therefore, in fast evaluations of the fitness. Noting that synthesis entails on average hundreds of thousands to millions of calls to a model checker, only simple evaluations can be considered. We have implemented six different combinations of selection and update mechanism to test our hypothesis: besides simulated annealing, we have used genetic programming both without crossover (as discussed in [21,22,23]) and with crossover. The tests we have run confirmed that simulated annealing performs significantly better than genetic programming. As a side result, we found that the assumption of the authors of [21,22,23] that crossover does not accelerate genetic programming did not prove to be entirely correct, but the advantages we observed were minor.

2 The approach in a nutshell

In a nutshell, our synthesiser (cf. Figure 1) consists of four main components: a modifier / seeder for programs (Program Generation), a compiler into a model checker format (Program Translation), a quantitative extension of a model checker, using NuSMV [3] as a back-end, and a selector that determines which program to keep (Simulated Annealing). The specification is provided in form of

a list of sub-specifications, which is then automatically extended to additional weaker specifications that are used to obtain a quantitative measure for partial satisfaction. Broadly speaking, the extension takes partial satisfaction of a specification into account by giving different weights to different weaker versions of sub-specifications (cf. Section 4). The result can be manually modified, but the results reported in Section 6 refer to the automatically produced extension.

The internal representation of a program is a tree. The seeder / modifier produces an initial seed. (Alternatively, one could start with an initial program provided by the user.) The modifier / seeder also produces modifications of existing programs by changing sub-trees (cf. Section 4). The programs are then translated to the input language of a model checker (NuSMV in our case), which is then called several times to determine the level of satisfaction, which is the core of the fitness (cf. Section 4) of a program.

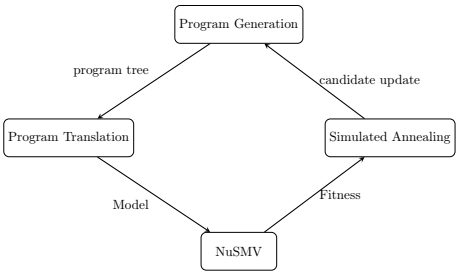

Figure 1. Synthesis Tool

Broadly speaking, the number of candidate programs kept depends on the search technique used. We have implemented both genetic programming approaches and simulated annealing in order to obtain a clean point of comparison.

3 Background

Simulated Annealing. Simulated annealing [4,16] is a general local search technique that is able to escape from local optima, easy to implementation, and has good convergence properties.

When applied to an optimisation problem, the fitness function (objective) generates values for the quality of the solution constructed in each iteration. The fitness of this newly selected solution is then compared with the fitness of the solution from the previous round. Improved solutions are always accepted, while some of the other solutions are accepted in the hope of escaping local optima in search of global optima. The probability of accepting solutions with reduced fitness depends on a temperature parameter, which is typically falling monotonically with each iteration of the algorithm.

Simulated annealing starts with an initial candidate solution. In each iteration, a neighboring solution is generated by mutating the previous solution. Let, for the i^{th} iteration, F_{i-1} be the fitness of the 'old' solution and F_i the fitness of its mutation constructed in the i^{th} iteration. If the fitness is not decreased ($F_i \geq F_{i-1}$), then the mutated solution is kept. If the fitness is decreased ($F_i < F_{i-1}$), then the probability p that this mutated solution is kept is

$$p = e^{\frac{F_i - F_{i-1}}{T_i}} ,$$

where T_i is the temperature parameter for the i^{th} step. The chance of changing to a mutation with smaller fitness is therefore reduced with an increasing gap in the fitness, but also with a falling temperature parameter. The temperature parameter is positive and usually non-increasing ($0 < T_i \leq T_{i-1}$). The development of the sequence T_i is referred to as the *cooling schedule* and inspired by cooling in the physical world [16].

Algorithm 1 Simulated Annealing algorithm

```
i := 0
loop local search with cooling
repeat
  i := i + 1
  derive a neighbor x' of x
  ΔF := F(x') − F(x)
  if ΔF < 0 then
    x := x'
  else
    derive random number p[0, 1]
    if p < e^(ΔF / T(i)) then
      x := x'
    end if
  end if
until the goal is reached or i = i_max
```

The effect of *cooling* on the simulation of annealing is that the probability of following an unfavorable move is reduced. In practice, the temperature is often decreased in stages. During each stage the temperature is kept constant until a balanced solution is reached. The set of parameters that determines how the temperature is reduced (i.e., the initial temperature, the stopping criterion, the temperature decrements between successive stages, and the number of transitions for each temperature value) is called the cooling schedule. We have used a simple cooling schedule, where the temperature is dropped by a constant in each iteration. The algorithm is described in Algorithm 1.

Genetic programming. Genetic programming [24] is a different general search technique that has been used for program synthesis in a similar setting [20,21,22,23]. In genetic programming, a population of λ candidate programs is first generated randomly. In each step, a small share of the population consisting of μ candidates (with $\mu \ll \lambda$) is maintained based on the fitness. Usually, a random function that makes it more likely for fitter candidate programs to be selected for spawning the next generation is applied. The selected candidates are then mated to retain a population of λ, and mutations are applied to a high share of the resulting programs (e.g., on all duplicates).

We have implemented genetic programming as a comparison point, using the values $\lambda = 150$ and $\mu = 5$ from [21]. We also use the $2,000$ iterations suggested

there as a cut-off point, where the algorithm is re-started. In its pure form, it uses the sum of the partial satisfaction values of all sub-specifications as a foundation of the fitness function.

We have additionally implemented a hybrid form that changes the selection technique over time. This technique works in layers: it first establish the safety properties, and then the liveness properties. Specifications with better values for the safety properties are always given preference, while liveness properties are—for equal values for the safety properties—used to determine the fitness. I.e., they are merely tie-breakers.

This approach has been used in [21,22,23]. We refer to it as a *hybrid approach* as it introduces a property known from simulated annealing: in the beginning, the algorithm is applying changes more flexibly, while it becomes more rigid later.

We have implemented the genetic approaches with and without crossover, and used both evaluation techniques for simulated annealing, where we refer to using the classic fitness function as a *rigid* evaluation, and to the hybrid approach as *flexible* evaluation.

Model checking. Model checking [6,2] is a technique used to determine whether a program satisfies a number of specifications. A model checker takes two inputs. The first of them, the specification, is a description of the temporal behavior a correct system shall display, given in a temporal logic. The second input, the model, is a description of the dynamics of the system that the user wants to evaluate. This might be a computer program, a communications protocol, a state machine, a circuit diagram, etc.

A model checker uses a symbolic representation of the model to decide efficiently if the model satisfies the specification. Standard temporal logic used in model checking are linear-time temporal logic (LTL) [32] and computation tree logic (CTL) [5]. We focus on the latter.

Given a finite set Π of atomic propositions, the syntax of a CTL formula is defined as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid A\psi \mid E\psi,$$
$$\psi ::= X\phi \mid \phi U \phi \mid G\phi,$$

where $p \in \Pi$. For each CTL formula ϕ we denote the length of ϕ by $|\phi|$.

Let $T = (V, E)$ be an infinite directed tree, with all edges pointing away from the root. (In model checking, this is the unraveling of the model.) Let $l : V \to 2^{\Pi}$ be a labeling function. The semantics of CTL is defined as follows. For each $v \in V$ we have:

- $v \models p$ if, and only if, $p \in l(v)$.
- $v \models \neg\phi$ if, and only if, $v \not\models \phi$.
- $v \models \phi \vee \psi$ if, and only if, $v \models \phi$ or $v \models \psi$.
- $v \models A\psi$ if, and only if, for all paths π starting at v, we have $\pi \models \psi$.
- $v \models E\psi$ if, and only if, there exists a path π starting at v with $\pi \models \psi$.

Let $\pi = v_1, v_2, \ldots$ be an infinite path in T. We have:

- $\pi \models X\phi$ if, and only if, $v_2 \models \phi$.
- $\pi \models \phi U \phi'$ if, and only if, there exists an $i \in \mathbb{N}$ such that $v_i \models \phi'$ and, for all j in the range $1 \leq j < i$, we have $v_j \models \phi$.
- $\pi \models G\phi$ if, and only if, $v_i \models \phi$ for all $i \in \mathbb{N}$.

Note that the ϕ and ϕ' here are state formulas.

The pair (T, l), where T is a tree and l is a labeling function, is a *model* of ϕ if, and only if, $r \models \phi$, where $r \in V$ is the root of the tree. If (T, l) is a model of ϕ, then we write $T, l \models \phi$.

For the candidate programs in our paper, the tree is the tree of all runs / interleaving of the programs under asynchronous composition, and the labels are the program states.

4 Synthesis tool architecture

Our tool consists of four main parts: a generator and mutator of abstract programs (Program Generation); a translator from abstract programs to models (Program Translator); a model checker as a basis for determining the fitness, and the simulated annealing mechanism for selecting the candidate program to continue with (cf. Figure 1).

We use NuSMV [3] as a model checker. The translator therefore translates the abstract programs into the model language of NuSMV. The other parts of the tool are written in C++. Figure 1 gives an overview on the main components of our tool.

When comparing simulated annealing to genetic programming, we merely replace the simulated annealing component by a similar component for the respective genetic programming variant and optionally add crossover to the available mutations.

The user provides specifications for the desired properties of a system in the form of a list of CTL specifications for the system dynamics that the program has to satisfy. The simulated annealing component then derives the intermediate specifications (full and partial compliance) that are used to determine the fitness of a candidate (cf. Section 4).

If the candidate program satisfied all required properties, then the synthesiser returns it as a correct program.

Otherwise, it will compare the fitness of the current candidate with the (stored) fitness value of the program it is derived from by mutation. (This is the currently stored candidate.) If the fitness is lower, then the tool will update the stored candidate with the probability $e^{\Delta F/T(i)}$ defined by the loss $\Delta F = F_i - F_{i-1}$ in fitness and the current temperature $T(i)$ taken from the cooling schedule. If the fitness is not lower, the tool will always replaces the stored candidate by the mutated one. When the end of the cooling schedule is reached, the tool aborts.

The synthesis process is then re-started, either with a fresh cooling schedule (usually with a higher starting temperature or slower cooling) or with the same cooling schedule. We have implemented the latter.Additional information about the tool can be found at: `http://cgi.csc.liv.ac.uk/~idresshu/index2.html`.

Model checking as a fitness function. We use model checking to determine the fitness of a candidate program in the same way as it has been used for genetic programming [21,22,23]. Based on the model checking results, we derive a quantitative measure for the fitness (as a level of partial correctness) of a program. This can be the share of properties that are satisfied so far, or mechanically produced simpler properties. For example, if a property shall hold on all paths, it is better if it holds on some paths, and yet better if it holds almost surely.

Our implementation considers the specification as a list of sub-specifications and assigns full marks for each sub-specification, which is satisfied by the candidate program. For cases where the sub-specification is not satisfied, we distinguish between different levels of partial satisfaction.

We offer an automated translation of properties with up to two universal quantifiers that occur positively. 100 points are assigned when the sub-specification is satisfied, 80 points if the specification is satisfied when replacing one universal path quantifier by an existential path quantifier, and 10 points are assigned if the specification is satisfied after replacing both universal path quantifiers by existential ones. (Existential quantifiers that occur negatively are treated accordingly.) Examples of this automated translation are shown in Section 5.

The output of the model checker is used to evaluate the fitness of the current candidate. The main part of the fitness is the average of the values for all sub-specifications in the rigid evaluation and the average of all liveness specifications in the flexible evaluation. Following [21], we apply a penalty for long programs by deducing the number of inner nodes of a program from this average when assigning the fitness of a candidate program. The resulting fitness value will be used by simulated annealing to compare the current candidate with the previous one when using rigid evaluation, and to make a decision whether the changes will be preserved or discarded. When using flexible evaluation, this only happens if the value for the safety specification is equal; falling resp. rising values for safety specifications always result in discarding resp. selecting the update when using flexible evaluation.

Programs as trees. The main form of the programs is a tree, in which each *leaf node* represents a parameter or constant, while each *parent node* represents an operation like assignments, comparisons, or algorithm instruction like *if* or *while*. The candidate programs are built from the root down to the terminal nodes (cf. [24,21]). Figure 2 shows the tree representation of the program

while (turn==me)
other=0

on the left, and two mutations of these programs in the middle and on the right.

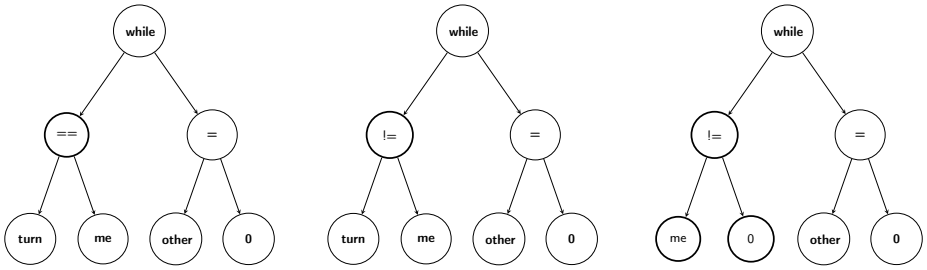

Figure 2. Program tree (left) with two mutations (middle and right)

Mutations are changes in the program tree. Changes can be applied as follows:

1. Randomly select a node to be changed.
2. Apply one of the following changes:
   (a) Replace a boolean comparator by a different boolean comparator. E.g., the middle program from Figure 2 can result from the left program when '==' is replaced by '!='.
   (b) Replace a leaf by a different parameter or constant from a user defined set.
   (c) Replace a sub-tree (which is no leaf) by a different sub-tree of size 3 with the same type. E.g., the right program from Figure 2 can result from the left program when by replacing the left sub-tree.
   (d) Add a new internal node, using the node that was there as one sub-tree and creating further offspring of minimal size (which is ≤ 3) to make the resulting tree well typed.

Crossovers between two programs P1 and P2 randomly select nodes N1 of P1 and N2 of P2, and swap the sub-trees rooted in N1 and N2. This way, they produce a proper mix of the two programs.

Besides standard commands—'while', 'if', assignments, boolean connectives and comparators—there are also variable names and constants. They have to be provided by the user. The user also needs to specify, which variables are local and which are global. She can provide an initial tree with nodes that the modifier is not allowed to alter. Examples of this are provided in Section 5.

To evaluate the fitness of the produced program, it is first translated into the language of the model checker NuSMV [3]. We have used the translation method suggested by Clarke, Grumberg, and Peled [6]. In this translation, the program is converted into very simple statements (similar to assembly language). To simplify the translation, the program lines are first labeled, and this label is then uses as a pointer that represents the program counter *(PC)*. From this intermediate language, the NuSMV model is then built by creating *(case)* and *(next)* statements that use the *PC*. Figure 3 shows the translation of a mutual exclusion algorithm. At first, each line in the source algorithm labelled, then a variable pc (which is local for each MODULE) is added to represent the control state.

```
process me
while (true) do
   noncritical section
   while (turn==me) do
      skip
   end while
   critical section
   turn=other
end while
```

'me' and 'other' are (different) variable valuations, in this example implemented as boolean variables. In other instances, they might be have a different (finite) datatype.

```
MODULE p(turn)
VAR
pc: {11, 12, 14,15};
ASSIGN
init(pc) := 11;
next(pc) :=
case
(pc=11) : {11, 12};
(pc=12)&(turn=me) : 14;
(pc=14) : 15;
(pc=15) : 11;
TRUE: pc;
esac;
next(turn):=
case
(pc=15): other;
TRUE :turn;
esac;
```

Figure 3. Translation example – source(left) and target (right)

5 Case studies

We have selected mutual exclusion [9] and leader election [29,23] as case studies, because these are the examples, for which genetic programming has been successfully attempted.

Mutual exclusion. In mutual exclusion, no two processes are allowed to be in the *critical section* at the same time. In addition, there are liveness properties that essentially require non-starvation.

For the mutual exclusion example, we consider programs that progress through four sections, a 'non-critical section', an 'entry section', a 'critical section', and an 'exit section'. The 'non-critical section' and 'critical section' parts are not targets of the synthesis process. In this example, we start with a small program tree that includes the non-critical section and the critical section as privileged commands that cannot be changed by the modifier. Neither can any of their ancestors in the program tree. The entry and exit sections, on the other hand, are standard parts of the tree that can be changed.

The modifier is also provided with the vocabulary it can use. Besides the standard commands and the privileged commands for the critical and non-critical sections, these are the variables 'me' and 'other' that identify the two processes involved and, depending on the benchmark, two or three global / shared boolean variables.

The mutual exclusion example uses one safety specification: only one process can be in the critical sections at a time. This is represented by the CTL formula

$$!EF(P0 \text{ in critical section } \&\ P1 \text{ in critical section }).$$

When using this sub-specification for determining the fitness, we assign

100 points when the sub-specification is satisfied, and
80 points when $!AF(P0$ in critical section & $P1$ in critical section) holds.

In addition, there is a non-starvation property that, whenever a process enters its entry section, it will eventually enter the critical section. For process, one this is

$$AG(P1 \text{ in entry section} \rightarrow AFP1 \text{ in critical section}).$$

When using this sub-specification for determining the fitness, we assign

100 points when the sub-specification is satisfied,
80 points when $EG(P1$ in entry section $\rightarrow AFP1$ in critical section) holds,
80 points when $AG(P1$ in entry section $\rightarrow EFP1$ in critical section) holds, and
10 points when $EG(P1$ in entry section $\rightarrow EFP1$ in critical section) holds.

Leader election. As a second case study, we consider synthesising a solution for the leader election problem [29,23]. For that purpose, we use clockwise unidirectional ring networks with two different sizes, three or four nodes, respectively.

For leader election, we do not consider any privileged commands. Again, the modifier needs to be provided with vocabulary. Besides the standard commands, this includes

- *id:* a specific integer value for each node in the ring, which have the values $1, \ldots, i$ for rings of size i.
- *myval,other,leaderID:* local variables; leaderID is initialized to 0.
- *Send (myval):* a command that refers to sending the value of 'myval' to the next node in the ring. (It is placed in a variable the next process can read using the following command.)
- *Receive (other):* a command that reads the last value sent by the previous node.

The specification for leader election requires the safety specification that there is never more than one leader, and the liveness requirement that a leader will eventually be elected. For both requirement, we assign

100 points when the sub-specification is satisfied on all paths, and
80 points when the sub-specification is satisfied on some path.

6 Results

We have implemented the simulated annealing and genetic programming algorithms as described, using NuSMV [3] as a solver when deriving the fitness of candidate programs. For simulated annealing, we have set the initial temperature to $20,000$. The cooling schedule decreases the temperature by 0.8 in each iteration. The schedule ends after $25,000$ iterations, when the temperature hits

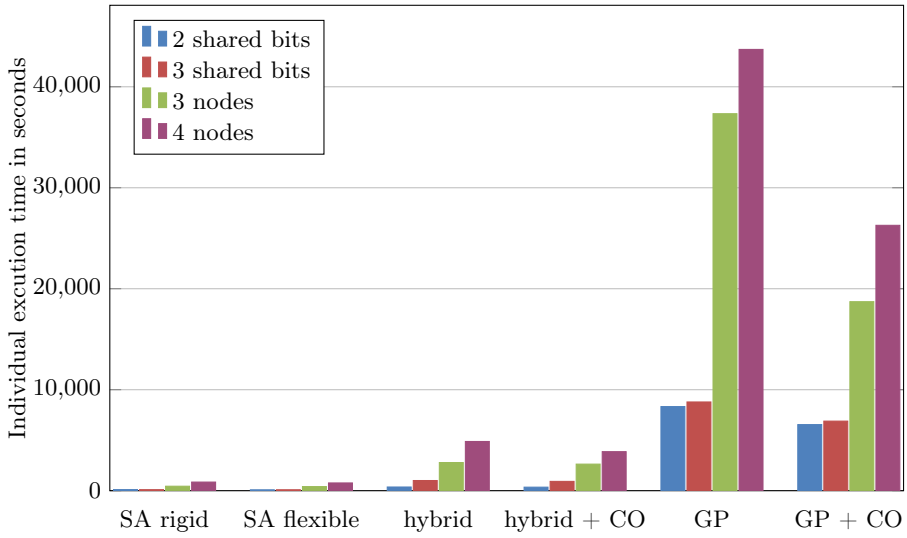

Figure 4. Average time required for synthesising a correct program

0. In a failed execution, this leads to determining the fitness of 25,001 candidate programs.

As described in Section 3, we have taken the values suggested in [21] for genetic programming: $\lambda = 150$ candidate programs are considered in each step, $\mu = 5$ are kept, and we abort after 2,000 iterations. In a failed execution, this leads to determining the fitness of 290,150 candidate programs.

For the mutual exclusion benchmark, we distinguish between programs that use two and three shared bits, respectively. For the leader election benchmark we use ring networks with three and four nodes, respectively. The results are shown in Figures 4 and 5 and summarised in Table 1. The experiments have been conducted using a machine with an Intel core i7 3.40 GHz CPU and 16GB RAM. Figure 4 shows the average time needed for synthesising a correct program. The two factors that determine the average running time are the success rate and the running time for a full execution, successful or not. These values are shown in Figure 5.

An individual execution of simulated annealing ends when a correct program is found or when the stopping temperature is reached after 25,000 iterations. Similarly, the genetic programming approaches stop when they have found a solution or when the number or iterations has reached its maximum of 2,000 iterations. Note that, while simulated annealing incurs more iterations before reaching its termination criterion, it needs to perform only a fraction of the model checking tasks in each iteration. While the number of iterations is slightly more than an order of magnitude higher, the number of programs, for which the fitness needs to be calculated, is slightly more than an order of magnitude lower (25,001 vs. 290,150).

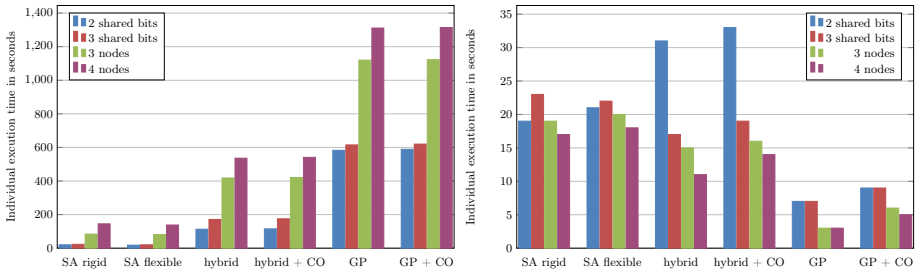

Figure 5. Average running time of an individual execution (left) and success rate of individual executions (right)

For the model checking community, success rates of around 20% may sound very low, but this is the appropriate range for such techniques. Note that it is very simple to drive the success rate up: one can decrease the cooling speed for simulated annealing and increase the number of iterations for genetic programming, respectively. However, this also increases the running time for individual full executions. A very high success rate is therefore not the goal when devising these algorithms, but a low expected overall running time. A 20% success rate is in a good region for achieving this goal. Table 1 shows the average running time for single executions in seconds, the success rate in %, and the resulting overall running time. The best values (shortest expected running time or highest success rate) for each comparison printed in bold. Both simulated annealing and the hybrid approach significantly outperform the pure genetic programming approach. The low success rate for pure genetic programming suggests that the number of iterations might be too small. However, as the individual execution time is already ways above the average time simulated annealing needs for constructing a correct program, we did not increase the number of iterations.

The advantage in the individual execution time between the classic and the hybrid version of genetic programming is in the range that is to be expected, as the number of calls to the model checker is reduced. It is interesting to note that simulated annealing, where the shift from rigid to flexible evaluation might be expected to have a similar effect, does not benefit to the same extent. It is also interesting to note that the execution time suggests that determining the fitness of programs produced by simulated annealing is slightly more expensive. This was to be expected, as the average program length grows over time. The penalty for longer programs reduces this effect, but cannot entirely remove it. (This potential disadvantage is the reason why an occasional re-start provides better results than prolonging the search.)

The advantage in running of simulated annealing compared to the hybrid approach reach from factor 4 to factor 10, and the comparison to pure genetic programming reach from factor 35 to factor 76. It is interesting to note that both the pure and the hybrid approach to genetic programming benefit from crossovers, but while the benefit for the pure approach is significant, almost

Table 1. Search Techniques Comparison

	Search Technique	single execution	success rate	overall time
2 shared bits	SA rigid	20	19	105.26
	SA flexible	**18**	21	**85.71**
	Hybrid w/o crossover	113	31	364.51
	Hybrid with crossover	115	**33**	348.48
	GP w/o crossover	583	7	8,328.57
	GP with crossover	589	9	6,544.44
3 shared bits	SA rigid	23	**23**	100
	SA flexible	**20**	22	**90.9**
	Hybrid w/o crossover	171	17	1,005.88
	Hybrid with crossover	175	19	921.05
	GP w/o crossover	615	7	8,785.71
	GP with crossover	620	9	6,888.88
3 nodes	SA rigid	84	19	442.1
	SA flexible	**81**	**20**	**405**
	Hybrid w/o crossover	418	15	2,786.66
	Hybrid with crossover	421	16	2,631.25
	GP w/o crossover	1120	3	37,333.33
	GP with crossover	1123	6	18,716.66
4 nodes	SA rigid	145	17	852.94
	SA flexible	**138**	**18**	**766.66**
	Hybrid w/o crossover	536	11	4,872.72
	Hybrid with crossover	541	14	3,864.28
	GP w/o crossover	1311	3	43,700.00
	GP with crossover	1314	5	26,280.00

halving the average time for synthesising a program in one case, the benefit for the superior hybrid approach is small.

7 Conclusion

We have implemented an automated programming technique based on simulated annealing and genetic programming, both in the pure form of [20] and the arguably hybrid form of [21,22]. The implementations from these papers were unavailable for comparison, but this is, in our view, a plus: the performance is naturally sensitive to the quality of the integration, the suitability of the model checker used, and hidden details, like how the seed is chosen or details of how the fitness is computed. The integrated comparison makes sure that all methods are on equal footage in these regards.

The results are very clear and in line with the expectation we had drawn from the literature [8,28,31]. When crossovers are not used, the main difference between the established genetic programming techniques and simulated annealing is the search strategy of using many and using a single instance, respectively.

The data gathered confirms that an increase of the number of iterations can easily overcompensate the broader group of candidates kept in genetic programming. In our experiments, we have used an increase that fell short of creating the same expected running time for a single full execution (with or without success), and yet outperformed even the hybrid approach w.r.t. the success rate on three of our four benchmarks. We have also added variations of genetic programming that include crossover to validate the assumption that crossovers do not lead to an annihilation of the advantage, but it proved that the hybrid approach, and thus the stronger competitor, does not benefit much from using crossover. The double advantage of shorter running time and higher success rate led to an improvement of 1.5 to 2 orders of magnitude compared to pure genetic programming (with and without crossover), and between half an order and one order of magnitude when compared to the hybrid approach (with or without crossover).

It will be interesting to see if future work will show that these factors are essentially constant, or if they depend heavily on the circumstances.

References

1. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *In CAV*, volume 1427 of *LNCS 1427*, pages 521–525. Springer, 1998.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
3. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *In CAV*, LNCS 2404, pages 359–364. Springer, 2002.
4. J. A. Clark and J. L. Jacob. Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology*, 43:891–904, 2001.
5. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *IBM Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.
6. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
7. C. David, D. Kroening, and M. Lewis. Using program synthesis for program analysis. In *LPAR*, volume 9450 of *LNCS*, pages 483–498. Springer, 2015.
8. L. Davis. *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers Inc., 1987.
9. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, page 569.
10. R. Ehlers. Unbeast: Symbolic bounded synthesis. In *Proc. of TACAS*, volume 6605 of *LNCS*, pages 272–275. Springer, 2011.
11. J. Fearnley, D. Peled, and S. Schewe. Synthesis of succinct systems. *Journal of Computer and System Sciences*, 81(7):1171–1193, 2015.
12. E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *In CAV*, volume 5643 of *LNCS*, pages 263–277. Springer, 2009.
13. B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *LICS*, pages 321–330. IEEE Computer Society Press, 2005.

14. B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.
15. E. M. Hahn, Y. Li, S. Schewe, A. Turrini, and L. Zhang. Iscasmc: A web-based probabilistic model checker. In *FM 2014*, volume 8442 of *LNCS*, pages 312–317. Springer, 2014.
16. D. Henderson, S. H. Jacobson, and A. W. Johnson. The theory and practice of simulated annealing. In *Handbook of metaheuristics*, pages 287–319. Springer, 2003.
17. T. A. Henzinger and J. Otop. From model checking to model measuring. In *CONCUR*, volume 8052 of *LNCS*, pages 273–287. Springer, 2013.
18. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
19. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, volume 3576 of *LNCS*, pages 226–238. Springer, 2005.
20. C. G. Johnson. Genetic programming with fitness based on model checking. In *EuroGP*, volume 4445 of *LNCS*, pages 114–124. Springer, 2007.
21. G. Katz and D. Peled. Model checking-based genetic programming with an application to mutual exclusion. In *TACAS*, volume 4963 of *LNCS*, pages 141–156. Springer, 2008.
22. G. Katz and D. Peled. Model checking driven heuristic search for correct programs. In *MoChArt*, volume 5348 of *LNCS*, pages 122–131. Springer, 2009.
23. G. Katz and D. Peled. Synthesizing solutions to the leader election problem using model checking and genetic programming. In *HVC*, volume 6405 of *LNCS*, pages 117–132. Springer, 2009.
24. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
25. O. Kupferman and M. Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245–263, June 1999.
26. O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.
27. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
28. J. Lahtinen, P. Myllymäki, T. Silander, and H. Tirri. Empirical comparison of stochastic algorithms. In *2NWGA*, pages 45–60, 1996.
29. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL*, pages 97–107. ACM, 1985.
30. P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In *ICALP*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001.
31. J. Mann and G. Smith. A comparison of heuristics for telecommunications traffic routing. In *Modern Heuristic Search Methods*, pages 235–254, 1996.
32. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society Press, 1977.
33. A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE Computer Society Press, 1990.
34. S. Schewe and B. Finkbeiner. Synthesis of asynchronous systems. In *LOPSTR 2006*, volume 4407 of *LNCS*, pages 127–142. Springer, 2006.
35. A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.
36. C. von Essen and B. Jobstmann. Program repair without regret. In *CAV*, volume 8044 of *LNCS*, pages 896–911. Springer, 2013.