A New Framework for Scalable Genetic Programming

Nassima Aleb
USTHB-FEI Faculty
BP 32 AL ALLIA Bab Ezzouar
Algiers, Algeria
naleb@usthb.dz

Samir Kechid
USTHB-FEI Faculty
BP 32 AL ALLIA Bab Ezzouar
Algiers, Algeria
skechid@usthb.dz

ABSTRACT
This paper presents a novel framework for scalable multi-objective genetic programming. We introduce a new program modeling aiming at facilitating programs' creation, execution and improvement. The proposed modeling allows making symbolic executions in such a way to reduce drastically the time of programs' executions and to allow well-founded programs recombination.

Categories and Subject Descriptors
I.2.2:**[Automatic Programming]**; D.2.5:**[Symbolic Execution]**; F.3.1:**[Pre and Post Condition].**

General Terms
Algorithm; Languages; Design; Performance.

Keywords
Genetic Programming; Program Representation; Weakest Precondition; Semantic Crossover.

1. INTRODUCTION
Genetic programming (GP) is an evolutionary-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. GP is a method of automatically generating computer programs to perform specified tasks [1]. It uses a genetic algorithm to search through a space of possible computer programs for one which is nearly optimal in its ability to perform a particular task. GP develops programs, usually represented in memory as trees. Trees are recursively evaluated. Every tree node has an operator function, and every terminal node has an operand. Accordingly traditionally GP favors predominantly the use of programming languages that naturally embody tree structures such as functional programming languages [3]. Usually, genetic operators are designed so that the resulting children are syntactically valid individuals. However there have been several attempts in using semantics to enhance GP in solving problems. The use of formal methods is one of these attempts which have been raised just recently. Formal methods are a class of mathematically based techniques for the specification, development and verification of software and hardware systems. [6, 7, 8, 9] are the first works which pioneered this area of research for GP. Two kinds of formal techniques have been used in GP, abstract interpretation [11] and model checking [5].

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

Abstract interpretation performs analysis on abstract domains instead of concrete ones. Using abstract interpretation, we can deduce information about some interesting program's properties.
In [6, 7], this information is used as a measure of the fitness. A placement problem is studied in [6]. With this kind of problem, it is very difficult to use traditional fitness measures that are based on a set of sample cases. Firstly, generating a set of sample cases is not easy in this situation. Secondly, even if a set of sample cases can be created, we cannot guarantee that the desired constraints will always be satisfied as the list of sample cases cannot cover all situations. In [14], abstract interpretation is used to check if an individual can be undefined in the whole range of input values. For example, if an individual contains the function log(x) and one can infer that the variable x can take negative values; this individual will be considered as an undefined individual, so it must be deleted from the population. Model checking is an algorithmic technique to verify a system description against a specification represented as a temporal logic formula. In [9], a system is modeled by a set of temporal logic formulas. The fitness function is measured by counting the number of satisfied formulas. Individual satisfies more propositions, it has a greater fitness value. The drawback of this approach is that a formula, which is nearly satisfied, will be considered as an absolutely unsatisfied formula. This weakness is considered by some later research in [12, 13], which used GP with model checking to check if a path in the graph that represents the behavior of the system, is satisfied by the formula. The fitness function is based on scores depending on the number of satisfied paths in the graph. The advantage of formal methods lies in their rigorous mathematical foundations, potentially helping GP to evolve computer programs. However, they are high in complexity and difficult to implement, which explains why they have been used mainly for fitness measures. In this paper, we present a new program representation for GP. It allows:

1. An easy program manipulation.
2. A rapid population evaluation.
3. "Semantically justified" programs' recombination.

The rest of the paper is organized as follows: Section 2 presents program modeling. In the section 3 we introduce our framework, we describe how is performed each of the well-known steps of GP. We focus on individual evaluation and semantic crossover operator. These points are illustrated by clarifying examples. Section 4 concludes by highlighting contributions of this paper, and exposing some possible future directions.

2. PROGRAM MODELING
Since individuals must be widely manipulated by: evaluation, recombination and mutation it is essential to design a program modeling which is as simple as efficient. We represent an individual by two tables. The first table, called: Variable Table, records the different expressions allowing computing program

variables values. The second table: Architecture Table describes the program structure.

2.1 Variable Table: VT

The variable table is composed of three columns the first one is an integer representing the statement location in the program. The second column represents the variable name; the last one is the expression allowing to compute the corresponding variable value. An expression could be: An input, a constant or a call to a predefined function.

2.2 Architecture Table: AT

The Architecture Table describes the program's structure. It contains constraints that make possible the execution of each statement of VT. AT models the conditionals loop statements.

Conditional statements: There are two sorts of conditional statements: alternative statement (with the else branch) and the simple conditional (without the else branch).

An alternative statement is modeled by *(C,CT,CF,End)* where:

- *C*: is a Boolean expression representing the condition of the statement.
- *CT*: is the location of the first instruction to perform if *Cd* is TRUE
- *CF*: is the location of the first instruction to perform if *Cd* is FALSE
- *End*: is the location of the first instruction after the conditional statement.

A simple conditional statement is represented by *(C,CT,End)* with *C,CT* and *End* having the same meaning as the alternative statement. *CF* is set to (-1)

Loops: A loop statement is modeled by *(C,CT,End)* where :

- *C*: is a Boolean expression representing the condition of the statement.
- *CT*: is the location of the first instruction in the loop.
- *End*: is the location of the first instruction after the loop.

CF is set to (-2).

2.3 Example of Individual Modeling

```
scanf("%d%d",&z,&x)
    if (z==x)
1:    { z=x+1;
2:       y=x-1; }
     else
3:   { z=x-1;
4:      y=x+1 ;}
5:  z=z+2;
    if (z<3)
6:    { t=z ; }
    else
7:   { t=-z;
      if (t==-1)
8:       {y=0;}
      else
9:       { y=1;}  }
```

VT

Loc	Var	Expression
1	Z	x+1
2	Y	x-1
3	Z	x-1
4	Y	x+1
5	Z	z+2
6	T	Z
7	T	-z
8	Y	0
9	Y	1

C	CT	CF	End
z=x	1	3	5
z<3	6	7	10
t=-1	8	9	10

AT

Figure 1. C-Program Modeling Example.

3. A NEW FRAMEWORK FOR GENETIC PROGRAMMING

As usual, in genetic programming four steps are used to solve problems:

(1) Generate an initial population of random compositions of functions and terminals of the problem.

(2) Execute each program of the population, with all the fitness cases, and assign it a fitness value according to how well it solves the problem.

(3) Select individuals for crossover and mutation

(4) Create a new population of computer programs.

i) Reproduce the best existing programs

ii) Create new computer programs by mutation.

iii) Create new computer programs by crossover.

(5) The best program that appeared in any generation, the best-so-far solution, is designated as the result of genetic programming [3].

In the subsequent, we will describe how each of these points is performed in our framework.

3.1 Initial Population

The initial population is constituted of a set of pairs (VTi,ATi) one pair for each individual. The tables are filled randomly. Problem functions and terminals are used in both variable table and architecture one. There are of course some syntactic rules to verify in the filling of (VTi,ATi), (like End>CT…).

3.2 Individual Evaluation

To evaluate an individual, we must execute it with all fitness cases. To reduce individual execution time, we perform symbolic executions. The idea is to compute a formula for each output of the genetic program: we call it Result Expression. So, for each output variable, the corresponding Result Expression summarizes all the expressions of the output variable with respect to input variables. Henceforth, for an individual the same obtained formula is used to evaluate all fitness cases by replacing input variables by their values and verify if the corresponding output value is correct. To perform symbolic executions, we use the concept of Weakest Precondition [4].

3.2.1 Weakest Precondition (WP)

Let $v=e$ be an assignment, where v is a variable and e is an expression of the appropriate type. Let P be a predicate. By definition, WP($v=e,P$) is P with all occurrences of v replaced with e. For example: WP($y=x+2$, $y>8$) = $(x+2)>8$. We denote WP(l,P) the weakest precondition of the predicate P with respect to (w.r.t.) the statement having the location l in the table VT. We extend the definition of WP to be applied on intervals of program locations, i.e.: a sequence of adjacent locations and on a whole element of AT: Let i and j be two locations:

$$(1)\quad \mathrm{WP}([i,j[,P)=\begin{cases} P & \text{if no variable occurs in } P \\ \mathrm{WP}(i,P) & \text{if } i=j \\ \mathrm{WP}([i,j[,\mathrm{WP}(j\text{-}1,P)) & \text{Otherwise} \end{cases}$$

Let e be an element of AT: e has the form *(C,CT,CF,End)*:

(2) WP(e,P)=

$$\begin{cases} C\wedge \mathrm{WP}([CT,CF[,P)\vee\neg C\wedge \mathrm{WP}([CF,End[,P) & \text{if } (CF>0) \\ C\wedge \mathrm{WP}([CT,End[,P)\vee\neg C\wedge P & \text{if } (CF=-1) \end{cases}$$

(3) WP([Si,Sj[$\cup$[Sk,Sl[,Cd)=WP([Si,Sj[,WP([Sk,Sl[,Cd)).

(4) **Loops**: Let *l* be an element (a line) in AT having the form *(C,CT,-2,End)*. Weakest precondition computing of a predicate P w.r.t. *l* is performed as follow:

C_0=C; P_0=P ; k=1;
While(True) { P_k=WP([CT,End[,P_{k-1}); C_k=WP([CT,End[,C_{k-1})
(If (C_k'=False)$\vee$(P_k=P_{k-1}) { WP(l,P)=P_k' ; exit }
k=k+1; }

C_k' and P_k' are obtained from C_k and P_k by replacing the variables modified in the loop, by their initial values(see example 2).

3.2.2 Symbolic Executions of Programs

Let $Prog_i$ = (VT_i,AT_i) be an individual. $Prog_i$ evaluation is performed as follows:

1. For each variable o_k, we add in VT_i and AT_i the lines corresponding to the following instruction *if* $(R_{ik}=o_k)$ *then* $R_{ik}=o_k$ (which has no effect on individual execution).
2. We compute backwards the successive Weakest Preconditions of the predicate$(R_{ik}=o_k)$ w.r.t. all the lines of AT_i from the end to the beginning.
3. The resulting expression $ExpR_{ik}$ represents, all the possible expressions of the output o_k with their corresponding conditions.
4. For each fitness case, replace the input variables in $ExpR_{ik}$. by their value and deduce the value of R_{ik} making $ExpR_{ik}$ True.

So, the same $ExpR_{ik}$ is used to compute all fitness cases for the individual Progi: This constitutes the first advantage of symbolic executions.

3.2.3 Individual Execution Examples

3.2.3.1 Example 1

Let's consider the program $Prog_i$ having as input *x* and *y* and as output *b*. Let's call r the expression of the result.

VT

Loc	Var	Exp
1	A	x
2	A	y
3	B	a+1
4	B	a-1
5	R	b

AT

	C	CT	CF	End
l1	x>y	1	2	3
l2	a>0	3	4	5
l3	r=b	5	-1	6

Figure 2. Execution Example 1.

WP(l2,r=b)=(a>0)$\wedge$WP([3,4[,r=b)$\vee$(a<=0)$\wedge$WP([4,5[,r=b)
=(a>0)$\wedge$WP(3,r=b)$\vee$(a<=0)$\wedge$WP(4,r=b)
=(a>0)$\wedge$(r=a+1)$\vee$(a<=0)$\wedge$(r=a-1)=D
WP(l1,D)=(x>y)$\wedge$WP([1,2[,D)$\vee$(x<=y)$\wedge$WP([2,3[,D)
=(x>y)$\wedge$WP(1,D)$\vee$(x<=y)$\wedge$WP(2,D)
=(x>y)$\wedge$[(x>0)$\wedge$(r=x+1)$\vee$(x<=0)$\wedge$(r=x-1)]$\vee$
(x<=y)$\wedge$[(y>0)$\wedge$(r=y+1)$\vee$(y<=0)$\wedge$(r=y-1] **(I)**

The expression (I) is the Result Expression. It represents exactly the result *r* expressed w.r.t all possible values of the input variables x and y. The evaluation of $Prog_i$ on all the fitness cases consists to replace x, y and *r* by their values and to check if (I) is True.

Let C1 and C2 be two fitness cases C1=(x=10,y=1,b=5) and C2=(x=-5,y=6,b=7). Let execute Progi with these two cases:
For C1: we replace in (I) : x by 10 and y by 1.
(I)= (10>1)$\wedge$[(10>0)$\wedge$(r=10+1)$\vee$(10<=0)$\wedge$(r=10-1)] $\vee$
(10<=1)$\wedge$ [(1>0)$\wedge$(r=1+1)$\vee$(1<=0)$\wedge$(r=1-1].
= (T)$\wedge$[(T)$\wedge$(r=11)$\vee$(F)$\wedge$(r=9)] $\vee$
(F)$\wedge$ [(T)$\wedge$(r=2)$\vee$(F)$\wedge$(r=0].
= (r=11). This expression is True if and only if r=11.
So, the result found by the execution of the individual Progi is 11 while the required output is b=5.
For C2:
(I) = (-5>6)$\wedge$[(-5>0)$\wedge$(r=-5+1)$\vee$(-5<=0)$\wedge$(r=-5-1)] $\vee$
(-5<=6)$\wedge$ [(6>0)$\wedge$(r=6+1)$\vee$(6<=0)$\wedge$(r=6-1].
=(F)$\wedge$[(F)$\wedge$(r=-4)$\vee$(T)$\wedge$(r=-6)] $\vee$(T)$\wedge$ [(T)$\wedge$(r=7)$\vee$(F)$\wedge$(r=5].
=(r=7) Which represents the expected output (b=7).
So for each individual the same formula is used to compute the results corresponding to all the fitness cases.

3.2.3.2 Example 2

Loop: input n, output s**;** Fitness Case: (n=3,s=5)

```
1: i=1;
2: s=0;
   while(i<n)
3: { s=s+i;
4:  i=i+1}
   if( r==s)
5: r=s
```

Loc	Var	Exp
1	i	1
2	s	0
3	s	s+i
4	i	i+1
5	r	S

	Cd	CT	CF	End
l1	i<n	3	-2	5
l2	r=s	5	-1	6

Figure 3. Execution Example 2.

WP(l1,r=s)=? C_0=(i<3);P_0=(r=s); k=1;
P_1=WP([3,5[,r=s)= (r=s+i)
C_1=WP([3,5[,i<n)=(i+1<n) so, C_1'=(1+1<3)=(2<3)=True
P_2=WP([3,5[,r=s+i)=(r=s+i+i+1);
C_2=WP([3,5[,i+1<n)=i+1+1<n so C_2'=(3<3)=False.
So, WP(l1,r=s)=P_2'=(r=3). So the result computed by the individual is 3 while the expected result is 5.

3.3 Individual Improvement

Usually, in genetic programming population improvement is performed in some chosen programs. Programs are elected depending on their fitness value. However, programs' combination and mutations are generally performed in an arbitrary way. Hence, there is no "semantic" explanation to the following questions:

- Why is it appropriate to perform **this form of mutation** on **this individual**?
- Why should we recombine **these** two **individuals** in **this way**?

In this paper, we do not focus on the first point but on programs' recombination. The aim of our work is to attempt to perform "semantically justified" recombination. To attain this objective, we exploit information deduced from executions. Firstly, let's introduce the following hypothesis:

1- We situate ourselves in the multi-objective context: So, the problem has several output variables: $o_1 \ldots o_m$ we call our recombination operator: Multi-crossover.

2- We note Fit_{ik} the fitness value of the program $Prog_i$ for the computing of the output o_k. In fact, in our approach, the fitness is not quantified by a unique value. This is justified by the fact that a program can compute very well an output variable o_r and fails dramatically to compute another output o_s. So, programs are judged relating to each output variable separately from the others.

3- For an individual $Prog_i$, we call S_i the set of all Result Expressions (one Result Expression for each output):
$S_i = \{ExpRes_{ir}, \ k=1..m\}$

4- Let F be a first order logical formula, we say that F is in the exclusive form if one of the two conditions holds:

- F is an atomic formula (i.e. it does not contain ∧ nor ∨)
- F is of the form C∧P∨¬C∧Q , where P and Q are two exclusive form formulas

Let's notice that weakest precondition computations give as result an exclusive form formula. Which implies that the expressions $ExpRe_{ik}$ are all in exclusive form.

3.3.1 Multi-Crossover

Our objective is to perform judicious recombination. So, we evaluate programs with respect to output variables. Consequently, programs' recombination is performed relatively to some output variable. We must take advantage of each program $Prog_i$ having a good fitness value F_{ir}. This is why our crossover operator does not create two programs but preserves the first program (the fittest on) and modifies only the second program. The first program could in its turn take advantage of another program which is better than it in the computing of another output. Symbolic executions make possible the isolation of statements computing a considered output. So, the idea of our crossover operator is to replace in S_j, the Result Expression $ExpRes_j$ by $ExprRes_i$ where $Prog_i$ is better than $Prog_j$ in the computing of the output o_k i.e. $Fit_{ik}>Fit_{jk}$. The obtained set S_j' is then translated into a new program $Prog_j$' where all the outputs o_r such that $r \neq k$ are computed as in $Prog_j$ and o_k is computed in the same manner than performed by $Prog_i$. This constitutes the second advantage of using symbolic executions instead of true executions. So, let $Prog_i$ and $Prog_j$ be two programs, we note $Prog_i \bowtie_r Prog_j$ the recombination of $Prog_i$ and $Prog_j$ w.r.t the output variable o_r. The algorithm of the figure 4 describes the crossover operation.

Algorithm: Crossover($Prog_1$,$Prog_2$,o_r)

Input: S_1, S_2, o_r
Output: $Prog_2$'=(VT_2',AT_2')
$S_2=S_2-\{ExpRes_{2r}\} \cup \{ExpRes_{1r}\}$
For each formula $ExpRes_{2k}$ in S_2
Do Translate ($ExpRes_{2k}$);
End.

Figure 4. Algorithm computing the Crossover of two programs

The translation of an exclusive form formula F in an individual Progi=(VTi,ATi) is performed as follows:

Algorithm: Translate(F)

1. Input: F: An EF Formula.
2. Output: VT,AT
3: If F is an atomic formula of the form v=exp
4: Then Let o be the output variable corresponding to v
5: Insert(o,exp) in the current line of VT
6: Else F of the form C∧P∨¬C∧Q
7: Let CT be the current line in VT.
8: insert(C,CT,CF,End) in AT
9: Translate (P);
10: Translate(Q)
End.

Figure 5. Algorithm constructing VT and AT from an exclusive formula

In the line (8), CF and End values are unknown. They will be updated respectively in the lines (9) and (10).

Example: Let's translate the expression (I) of the example 1 section 3.2.3.1 we have:

(I) =(x>y)∧[(x>0)∧(r=x+1)∨(x<=0)∧(r=x-1)]∨
(x<=y)∧[(y>0)∧(r=y+1)∨(y<=0)∧(r=y-1]

Loc	Var	Function
1	b	x+1
2	b	x-1
3	b	y+1
4	b	y-1

C	CT	CF	End
x>y	1	3	5
x>0	1	2	3
y>0	3	4	5

The variable r corresponds to the output variable b. We remark that VT and AT are quite different from the initial tables, because in 3.2.3.1 we used an intermediary variable a which has disappeared in (I), since in (I) we have just outputs and inputs variables.

The translation of a set of formulas is performed by translating each formula independently of the others. The order in which we do translations is not important since in the Result Expressions each output is expressed only by using input variables.

3.3.2 Multi-Crossover Example

Let's consider a population constituted of Prog1, Prog2, and Prog3. We suppose that our problem has four input variables a, b, c and d and three output variables o1, o2 and o3. o1=Max(a,b)*c*d ; o2=|a*c|-b-d ; o3=Min(a,c)+Max(b,d). Of course these functions are unknown in the problem. The three individuals are:

Prog1:

Loc	Var	Exp
1	mx	a
2	mx	b
3	o1	mx*c*d
4	o2	a*c-b-d
5	mn	c
6	mn	a
7	o3	mn+mx
8	res1	o1
9	res2	o2
10	res3	o3

CD	CT	CF	END
a>b	1	2	3
a>c	5	6	7
res1=o1	8	-1	9
res2=o2	9	-1	10
res3=o3	10	-1	11

ExpRes11=(a>b)∧(Res1=a*c*d)∨(a<=b))∧(Res1=b*c*d)
ExpRes12=(Res2=a*c-b-d)
ExpRes13=(a>c)∧(a>b∧ (Res3=c+a)∨(a<=b)∧(Res3=c+b))∨
(a<=c)∧(a>b∧ (Res3=2*a)∨(a<=b)∧(Res3=a+b))

Prog2:

Loc	Var	Exp
1	o1	a*c*d
2	abs	a*c
3	abs	-a*c
4	o2	abs-b-d
5	m	b
6	m	d
7	o3	m+a
8	res1	o1
9	res2	o2
10	res3	o3

CD	CT	CF	END
a*c>0	2	3	4
b>d	5	6	7
res1=o1	8	-1	9
res2=o2	9	-1	10
res3=o3	10	-1	11

ExpRes21= (Res1=a*c*d)
ExpRes22=(a*c>0)∧(Res2=a*c-b-d)∨(a*c<=0)∧(Res2=-a*c-b-d)
ExpRes23=(b>d)∧(Res3=b+a)∨(b<=d)∧(Res3=d+a)

Prog3:

Loc	Var	Exp
1	o1	b*c*d
2	o2	c-b-d
3	mi	a
4	mi	c
5	ma	b
6	ma	d
7	o3	mi+ma
8	res1	o1
9	res2	o2
10	res3	o3

CD	CT	CF	END
a<c	3	4	5
b>d	5	6	7
res1=o1	8	-1	9
res2=o2	9	-1	10
res3=o3	10	-1	11

ExpRes31=(Res31=b*c*d) ;
ExpRes32=(Res2=c-b-d)
ExpRes33=(b>d)∧((a<c)∧ (Res3=a+b)∨(a>=c)∧(Res3=c+b))∨
(b<=d)∧((a<c)∧ (Res3=a+d)∨(a>=c)∧(Res3=c+d))

In a real problem, we have not the expression of the expected functions, but instead we have fitness cases. So, let's suppose that after executing all fitness cases, we find that each individual Progi is the best in computing the output oi computing. Hence, let's compute Prog1$\bowtie_1$Prog2=(Prog1,Prog2')

We will then have the following Result Expressions for Prog2':
ExpRes21'=ExpRes11;
ExpRes22'=ExpRes22;
ExpRes23'=ExpRes23

Now let's compute Prog3$\bowtie_3$Prog2'=(Prog3,Prog2''). We will then have the following expressions ExpRes for Prog2'':
ExpRes21"=ExpRes11;ExpRes22"=ExpRes22;ExpRes23"=ExpRes33.
The translation of this set of Result Expressions gives:

Loc	Var	Exp
1	o1	a*c*d
2	o1	b*c*d
3	o2	a*c-b-d
4	o2	-a*c-b-d
5	o3	a+b
6	o3	c+b
7	o3	a+d
8	o3	c+d

CD	CT	CF	END
a>b	1	2	3
a*c>0	3	4	5
b>d	5	7	9
a<c	5	6	7
a<c	7	8	9

This corresponds to the following C-program:

```
If  (a>b)
      o1=a*c*d
else o1= b*c*d
if (a*c>0)
       o2=a*c-b-d
else  o2= -a*c-b-d
if (b>d)
    if (a<c)
          o3=a+b
    else o3=c+b
else if (a<c)
           o3=a+d
      else o3=c+d
```

We remark that this program, automatically generated from Prog1, Prog2 and Prog3, computes the three output o_1, o_2 and o_3.

4. CONCLUSION

We have presented an original and efficient approach to program improvement in multi-objective genetic programming. The modeling we have used is as simple as effective. Our work presents several contributions:

1. The computing of the fitness value with respect to each output is as realistic as advantageous: A same program could be fit in the computing of some output but unfit for another one.
2. Result Expression translation to a program is the simplest way to guarantee that the constructed program computes as desired the corresponding output. Furthermore, the translation is effortless.
3. Our crossover operator guarantees that the obtained program is effectively better than its predecessor.
4. All Result Expressions must be computed to run fitness cases, consequently, the crossover operator does not need further computations.

5. It is not an obvious task to decompose a program in such a way to obtain just the required treatments for the required output: which we performed in our work.
6. Backwards computations could be seen as an abstraction technique since in a backward investigation we track just the desired variables.
7. Every extra-treatment, i.e. statements which do not contribute in the computing of any output, will disappear in the final solution since they will not appear in the Result Expressions.

However, several future directions could be envisaged for this work: The first is the experimentation of the method on real world problems. The second consists to explore the use of other program's properties to create individuals verifying the desired behavior.

5. REFERENCES

[1] Banzhaf W., Beslon G.,Christensen S.,Foster J.,Kepe F., Lefort F.,Miller J.,Radman M., and Ramsden J. 2006. From artificial evolution to computational evolution: a research agenda. *Nat. Rev. Genet*.7(9), 729–735 (2006).

[2] Beadle L. and Johnson C.G. 2008. Semantically driven crossover in genetic programming. In *Proceedings of the IEEE World* Congress on Computational Intelligence, pages 111–116. IEEE Press, 2008.

[3] Brameier M.,and Banzhaf W. 2007. Linear Genetic Programming. No. XVI in *Genetic and Evolutionary Computation* (Springer, Berlin, 2007)

[4] Dijkstra E. 1976. A discipline of programming *Prentice-Hall,* 1976.

[5] Grumberg O., Clarke E.M.,and Peled D. 1999. Model Checking. MIT Press, 1999.

[6] Johnson C.G. 2002. Deriving genetic programming fitness properties by static analysis. In *Proceedings of the 4th* European Conference on Genetic Programming *(EuroGP2002),* pages 299–308. Springer,2002.

[7] Johnson C.G. 2002. Genetic programming with guaranteed constraints. In Recent Advances in Soft Computing, pages 134–140. The Nottingham Trent University, 2002.

[8] Johnson C.G. 2002. What can automatic programming learn from theoretical computer science. In *Proceedings of the UK* Workshop on Computational Intelligence. University of Birmingham, 2002.

[9] Johnson C.G. 2007. Genetic programming with fitness based on model checking. In Proceedings of the 10th European Conference on Genetic Programming (EuroGP2007), pages 114–124. Springer, 2007.

[10] Johnson C.G. Genetic programming crossover: Does it cross over? In Proceedings of the 12th European Conference on Genetic Programming (EuroGP2009), pages 97–108. Springer, 2009.

[11] Jones N.D., and Nielson F. 1995. Abstract interpretation: a semantics based tool for program analysis. Handbook of Logic in Computer Science, 1995.

[12] Katz G., and Peled D. 2008. Model checking-based genetic programming with an application to mutual exclusion. Tools and Algorithms for the Construction and Analysis of Systems, 4963:141–156, 2008

[13] Katz G., and Peled D. 2008. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. Automated Technology for Verification and Analysis, Lecture Notes in Computer Science, 5311:33–47, 2008.

[14] Keijzer M. 2003. Improving symbolic regression with interval arithmetic and linear scaling. In Proceedings of EuroGP'2003, pages 70–82. Springer-Verlag, April 2003.