ABSTRACT
This paper presents a new crossover operator for genetic programming. We exploit two concepts of formal methods: Weakest precondition and Craig interpolation, to perform semantically aware crossover. Weakest preconditions are used to locate “faulty” parts of a program and Craig interpolation is utilized to “correct” these ones.

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

General Terms
Algorithm; Languages; Design; Performance.

Keywords
Genetic Programming; Craig Interpolant, Semantic Crossover, Symbolic Execution, Weakest Precondition.

1. INTRODUCTION
Genetic Programming (GP) 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. It develops programs, usually represented in memory as trees. Trees are recursively evaluated. According to traditional GP favors predominantly the use of programming languages that naturally embody tree structures such as functional programming languages [4]. 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. Formal methods are a class of mathematically based techniques for the specification, development and verification of software and hardware systems. [7, 8, 9, 10] are the first works which pioneered this area of research for GP. Two kinds of formal techniques have been used in GP, abstract interpretation [12] and model checking [6]. Abstract interpretation performs analysis on abstract domains instead of concrete ones. In [7, 8], abstract interpretation is used as a measure of the fitness. Model checking is an algorithmic technique to verify a system description against a specification represented as a temporal logic formula. In [10], a system is modeled by a set of temporal logic formulas. The fitness function is measured by counting the number of satisfied formulas. 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 [1], a new framework for multi-objective genetic programming is presented. A semantic crossover operator is defined. Our paper is related to the work presented in [1]. In this paper, we use the same program representation and execution procedure as in [1], and we define a new recombination operator based on Craig interpolation [16]. We call it CIC: Craig Interpolant based Crossover. Craig interpolation is a concept of mathematical logic that relates inconsistent formulas. Intuitively an interpolant Â of a pair (A, B), of inconsistent formulas, is a weakening of A which uses only the common variables of A and B, and which is inconsistent with B. Â explains and summarizes why A is inconsistent with B in their shared language. Craig interpolants have shown a great success in program model-checking. In this work, we use them to define semantic aware crossover operator. Weakest preconditions are used to locate faulty parts of a program Pi, and Craig interpolation is exploited to explain why Pi fails in computing a given fitness case c while another program Pj succeeds; and to take advantage of Pj to improve Pi. . We are not aware of any work presenting this idea. Craig interpolants have been used very successfully in software verification and testing techniques and we they could be also appropriate for automatic program generation. , this is the main contribution of this paper.

The rest of the paper is organized as follows: Section 2 introduces some preliminaries essential for the reading of this paper. It begins by introducing the notion of Craig interpolation, and recalls briefly program modeling and symbolic executions defined in [1]. In the section 3, we describe our crossover operator. An illustrating example is presented in the section 4. Finally, section 5 concludes by highlighting contributions of this paper and exposing some of its possible applications.
2. PRELIMINARIES

2.1 Craig Interpolation

A Craig interpolant for a mutually inconsistent pair of formulas (A, B) is a formula that is (1) implied by A, (2) inconsistent with B, and (3) expressed over the common variables of A and B. An interpolant can be efficiently derived from a refutation of A\(\land\)B [16], for certain theories and proof systems. Interpolants can be derived from resolution proofs in propositional logic, and for systems of linear inequalities over the real numbers. These methods have recently been extended to combine linear inequalities with uninterpreted function symbols, and to deal with integer models. One key aspect of these procedures is that they yield quantifier-free interpolants when the premises A and B are quantifier-free[16].

Definition (Craig interpolant [16]): For a pair of formulas (A, B) such that A\(\land\)B is unsatisfiable, an interpolant \(\tilde{A}\) is a formula referring only to common variables of A and B such that A\(\Rightarrow\tilde{A}\) and \(\tilde{A}\Rightarrow\neg B\).

Craig showed that for first-order formulas, an interpolant always exists for inconsistent formulas. A more practical interest is that \(\tilde{A}\) interpolant can be efficiently derived from a refutation of A\(\land\)B in linear time. In [16] it is shown that linear-size interpolants can be derived from refutations in a first-order theory with uninterpreted function symbols and linear arithmetic. This translation has the property that whenever A and B are quantifier-free formulas, the derived interpolant \(\tilde{A}\) is also quantifier-free[16].

Examples: Let A and B be two formulas such that:

1. A = p\(\land\)q and B = \neg q\(\land\)r : A\(\land\)B is unsatisfiable. \(\tilde{A}=q\) is an interpolant for the pair of formulas (A, B) since: \(\tilde{A}\) refers only to common variables of A and B ; p\(\land\)q \(\Rightarrow\tilde{A}\) and \(\tilde{A}\Rightarrow\neg q\land r\).

2. A = (x>y)\(\land\)(z=x-y) B = (z=0).

\(\tilde{A}=(z>0)\) In fact : since x>y then x-y>0. We remark that the term (z=0) does not exist explicitly in A, but it is a consequence of A. This is one of the notable features that attract us in Craig interpolation since it is not based only on syntax.

3. A = (0<x, x=1+2x, 0<=x, x+1) B = (0<=x, x-1, 0<=x, x+1)

\(\tilde{A}=(0<=4x+1)\)

In this example, taken from [2], we see that Craig interpolants are directly related to semantic and not syntax. In fact, we cannot deduce or presume \(\tilde{A}\) only by observing the syntax of A and B.

2.2 Program Modeling

As we said before, we use the same program representation as in [1], so, let’s recall it briefly. A program is represented by two tables: The Variable Table (VT), records the different expressions allowing computing program variables values. The second: Architecture Table (AT) describes the program structure. Each row of VT is a triplet (I, v, e). Where I is the location of the assignment statement in the program, v is the name variable and e the expression allowing computing the value of v. An expression could be: An input, a constant or a call to a function. AT models conditionals and loop statements. A conditional statement is represented 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 and End is the location of the first instruction after the conditional statement. Ina simple conditional statement (without the Else branch) CF is set to (-1). A loop is represented in a same manner, where CF is set to (-2).

![Figure 1. A program, its representation and its symbolic Execution[1]](image)

2.3 Symbolic Execution

Each individual of the population must be evaluated by executing it with all fitness cases. In [1], a special variable called Result expression and noted ExpRes, is associated to each output variable \(o_i\). It summarizes the expression of the output variable \(o_i\) in terms of inputs. Programs are executed symbolically by computing successive weakest preconditions, to compute the result expression of each output variable.

Definition (Weakest Precondition): 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. In [1] weakest precondition is defined for conditional statements and loops. The figure 1 presents an example program, its modeling and its symbolic execution, for more details see [1].
Definition (Exclusive Form Formula [1]): 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 \( \land \) nor \( \lor \))
- \( F \) is of the form \( C \land P \lor \neg C \land Q \), where \( P \) and \( Q \) are two exclusive form formulas

Weakest precondition computations result in an exclusive form formula. So result expressions provide as a result an exclusive form formula.

3. INTERPOLATION BASED Crossover

As usual in genetic programming some good individuals are selected for recombination. Traditionally, recombination is performed randomly by swapping sub-trees of some chosen individuals. So, there is no semantic justification of the operation, nor guarantees that the obtained offspring is effectively better than its predecessors. We use Craig interpolation to address this issue.

If a program does not compute well a fitness case, we use its Result expression and that of another program which computes correctly the same fitness case, to explain ‘faulty’ parts of the program. To attain this achievement, we generate in an adequate manner, an interpolant for the pair of formula constituted by the two Result Expressions. Then, we perform the suitable modifications on the faulty program so that it computes the output as required. So, weakest preconditions are utilized to execute programs and to locate parts of program which are targeted by a given input values; while Craig interpolants are exploited to detect inconsistencies and to improve programs. The algorithm computing the crossover of two programs is given in the figure 2.

Let \( Prog_f \) and \( Prog_u \) be two programs and let \( f \) be a fitness case, such that \( Prog_f \) computes well \( f \) and \( Prog_u \) fails to compute it. Each fitness case corresponds to a possible instantiation of the tuple \( (x_1,x_2,\ldots,x_n,y_1,y_2,\ldots,y_k) \) where \( x_1,\ldots,x_n \) are the input variables and \( y_1,\ldots,y_k \) the output variables. Since for each output variable is associated a result expression independently of the other output variables, then we consider only the output variable we are interested in. So, we consider tuples of the form \( (x_1,x_2,\ldots,x_n,y) \). In the subsequent, a fitness case \( f \) has the form: \( \langle (1,2,\ldots,n,w), \text{ where } v_1,\ldots,v_n \text{ are the input values and } w \text{ is the corresponding output.} \)

Let \( RE_f \) and \( RE_u \) be the Result Expressions corresponding, respectively, to the “fit” program: \( Prog_f \) and “unfit” program: \( Prog_u \). As we stipulate in the previous section, \( RE_f \) and \( RE_u \) are in the exclusive form. So, they have the form:

\[
\begin{align*}
RE_f &= C_1 \land P_1 \lor \neg C_1 \land Q_1 \\
RE_u &= C_2 \land P_2 \lor \neg C_2 \land Q_2
\end{align*}
\]

Where \( P_1, Q_1, P_2 \) and \( Q_2 \) are all in the exclusive form.

Let \( C(f) = C[x_i/v_i] \) be the condition \( C \) where each variable \( x_i \) is substituted by the corresponding value \( v_i \) of the fitness case \( f \). \( C(f) \) is either true or false. So, in each case, only one branch of the Result Expression is true: we call it \( AB(RE,f) : \text{‘Activated Branch’} \) in the result expression \( RE \) for the fitness case \( f \).

\[
\begin{align*}
AB(C \land P \lor \neg C \land Q,f) &= \begin{cases} 
C \land AB(P,f) & \text{if } C(f)=\text{True} \\
\neg C \land AB(Q,f) & \text{else}
\end{cases}
\end{align*}
\]

If \( P \) is an atomic formula \( AB(P,f)=P[x_i/v_i] \)

Algorithm CIC(Prog_f, Prog_u, f)

1: Input: RE_f, RE_u : result expressions of Prog_f and Prog_u
   \( RE_f=C_1 \land P_1 \lor \neg C_1 \land Q_1 \)
   \( RE_u=C_2 \land P_2 \lor \neg C_2 \land Q_2 \)
   \( f=(v_1,v_2,\ldots,v_n,w) \) a fitness case.

2: Output: RE'_f, RE'_u

3: Initializations: Let \( AB_f=C_1 \land P_1 \); \( AB_u=C_2 \land P_2 \ :
   \text{Activated branches, for the fitness case } f \text{, respectively in } RE_f \text{ and } RE_u\)
   Let \( AB_f'=AB_f\land(r=w') \); \( AB_u'=AB_u\land(r=w') \)

4: Compute the interpolant \( A \) of \( (AB_f',AB_u') \)

5: Transform \( AB_u' \):
   5.1. Let \( X \) the formula such that \( AB_u=X \land (r=e_1) \)
   5.2. Let \( Y \) the formula such that \( \Lambda=Y \land (r=e_1) \)
   5.3. \( AB_u''=X \land (Y \land (r=e_1) \lor \neg Y \land (r=e_0)) \)

6: Replace \( AB_u' \) by \( AB_u'' \) in \( RE_u \), so: \( RE_u'=RE_u[AB_u'/AB_u''] \)

7: End.

Figure 2. Algorithm CIC(Prog_f, Prog_u)

So, each fitness case activates some parts of a program and does not execute at all other parts. Consequently, for each fitness case, we are not interested in the whole program but only on activated branches. \( AB(RE,f) \) is determined when executing symbolically the program. Let’s remark that a true execution does not allow distinguishing, in a program, which parts are executed. So, this is an advantage of performing symbolic executions instead of true ones. In the algorithm, we initialized \( AB(RE,f) \) with \( C \land P \), just to lighten the notation (it could be \( \neg C \land Q \)). \( AB_f' \land AB_u'' \) is unsatisfiable as it contains: \( (r=w) \land (r=w') \) which is unsatisfiable since \( Prog_u \) does not compute correctly the fitness case \( f \) so \( w \neq w' \)

We recall that the variable \( r \) is used to store the value of the result (i.e. the output value). More details about symbolic execution and result expressions are given in [1]. The translation of \( RE_u'' \) in an individual represented by its two tables VT, and AT is straightforward; the translation procedure is explained in [1].

4. ILLUSTRATING EXAMPLE

To illustrate our approach, we present how it works on an example. Let \( Prog_1 \) and \( Prog_2 \) be two individuals. We present the tree-based representation for more clarity and to show that in classical GP, (tree-based GP) whatever transformations (crossover and mutation) can never achieve this

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
VT & AT \\
\hline
Lo & Var & Exp & Cd & C & CF & En \\
\hline
1 & L & j-k & \multicolumn{3}{c|}{L} & \multicolumn{3}{c|}{1} & 4 & 5 \\
2 & L & j+k & \multicolumn{3}{c|}{L} & \multicolumn{3}{c|}{j-k} & \multicolumn{3}{c|}{1} & 2 & 4 \\
3 & L & K & \multicolumn{3}{c|}{L} & \multicolumn{3}{c|}{L} & \multicolumn{3}{c|}{J} & \multicolumn{3}{c|}{1} & 2 & 3 \\
4 & R & L & \multicolumn{3}{c|}{L} & \multicolumn{3}{c|}{R} & \multicolumn{3}{c|}{L} & \multicolumn{3}{c|}{r=k} & \multicolumn{3}{c|}{1} & 4 & -1 & 5 \\
\hline
\end{tabular}
\end{table}

\textbf{Prog 1}

\begin{align*}
&\text{if}(j>e) \\
&\text{if}(e<k) \\
&\text{if}(j<k) \\
&\text{else} \\
&\text{else} \\
&\text{else} \\
&\text{else}
\end{align*}

\begin{align*}
&\text{if}(r==l) r=l; \\
&\text{l=k; else} \\
&\text{else} \\
&\text{else} \\
&\text{else} \\
&\text{else}
\end{align*}

\textbf{Prog 2}

\begin{align*}
&\text{if}(j>e) \\
&\text{if}(e<k) \\
&\text{if}(j<k) \\
&\text{else} \\
&\text{else} \\
&\text{else} \\
&\text{else}
\end{align*}

\begin{align*}
&\text{if}(r==l) r=l; \\
&\text{l=k; else} \\
&\text{else} \\
&\text{else} \\
&\text{else} \\
&\text{else}
\end{align*}
Prog 2

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>If (k&gt;0)</td>
<td>VT</td>
<td>AT</td>
<td></td>
</tr>
<tr>
<td>1: l=j</td>
<td>L</td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td>2: l=k</td>
<td>L</td>
<td>K</td>
</tr>
<tr>
<td></td>
<td>Else</td>
<td>3: l=j*k</td>
<td>L</td>
</tr>
<tr>
<td>4: If(r==l) r=l;</td>
<td>4: R</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1. Result Expressions Computing:

RE1 = (j>e) ∧ (e>k) ∧ (j+k<s) ∧ (r=j-k) ∨ (j>0) ∧ (r=k) ∨ (j<=e) ∧ (r=k-1)

RE2 = (k>0) ∧ (r=j) ∨ (k<=0) ∧ (j<2) ∧ (r=k) ∨ (j>=2) ∧ (r=j*k)

Let f1 be the fitness case (e=0, j=1, k=-2, l=3, s=10). Prog1 computes well f1; while Prog2 fails. So, for this fitness case, REf = RE1 and REu = RE2

2. CIC(Prog2, Prog1, f):

- Activated Branches Computing:
  
  ABf = (j>e) ∧ (e>k) ∧ (j+k<s) ∧ (r=j-k)
  
  ABu = (k<=0) ∧ (j<2) ∧ (r=k)

  So, we have located the “faulty” part of Prog2

- ABf' and ABu' computing:
  
  ABf' = (j>e) ∧ (e>k) ∧ (j+k<s) ∧ (r=j-k)
  
  ABu' = (k<=0) ∧ (j<2) ∧ (r=k)

  ABf' ∧ ABu' is unsatisfiable.

- Craig interpolant computing:
  
  An Interpolant of (ABf', ABu') is Â = (j>k) ∧ (r=j-k)

  In fact:
  
  1- (j>e) ∧ (e>k) ∧ (j+k<s) ∧ (r=j-k) ⇒ (j>k) ∧ (r=j-k)
  
  2- (j>k) ∧ (r=j-k) and (k<=0) ∧ (j<2) ∧ (r=k) are inconsistent since in (r=j-k) r is positive while it is negative in (r=k).
  
  3- Â uses only common variables of ABf' and ABu'. This property excludes the terms: (j+k<s) since the variable s does not appear in ABu', and (j>e) since ABu does not contain the variable e. An important feature of Craig interpolant is that they track down semantic relations between formulas rather than just syntactic ones. For example in the expression (j>e) ∧ (e>k) ∧ (j+k<s) ∧ (r=j-k) we have not an explicit comparison between the variables j and k, even so, the semantic relation between j and k is captured in (j>k) by the interpolant Â

- ABf Transformation
  
  Let X=(k<=0) ∧ (j<2), so ABf" = X ∧ (r=k)
  
  Let Y = (j>k), Â = Y ∧ (r=j-k)
  
  ABu" = X ∧ (Y ∧ (r=j-k) ∨ ¬Y ∧ (r=k))
  
  = (k<=0) ∧ (j<2) ∧ ([j>k] ∧ (r=j-k) ∨ (j<=k) ∧ (r=k))

- REu Transformation : replace ABu by ABu" in REu
If we assume that the activated branch of Prog2 was correct, then, Prog1 is corrected for all the fitness cases where $k < j < 2$

5. CONCLUSION

We have presented a new approach for program improvement in genetic programming. The main contribution of this paper is the using of Craig interpolation in genetic programming. Craig interpolation is a notion which has recently demonstrates its efficiency in software verification and testing area. We are optimistic that this concept could bring some new methodologies in the genetic programming research domain. In this paper, our objective was to introduce this idea and to highlight its suitability in this domain.

Some Possible Applications of the Approach: Since Craig interpolants are related to logical formulas, our approach could be applied to problems where several situations are possible depending on the properties checked by the input data. So, for example, all the problems commonly treated by decision trees, classification trees, regression trees, could be good examples of using this method. Indeed, a binary tree is easily expressible by a program consisting of a set of IF-THEN-ELSE statements. Furthermore, This is particularly desirable since, in these trees, conditions of nodes are conjunctions and disjunctions of several formulas, which is well-handled by Craig interpolation.

Another possible application is in symbolic regression to find logical expressions using the three functions: AND, OR, NOT. For example for electronic circuits.

6. REFERENCES


