ЭВОЛЮЦИОННЫЙ ПОДХОД К ФУНКЦИОНАЛЬНОМУ ТЕСТИРОВАНИЮ ЦИФРОВЫХ СХЕМ

Скобцов Ю.А.
Донецкий национальный технический университет, кафедра АСУ
Иванов Д.Е.
Институт прикладной математики и механики, отдел ТУС
E-mail: skobtsov@iamm.ac.donetsk.ua, ivanov@iamm.ac.donetsk.ua

Abstract
Skobtcov Y.A., Ivanov D.Y. Evolutionary approach for digital circuits functional testing. *In paper an evolutionary approach for functional testing of digital circuits is considered. A genetic algorithm for digital multiplier testing is proposed. A construction of genetic algorithm is proposed: coding of individual, population and input sequence, building of genetic operators (selection, crossingover, mutation). Experimental data of program realization is also presented.*

Особенность идей теории эволюции и самоорганизации заключается в том, что они находят свое подтверждение не только для биологических систем, успешно развивающихся многие миллиарды лет. В настоящее время бурно развивается новое направление в теории и практике искусственного интеллекта эволюционные вычисления – термин, обычно используемый для общего описания алгоритмов поиска, оптимизации или обучения, основанных на некоторых формализованных принципах естественного эволюционного отбора. Эволюционные вычисления используют различные модели эволюционного процесса, которые отличаются различными формами представления решений и генетическими операторами. Данный подход с успехом использовался при тестировании цифровых схем [1,2]. В настоящей работе рассматривается его применение к функциональному тестированию цифровых схем.

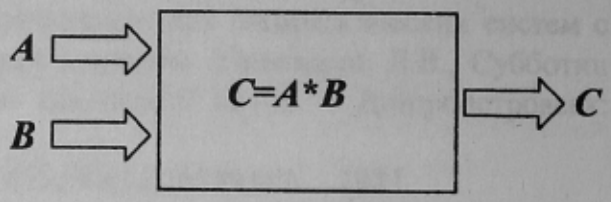


Рисунок.1 Умножитель.

В качестве примера рассмотрим построение функционального теста для комбинационной схемы – умножителя, которая представлена на рис.1. Для простоты ограничимся случаем, когда операнды *A*, *B* и результат *C* являются целыми числами. Необходимо построить функциональный тест, проверяющий данный умножитель, по возможности минимальной длины. При решении этой задачи используется генетический алгоритм.

Генетические алгоритмы (ГА) [3], являясь одной из парадигм эволюционных вычислений, представляют собой алгоритмы поиска, построенные на принципах, сходных с принципами естественного отбора. ГА используют случайный направленный поиск для построения (суб)оптимального решения данной проблемы. В ГА из всего пространства поиска выделяется некоторое множество точек этого пространства (потенциальных решений), которое в терминах натуральной селекции и генетики называется популяцией. Каждая особь популяции – потенциальное решение задачи, представляется хромосомой – структурой элементов – генов. В простейшем случае особью может быть двоичная строка (например, 0011101). Это делает ГА привлекательным для решения задач генерации проверяющих тестов логических схем, где решение задачи представляется в виде двоичных наборов или их последовательностей. На множестве решений определяется целевая (fitness)

функция (ЦФ), которая позволяет оценить близость каждой особи к оптимальному решению – способность к выживанию. Генетический алгоритм поиска решения заключается в моделировании эволюции подобной искусственной популяции. Популяция развивается (эволюционирует) от одного поколения к другому. Создание новых особей в процессе работы алгоритма происходит на основе моделирования процесса размножения. В каждом поколении множество особей-потомков создается, используя части особей-родителей и добавляя новые части с «хорошими свойствами». При этом ГА эффективно используют информацию, накопленную в процессе эволюции.

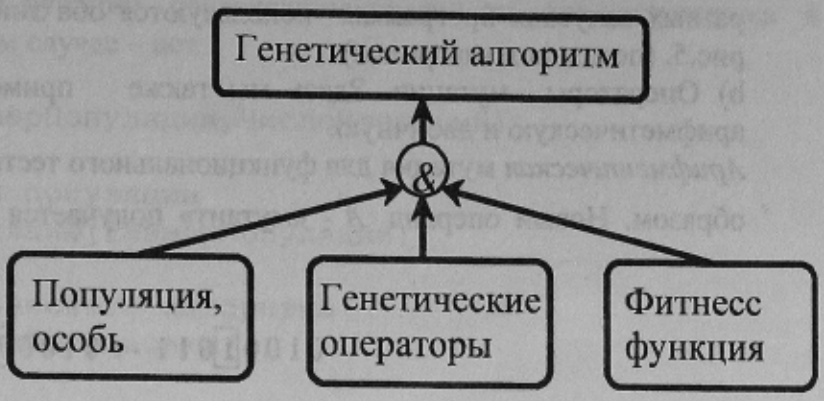


Рисунок.2 Построение генетического алгоритма.

Для применения ГА к конкретной задаче необходимо прежде всего: 1) выбрать форму представления (способ кодирования) решения - хромосомы; 2) определить на этом представлении генетические операторы кроссинговера и мутации; 3) определить фитнесс-функцию, которая позволяет оценить качество решения (его близость к оптимальному) (рис.2).

Далее мы рассмотрим эти аспекты применительно к нашей задаче.

1) Очевидно, что один тестовый набор образуют два целых числа A,B. Поэтому решение представляется вектором целых чисел (A,B) из двух компонент – значений операндов. Наряду с этим мы также будем использовать и двоичный код этого вектора.

2) При решении задачи используются два типа генетических операторов:

а) Операторы кроссинговера. Мы применяем два вида этого оператора: арифметический и стандартный двоичный.

Арифметический кроссинговер для функционального тестирования выполняется следующим образом. Для двух родителей (операндов) A и B потомок $\widetilde{A}$ (новое значение операнда) находится согласно

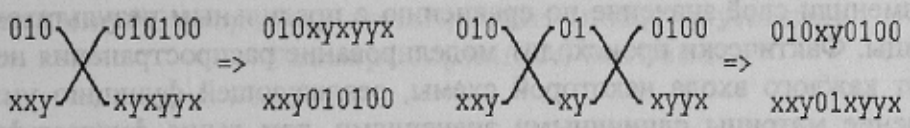


Рисунок.3. Одноместное и двуместное скрещивание.

$\widetilde{A} = (1-\alpha) * A + \alpha * B$, где $\alpha \in (0;1)$ - параметр.

Двоичный кроссинговер выполняется по стандартной схеме: для родителей – двоичных кодов операндов A и B потомки – новые коды двоичных операндов $\widetilde{A}, \widetilde{B}$ получаются следующим образом (Рис.3).

При этом каждый тип оператора выполняется со своей вероятностью P_b^c (двоичный) и P_a^c (арифметический). Для них справедливо равенство $P_b^c + P_a^c = 1$. Отметим, что при

разных запусках программы используются оба типа оператора, как это показано на рис.5. (псевдокод алгоритма).

b) Операторы мутации. Здесь мы также применяем два вида этого оператора: арифметическую и двоичную.

Арифметическая мутация для функционального тестирования выполняется следующим образом. Новый операнд $A^{'}$ - «мутант» получается из старого значения A согласно

$$0100\boxed{1}011 \rightarrow 0100\boxed{0}011$$

Рис.4 Битовая мутация.

формуле $A^{'} = A \pm \Delta * A$, где Δ - малое число.

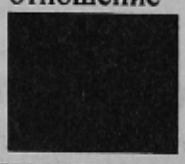
Двоичная мутация выполняется по стандартной схеме (Рис.4)

При этом, как и в предыдущем случае, каждый тип мутации выполняется со своей вероятностью P_b^m (двоичный) и P_a^m (арифметический) и $P_b^m + P_a^m < 1$.

3) Фитнесс функция на предварительном этапе для потенциального тестового набора вычисляется следующим образом. Для каждого изменения двоичного разряда текущего тестового набора оценивается число изменений в двоичных разрядах результата (произведения). Эксперимент показал, что всегда можно найти тестовый набор (пару целых чисел), что изменение любого разряда операнда влияет хотя бы на один разряд произведения.

Нашей целью является построение множества тестовых наборов, на которых на изменение входных разрядов реагируют максимальное число разрядов произведения. В идеале (полнота - 100%). Необходимо построить множество входных наборов (пар чисел), в котором изменение каждого входного набора влияет на каждый разряд произведения.

Таким образом мы можем составить матрицу с размерностью *(2*N,M)*, где *N* – разрядностьоперанда и *M* - разрядность_результата. Элемент матрицы a_{ij}=1 в том случае, если инверсия *i*-го входного бита влияет на значение *j*-го бита результата. Матрица заполняется следующим образом. В начальный момент времени все элементы матрицы равны 0. Из входной последовательности выбирается очередная пара операндов. В ней последовательно изменяется каждый входной бит. В зависимости от того, какие из битов результата изменили своё значение по сравнению с правильным результатом, заполняются ячейки матрицы. Фактически происходит моделирование распространения неопределённого значения *u* от каждого входа некоторой схемы, реализующей функцию умножителя. Чем выше заполнение матрицы единичными значениями, тем выше фитнесс-функция данной входной последовательности. Таким образом, в качестве фитнесс-функции входной последовательности операндов выбрано отношение



Далее рассмотрим генетический алгоритм построения функционального теста в целом, псевдокод которого приведен на рис.5. Построение входной тестовой последовательности из одиночных операндов происходит в функции «ДобавитьЛучшийВходВТест()». После очередной итерации генерации нового поколения происходит изменение входной тестовой последовательности. Если при добавлении лучшей особи из популяции (пары операндов)

происходит увеличение фитнесс-функции последовательности, то она включается в результирующий тест. В противном случае – нет.

```
ГенетическийАлгоритм(РазмерПопуляции,ЧислоИтераций)
{
  // Построение стартовой популяции
  ПостроитьНачальнуюПопуляцию(РазмерПопуляции);
  НомерПоколения=0;
  // Основной цикл генетического алгоритма
  while(НеДостигнутКритерийОстановки())
  {
    НоваяПозиция=0;
    for( int i=0 ; i< РазмерПопуляции ; ++i )
    {
      // выбор родителей для генетических операций
      Селекция(РодительА,РодительБ);
      // скрещивание особей
      Скрещивание(РодительА,РодительБ,Потомок);
      // мутация особи
      Мутация(Потомок);
      // добавить потомка в промежуточную популяцию
      ДобавитьВПромежуточнуюПопуляцию(Потомок, НоваяПозиция);
      ++ НоваяПозиция;
    }
    ПостроитьНовоеПоколение(РазмерПопуляции);
    ДобавитьЛучшийВходВТест();
    // увеличить номер поколения
    ++ НомерПоколения;
  }
  // печать отчёта
  Отчёт();
} // конец генетического алгоритма
Селекция(РодительА,РодительБ);
{
  ПересчитатьФитнесс(Популяция,РазмерПопуляции,Фитнесс);
  ПропорциональныйОтбор(РодительА,РодительБ,Популяция,
                        РазмерПопуляции,Фитнесс);
}
Мутация(Потомок);
{
  // выбор с малой вероятностью - производить ли мутацию
  if( НеобходимаМутация() )
  {
    // выбор схемы мутации
    ВыбратьСхемуМутации();
    if( ФункциональнаяМутация )
    {
      ФункциональнаяМутация(Потомок);
    }
    else
    {
```

```
      БитоваяМутация(Потомок);
    }  }}
Скрещивание(РодительА,РодительБ,Потомок);
{  // выбор схемы для оператора скрещивания
  ВыбратьСхемуСкрещивания();
  if( ФункциональноеСкрещивание )
  {
    ФункциональноеСкрещивание(РодительА,РодительБ,Потомок);
  }
  else
  { БитовоеСкрещивание(РодительА,РодительБ,Потомок); }
}
ПостроитьНовоеПоколение(РазмерПопуляции);
{
  // построить временную популяцию объединением
  // основной и промежуточной популяций
  ОбъединитьПопуляции(Популяция,ПромежуточнаяПопуляция);
  // отсортировать временную популяцию по убыванию фитнесс
  СортироватьВременнуюПопуляциюПоФитнесс();
  // выбрать в качестве популяции для следующей итерации ГА
  // лучших особей из временной популяции
  КопироватьЛучшихОсобей(РазмерПопуляции);
}
ФункциональнаяМутация(Потомок)
{
  ВыбратьМалоеАльфа();
  Потомок=Потомок±Альфа*Потомок;
}
ФункциональноеСкрещивание(РодительА,РодительБ,Потомок)
{
  ВыбратьМалоеАльфа();
  Потомок=(1-Альфа)*РодительА+Альфа*РодительБ;
}
```

Рисунок 5 Псевдокод ГА.

Описанный выше алгоритм был реализован программно на языке C++ в среде C++ Builder. Исходный текст программы составил около 1000 строк. Экспериментальным путём определены значения $P_{мут}$=0.01, $Alpha_{скр}$=0.5, $Alpha_{мут}$=0.5, число особей в популяции - 100. С этими параметрами проводилось исследование зависимости фитнесс-функции последовательности от числа поколений. Экспериментальные данные, представленные на рис.6, показывают, что значение фитнесс-функции (полнота теста) стабилизируется достаточно быстро. На этом основании выбрано граничное значение числа поколений

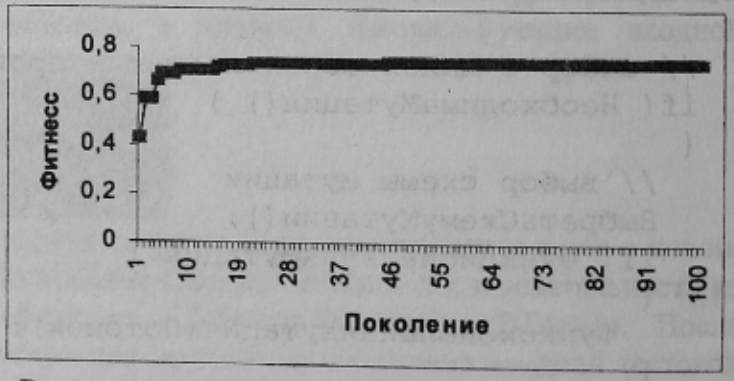


Рисунок 6. Рост фитнесс-функции в зависимости от поколения.

40. Далее проводились эксперименты по выявлению зависимости полноты теста от разрядности операндов. Средние данные по 10 экспериментам приведены на рис.7., где представлена зависимость длины полученного теста от разрядности. В табл.1 в качестве примера приведен один из полученных функциональных тестов из 11 наборов для 32-разрядного умножителя. В настоящее время проводятся исследования по определению полноты теста относительно одиночных константных неисправностей логической схемы на вентильном уровне, которая реализует 32-разрядный умножитель.

Таким образом, в данной работе исследован эволюционный подход к построению функциональных тестов на примере умножителя. Фактически выполнена формализация эвристических приемов построения функциональных тестов – «фольклора» тестирования цифровых систем.

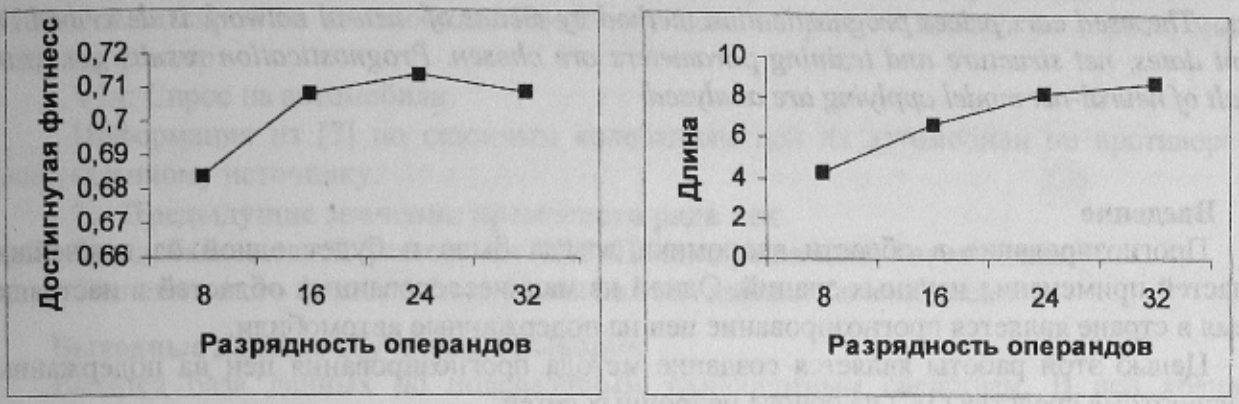


Рисунок 7. Зависимость достигнутой фитнесс-функции и длины тестовой последовательности от разрядности операндов.

Таблица1. Пример теста в десятичном и двоичном виде.

В десятичном виде		В двоичном виде	
A	B	A	B
1373907781	127562156	01010001111001000010101101000101	01001100000010000111000010111001
-254785975	376343075	11110000110100000100011001001001	00010110011011101000101000100011
1630710233	579247396	01100001001100101010100111011001	00100010100001101001110100100100
1373907165	1277006115	01010001111001000010100011011101	01001100000111011001000100100011
13649481	1382976035	00000000110100000100011001001001	01010010011011101000101000100011
67977233	70853037	00000100000011010100000000010001	00000100001110010010000110101101
122962474	458015755	00000111010101000100001000101010	00011011010011001100010000001011
1376166070	419570069	01010010000001101010000010110110	00011001000000100010000110010101
92684874	1309303734	00000101100001100100001001001010	01001110000010100110001110110110
26034540	646202154	00000001100011010100000101101100	00100110100001000100001100101010

Литература

1. P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms" In Proc. *Int. Test Conf.*, 1994, pp. 240-249.
2. E.M. Rudnick, J.H. Patel, G.S. Greenstein, T.M.Niermann, "Sequential Circuit Test Generation in a Genetic Algorithm Framework". In Proc. *Design Automation Conf.*, 1994, pp.40-45.
3. Goldberg D.E., Genetic Algorithm in Search, Optimization, and Machine Learning.- Addison-Wesley.- 1989.
4. podem RAPS (Abramovici)
5. Иванов Д.Е., Скобцов Ю.А. Ускорение работы генетических алгоритмов при построении тестов // Искусственный интеллект.- №1, 2001.- С.52-60.