GRAPH-BASED GP APPROACH TO MICROPROCESSOR SYSTEM TESTING

Ermolenko M.L.
Institute of Applied Mathematics & Mechanics NANU, Donetsk, Ukraine
E-mail: marker@court.gov.il

Abstract

Ermolenko M.L. Graph-based GP approach to microprocessor system testing. *Testing of microprocessor system is a critical issue not only because of their complexity, but also because of their specific characteristics to intensify all difficulties . Functional testing is an effective solution which consists in suitable test program executing by microprocessor. This paper presents an approach to automatic test program generation and implementation exploiting an genetic programming techniques based on the test program representation as direct graph . It overcomes the main limitation of previous techniques and provides significantly better results.*

Ермоленко М.Л. Подход к тестированию микропроцессорных систем, основанный на генетическом программировании, где программа представляется в виде ориентированного графа. *Тестирование микропроцессорных систем является серьезной проблемой не только из за их сложности, но также из-за специфических свойств этих систем, которые осложняют процесс тестирования. Функциональное тестирование является эффективным методом диагностики, который основан на выполнении тест-программы на ассемблере. В статье представлен подход к автоматической генерации тест-программ, основанный на генетическом программировании, где программа представляется в виде ориентированного графа. Этот подход позволяет преодолеть многие ограничения предыдущих методов и получить существенно лучшие результаты.*

1. Introduction to Graph GP

In this chapter a graph-based representation of GP programs will be introduced. It provides the flexibility to choose different execution paths for different inputs and to create programs with more higher complexity.

The program flow of a graph program is more natural than linear or tree GP-programs and similar to program flow of hand written programs.

In graph-based GP each program p is represented by a direct graph of Np nodes. Each node can have up to Np outgoing edges. Each node in the program have two parts , *sequential branch* and *conditional branch*. The *sequential branch* part is either a constant or a function which will be executed when the node is reached during the interpretation of the program. After the *sequential branch* of a node is executed an outgoing edge is selected according to the *conditional branch*. This decision is made by a *conditional function* which determines the edge to the next node .

Each program has two special nodes , a *start* and a *stop* node. The start node is always the first node to be executed when the interpretation of the program begins. After the *stop* node is reached ,its *sequential branch* is executed and the program halts. Since the graph structure inherently allows loops and recursion , it is possible that the stop node is never reached during the interpretation. In order to

avoid that a program runs forever it is terminated after a certain time threshold is reached. The threshold can be implemented as a fixed maximum number of nodes which can be executed during interpretation.

1.1 Recombination of Graph-based program

A crossover operator combines the genetic material from two parent programs by swapping certain program parts. The crossover for a graph based program can be realized in many ways. The first way is to perform the crossover by exchanging sub-graphs, it is similar to exchanging sub-trees in tree-based GP.

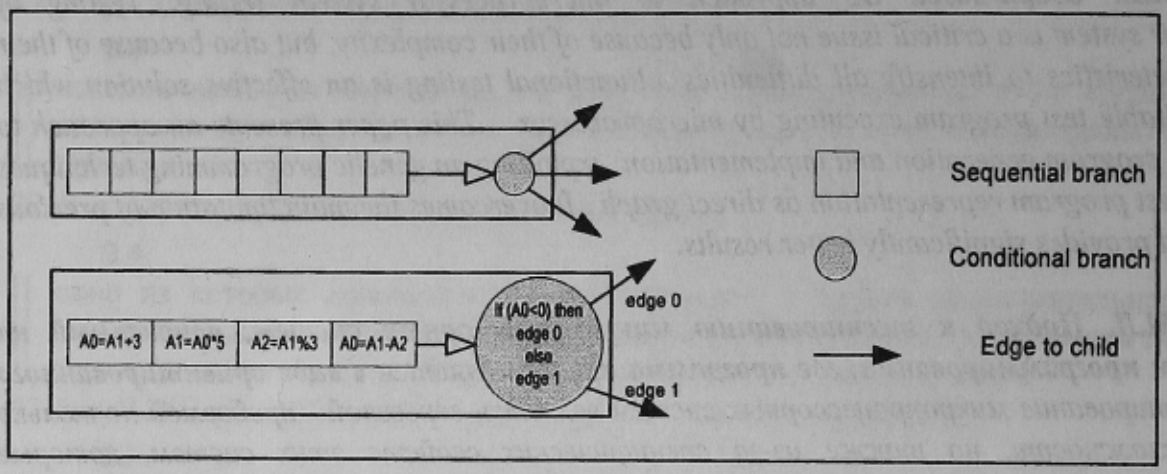

Fig. 1: The structure of a node in a graph-based GP program (top) and an example Node (bottom)

The basic crossover operator of graph-base GP, which called *Sub-graph crossover(SGC)*, is a generalization to graphs of the crossover used in GP to recombine trees. *SGC* crossover works as follows: 1) a random node is selected in each parent (crossover point); 2) a sub-graph including all the nodes which are used to compute the output value of the crossover point in every parent is extracted; 3) the sub-graph of the firs parent is inserted in the second parent to generate the offspring 4) the sub-graph of the second parent is inserted in the first parent to generate the offspring. An example of *SGC* crossover is shown in Figure 2. Obviously, for *SGC* crossover to work properly some care has to be taken to ensure that the depth of the sub-graph being inserted in the first or second parent is compatible with the maximum allowed depth. A simple way to do this is to select one of the two crossover points at random and choose the other with the coordinates of the first crossover point and the depth of the sub-graph in mind or if the depth of the sub-graph is too big for it to be copied into the second parent, the lowest nodes of the sub-graph are pruned to make it fit .

The idea behind this form of crossover is that connected sub-graphs are functional units whose output is used by other functional units. Therefore, by replacing a sub-graph with another sub-graph, we tend to explore different ways of combining the functional units discovered during evolution.

1.2 Mutation

The difference between crossover and mutation is that mutation operates on a single program only. After applying crossover operator to the population a program is chosen with a given probability for mutation. The random mutation operator select a subset of nodes randomly and change either a

node of a linear program, a branching function , or the number of outgoing edges. The altered program is then placed into the population.

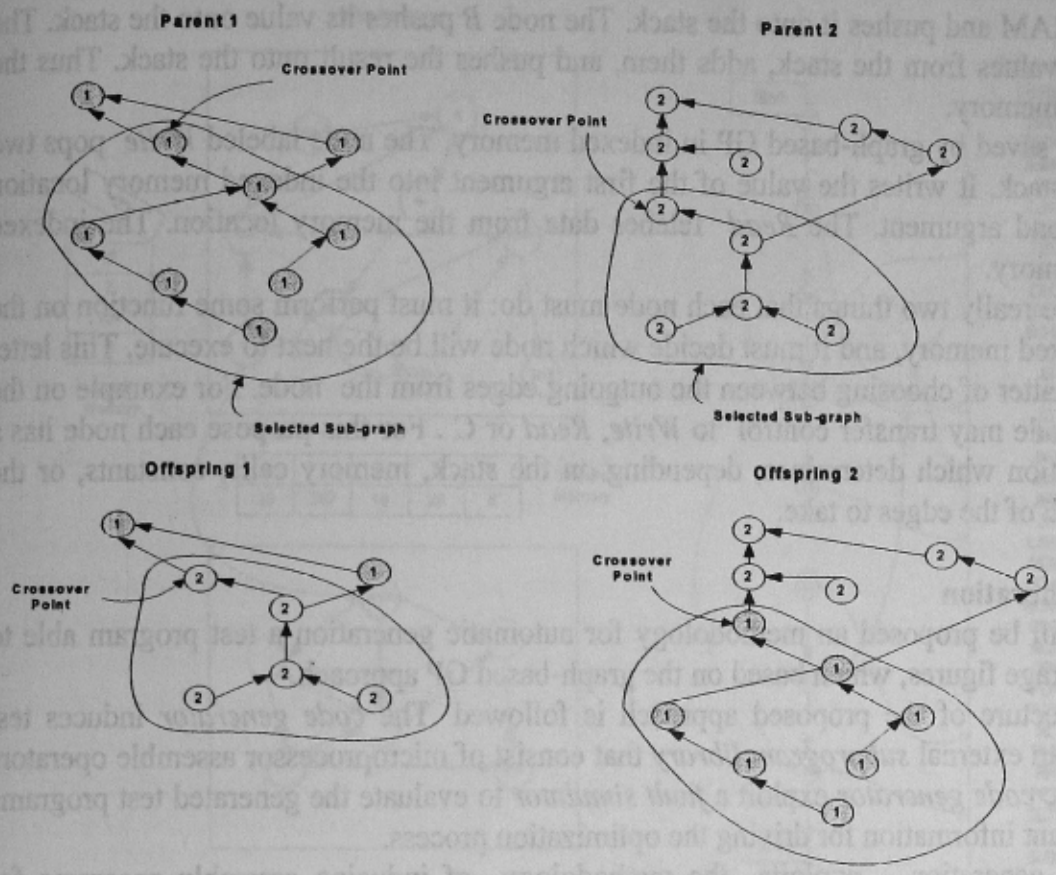

Fig. 2: Sub-graph crossover (SGC).

2. Programmatically representation of graph-based GP system

The graph-based GP system are capable of representing very complex program structures compactly. Figure 3 is a diagram of a small graph-based program.

Each program has a stack and an indexed memory for its own use of intermediate values and for communication. There are also the following special nodes in the program:

- Start node
- Stop node
- Subprogram calling nodes
- Library subprogram calling nodes

There are also parameters stating, for example, the minimum and the maximum time for a program to run.

Execution begins in the *Start* node. When the system hits the *End* node or another preset condition, execution is over. Thus, the flow of execution is determined by the edges in the graph. If a particular program stops earlier, it is simply restarted from the start node, which values accumulated in stack or memory reused. *Library subprograms* are available to all nodes, not just the one which is calling, whereas subprograms without that provision are for program's "private" use only.

Library subprogram calling nodes Execution begins in the *Start* node. When the system hits the *End* node or another preset condition, execution is over. Thus, the flow of execution is determined by the edges in the graph.

Like all GP system ,graph-based GP system needs memory to give its nodes the data upon which to operate. Here, data is transferred among nodes by means of a stack. Each of the nodes executes a function that reads from and/or writes to the stack. For example, the node *A* in Figure 3 reads the value of the input *A* from RAM and pushes it onto the stack. The node *B* pushes its value onto the stack. The node *Plus* pops two values from the stack, adds them, and pushes the result onto the stack. Thus the system has localized memory.

Data may also be saved by graph-based GP in indexed memory. The node labeled *Write* pops two arguments from the stack. It writes the value of the first argument into the indexed memory location indicated by the second argument. The *Read* fetches data from the memory location. The indexed memory is global memory.

Note that there are really two things that each node must do: it must perform some function on the stack and/or the indexed memory, and it must decide which node will be the next to execute. This letter function is really a matter of choosing between the outgoing edges from the node. For example on the Figure 3. The *Plus* node may transfer control to *Write*, *Read* or *C* . For this purpose each node has a branch-decision function which determines, depending on the stack, memory cells, constants, or the foregoing node, which of the edges to take.

3. Test program generation

In this chapter will be proposed an methodology for automatic generation a test program able to attain high fault coverage figures, which based on the graph-based GP approach.

The overall architecture of the proposed approach is followed .The *code generator* induces test programs exploiting an external *subprogram library* that consist of microprocessor assemble operators and subprograms. The *code generator* exploit a *fault simulator* to evaluate the generated test programs and to gathered relevant information for driving the optimization process.

Test program generation exploits the methodology of inducing assembly programs for microprocessors cores to reach a specific goal. It utilizes a program representation as *directed graph* for representing the syntactical flow of a program, and an *subprogram library* for describing the assembly instructions and subprograms.

3.1 Program representation

Each node of the *directed graph* (Figure 4) contains a pointer inside the *subprogram library*. The instruction library describes the assembly syntax, listing the possible operators and subprograms with the syntactically correct operands. The *directed graph* are built according to the following rules:

- *Start* and *End* nodes are always present in the graph-based program representation. The *Start* has no parent node, while the *End* has no children. These nodes may never be removed from the program, nor changed.
- *Sequential-branch* nodes represent common operations, such as arithmetic or logic ones (e.g., node **B**). They have out-degree 1 and the number of parameters changes from instruction to instruction.
- *Conditional-branch* nodes are translated to assembly-level conditional-branch instructions (e.g., node **A**). All common assembly languages implement some jump-if-condition mechanisms. All conditional branches implemented in the target assembly languages must be included in the library.

3.2 Program Implementation

Test programs are implemented by modifying *directed graph* topology and by mutation

parameters inside *directed graph* nodes. Both kinds of modification are embedded in an evolutionary algorithm using a (μ+λ) strategy.

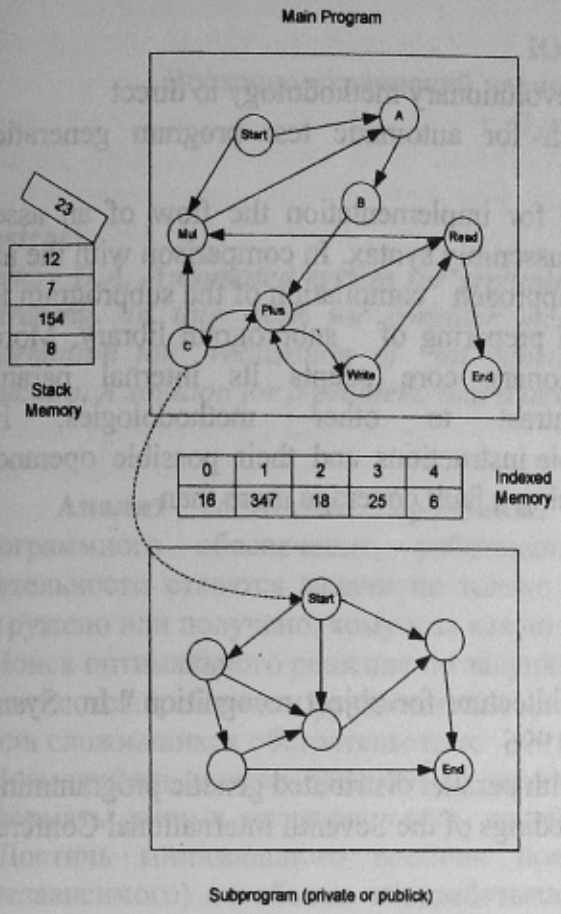

Fig. 3:The representation of a program and subprogram

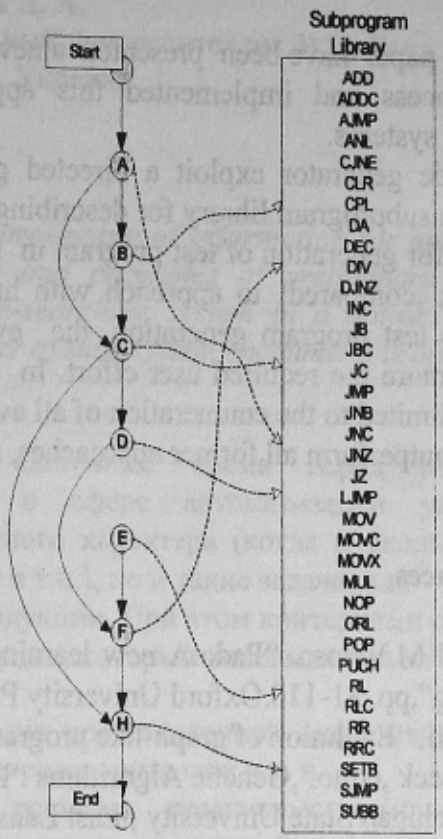

Fig. 4: Directed Graph and Subprogram Library

In more details , a population of μ individuals is cultivated, each individual representing a test program. In each step , an offspring of λ new individuals are generated. Parent are selected using tournament selection with tournament size τ
(i.e., τ individuals are randomly selected and the best is picked). Each new individuals is generated by applying genetic operators. After creation new λ individuals, the best μ programs in the population of (μ+λ) are selected for surviving.

The evolution process iterates until the population reaches a steady state condition
Three mutation and crossover operators are implemented and applied with probability respectively.

- **Add node:** a new node is inserted into the *directed graph* in a random position. The new node can be either a *sequential branch* or a *conditional branch*. In both cases , the instruction referred by the node is randomly chosen. If the inserted node is a branch, ether unconditional or conditional, one of the subsequent nodes is randomly chosen as the destination. Remarkable, when an unconditional branch is inserted, some nodes in the *directed graph* may become unreachable.
- **Remove node:** an existing internal node (except *start* or *end*) is removed from the *directed graph* . If the removed node was the target of one or more branch, parents' edges are updated.
- **Modify node:** all parameters of an existing internal node are randomly changed.
- **Crossover:** two different programs are mated to generated a new one. First, parents are

analyzed to detect potential cutting points, i.e., vertices in the *directed graph* that if removed create disjoint sub-graphs (e.g., node C in Figure 4). Then a *SAG* 1-point crossover is exploited to generate the offspring.

4. Conclusion

In this paper have been presented a newest evolutionary methodology to direct the search process and implemented this approach for automatic test program generation of microprocessor systems.

The code generator exploit a directed graph for implementation the flow of an assembly program and an subprogram library for describing the assembly syntax. In comparison with the anthers methodologies for generation of test program in this approach compilation of the subprogram library is a trivial task ,compared to approach with human preparing of subprogram library. Moreover, concurrently to test program generation, the evolutionary core adapts its internal parameters, reducing even more the required user effort. In contrast to other methodologies, human intervention is limited to the enumeration of all available instructions and their possible operands. This approach outperform all former approaches, reaching a fault coverage more than 90%.

5. References

[1]A.Teller and M.Veloso. "Pado:A new learning architecture for object recognition." In Sysmbolic Visual Learning",pp. 81-116.Oxford University Press,1996

[2]Riccardo Poli, "Evolution of graph-like programs with parallel distributed genetic programming .",In Thomas Back ,editor ,Genetic Algorithms : Proceedings of the Seventh International Conference, pp.346-353,Michigan State University , East Lansing , MI,USA,19-23 July 1997.Morgan Kaufmann

[3] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. "Genetic Programming: An Introduction." Morgan Kaufmann, Inc., San Francisco, USA, 1998.

[4] Kantschik, W., P. Dittrich, M. Brameier, and W. Banzhaf. 1999. "MetaEvolution in Graph GP", Proceedings of EuroGP'99, LNCS, Vol. 1598. SpringerVerlag, pp. 15-28.

[5] L. Chen, S. Dey, "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors", IEEE VLSI Test Symposium, 2000, pp. 255-262

[6] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, "On the Test of Microprocessor IP Cores", IEEE Design,Automation & Test in Europe, 2001, pp. 209-213

[7] W. Kantschik, W. Banzhaf, "Linear-Graph GP -- A new GP Structure", EuroGP2002: 4 tn European Conference on Genetic Programming, 2002, pp. 83- 92

[8] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Evolutionary Test Program Induction for Microprocessor Design Verification", 11th Asian Test Symposium, 2002, pp. 368-373

[9] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Efficient Machine-Code Test-Program Induction",Congress on Evolutionary Computation , 2002,pp. 1486-1491

[10] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero," Fully Automatic Test Program Generation for Microprocessor Cores",DATE2003: Design, Automation and Test in Europe, Munich, Germany, March 3-7, 2003, pp. 1006-1011