Automatic Verilog Code Generation through Grammatical Evolution

Ulya R. Karpuzcu
Department of Computer Engineering
Istanbul Technical University
Maslak 34469 Istanbul, Turkey
karpuzcu@itu.edu.tr

ABSTRACT
This work aims to investigate the automatic generation of Verilog code, representing digital circuits through Grammatical Evolution (GE). Preliminary tests using a simple full adder generation problem have been performed.

Categories and Subject Descriptors
I.2.2 [**Artificial Intelligence**]: Automatic Programming – *program synthesis, program verification.*

General Terms
Design, Verification.

Keywords
Grammatical Evolution, Automatic Code Generation, Verilog.

1. INTRODUCTION
Application of a HDL (Hardware Description Language) represents a basic step in the ASIC (Application Specific Integrated Circuit) design flow. In this context, (mostly) a digital circuit is represented in Verilog or VHDL. Then simulations are performed to check the functionality of the generated circuit representation. If the circuit model generated by a HDL shows the expected behavior, synthesis of the circuit, the conversion of the HDL code to an ensemble of logic gates, follows [2]. Because the syntax of Verilog is simpler, Verilog code generation is aimed for in this study.

In Verilog, circuits can be described at various abstraction levels: behavioral, structural and RTL (Register Transfer Level) coding [4]. Behavioral coding provides a "black box" representation of the circuit to be designed and cannot be synthesized. Structural coding reflects components of a design (e. g. logic gates) and interconnections between them. RTL coding, representing the functionality of a design at register transfer level, employs only the synthesizable portion of Verilog syntax. The most descriptive type is behavioral, then RTL and finally structural coding [2].

Genetic and Evolutionary Computation Conference (GECCO)'05, June 25–29, 2005, Washington DC, USA.

Generation of synthesizable Verilog code, using RTL coding, is intended within the scope of this work. Circuit models generated by synthesis tools stand in strong correlation with the HDL code written to represent the functionality of the circuit to be designed [2]. Automatic Verilog code generation through Grammatical Evolution covers this step.

2. DETERMINATION OF THE EVOLUTIONARY TECHNIQUE SUITED TO VERILOG CODE GENERATION

2.1 Genetic Programming (GP)
The first technique considered was Genetic Programming (GP). Since each element in the solution space of the problem is depicted as a tree, GP provides a proper representation mechanism for the automatic code[1] generation paradigm [1]. Main limitation of GP over crossover and mutation is closure. Closure allows any two points in a tree representing a computer program to be crossed over without loss of validity [11]. The set of syntactically valid individuals should be closed over the genetic operations. To achieve this, each non-terminal node of the tree representation of a program is expected to have the same arity and a return value of the same type [10].

Verilog syntax permits the use of code blocks such as "if" and "always", which should be protected against the destructive operation of crossover. GP's modularization scheme, Automatically Defined Functions (ADFs) can be used to achieve this. Besides, Verilog syntax constraints include:

1. An input variable on the left hand side of an assignment is not allowed.
2. An assignment cannot be generated as the condition statement of an "if" or "always" block.
3. Depending on the nature of the circuit to be represented, zero or more "always" blocks or alternatively zero or more assignment statements should be brought together. For instance, an expression defining the right hand side of an assignment, or a conditional statement (expected to be used in the context of an "if" or "always" block) should not be generated without context to represent a line of code.

If standard GP is used, the set of non-terminals consist of Verilog operators. To define constraint 1, e. g., the left child of a subtree having an assignment operator in its root should be restricted to include an element from the set of output variables, while various logical, bitwise, reduction or concatenation operators can be placed at the right hand side child. As no rules can be defined to

[1] In this work, "code" is used to specify Verilog code.

manipulate the selection on the function (non-terminal) and terminal set to determine the function or terminal corresponding to a certain node of the tree, generation of syntactically correct Verilog code cannot be guaranteed if standard GP is employed.

To overcome the mentioned problem of standard GP, GP variants such as Grammatically Based Genetic Programming [12], Strongly Typed Genetic Programming (STGP) [3] or Derivation Tree Based Genetic Programming (DTGP) [10] are introduced. Grammatically Based Genetic Programming benefits from context free grammars to meet structural requirements. The derivation steps for each individual (according to the preset grammar defining constraints to be followed) are given as the nodes of the tree representing the individual. The program code of the individual can be detected by traversing the leaves of the tree (derivation tree) from left to right [12]. In STGP, the number and type of arguments for each function and the return type of the function (the "signature" of a function) is defined before the construction of the initial population to force the generation of valid individuals only. Since the signatures are to be preserved over genetic operations, more sophisticated operators than the ones used by standard GP are needed. STGP uses the basic tree structure of the standard GP [3]. DTGP aims to design solely syntactically correct individuals by introducing constraints as derivation trees [10]. Derivation trees represent individuals as well.

The modified approaches can be used to assign different subsets of function and non-terminal sets as arguments to functions, thus all appear to be suitable to define Verilog syntax constraints.

2.2 Probabilistic Incremental Program Evolution (PIPE)

Another technique suitable for automatic code generation is PIPE (Probabilistic Incremental Program Evolution) [9]. Individuals are represented by n-ary trees, n being the maximum arity. Programs are generated in accordance with the probabilistic prototype tree (PPT) suited to the problem at hand. The PPT is a complete n-ary tree. Each node of PPT contains a variable probability vector and each element of the vector corresponds to an element of the function or terminal set. Nodes are characterized by their depth and width coordinates. To determine the terminal or function corresponding to a certain node of an individual, the node of PPT having the same coordinates is examined. This node gives the probability of association of a certain terminal or function with the node in the individual tree under operation. Initially the PPT contains only the root node. Subtrees of PPT are created dynamically, when a function is selected for a node in the individual tree, and no node (or an insufficient number of nodes) is present to define its arguments in the PPT (*growing*). In a similar manner, subtrees of PPT not required as function arguments are pruned. After a population gets evaluated, *learning from population* (PPT probabilities are modified to increase the probability of generating the best program found in the current generation.) and mutation of the PPT (influenced by the current best solution) follows. This is the only genetic operation used.

If a node in the PPT has a dominant probability for an assignment, e.g., the children of this node, corresponding to the arguments, should be organized to generate an output variable on the left hand side to stay in accordance with constraint 1. This could be achieved by setting the probability vectors corresponding to arguments accordingly, but PIPE does not permit such a modification. As long as no *growing* is required, the nodes of PPT cannot be manipulated in the generation phase of an individual. If *growing* were required, the new node to be inserted to generate an argument could not be determined in advance without any knowledge about the function associated with the parent node (since arguments are function dependant) and PIPE does not provide such an "inter-node communication" mechanism. Hence, PIPE turns out not to be suited to the Verilog code generation problem.

2.3 Grammatical Evolution (GE)

A method making use of context free grammars, Grammatical Evolution (GE) [7] was examined next. GE is capable of generating compilable code in any language, provided that the search space is limited by the BNF (Backus Naur Form) representation of the language concerned. Depending on the nature of the problem, a subset of the BNF representation can be employed to further restrict the search space. The first step in applying GE to a problem covers the determination of a BNF representation [8]. BNF representation defines a set of production rules that map non-terminals to terminals.

In GE, the individuals are represented as binary strings of fixed length. The binary string is organized as a set of adjacent codons of 8 bits [7]. However, which (adjacent) portion of codons is to be reflected on the phenotype changes from individual to individual. GE can be regarded as a variable length linear genome system, since the size of genetic material to be reflected on the phenotype changes from individual to individual [8].

To generate an individual, first the start symbol, a non-terminal, is mapped. Production rules provide all possible mappings (various non-terminals or terminals) for a non-terminal. To determine which mapping to use, the integer value of each codon of 8 bits is calculated. Then, a rule to map a non-terminal is determined by the modulo operation of the *integer codon value* by the *number of production rules associated with the current non-terminal* [8]. If a non-terminal characterized by the 3 production rules <binary-op> :: <bitwise-and> (0) | <bitwise-or> (1) | <bitwise-xor> (2) is to be mapped and the codon value read is 7, e.g., rule 1 (*7 mod 3*), namely bitwise-or will be selected. IN GE, to determine a production rule, each time a different codon is read and the binary string corresponding to an individual is traversed [7]. It is probable that the generated code of an individual contains non-terminals, when the genotype of the individual is fully traversed. Then, the codons of the individual are reused by wrapping the binary string genome. In this manner, a codon might be used multiple times until all non-terminals in the generated code are mapped. The relationship between codons and integers are one-to-one, but depending on the non-terminal under operation, the same codon may cause different production rules to be selected [7]. If a codon value results in the same production rule to be selected over and over, invalid individuals are very likely to occur. To help preventing this situation, the maximum number of wrapping events is defined as a parameter of GE. If an individual remains still invalid after the preset permitted maximum numbers of wrappings are carried out, it gets penalized with the lowest possible fitness value. To get rid of invalid individuals –as well as to eliminate those with low fitness values in an effective manner- steady state replacement techniques can be used [5, 6].

2.4 Evaluation of the Techniques

Bias is defined as all factors having an influence on the form of the solutions. A bias is characterized by its strength and correctness. The more constraints a bias defines, the stronger the bias is. Correctness is taken as a measure of how well a bias is suited to a problem [11]. The solution space should be biased to accelerate the system to find a solution. Verilog syntax represents the correct bias for the automatic Verilog code generation problem. The smaller a subset of the syntax is chosen, the stronger the bias will be. With no bias, individuals violating the Verilog syntax will occur more frequently. They can be penalized by worst-case fitness values; however, they often cannot be evaluated by a fitness function. Even if they can be corrected, they consume system resources [10].

Neither standard GP nor PIPE is suited for a bias (Verilog syntax or a subset thereof) definition for the problem at hand, thus both get eliminated. Candidate techniques are the GP variants mentioned in Section 2.1 and GE. From the GP variants, DTGP and grammatically based GP suffer from extensive space consumption, because the representation (derivation tree) includes not only the generated code of an individual but also the production rules employed. STGP uses the same representation as standard GP, hence is simpler from this viewpoint. But the simplest representation is introduced by GE. Moreover, for DTGP, grammatically based GP and STGP, the operator complexity is higher when compared to the standard GP. GE turns out to be the simplest method with respect to operators as well [12, 10]. Various kinds of GA bit operators can be applied with GE. Since BNF notation (*given in IEEE Std 1362-2001: Verilog Hardware Description Language*) represents the most powerful definition of Verilog syntax and because of its simple representation and operators, in this study, GE is chosen as the evolutionary technique to be employed.

3. AUTOMATIC VERILOG CODE GENERATION EXAMPLE

3.1 Preparation

As a simple first example, a one bit full adder is chosen. The full adder is expected to calculate the sum *s* of its one bit inputs *a* and *b* by taking the input carry *cin* into consideration. Moreover, the value of the output carry *cout* has to be determined. The fitness of an individual is the number of valid outputs generated over the whole 8 test cases (outputs are combined to a vector and evaluated simultaneously over the 8 test cases). The evaluation stops when an individual of fitness 8 (an optimal solution) is encountered or the maximum number of fitness evaluations is encountered. The truth table is given Figure 1. Open source Icarus Verilog [14] is employed as the Verilog compiler and simulator. The GE tool from [13] used.

The BNF for the subset of Verilog syntax used in this example problem is given below:

< S > :: < blocking-assignment-s > < blocking-assignment-cout >
< blocking-assignment-s > :: assign s = < rhs > ;
< blocking-assignment-cout > :: assign cout = < rhs > ;
< rhs > :: < binary-op > | < logical-not >
< binary-op > :: < bitwise-and > | < bitwise-or > | < bitwise-xor >
< bitwise-and >:: (< argument > & < argument >)
< bitwise-or > :: (< argument > | < argument >)
< bitwise-xor > :: (< argument > ^ < argument >)
< logical-not > :: ! (< argument >)
< argument > :: < invar >| < binary-op-out > | < logical-not-out >
< argument-out > :: < invar > | <binary-op-in>| < logical-not-in >
< binary-op-out > :: < bitwise-and-out > | < bitwise-or-out >
| < bitwise-xor-out >
< bitwise-and-out > :: (< argument-out > & < argument-out >)
< bitwise-or-out > :: (< argument-out > | < argument-out >)
< bitwise-xor-out > :: (< argument-out > ^ < argument-out >)
< binary-op-in > :: < bitwise-and-in > | < bitwise-or-in >
| < bitwise-xor-in >
< bitwise-and-in > :: (< invar > & < invar >)
< bitwise-or-in > :: (< invar > | < invar >)
< bitwise-xor-in > :: (< invar > ^ < invar >)
< logical-not-out > :: ! (< argument-out >)
< logical-not-in > :: ! (< invar >)
< invar > :: a | b | cin

To strengthen the bias, all output variables are forced to be assigned a proper expression. Moreover, to prevent an unlimited growth of nested statements (since Verilog operators can accept statements including Verilog operators as arguments), maximum number of nested statements is bounded to be 3.

Bitwise mutation at a rate of 0.01 and uniform crossover with probability 0.5 are employed. Two parents are selected through binary tournaments to participate in reproduction. If the generated child (the corresponding binary string) already exists in the population or if its fitness is less than the fitness of the worst individual in the population, it is discarded (duplicate elimination). After each insertion into the population, the worst individual is eliminated. Hence, the individuals in the population are maintained by a steady state approach. Codons of 8 bits are used. An individual consists of a chromosome of length 160 bits and there are 200 individuals in the population. The maximum number of fitness evaluations is limited to be 100000. All parameters are determined empirically and further tests are being carried out to find optimal settings.

Table 1. Truth Table of the Full Adder

a	b	cin	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3.2 Results

An optimal solution (an individual of fitness 8) was generated after 50369 fitness evaluations. The generated individual employs 24 codons (and a wrapping, since a maximum of 20 codons per individual is available). The Verilog code evolved is as given in the first column of Table 2. Another individual of fitness 8 according to Table 1 is generated after 19772 fitness evaluations. Depicted on the right column of Table 2, this individual employs 26 codons, thus a wrapping as well. (In Table 2, Verilog operators &, |, ^ and ! represent logical and, or, xor and not respectively.)

35 runs were performed, and only 2 of them resulted in a (optimal) solution. Hence, in spite of the strong bias employed, the *probability of success* [12] was only about %5.7. Obviously, runs were terminated too early (“Failing” runs were terminated after 100000 fitness evaluations were carried out and the majority of them ended up with a best individual of fitness 7.). Since

evolutionary techniques are stochastic, statistical methods should be used to determine the expected value range for success.

Table 2. Examples of Generated Verilog Codes

module adder(a,b,cin,s,cout);
input a; input b; input cin;
output s; output cout;
assign s=(a^(b^cin));
assign cout=(((b^a)&(a^cin))
^a);
endmodule //adder
	module adder(a,b,cin,s,cout);
	input a; input b; input cin;
	output s; output cout;
	assign s=(!((a^cin))^!(b));
	assign cout=((cin|b)^ (
	(cin^b)&!(a)));
	endmodule //adder

On the other hand, when the problem is partitioned, so that expressions for *s* and *cout* are evolved separately, a *probability of success* of 1 was encountered for both, *s* and *cout*, when 20 runs per each output were performed. Each run was able to yield a solution of fitness 8 with less than 100000 fitness evalutions by then. The %95 confidence interval for number of fitness evaluations required to locate the optimum is found out to be

%95 CI = 242.1 to 500.9 for s and

%95 CI = 4424.2 to 9819.1 for cout.

4. DISCUSSION

According to the results of experiments, introduction of an encapsulation scheme is suspected to be capable of improving performance dramatically.

Although the full adder problem is of minimal complexity and the BNF specification represents a strong bias (*s* and *cout* are forced to participate in assignments) it proved to be a good starting point. However, more work is required to tune the current method and to find better parameter settings to improve average performance.

5. FUTURE WORK

Measures to improve efficiency and average performance should be taken. After having stabilized an effective scheme, multiple bit adder code is intended to be generated. After this, the next step will be to model a sequential circuit. The duplication operator mentioned in [7] is not currently employed. Introduction of this operator may improve performance. Furthermore, a scheme allowing definition of zero or more (up to a limit) temporary variables can be added to the grammar, which can be useful in applications requiring multiple nested statements. Also an adaptive method for the grammar representation can be introduced. For instance, if an operator taking input variables as arguments has selected one of the inputs and needs more inputs to be placed within the same statement, it can be forced to select another input at the next step. This will prevent occurrence of statements like ($input_i$ ^ $input_i$) & ($input_i$ ^ $input_i$), which do not seem to be very useful if constants 1 and 0 are presented in the grammar. (The current implementation does not employ any constants.)

The results obtained from the preliminary experiments seem to be promising. Current work is being carried out to address some of the issues mentioned above.

6. ACKNOWLEDGMENTS

This work is being carried out as a senior year graduation project under the supervision of Asst. Prof. Dr. Sima Uyar in the Computer Engineering Department of Istanbul Technical University.

7. REFERENCES

[1] Koza, J., Bennett III F. H., Andre D., Keane M. A. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[2] Lee, W. F. *Verilog Coding for Logic Synthesis*. Wiley-Interscience, Hoboken, NJ, 2003.

[3] Montana, D. J., *Strongly Typed Genetic Programming.* Technical Report BBN 7866 Bolt Banek and Newman Inc., Cambridge, MA, 02138, 1994.

[4] Navabi, Z. *Verilog Digital System Design*. McGraw-Hill, New York, NY, 1999.

[5] O'Neill M., Ryan C. Grammatical Evolution: A Steady State approach. In *Late Breaking Papers at the Genetic Programming 1998 Conference* (Madison, WI, USA, July 22-25, 1998). Madison, WI, Omni Press, 1998.

[6] O'Neill M., Ryan C. Under the Hood of Grammatical Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99* (Orlando, FL, USA, July 13-17, 1999). Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[7] O'Neill M., Ryan C. Grammatical Evolution. *IEEE Transactions on Evolutionary Computation, 5,* 4, (August 2001), 349-358.

[8] Ryan C., Collins J.J., O'Neill M. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Proc. of the First European Workshop on Genetic Programming* (Paris, France, April, 1998). Lecture Notes in Computer Science Volume 1391**,** Spriger Verlag, London, UK, 1998, 83-96.

[9] Salustowicz, R.P. and Schmidhuber, J. Probabilistic Incremental Program Evolution. *Evolutionary Computation, 5,* 2 (1997), 123-141.

[10] Vanyi R., Zvada S. Syntactically Correct Genetic Programming. In *Proc. of the Grammatical Evolution (GEWS 2004) Satellite Workshop to Genetic and Evolutionary Computation Conference (GECCO 2004)* (Seattle, Washington, USA, June 26-30, 2004). S. CDROM, 2004.

[11] Whigham, P. A. Inductive Bias and Genetic Programming. In *Proc. of First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications* (12-14 September 1995) Conference Publication No. 414, IEEE, London, UK, 1995, 461-466.

[12] Whigham, P. Grammatically-based Genetic Programming. In *Proc. of the Workshop on Genetic Programming: From Theory to Real-World Applications* (Tahoe City, California, USA, October, 1995), Morgan Kaufmann Publishers, 1995, 33-41.

[13] http://www.grammatical-evolution.org.

[14] http://www.icarus.com/eda/verilog.