Using Genetic Programming and High Level Synthesis to Design Optimized Datapath

Sérgio G. Araújo, A. Mesquita, Aloysio C. P. Pedroza

Electrical Engineering Dept.
Federal University of Rio de Janeiro
C.P. 68504 - CEP 21945-970 - Rio de Janeiro - RJ - Brazil
Tel: +55 21 2260-5010 - Fax: +55 21 2290-6626
{granato,aloysio}@gta.ufrj.br mesquita@coe.ufrj.br

Abstract. This paper presents a methodology to design optimized electronic digital systems from high abstraction level descriptions. The methodology uses Genetic Programming in addition to high-level synthesis tools to automatically improve design structural quality (area measure). A two-stage, multiobjective optimization algorithm is used to search for circuits with the desired functionality subjected additionally to chip area constraints. Experiment with a square-root approximation datapath design targeted to FPGA exemplifies the proposed methodology.

1 Introduction

Recently, integrated circuit (IC) design has focused on meeting time-to-market and costumers specific requirements for more and more complex systems [1]. To address this new trend, automation of the entire design process from conceptualization to silicon or a describe-and-synthesize methodology [2] has become necessary. Using a top-down design approach, the system functionality is specified through a behavioral model, simulated to verify correct operation and synthesized using high-level synthesis (HLS) tools.

A chip design process using HLS involves several steps, the first being the specification of the system functionality. To this end a variety of conceptual models can be used as discussed in detail in [3]. Finite-state machines (FSM), finite-state machines with datapath (FSMD) and Petri nets are some examples. Programming languages are conceptual models of particular interest since they can support heterogeneous modeling, i.e., they can describe data, activity and control simultaneously. Hardware description languages (HDLs) such as VHDL [4], [5], are able to capture the system functionality in a machine-readable and simulatable form. The system functional specification step is followed by the HLS one, where the behavioral model is translated into a register-transfer level (RTL) structural model. Logic and layout synthesis steps follow in the sequence.

Although more and more EDA tools are available providing high abstraction level design facilities, the market still resists to the automatic behavioral synthesis approach. Gajski et al. [6] point out two reasons for this: poor-quality results and lack of interactivity during the synthesis process. High-level behavioral specifications are usually considered not quite "synthesizable" since at this abstraction level many low-level implementation details are masked (by definition) from the designer [7]. To cope with this problem most behavioral synthesis tools constrain the design space by assuming a "target architecture" to which all high level specifications are mapped. The result is a reasonable average solution locally optimized for each design problem.

On the other hand, layout synthesis and hardware implementations are considerably simplified by FPGA (Field Programmable Gate Arrays) prototyping. By eliminating the need to cycle through an IC production facility, both time-to-market and financial risk can be substantially reduced. Recent statistics [8] indicate that, today, approximately one half of all chip designs are started using FPGAs. This trend is followed in modern synthesis tools that offer a smooth migration path between FPGA and ASIC technologies, providing direct mapping of an FPGA design into ASIC libraries [9].

This work presents a digital IC design methodology based on Evolvable Hardware (EHW) techniques [10], [11]. Genetic Programming is used to automatically explore the design space and improve the design quality. To accomplish this, a multi-criteria fitness function was defined to rank candidate solutions according to their area sizes. An experiment with a square-root approximation datapath design targeted to FPGA is detailed. Simulation results confirm the automatic generation of optimized structures starting from a high level specification.

The paper is organized as follows: Section 2 presents related works in order to establish the context in which the proposed methodology is applied. In Section 3 the Evolutionary Computation and Genetic Programming concepts of interest to the proposed methodology, are discussed. The design methodology is detailed in Section 4 and the example of a datapath design is presented in Section 5. Conclusions are drawn in Section 6.

2 Related Works

Evolutionary Algorithms have already been used in lower levels of electronic design, including routing, partitioning and placement [1], [12], and more recently in the automation of the structural design of digital circuits, as well as in the search for quality implementations, as related below.

Recent works [13] - [16] present evolved structures entirely in software using computer simulations (*extrinsic* evolution) to evaluate intermediate solutions. In [13] an algorithm capable of evolving 100% functional arithmetic circuits is presented. Based on a rectangular array of uncommitted logic cells of FPGAs, the algorithm is able to re-discover conventional optimum designs for one and two-bit adders, eventually improving the gate count of human produced designs of the two-bit multiplier.

Kalganova & Miller [14] addressed a two-stage *multiobjective* fitness function. In the first stage a 100% functional solution, which occurs when the truth table is matched, is sought. In the second stage the number of gates actually used in each candidate solution is taken into account in the fitness function. This allows circuits to evolve with the desired functionality minimizing their number of active gates. The authors limited their focus to combinational logic circuits, containing no memory elements and no feedback paths. A similar multiobjective optimization technique was proposed by Coello et al. [15], the main difference being the use of a population-based technique to split the search task among several sub-populations. In this approach *each* objective is assigned to a small sub-population, which is merged with the rest of the individuals when one of the objectives is satisfied. The merged populations contribute to minimize the total amount of mismatches produced between the encoded circuits and the truth table.

Hounsell & Arslan [16] addressed an EHW environment called *Virtual Chip* in which timing constraints are taken into account. In this environment not only gates but also functional macros, such as half-adders and compound gates, are available to the evolutionary process. In order to speed up the overall process a *single* VHDL program describing all candidate circuits is used in the simulations. In this work a *phased* approach was introduced to ease the evolution of complex systems. The phased evolution consists of evolving a circuit in stages: the system initially evolves a "sub-circuit" for each output of the desired circuit; in the sequence, redundant logic between the evolved sub-circuits is removed as they are combined to generate the required circuit. Using this approach a 3-bit multiplier was evolved with results as good as CAD based circuits.

An alternative to the above approaches is the use of formal grammars to represent the evolution of hardware designs, as in the work of Hemmi et al. [17]. In this approach, hardware specifications, which produce not only hardware structures but also behaviors, are automatically generated as HDL programs. To create HDL-descriptions the authors used Genetic Programming (GP) to evolve trees from productions (rephrased rules in the HDL grammar). The fitness is evaluated using simulation tools and hardware implementation is created *after* the learning task is completed. This system was used to automatically generate a sequential binary adder.

The approach to be presented in this paper uses the main techniques reported above, together with new improvements. Similar to [17], formal grammar of an HDL language was used to build candidate solutions. However, the place-and-route task was placed *inside* the evolutionary process in order to supply the fitness function with chip area information. The multiobjective fitness function adopted closely parallels the two-stage approach suggested in [14], although applied in a different framework. Also, an experience was made using a single VHDL program, as suggested in [16], in order to reduce computing effort.

3 Evolutionary Computation and Genetic Programming

Evolutionary Computation (EC) consists of a class of probabilistic algorithms inspired by the principles of Darwinian natural selection and variations that can greatly benefit from the increased simulation power for complex design, control and knowledge discovery applications [18]. Genetic Algorithm (GA) and Genetic Programming (GP) are the instances of EC most widely used in Evolvable Hardware [19].

GA [20] is based on the evolutionary process occurring in nature where a population is shaped by the survival of best-fit individuals. GA usually works with strings of fixed length in which one or more parameters are coded. GP is a branch of GA, the main difference being the solution representation. Unlike GA, GP can easily code genotypes of different sizes, which increases the capacity in structure creation.

Koza introduced canonical GP [21] with tree-based genome using LISP language. Since that, other genome types have been proposed like linear and graph genomes [22]. Other GP systems were developed to automatically create programs in arbitrary languages (other than LISP) [23], [24]. An alternative way to encode domain-knowledge is by formal grammars [25]. Programs may be produced by combining a context-free grammar (CFG) with GP [26] - [29], known as G^3P (Grammar-Guided Genetic Programming). G^3P provides a framework for automatic creation of error free programs in arbitrary language. Genetic Programming Kernel (GPK) [28] was used as the core engine for the GP system used in this paper. GPK is a complex system that evolves programs in any language once a Backus-Naur form (BNF) is provided as input. However, the initialization procedure was changed to overcome GPK difficulty associated with creating the first generation, as pointed out in [29].

4 Design Methodology

The proposed design methodology evolves independent VHDL programs trying to breed a solution (hardware description) that implement the desired functionality *and* satisfies design constraints for area. This methodology uses a mixture of C code and VHDL, with HLS tools. The execution flow of the proposed methodology is drawn in Fig. 1. This figure shows the two main components of the methodology: the GP core and the Valuation function. Also, it can be seen that, differently to other methodologies [16], [17], synthesis and place-and-route are performed for each candidate solution, which gives an exact value for the area.

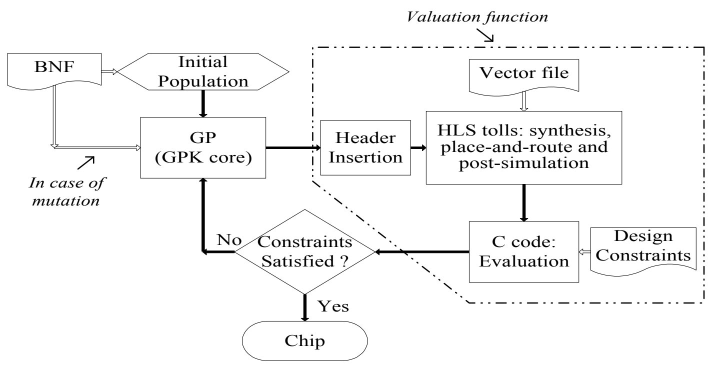


Figure 1. Execution flow of the methodology

The whole methodology is based on the use of the GP algorithm. Fig. 2 shows the flowchart of the GP system used, adapted from [21]. In this figure, the index “i” refers to an individual in the population of size M. The variable “Gen” is the number of the current generation. A population of M individuals (programs) is randomly created with the restriction of a pre-defined BNF containing a subset of VHDL grammar obeying the *sufficiency* property [21]. The system runs until the termination criterion is satisfied. Since the proposed methodology is concerned with unknown solutions, the termination is achieved when the number of generations exceeds a maximum pre-defined number G (*generational predicate*).

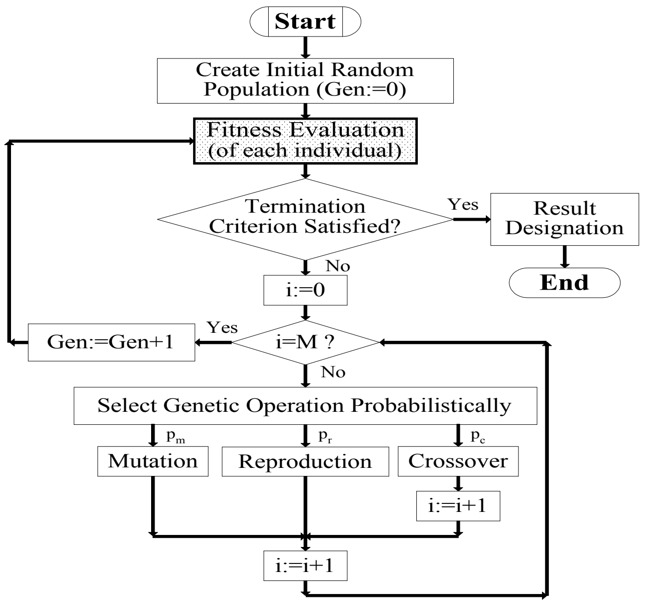

Start
Create Initial Random Population (Gen:=0)
Fitness Evaluation (of each individual)
Termination Criterion Satisfied?
Yes
Result Designation
End
No
i:=0
Yes
Gen:=Gen+1
i=M ?
No
Select Genetic Operation Probabilistically
p_m
p_r
p_c
Mutation
Reproduction
Crossover
i:=i+1
i:=i+1

Figure 2. Flowchart of the GP system

To create VHDL source codes, the entity declaration and the architecture declarative part of the VHDL program are maintained fixed, as they are common to all individuals. The architecture statement part of the VHDL code will evolve.

The evaluation of the objective function is a key procedure of the GP system. The steps involved in this operation are depicted in Fig. 3. After a new population is created it is necessary to complete the VHDL source code by inserting the entity declaration and the architecture declarative part of each individual. Then, the individuals are synthesized using Altera MAX+PLUS II compilation tool [30]. To effectively evaluate the fitness of an individual it is necessary to simulate each of the fitness cases and check the results against target values.

The objective function to be minimized is a multi-parameter error function combining functionality with the area parameter in a two-stage approach similar to that proposed in [14]. In this process the functional specification (F_1 *fitness*) must be first matched to trigger the search for an optimal area implementation (F_2 *fitness*). Let the average functionality error, fe_{av}, be defined as:

$$fe_{av} = \frac{1}{N}\sum_{i=1}^{N} w_i \left|D_i - S_i\right| \tag{1}$$

Where, D_i is the desired value for fitness case i, S_i is the simulated value obtained for fitness case i, w_i is a weighting factor and N is the number of fitness cases.

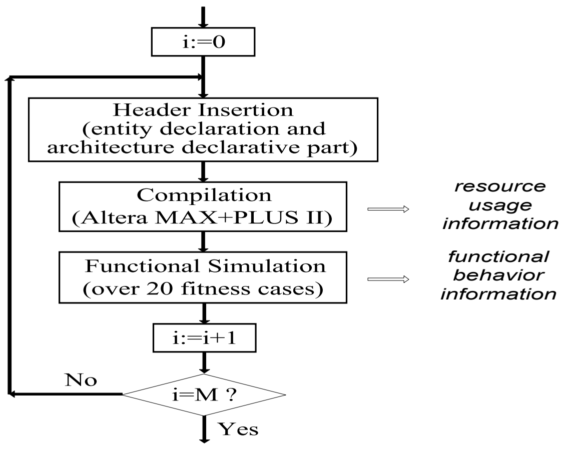


Figure 3. Valuation block of the Fig. 2

The fitness function *F* is defined as follows:

$$F = \begin{cases} F_1 = fe_{av} & \text{if } \exists i \in \{1, \cdots, N\} \mid \left| \dfrac{D_i - S_i}{D_i} \right| > te \\ F_2 = fe_{av} * k(lu) & \text{otherwise} \end{cases} \tag{2}$$

Where, *te* is the target functional specification error for the fitness cases, *lu* is the number of used logic elements (Cf. Subsection 5.3) and *k* is a function of *lu*. The *k(lu)* function is a weighting factor that considers the effect of the area used to implement the desired functionality. It is an exponential function of *lu* that privileges the individuals containing the lowest quantities of logic elements. The *k(lu)* function will take the form:

$$k(lu) = ae^{b*lu} \tag{3}$$

Where, *a* and *b* are constants that must be defined by solving the following system:

$$\begin{aligned} k_{\min} &= ae^{b*lu_{\min}} \\ k_{\max} &= ae^{b*lu_{\max}} \end{aligned} \tag{4}$$

Where, lu_{min} *and* lu_{max} are minimum and maximum quantities of logic elements and k_{min} and k_{max} are the minimum and maximum values of *k(lu)*, respectively.

5 Experiment

To exemplify the proposed methodology, it was selected a datapath design taken from [6]. The aim is to compute the square-root approximation (SRA) of two integers, *a* and *b*. In [6] the following approximation formula is given:

$$\sqrt{a^2 + b^2} \cong \max((0.875x+0.5y),x) \qquad (5)$$

Where $x = max(|a|, |b|)$, and $y = min(|a|, |b|)$.

In this problem of *symbolic regression* the goal is to find a function in symbolic form that is a good, or an exact, fit to a group of 20 sets of numerical data points. The *black box* of the SRA is given in Fig. 4. In this figure, the “start” input and “done” output signals are present for interface control issues.

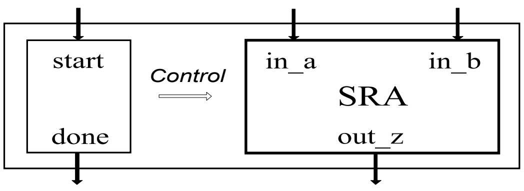


Figure 4. *Black box* of the SRA

5.1. BNF Definition

A BNF grammar describes admissible structures of a language through a 4-tuple $\{S,N,T,P\}$ where *S* denotes the *start* symbol, *N* the set of *non-terminal* symbols, *T* the set of *terminal* symbols and *P* the *productions*, i.e., rewritten rules that map the elements of *N* to *T*. The BNF that defines the syntax of the VHDL statement part of the SRA description is defined as:

```
S      ::= <fe>;
<fe>   ::= "PROCESS BEGIN WAIT UNTIL (clk'event and clk='1');
           IF in_a>=in_b THEN x<=in_a; y<=in_b; ELSE y<=in_a;
           x<=in_b; END IF; t6<=" <expr> "; IF t6>=x THEN
           t7<=t6; ELSE t7<=x; END IF; out_z<=t7; done<='1';
           END PROCESS;";
<expr>::= "(" <expr> ")" <op> "(" <expr> ")" | <var> |
           <var> "(7 downto " <shf_size> ")";
<op>   ::= " + " | " - ";
<var> ::= " x " | " y " | " w ";
<shf_size> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7";
```

In the above BNF description the signals “x”, “y”, “w”, “t6” and “t7” are internal signals of the VHDL architecture body. The signal “w”, assigned zero, is present to make ease the insertion or removal of a terminal. *max* and *min* operations are already fixed in BNF through *IF* clause. The original problem was restricted to use input and output operands as unsigned type instead of integer type, reducing computing effort.

5.2. GP Parameters

The main control parameters of the GP for the *first stage* (*F_1 fitness*) of the proposed experiment are shown in Table 1. In the *second stage* of the evolution (*F_2 fitness*) the GP parameters are the same, except for the objective and raw fitness rows of Table 1 that change to incorporate the search for a design with optimal area.

Table 1. GP tableau for the first stage of the SRA design

Objective:	Find a VHDL program that implements a two argument function that fits a sample of 20 I/O sets $\{a_i, b_i, F(a_i,b_i)\}$, where the target function is $F(a,b) = sqrt(a^2 + b^2)$.
Terminal symbols:	x, y, w, +, –, 7 downto (*logical shift-right operator*), 1, 2, 3, 4, 5, 6, 7 (*index of shift-right operator*) and the sentences between quotation marks on the right-hand side of <fe>.
Non-terminal symbols:	<fe>, <expr>, <op>, <var>, <shf_size>.
Fitness cases:	Sample of 20 sets of data points $\{a_i, b_i, F(a_i, b_i)\}$, a_i and b_i randomly chosen from the interval [0, 255] and satisfying the restriction $sqrt(a_i^2 + b_i^2) \leq 255$.
Raw fitness:	The average, taken over 20 fitness cases, of the absolute value of the difference between the value of the dependent variable produced by simulating synthesized VHDL program and the target value of the dependent variable (Cf. eq.(1)).
Selection Method:	Fitness-proportionate.
Main GP Parameters:	M=100, G=51, p_c=0.73, p_m= 0.12, without elitism.

5.3. Results

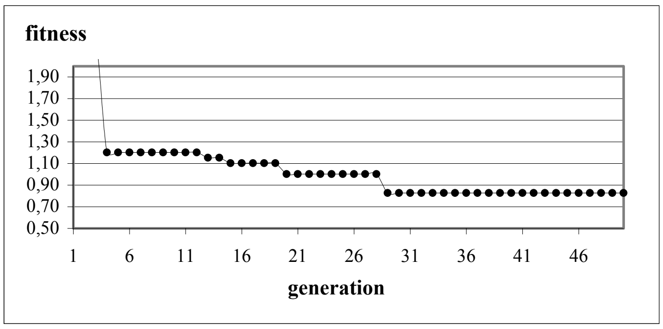
The EPF8282A FPGA from Altera FLEX 8000 family, with up to 282 logic elements (LEs), was used as target device. Each LE consists of a look-up table (LUT), a function generator that computes any function of four variables, and a programmable register. In this experiment, area will be measured in number of LEs. The target functional specification error (*te*) was set to 2.5%. It establishes for *each* fitness case the maximum relative error between the simulated value of the dependent variable, produced by the synthesized VHDL programs, and the desired value.

A population size of 100 individuals was used, evolving up to a maximum of 51 generations. Experiments have been carried out with N=20, w_i=1 and $k(lu)=0.25e^{(0.0139*lu)}$, which guarantees $k(lu) \in [0.5,1.0]$ for $lu \in [50,100]$ (Cf. eq.(3) and eq.(4)). The initial population was created using the ramped half-and-half generative method [21], replacing GPK initialization procedure.

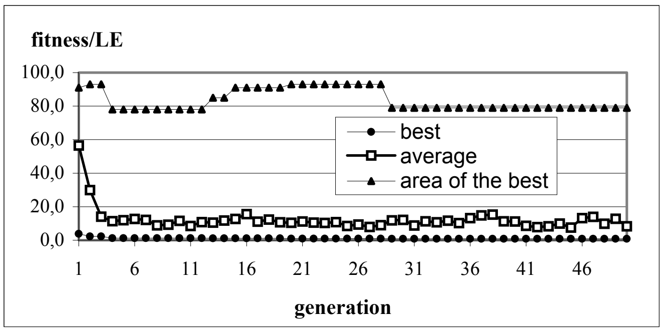
The first functional individual, i.e., one that met all fitness cases evaluations with the functional specification target error of 2.5% was created at generation 18. The system started, then, the search for an implementation with optimal area. During the second stage of the evolution a variety of individuals agreeing with the specified *te* value emerged, although it happened with a non-optimal area use. Functional individuals with 95, 92 and 90 LEs have been reported. After 51 generations the best individual was on hand. It was selected at generation 29. Its main fitness (*F*) was 0.827 and the average functionality error (fe_{av}), 1.103. This individual was synthesized with 78 LEs. The statement part of this VHDL program was:

```
BEGIN
 w <="00000000";
 PROCESS BEGIN WAIT UNTIL (clk'event and clk='1');
   IF in_a >= in_b THEN x <= in_a;  y <= in_b;
   ELSE y <= in_a;  x <= in_b; END IF;
   t6 <=((y(7 downto 1))+((x)-(x(7 downto 3))))+(y(7 downto 6));
   IF t6 >= x THEN t7 <= t6;
   ELSE t7 <= x; END IF;
   out_z <= t7;  done <= '1';
 END PROCESS;
```

Fig. 5 shows (a) the evolution of the fitness of the best individual and (b) its area versus generation curve. In Fig. 5 (b) it can be noted that at generation 13 the best individual increases its area in order to achieve better accuracy. At generation 18 it reaches the desired accuracy and begins to look for optimal area, which is found at generation 29.



a)



b)

Figure 5. (a) Histogram of the fitness and (b) of the area of the best individual

Table 2 reports optimal area implementation (LE column) obtained for other values of the functional specification target error (*te*) with the respective hardware descriptions. An interesting result shown in Table 2 refers to the individual with optimal area for *te*=25% that do not use the internal signal *y* (minimum between the two inputs). This indicates that the system actually looks for optimal area, taking into account route and resource issues of the synthesis tool.

In some independent runs it could be noted that the first functional individual was also the one with optimal area. Nevertheless, this is not always true, as was shown in the SRA design for *te*=2.5%. Also, an experience was made adding the multiplier and shift-left operators on the right-hand side of <*op*> and <*expr*> non-terminal symbols, respectively (Cf. Subsection 5.1). The system showed to be robust by eliminating individuals with these two operators from the early generations.

Table 2. LE quantities for other *te* values with respective hardware descriptions

te	LE	Hardware description
25%	51	`x + x(7 downto 3)`
15%	58	`x + y(7 downto 2)`
12%	59	`x + y(7 downto 1)`
5%	70	`x - x(7 downto 3) + y(7 downto 1)`

The drawback of the proposed methodology is its excessive time consumption. Currently available tools for compiling VHDL programs are slow, particularly the place-and-route ones [31]. To overcome this problem a pre-stage, to be run before the start of the main optimization process shown in Fig. 1, was introduced. In the pre-stage the system evolves a single VHDL program as in [16], each individual is described in a single VHDL PROCESS to bring the whole population close to the target functionality. No place-and-route is accomplished at this time. The pre-stage finishes when the first functional individual comes out.

With this revised approach the system runs three times faster while remaining at the pre-stage. It was observed that the performance of the overall design process improves considerably when compared with the original approach (Fig. 1).

6 Conclusion

A methodology based on Genetic Programming paradigm to evolve optimized implementations was presented. It implies the use of a multiobjective evolutionary technique. The methodology is fully automatic in the sense that it is not necessary to write code but define a BNF grammar and specify the design constraints. Cycling through a place-and-route process makes it possible to consider the exact area used by each implementation in the optimization.

The proposed methodology is capable of synthesizing combinational circuits, using basic digital or mathematical operators declared in a BNF, which approximates a given function with a specific accuracy while looking for optimal area. From this work it can be concluded:

i) It is not necessary to exactly specify the basic operators in the BNF.
ii) Performance optimization may be achieved instead of (or in addition to) area if one considers timing parameters in the objective function.
iii) The methodology is synthesis-tool independent.

Finally, the methodology can be used to implement a new function in a semi-filled FPGA, i.e., in addition to an already described function (in a VHDL PROCESS of the VHDL program). In this case the new function is described in a new VHDL PROCESS. The code of the already described function is maintained fixed while the new code will evolve. The system will look for synthesizing the new function using any common construct between both functions in order to reduce area usage.

References

1. Y-L Lin, *Recent Development in High Level Synthesis*, ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No. 1, 1997.
2. D. Gajski, N. Dutt, A. Wu and S. Lin, *High Level Synthesis: Introduction to Chip and System Design*, Kluver Academic Publishers, 1992.
3. D. Gajski, J. Zhu and R. Dörner, *Special Issues in Codesign*, Technical Report ICS-97-26, 1997.
4. IEEE Std. 1076-1993, *VHDL Language Reference Manual*, 1993.
5. Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, McGraw Hill, 2^{d} Edition, 1998.
6. D. Gajski, I. Tadatoshi, V. Chaiyakul, H. Juan and T. Hadley, *A Design Methodology and Environment for Interactive Behavioral Synthesis*, Technical Report 96-29, 1996.
7. K. Kuusilinna, T. Hämäläinen and J. Saarinen, *Practical VHDL Optimization for Timing Critical FPGA Applications*, Microprocessors and Microsystems 23, pp. 459-469, 1999.
8. D. Landis, *Programmable Logic and Application Specific Integrated Circuits*, Handbook of Components for Electronics, Chapter II, Vol. 1, 1995.
9. Synopsys Inc., *Synopsys Online Documentation*, 2001.
10. H. DeGaris, *Evolvable Hardware: Genetic Programming of a Darwin Machine*, McGraw-Hill, Artificial Neural Nets and Genetic Algorithms, Springer-Verlag, NY, 1993.
11. R. Zebulum, M. Pacheco and M. Vellasco, *Evolvable Systems in Hardware Design Taxonomy, Survey and Applications,* Evolvable Systems: From Biology to Hardware, (ICES96), pp. 344-358, 1996.
12. F. Bennett III, J. Koza, J. Yu and W. Mydlowec, *Automatic Synthesis, Placement and Routing of an Amplifier Circuit by Means of Genetic Programming*, Proc. of the Third Int. Conf. on Evolvable Systems, ICES 2000, pp. 1-10, Edinburgh, Scotland, UK, 2000.
13. J. Miller, P. Thomson and T. *Fogarty, Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study*, Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, edited by D. Quagliarella et al., Publisher: Wiley, 1997.
14. T. Kalganova and J.Miller, *Evolving More Efficient Digital Circuits by Allowing Circuit Layout Evolution and Multi-Objective Fitness*, Proc. of the First NASA/DoD Workshop on Evolvable Hardware, pp. 54-63, Los Alamitos, CA, IEEE Computer Society Press, 1999.
15. C. Coello, A. Aguirre and B. Buckles, *Evolutionary Multiobjective Design of Combinational Logic Circuits,* Proc. of the Second NASA/DoD Workshop on Evolvable Hardware, pp. 161-170, IEEE Computer Society, Los Alamitos, CA, 2000.
16. B. Hounsell and T. Arslan, *A Novel Evolvable Hardware Framework for the Evolution of High Performance Digital Circuits*, GECCO-2000, pp. 525-532, 2000.

17. H. Hemmi, J. Mizoguchi and K. Shimohara, M. Tomassini, *Development and Evolution of Hardware Behaviors*, Towards Evolvable Hardware: The Evolutionary Engineering Approach, Int. Workshop, Lausanne, edited by E. Sanchez and M. Tomassini, Springer-Verlag, LNCS 1062, pp. 250-265, 1996.
18. J. Rosca, *Hierarchical Learning with Procedural Abstractions Mechanisms*, PhD Thesis, University of Rochester, New York, 1997.
19. M. Tomassini, *Evolutionary Algorithms*, Towards Evolvable Hardware: The Evolutionary Engineering Approach, Int. Workshop, Lausanne, edited by E. Sanchez and M. Tomassini, Springer-Verlag, LNCS 1062, pp. 19-47, 1996.
20. J. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan, 1st Edition, 1975.
21. J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
22. W. Banzhaf, P. Nordin, R. Keller and F. Francone, *Genetic Programming: An Introduction,* San Francisco, CA, Morgan Kaufmann and Heidelberg, 1997.
23. R. Keller and W. Banzhaf, *Genetic Programming Using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes*, Proc. of Genetic Programming 1996, pp. 116-122, MIT Press, 1996.
24. N. Paterson and M. Livesey, *Evolving Cache Algorithms in C by GP*, Genetic Programming 1997, pp. 262-267, MIT Press, 1997.
25. A. Ratle and M. Sebag, *Genetic Programming and Domain Knowledge: Beyond the Limitations of Grammar-Guided Machine Discovery,* Parallel Problem Solving from Nature, 2000.
26. M. Wong and K. Leung, *Applying Logic Grammars to Induce Sub-functions in Genetic Programming*, Proc. of 1995 IEEE Conf. on Evolutionary Computation, pp. 737-740, USA:IEEE Press, 1995.
27. P. Whigham, *Grammatically-Based Genetic Programming*, Proc. of the Workshop on Genetic Programming: From Theory to Real-World Applications, pp. 33-41, Morgan Kauffmann Publishers, 1995.
28. H. Hörner, *A C++ Class Library for Genetic Programming*, Release 1.0 Operating Instructions, Viena University of Economy, 1996.
29. C. Ryan and M. O'Neill, *Grammatical Evolution: Evolving Programs for an Arbitrary Language,* LNCS 1391, Proc. of the First European Workshop on Genetic Programming, pp. 83-95, Springer-Verlag, 1998.
30. Altera Inc., *Altera Digital Library Databook 2001*, 2001.
31. D. Montana, R. Popp, S. Iyer and G. Vidaver, *EvolvaWare: Genetic Programming for Optimal Design of Hardware-Based Algorithms*, Genetic Programming 1998, pp. 869, 1998.