

## Design Transformations to Improve Quality of Results

Steve Carlson, Escalade  
Gregory Pogossiants, VeriBest

### Abstract

High-Level Design Automation (HLDA) methodologies depend, to a large extent, on the quality of implementation that HDL synthesis tools can provide. Because of the computational complexity of the synthesis problem, the quality of the input coding style is, in many instances, the largest determining factor of quality (the old adage: garbage-in, garbage-out, still holds). This paper explores the requirements and proposes an architecture for a structured approach to language-based transformations to improve design quality.

There are five major steps in the process of this work: language processing, pattern matching, transformation analysis, transformation, code generation. The language processing component of this work is well documented at this point and the paper focuses on the latter four steps of the process. The pattern matching component of the process works from two standpoints: common coding practice errors and actions on the predictable portion of the synthesis heuristics. Transformation analysis is an estimation process based on the generation of a hardware-like model of the processed language. The transformation actions are algorithmic generators that operate on the native process architecture structures. The final code generation step is an attributable, parameterizable step that can be used to create additional degrees of freedom. Examples are used to describe the details of the process.

### Background

High-Level Design Automation (HLDA) methodologies depend, to a large extent, on the quality of implementation that HDL synthesis tools can provide. Coding style, constraint settings and command options together form the design space degrees of freedom for a given function in the HLDA implementation space. Because of the computational complexity of the synthesis problem and the resultant set of heuristics that are used, the quality of the input coding style is, in many instances, the largest determining factor of quality of design results. The intricacies of the synthesis heuristics and the interdependencies of language constructs make predictability hard to achieve. Thus, experimentation and iteration are a common part of the HLDA experience.

A common methodology used by many HLDA users is to first write easy to validate, easy to debug functional code, and then refine only that code on the critical paths that cannot be made to meet timing through other methods. A problem that exists for the HLDA designer, however, is that functional level coding leads to hard to predict results in synthesis. An approach of not changing the design source alleviates some of the re-validation responsibility during the design refinement process, but removes the most powerful form of control (the coding style) from the designer's first options to fix performance problems.

Another alternative is to use a "meta-netlisting" approach to code where each construct used is carefully chosen to yield a predictable gate-level translation. The drawback of this approach is that the

productivity benefits of abstraction are mostly lost, more source code results, and in a style that is harder to debug.

Examining the operation of synthesis tools reveals a common first transformation action from the language construct domain into a technology independent hardware element domain. The immediacy of the transformation means that some of the derivable intent from the original source can be lost. This means that some opportunities for transformation are lost. For example there are known common coding mistakes that lead to poor synthesis results, but once the transformation to hardware has been made, the original situation is not recognizable.

The basic premise for the work reported in this paper is that the efficiency and the scope of optimization can be increased by adding language based transformations prior to the translation to hardware constructs performed by synthesis tools. Figure 1 depicts relative scope of design space coverage offered by using the basic controls over synthesis (coding style, command options, design constraints). As the figure implies, a much broader design space coverage is expected using these techniques.

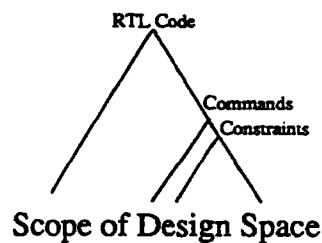


Figure 1

The idea of better quality of results through more effective design space exploration is not a new one. The methodology shift from gate-level to register transfer-level was prompted by a desire to get equivalent or better results in less time by letting the synthesis tool try more gate-level configurations faster. The higher level source code means that explorations taking place beneath the source level need not trigger a re-validation step for the design. Additionally, each increment in abstraction used increases the technology portability of a design description and its prospect for reuse. Moving from an HDL construct specific representation to a symbolic level does in fact increase the abstraction and thus the optimization scope, retargetability, and allows for easier language translation.

There are other applications of this basic approach as well. One common problem encountered by HLDA users is that the coding style needed to obtain good synthesis results is not the same style that will yield the fastest running simulation model of the function. Working from a language transformation methodology, models can be derived that are optimized for their target process (e.g., simulation and synthesis). The framework and tools to create the language transformation solution can also be applied to create custom solutions for coding style analysis for design teams that desire uniformity. Further, common coding problems can be coded, along with solution actions to use known remedies to frequently occurring errors.

### Requirements

The chosen solution approach has a number of requirements in order to be broadly adopted. The system input is RTL VHDL (and Verilog), with RTL VHDL (and Verilog) being generated as the output along with a series of reports. The solution implementation works at multiple levels of intrusiveness. Level 1 is as a warning mechanism for constructs and construct combinations that may present quality of results problems during the synthesis process. Level 2 includes the inclusion of suggested actions for the problems identified by level 1 issues. Level 3 includes automation, with user guidance and overrides, of the transformation into more desirable coding styles. The transformation engine needs to be both

hardware and synthesis cognizant in the analysis and code generation portions of its implementation. Finally, the solution should be easily extensible to take advantage of new knowledge, and local design customs. The customization may extend into new areas that include checking for synthesizability, testability, and analysis of various module attributes.

When considering code re-writing transformations, a number of more formal requirements appear. First and foremost, functional equivalence must be maintained. More formally, if  $P_0$  is the initial program and we want to obtain  $P_n$ , the final program, with the same semantic value ( $Sem(P_0) = Sem(P_n)$ ) for some semantic function  $Sem$ . Transformations are often performed as a series of steps such that we require:

$$\langle P_0, \dots, P_n \rangle \ni Sem(P_i) = Sem(P_{i+1}), \quad 0 \leq i < n$$

In addition to functional equivalence, the user may wish to use the results of the transformation engine as the new source for the design process. In this case, and to simplify the downstream debugging process, the basic code architecture, naming conventions, ports, and comments should be maintained to the largest extent possible.

User control over the application of transformations must be maintained, and user assistance must be permitted. In the final analysis, no tool can understand the functional intent to the same level as the designer. Giving the user control over the application of the transformations ensures that the system can be useful to all designers (rather than just those designers that get satisfactory results from black box operation of the system).

Non-determinism is an issue in finite state machine design. Sometimes the designer's initial code may be non-deterministic. This is usually unintentional and/or undesirable. This means that we may allow transformations that are *sound*, but not *complete*, in the sense that:

$$Sem(P_n)(v) \subseteq Sem(P_0)(v) \quad \text{for any input value } v$$

This means that we allow options for the improvement of code that can include transformation of state machines into safe state machines that recover to the introduction of unexpected states, and the use of don't care assignments to enhance the logic optimization provided by synthesis tools.

The requirements outlined thus far call for a very ambitious research and development program to achieve success. Fortunately, there is a rich body of research available from the functional and logic programming transformation domain. The challenge becomes the specialization, simplifications and optimization of these general purpose programming transformation techniques for use with HDLs and HLDA tools. An excellent survey of the research is given in [Petrossi and Proietti, 1996]. There are host of approaches that have been reported in the literature. The primary research leverage in this work is from:

- Composition Strategy [Burstall and Darlington, 1977]
- Internal Specialization [Scherlis, 1981]
- Deforestation [Wadler, 1988]
- Tupling Strategy [Burstall and Darlington, 1977]
- Generalization [Boyer and Moore, 1975]
- Schemata Approach [Partsch, 1990]
- Partial Evaluation [Ershou, 1982]
- Finite Differencing [Paige and Koenig, 1982]
- Combinatory Techniques [Turner, 1979]

### **Solution Architecture**

The bulk of the solution architecture is based on the schemata approach to program transformation. We have borrowed specific techniques from other solution approaches that are applied in a focused manner to

specific HDL construct combinations. The schemata approach is not computationally expensive compared to some of the alternative approaches. The application of a schema is really just the application of a substitution in the source code. Where the computational cost of the schemata approach can get high is in deciding where substitutions should take place. The matching of a suitable schema is a pattern matching operation that must be efficient in its implementation. Further, the power of the schemata approach is limited by the knowledge encompassed in its dictionary.

In the schemata approach to program transformation a dictionary of ordered pairs of program schemata is given. If a program  $P$  matches schema  $S1$  by a substitution  $\theta$  and  $\langle S1, S2 \rangle$  is a pair in the dictionary, then we can replace  $P$  by  $S2\theta$ . The dictionary must ensure that  $P \equiv S2\theta$ . Further, the transformation should have known improvement effects (predictable or measurable transformation results).

The implementation of the schemata approach has been divided into five processing steps. The first step is basic language parsing (Figure 2), where the RTL source code is transformed into an equivalent program tree representation. This step uses well understood scanner/parser technology. There are special requirements that are placed on the implementation to link the original source and the transformed source. All data is stored at the program tree representation, such that bi-directional queries can be made. The program tree formulation is based on an object oriented model. The semantic representation is the union of VHDL and Verilog. The end result is a language neutral representation that can be used as the source for subsequent steps, independent of the source language. An additional set of linking structures augment the program tree to improve the performance of the pattern matching algorithm.

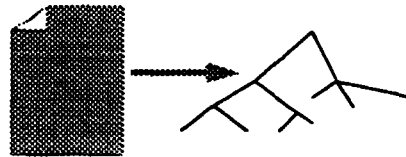


Figure 2

The second step of processing is pattern matching (Figure 3). The dictionary is searched for applicable schema that can be applied to different branches of the program tree. There are two basic purposes for a schema, to identify common coding errors (which can have an identify only, or an identify and fix operation), and to make a transformation that has a predictable reaction by the synthesis heuristics. The schema are coded to account for construct analysis and interplay that allows hardware construct matching (e.g., recognition of counters, finite state machines, etc.). These types of matching allow the interleaving of specialized techniques for more accurate analysis, and directed transformations. The schema have been implemented using a meta-rule formulation for the patterns and the transformations. This has led to a flexible customizable solution, that has been critical in early experimentation.

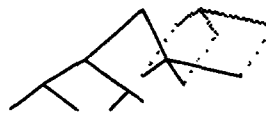


Figure 3

The third step of the process is the cost determination for the matching schema (Figure 4). Here cost (in terms of speed and area, or simple design rule improvement criteria) are applied to make trade-offs. The transformation analysis capability is present to tell users what they have in terms of language constructs and their hardware implications. In the application of a transformation, the analysis estimates design parameters such as speed, area, power, synthesis run time, simulation run time, and adherence to design rules. The transformation analysis is an estimation process that is based on the generation of a hardware-like model of the relevant portions of the program tree.



Figure 4

The fourth step of the process is the substitution of the schemata that re-writes a portion of the program tree (Figure 5). The transformations are implemented as algorithmic generators that operate on the native process and architecture scoped structures of the program tree. The operations are language neutral because the source and target are the program tree.

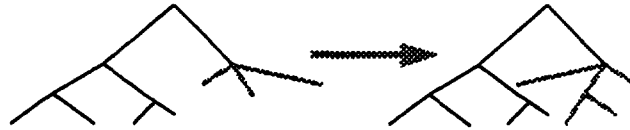


Figure 5

Finally, the fifth step in the transformation process is code generation where the program tree is traversed and coding style rules are applied to create the RTL output (Figure 6). The VHDL and Verilog output include an attributable, parameterizable code generation capability that includes meta-comment generation for issuing synthesis directives. Some of the program tree node types have alternative code generation strategies that allow the creation of additional degrees of freedom for the design exploration process. These degrees of freedom span language construct formulation and hardware alternatives.

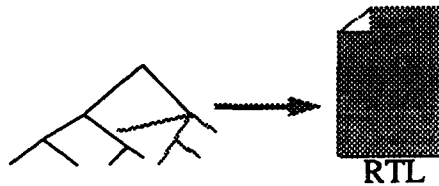


Figure 6

This overview in the five steps of design transformations through program re-writing just scratches the surface of the possibilities. Again we direct interested parties to the references provided. There is a wealth of established methods and reports on the results of the various approaches.

To understand the scope and the limitations of this work, we present some of the mechanics associated with design transformation. The first example shows a multi-process scope for the identification of common sub-expressions.

<pre>entity e is port (x, y, z : in       std_logic_vector(31 downto 0);       out1, out2 : std_logic); end e;  architecture a of e is begin x update:   process (x,y,z) begin     x &lt;= y + z;   end process; </pre>	<pre>x_read1:   process (x,w) begin     if (x+1 &lt; w) then out1 &lt;= '1';     else out1 &lt;= '0';     end if;   end process;  x_read2:   process (x,r) begin     if (x+1 &lt; r) then out2 &lt;= '0';     else out2 &lt;= '1';     end if;   end process; end a;</pre>
---	--

Common sub-expression

Example 1a: Multi-process Common Sub-expression Matching

In Example 1, the sub-expression  $(x+1)$  can be replaced by a new intermediate variable that contains the incremented value of  $x$ . This type of code duplication is common in the conditional part of HLDA code. Because  $x$  is updated in a single location, and the sensitivity lists are complete in  $x$ , the transformation is sound. Some synthesis tools will identify the common sub-expression, but most will not increase the search scope across multiple processes. The result of making a substitution is faster running synthesis, with better quality of results.

```

entity e is
port (x, y, z : in
      std_logic_vector(31 downto 0);
      out1, out2 : std_logic);
end e;
architecture a of e is
  signal x_p_1 :
    std_logic_vector(31 downto 0);
begin
  x_update:
    process (x,y,z) begin
      x <= y + z;
      x_p_1 <= x + 1;
    end process;
  x_read1:
    process (x_p_1,w) begin
      if (x_p_1 < w) then out1 <= '1';
      else out1 <= '0';
      end if;
    end process;
  x_read2:
    process (x_p_1,r) begin
      if (x_p_1 < r) then out2 <= '0';
      else out2 <= '1';
      end if;
    end process;
end a;

```

Common sub-expression

### Example 1b: Multi-process Common Sub-expression Matching

In the second example we look at a more significant code re-writing operation. This particular schemata comes from design for configurable logic block (CLB) FPGAs. The purpose of this schemata is to take advantage of the special resources available in this target technology to implement wide multiplexing operations. As the diagram below (Figure 7) indicates, for a single bit multiplexing function, the use of the configurable logic resources makes sense. However, for an 8-bit multiplexing function, less CLB resources need to be used if the internal tri-state resources are applied to implement the function.

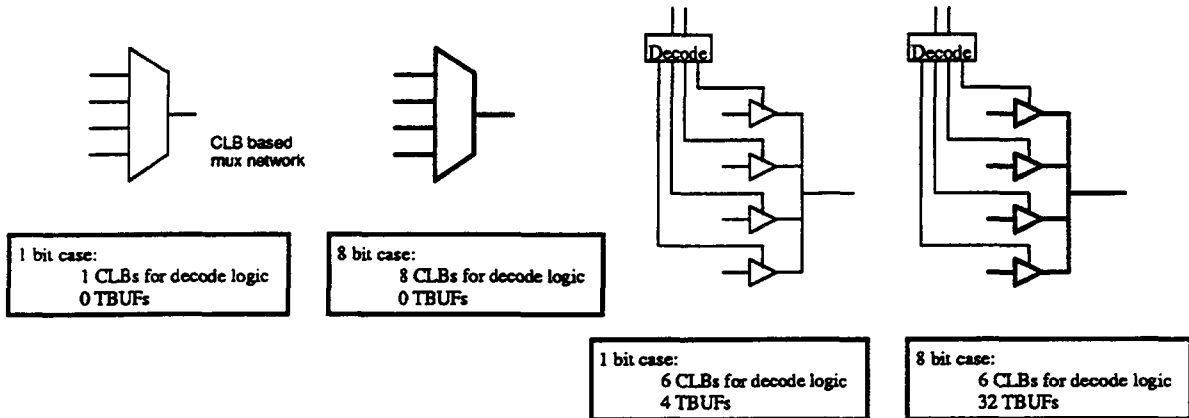


Figure 7

In the code shown below, we show the original designer source. The wide case statement is interpreted as a multiplexing function by the synthesis tools.

```

library IEEE;
USE IEEE.std_logic_1164.all;
ENTITY sample IS
  PORT(
    clk, rst      : IN  STD_LOGIC;
    data1_in     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    data2_in     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    mask         : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    s            : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    data_out     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
  );
END sample;

ARCHITECTURE behavior OF sample IS
  SIGNAL e: std_logic_vector (0 to 3);
  BEGIN
  PROCESS (rst, clk, s)
  BEGIN
    IF (rst = '0') THEN
      data_out <= "0000";
    ELSIF (clk'EVENT and clk = '1') THEN
      data_out <= e;
    END IF;
    CASE s IS
      WHEN "0000" => e <= "0000" ;
      WHEN "0001" => e <= "1111" ;
      WHEN "0010" => e <= data1_in;
      WHEN "0011" => e <= not data1_in ;
      WHEN "0100" => e <= data2_in;
      WHEN "0101" => e <= not data2_in ;
      WHEN "0110" => e <= data1_in and data2_in;
      WHEN "0111" => e <= data1_in or data2_in;
      WHEN "1000" => e <= mask;
      WHEN "1001" => e <= not mask;
      WHEN "1010" => e <= data1_in and mask;
      WHEN "1011" => e <= (not data1_in) and mask;
      WHEN "1100" => e <= data2_in and mask;
      WHEN "1101" => e <= (not data2_in) and mask;
      WHEN "1110" => e <= (data1_in and data2_in) and mask;
      WHEN "1111" => e <= (data1_in or data2_in) and mask;
      WHEN OTHERS => NULL;
    END CASE;
  END PROCESS;
END BEHAVIOR;

```

The wide-case schema is used to match the code and detect the applicability of the transformation to a tri-state hardware implementation because of the target technology library. The code can then be re-written (as shown below), such that the synthesis tools will immediately interpret the need for a tri-state implementation.

```

library IEEE;
USE IEEE.std_logic_1164.all;
ENTITY sample IS
  PORT(
    clk, rst      : IN  STD_LOGIC;
    data1_in     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    data2_in     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    mask         : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    s            : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
    data_out     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
  );
END sample;

ARCHITECTURE behavior OF sample IS
  SIGNAL e: std_logic_vector (0 to 3);
  BEGIN
    e <= "0000"          WHEN s = "0000" else "ZZZZ";
    e <= "1111"          WHEN s = "0001" else "ZZZZ";
    e <= data1_in        WHEN s = "0010" else "ZZZZ";
    e <= not data1_in    WHEN s = "0011" else "ZZZZ";
    e <= data2_in        WHEN s = "0100" else "ZZZZ";

```

```

e <= not data2_in          WHEN s = "0101" else "ZZZZ";
e <= data1_in and data2_in WHEN s = "0110" else "ZZZZ";
e <= data1_in or data2_in  WHEN s = "0111" else "ZZZZ";
e <= mask                  WHEN s = "1000" else "ZZZZ";
e <= not mask              WHEN s = "1001" else "ZZZZ";
e <= data1_in and mask     WHEN s = "1010" else "ZZZZ";
e <= (not data1_in) and mask WHEN s = "1011" else "ZZZZ";
e <= data2_in and mask     WHEN s = "1100" else "ZZZZ";
e <= (not data2_in) and mask WHEN s = "1011" else "ZZZZ";
e <= (data1_in and data2_in) and mask WHEN s = "1110" else "ZZZZ";
e <= (data1_in or data2_in) and mask WHEN s = "1111" else "ZZZZ";

PROCESS (rst, clk, s)
BEGIN
  IF (rst = '0') THEN data_out <= "0000";
  ELSIF (clk'EVENT and clk = '1') THEN data_out <= e;
  END IF;
END PROCESS;
END BEHAVIOR;

```

One of the more complicated patterns to match is that of a finite state machine. This is an important pattern because of the specialized analysis, optimizations, and code generation options that exist for state machines. Revealing one more layer of the transformation system, the partial code for the recognition of state machines from the program-tree is shown below.

### Procedural Pattern Matching

In this short section we introduce the pattern language used for this work. The introduction uses the example of a state machine recognizer based on the common two process style recommended by most synthesis vendors today

The Procedural Pattern Language(PPL) is associated with an interpreter. It operates on the hardware description language program tree. There are two simple variable types supported: string and integer. In addition there are two aggregate types : records and arrays. There are a number of variables that have a special relationship to the interpreter. These variables begin with @ . Variables in the example below that are specific to the base FSM schemata begin with \$.

The number, semantics and behavior of interpreter variables are application specific. For our example the special interpreter variables are:

```

@design_region - path to the root of interpreted subtree
@design_entity - record for current design entity
@process      - record for current process
@seq_statement - record for current statement in the process

```

The interpreter is initialized with a call to the init() procedure. This procedure takes @design\_region and sets the initial values for @design\_entity and @process. The construct : next(@design\_entity) permits us to iterate through the program tree design\_entity set. As a side effect of this switching, the @process is changed to the "first" process of new design entity. The same iterator is used for iterating through many program tree objects; for example: next(@process). We can also use "alter", an advanced iterator loop with alternatives. It combines iterator loop for the first parameter and a "switch/case" structure inside (see example below). The second parameter of "alter" is an expression for the calculation of the current case value. The case label contains the case value (which may be represented symbolically) and ,as an option, it can be a case\_condition expression in "{}". In our example each case\_condition means that the alternative can occur once and only once. Of course we have kept in mind a lot of expressions (including the function calls) that are useful as a case\_conditions. The variable @alter\_success is true if all present case\_conditions were satisfied in the last completed iteration loop.

```

MODULE FSM_PATTERN
  @design_region = path_to_design_subtree;
#   fill in some fields in $design_style record
  $design_style.general = SYNOPSIS;
  $design_style.FSM = FSM_uncode_style;
#   .....more init assignments .....
  init();
  do{
    alter( @process, CheckFSM(@process, $style) ){
      FSM_SEQ{#1}:
        clock_name = $FSM_clock_name;
        reset_name = $FSM_reset_name;
#       .....
      FSM_STATE{#1}:
        number_of_inputs = $FSM_number_of_inputs;
        number_of_outputs= $FSM_number_of_outputs;
        number_of_states = $FSM_number_of_states;
        implementation_name = $FSM_arch_name;
        symbol_name       = $FSM_entity_name;
#       .....
    }
    if( @alter success){
#     Check parameters and call FSM or some
#     transformation routine. Use @design_entity
#     as a pointer to FSM subtree. Use application calls
#     FSM_next_input(@process) and FSM_next_output(@process)
    }
#     .....
  }while(next(@design_entity));
checkFSM( @process, @style)
{
  if(@style.FSM == FSM_uncode_style){
#     next two assignments stand here only for clarification purpose
#     actually both of them can be placed in special section of init()
#     and their values depend on values of some @design_style fields.
    FSM_reset_cond_pattern = "$signal = $ZERO_OR ONE";
    FSM_clock_rise_pattern = "$signal'event and $signal = $ZERO_OR ONE";
#     try matching SEQ process
    if(@process.sens_list_size ==2)
      alter(@seq_statement, @seq_statement.name){
#       check if condition match reset pattern.
#       extract name of reset signal, if matching happened
        IF{#1}: /FSM_reset_cond_pattern/ @seq_statement.cond
              {FSM_reset_name = $signal;}
#       .....
#       check if condition match clock pattern.
#       extract name of clock signal, if matching happened
        ELSIF{#1}: /FSM_clock_cond_pattern/ @seq_statement.cond
              {$FSM_clock_name = $signal;}
      }
      if(@alter_success) return FSM_SEQ;
#     try matching combo process
#     .....
      alter(...){
      }
    }
    else if(@style.FSM == FSM_encode_style){
#     try matching encoding style FSM
#     .....
    }
  }
}
END MODULE FSM_PATTERN

```

### Conclusions & Future Work

This overview of the language transformation system has introduced the basic requirements and framework for the creation of such a tool. The tests to date have been very encouraging, and the general purpose nature of the architecture allows for many more experiments in the near future. One of the near term goals for the project is the transformations based on ASIC Libraries models. It is a good target for

this work because of the restricted language usage in the models. This makes the recognition patterns easy to describe. It is also easy to classify the types of models, which makes specialization of techniques straightforward. Finally, the basic elements of the model usually have nice language neutral formulations that lead to easy targeting optimizations for many different simulation engines, without regard to base language. Some preliminary experiments show significant potential model complexity reduction (thus faster simulation run times for the same function).

Another avenue of exploration is in general control and data-path partitioning. One of the more important reasons for performing this partitioning is to take advantage of alternate synthesis strategies for structured (as in data-path structures), versus more random logic (as in control dominated structures).

Another use of the partitioning application of transformations is to permit library mapping to pre-optimized structures. The structures may or may not be accessed via HDL synthesis tools (the operations can be specialized to run technology specific module generation programs that are merged at the netlist level).

Finally, there are many opportunities to apply existing research results from the software domain to the HLDA solution space. We have only begun to scratch the surface in terms of specializing existing work to HDL re-writing.

## References

Petrossi and Proietti 1996. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, Vol. 28, No. 2, June 1996

Boyer, R. S. and Moore, J. S. 1975. Proving theorems about LISP functions. *JACM* 22, 1, 129—144.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *JACM* 24, 1, 44—67.

Ershov, A. P. 1982. Mixed computation: Potential applications and problems for study. *Theor. Comput. Sci.* 18, 41—67.

Paige, R. and Koenig, S. 1982. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.* 4, 3, 402—454.

Partsch, H. A. 1990. *Specification and Transformation of Programs*. Springer Verlag, New York.

Scherlis, W. L. 1981. Program improvement by internal specialization. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages* (Williamsburgh, VA), 41—49.

Turner, D. A. 1979. A new implementation technique for applicative languages. *Softw. Pract. Exper.* 9, 31—49.

Wadler, P. L. 1985. Listlessness is better than laziness. Computer Science Department, CMU-CS-85-171, Carnegie Mellon University, Pittsburgh, PA, Ph.D. Thesis.