Methodology for Separation of Design Concerns Using Conservative RTL Flipflop Inference

Maya H. Safieddine, Fadi A. Zaraket, Rouwaida Kanj
American University of Beirut, Lebanon
Ali Elzein, Wolfgang Roezner
IBM Systems and Technology Group, Austin TX
Outline

- Aspect Oriented Design
  - Hardware Perspective
- Methodology for Separation of Hardware Design Concerns
  - RTL-to-RTL Transformation Block
  - Correctness
  - Concern Insertion Block
- Implementation and Results
- Conclusions
Aspect Oriented: What’s Behind the Big Name

• Concept borrowed from software: Aspects support separation of concerns
  – Example: log every function call w/o modifying functions
  – Languages (AspectC, JAspect) separate coding of aspects

• Easy to make changes in every aspect

• Did not take off due to the lack of useful applications, legacy software, infrastructure and integration
Functional Queue class

```cpp
class Queue { public:
    Item * first, *last;

    #ifdef COUNTING_ASPECT
    int counter;
    #endif

    #ifdef LOCKING_ASPECT
    os::Mutex lock;
    #endif

    Queue() : first(0), last(0) {
        #ifdef COUNTING_ASPECT
        counter = 0;
        #endif
    }

    void enqueue(Item* item) {
        #ifdef LOCKING_ASPECT
        lock.enter();
        try {
            #ifdef COUNTING_ASPECT
            if (counter > 0) --counter;
            #endif
            #ifdef ERRORHANDLING_ASPECT
            if (item == 0)
                throw QueueInvalidItemError();
            #endif
            if (last) {
                last->next = item;
                last = item;
            } else first = first->next;
        } catch(...) {
            lock.leave(); throw;
        }
        lock.leave();
        #endif
    }

    void dequeue (Item *item) {
        Item* res = first;
        #ifdef LOCKING_ASPECT
        lock.enter();
        try {
            #ifdef ERRORHANDLING_ASPECT
            if (res == 0)
                throw QueueEmptyError();
            #endif
            #ifdef LOCKING_ASPECT
            } catch(...) {
                lock.leave(); throw;
            }
            lock.leave();
            #endif
        } return res;
    }

    #ifdef COUNTING_ASPECT
    int count() { return counter; }
    #endif
}; // class Queue
```
Aspect Oriented Design

- Cross-cutting (coupling) of concerns exists in hardware
  - Designers have to deal with highly coupled code
  - Verification is challenging and late in design cycle
- We identify several hardware concerns that can be isolated into aspects
Advantages of Aspect Oriented Design

• Increase Designers productivity
  – Automatic insertions of several aspects
  – Designers work in parallel coding 1) aspects, 2) aspect supporting HDL, and 3) functional HDL
  – Internally, aspects are referred to as recipe files

• Faster Verification
  – Smaller model: Aspect-specific verification happens on the model that only contains the corresponding aspect
  – Verification environment is more stable
    – Changes to aspects that are not part of the verification task have no effect on the environment
    – Easier to debug
Old Design Paradigm

1. Environment constantly changes due to Physical Design (PD) changes.
2. Slower Verification: All tasks share the same model containing all design aspects.

Iterative back-annotation of PD artifacts makes implementation and integration very inefficient and increasingly time consuming.
Aspect Oriented Based Design Paradigm

Asics Style HDL algorithmic

Transformation

High Perf. HDL (FSMs)

Weaving

Scan Chains

ABIST

VHDL Logical Hierarchy

Weaving

Complete Design Physical Hierarchy

Functional Verification

Pervasive Verification/ DFT Analysis

Recipe

Recipe

Physical tools

Logic Engineers

Test/Pervasive Engineers

Integrators
Aspect Oriented Based Design Paradigm

- More Stable Verification Env
- Faster simulation (smaller model)

Functional Verification
- High Perf. HDL (FSMs)
- Weaving
- Scan Chains ABIST
- VHDL Logical Hierarchy
- Weaving
- Complete Design Physical Hierarchy

Physical tools

Logic Engineers
Test/Pervasive Engineers
Integrators

- Feedback now goes separately to different Engineers.
- Every Engineer is now in control of the changes needed to address the issues related to his/her area of expertise

Asics Style HDL algorithmic Transformation
Four Transforms

1. **Structural Based**: Identifies RTL sequential blocks that can directly be mapped to DFFs
   \[\Leftrightarrow \text{Simple Blocks}\]

2. **Algorithmic Based**: Maps generic RTL sequential blocks into RTL combination blocks and flipflop instantiations
   \[\Leftrightarrow \text{Advanced Blocks}\]

3. **Synthesis Based**: Maps RTL sequential blocks into low level netlists
   \[\Leftrightarrow \text{Algorithmic Engine Fails}\]

4. **Synthesis-Algorithmic Based**: Synthesis results enable algorithmic transformation
   \[\Leftrightarrow \text{Advanced Engine}\]
   - Not implemented yet: Currently reverting to Synthesis Based Transform
//set-resets in structural always block
//structural

module set1(in,clk,set,reset,out);
    input in, clk, set, reset;
    output reg out;

    always @ (posedge clk or posedge reset)
    if (reset)
        out <= 0;
    else if (set)
        out <= 1;
    else
        out <= in;
endmodule

module set1(in,clk,set,reset,out);
    input in, clk, set, reset;
    output out;
    wire out;

dff out_0(
    .d(in), .clk(clk),
    .async_reset(reset),
    .async_data(0),
    .sync_reset(set),
    .sync_data(1), .gate(1'b1),
    .q(out));
endmodule
Algorithmic Based Transformation

Advanced Blocks:

1. Any signal set inside an always block with posedge or negedge (or any combination of them) is inferred flipflop

2. Extract clocking and asynchronous logic: create the DFF model

3. Create the combination always block

4. Generate equivalent HDL
Combinational Block

1. Drop from the original process the asynchronous set/reset logic as well as the clocking construct.

2. For every flipflop signal assigned

<table>
<thead>
<tr>
<th>Flip Flop Signal</th>
<th>Assignment</th>
<th>Target of assignment</th>
<th>Referenced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blocking</td>
<td>Non-blocking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Replace it with FF input</td>
<td>Replace it with FF input</td>
<td>Keep it</td>
</tr>
</tbody>
</table>

- For all flipflops add an assignment at the beginning of the process, assigning the output of the flipflop to its input
  - Ensures no latches inferred inside the combination always block
  - Alleviates understanding latch inference algorithms
Algorithmic Transform

module example (  
inputs cond1, cond2,  
    a, b, c, d,  
outputs reg r, r2,  
inputs clk, ...);  
reg r3, x;  
always@( posedge clk, ...)  
begin  
    if(cond1==1) begin  
        r3 <= a;  
        x = a || b;  
    end  
    r <= x;  
    if(cond2==1) begin  
        x = c && d;  
    end  
    r2 <= r3 && x;  
end  
endmodule

Step 1: Flipflop Inference
module example (  
inputs cond1, cond2,  
    a, b, c, d,  
outputs reg r, r2,  
inputs clk, ...);  
reg r3, x;  
always@( posedge clk, ...)  
begin  
    if(cond1==1) begin  
        r3 <= a;  
        x = a || b;  
    end  
    r <= x;  
    if(cond2==1) begin  
        x = c && d;  
    end  
    r2 <= r3 && x;  
end  
endmodule

Inferred flipflops:  
r, r2, r3, x

Step 1: Flipflop Inference
Step 2: Add wire and reg variables to map to the output and next state of the flipflops

```
module example (  
    inputs cond1, cond2,  
    a, b, c, d, 
    outputs wire r, r2, 
    input clk, ...); 
wire r3, x;  
always@( posedge clk, ...) 
begin 
    if(cond1==1) begin 
        r3 <= a;  
        x = a || b; 
    end  
    r <= x;  
    if(cond2==1) begin 
        x = c && d;  
    end  
    r2 <= r3 && x; 
end 
endmodule 
```
module example (  
    inputs cond1, cond2,  
        a, b, c, d,  
    outputs reg r, r2,  
    inputs clk, ...);  
reg r3, x;  
always@( posedge clk, ...)  
begin  
    if(cond1==1) begin  
        r3 <= a;  
        x = a || b;  
    end  
    r <= x;  
    if(cond2==1) begin  
        x = c && d;  
    end  
    r2 <= r3 && x;  
end  
endmodule

Step 3: Flipflop Instantiation

module example (  
    inputs cond1, cond2,  
        a, b, c, d,  
    outputs wire r, r2,  
    input clk, ...);  
wire r3, x;  
reg r_in, r2_in, r3_in, x_in;  
ff ff_r(r_in, r,0, clk ...);  
ff ff_r2(r2_in,r2,0,clk ...);  
ff ff_r3(r3_in,r3,0,clk ...);  
ff ff_x(x_in, x,0, clk ...);  
...
module example (  
inputs cond1, cond2,  
a, b, c, d,  
outputs reg r, r2,  
inputs clk, …);  
reg r3, x;  
always@( posedge clk, …) 
begin 
  if(cond1==1) begin 
    r3 <= a;  
    x = a || b;  
  end  
  r <= x;  
  if(cond2==1) begin 
    x = c && d;  
  end  
  r2 <= r3 && x;  
end 
endmodule

module example (  
inputs cond1, cond2,  
a, b, c, d,  
outputs wire r, r2,  
input clk, …);  
wire r3, x;  
reg r_in, r2_in, r3_in, x_in;  
ff ff_r(r_in, r,0, clk …);  
ff ff_r2(r2_in,r2,0,clk …);  
ff ff_r3(r3_in,r3,0,clk …);  
ff ff_x(x_in, x,0, clk …);  
…

Concern weaving requires only instantiating the corresponding flipflop library by substituting ff

Sequential Block

Step 3: Flipflop Instantiation

Input RTL
module example (  
inputs cond1, cond2,  
a, b, c, d,  
outputs reg r, r2,  
inputs clk, ...);  
reg r3, x;  
always@( posedge clk, ...)  
begin  
  if(cond1==1) begin  
    r3 <= a;  
    x = a || b;  
  end  
  r <= x;  
  if(cond2==1) begin  
    x = c && d;  
  end  
  r2 <= r3 && x;  
end  
endmodule

Step 4: Construction of the combination logic block
- Explicit connection of flipflop input and output signals

...  
always@(*) begin  
  if(cond1==1) begin  
    r3_in <= a;  
    x_in = a || b;  
  end  
  r_in <= x_in;  
  if(cond2==1) begin  
    x_in = c && d;  
  end  
  r2_in <= r3 && x_in;  
end
module example (  
  inputs cond1, cond2,  
              a, b, c, d,  
  outputs reg r, r2,  
  inputs clk, ...);  
regr3, x;  
always@( posedge clk, ...)  
begin  
  if(cond1==1) begin  
    r3 <= a;  
    x = a || b;  
  end  
  r <= x;  
  if(cond2==1) begin  
    r3 <= a;  
    x_in = a || b;  
  end  
  r2 <= r3 && x;  
end  
endmodule  
always@(*)  begin  
  r_in = r;  
  r2_in = r2;  
  r3_in = r3;  
  x_in= x ;  
  if(cond1==1) begin  
    r3_in <= a;  
    x_in = a || b;  
  end  
  r_in<= x_in;  
  if(cond2==1) begin  
    r3_in <= a;  
    x_in = a || b;  
  end  
  r2_in <= r3 &&x_in;  
end  
Combinational Block  
  
Inferred flipflops: r, r2, r3, x  
  
Step 5: Combination logic block  
• Removal of all possible flipflop inferences
RTL-to-RTL Transformation

module example (  
    inputs  cond1, cond2,  
        a, b, c, d,  
    outputs wire r, r2,  
    input clk, ...);  
wire r3, x;  
reg r_in, r2_in, r3_in, x_in;  

    ff ff_r(r_in, r, 0, clk ...);  
    ff ff_r2(r2_in, r2, 0, clk ...);  
    ff ff_r3(r3_in, r3, 0, clk ...);  
    ff ff_x(x_in, x, 0, clk ...);  

always@(*) begin  
    r_in = r;  
    r2_in = r2;  
    r3_in = r3;  
    x_in = x;  
    if(cond1==1) begin  
        r3_in <= a;  
        x_in = a || b;  
    end  
    r_in <= x_in;  
    if(cond2==1) begin  
        x_in = c && d;  
    end  
    r2_in <= r3 && x_in;  
end

Sequential Part  
Instantiated Generic Flipflops  

Combinational Part  
Logic gates and Connections
Sketch of the Transformation
Correctness Proof

Given

- Mealy Machine M
- Mealy Machine M’ with separated Combinational and Sequential Structures

If \{L_1, \ldots, m \equiv L_0^{0,1,\ldots,m}\} \& \{\delta_1, \ldots, k \equiv L_{\text{nsf}}^{0,1,\ldots,k}\} then \ M' \equiv M

ff: Flipflops / Registers
RI: Register initial values
I: Input wires
O: Output wires
\delta: Next State functions
L: Output functions

Mealy Machine M = <ff, RI, I, O, \delta, L>
Mealy Machine M’ = <ff’, RI’, I’, O’, \delta’, L’>

M’ = M^c \times M^s
Sketch of the Transformation

Correctness Proof

\[ \{ L_1, \ldots, m \equiv L^o_{1, \ldots, m} \} \& \{ \delta_1, \ldots, k \equiv L^{nsf}_{1, \ldots, k} \} \]

- Original RTL synthesized to \( M \)
- Transformed RTL synthesized to \( M' \)

\[ I = I' = <PI', clk> = <PI, clk> \]

\[ M' <ff', RI', I', O', \delta', L'> = M^c \times M^s \]
Program P

**P(V, S)**

**Variables V:**
- Inputs I
- Outputs O
- Registers R
- Wires W

**Statements S:**
- Blocking assignments \((BS) \ "u = exp"\)
- Non-blocking assignments \((NBS) \ "u <= exp"\)
- Other Statements

**Original RTL Module**

\[ P(V, S) \]

**Transformation**

**Transformed RTL Module**

\[
\begin{align*}
P^c(V^c, S^c) & \quad \text{Combinational} \\
P^s(V^s, S^s) & \quad \text{Sequential}
\end{align*}
\]
Program P synthesis

\[ P(V, S) \]

\[ V = \{ I, O, W, R \} \]

\[ S = \{ BS, NBS, Other \} \]

Assignment statements are in the form: \( u = \text{exp} \)

\[ ff = \text{Synthesis}(R) \]
\[ I = \text{Synthesis}(I) \]
\[ O = \text{Synthesis}(O) \]
\[ \delta = \text{Synthesis}(S \mid u \in R) \]
\[ L = \text{Synthesis}(S \mid u \in O) \]
**Program P**

<table>
<thead>
<tr>
<th>Original RTL Module</th>
<th>Transformation</th>
<th>Transformed RTL Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P( V, S) )</td>
<td></td>
<td>( P^c( V^c, S^c) )</td>
</tr>
</tbody>
</table>

\[ P^c( V^c, S^c) \]

\( V^c = \{ I^c, O^c, R^c, W^c \} \)

\( R^c = \{ \} \)

\( W^c = V \cup \{ u_{in} : u \in R \} \)

\( O^c = O \cup \{ u_{in} : u \in R \} \)

\( I^c = I \cup R \)

\( U_{tb} = \{ u: u \in R, u \text{ is target of blocking statement} \} \)

\( U_{tn} = \{ u: u \in R, u \text{ is target of nonblocking statement} \} \)

\( S^c \) is defined iteratively as follows

Let \( S^{c(0)} = S, h_n = |U_{tn}| \) and \( h_b = |U_{tb}| \)

\[
S^{c(i)} = \begin{cases} 
S^{c(i-1)}_{u_{i-1}} & i = 1, \ldots, h_b \\
S^{c(i-1)}_{u_{i-1}} & i = h_b + 1, \ldots, h_b + h_n 
\end{cases}
\]

and \( u_i \) is a target

**Notes:**
- **V:** Variables
- **I:** Inputs
- **O:** Outputs
- **R:** Registers
- **W:** Wires
- **S:** Statements
- **BS:** Blocking assignments
- **NBS:** Non-blocking assignments

**Abbreviations:**
- **ff:** Flipflops /Registers
- **RI:** Register initial values
- **O:** Output wires
- **δ:** Next State functions
- **L:** Output functions

---

3/2/2022

Maya H. Safieddine, American University of Beirut
**Pc Synthesis**

\[ R^c = \{ \} \]
\[ W^c = V \cup \{ u_{in} : u \in R \} \]
\[ O^c = O \cup \{ u_{in} : u \in R \} \]
\[ I^c = I \cup R \]

\{ PI', Iff \} = Synthesis(I^c)

\{ O', O_{nsf} \} = Synthesis(O^c)

\{ L_{c,O}, L_{c,nsf} \} = Synthesis(S^c | u \in O^c)

\[ I = I' = <PI', clk> = <PI, clk> \]

**Abbreviations:**
- **V:** Variables
- **I:** Inputs
- **O:** Outputs
- **R:** Registers
- **W:** Wires
- **S:** Statements
- **BS:** blocking assignments
- **NBS:** Non-blocking assignments

**Notations:**
- \( f_f \): Flipflops / Registers
- \( R_I \): Register initial values
- \( I \): Input wires
- \( O \): Output wires
- \( \delta \): Next State functions
- \( L \): Output functions

Transformed RTL Module

Combinational

Sequential
Program $P_s$

Original RTL Module
$P(V, S)$

Transformation

Transformed RTL Module

\[ P_c(V^c, S^c) \]
\[ P_s(V^s, S^s) \]

$P_s(V^s, S^s)$

$V^s = \{ I^s, O^s, R^s, W^s \}$

$R^s = R$

$I^s = I + \{ u_{in} : u \in R \}$

$O^s = R$

$W^s = \{ \}$

$S^s = \{ s_i^s : \text{ff} \_u(u_{in}, u, 0, \text{clk}, ...) \}$, where $u \in R$

\[ ff: \text{Flipflops/Registers} \]
\[ O: \text{Output wires} \]
\[ RI: \text{Register initial values} \]
\[ \delta: \text{Next State functions} \]
\[ I: \text{Input wires} \]
\[ L: \text{Output functions} \]
Ps Synthesis

\[ R^s = R \]
\[ I^s = I + \{ u_{in} : u \in R \} \]
\[ O^s = R, \quad \text{(Simple Assignment)} \]
\[ W^s = \{ \} \]
\[ S^s = \{ s_i^s : ff_u(u_{in}, u, 0, clk, \ldots) \} \]
where \( u \in R \) \quad \text{(Simple Declaration)}

ff\textsuperscript{s} = Synthesis(R\textsuperscript{s})

\{I_{clk}, \ldots, I_{nsf}\} = Synthesis(I\textsuperscript{s})

O\textsuperscript{ff} = Synthesis(O\textsuperscript{s})

\[ \delta^s = \{ \} \quad \text{(Empty Functions (wiring))} \]
\[ L^s = \{ \} \quad \text{(Empty Functions (wiring))} \]
P' Synthesis into M'
Functional Equivalence by structural induction

- Equivalence of the synthesis of expressions of original and transformed programs

\[
\forall \exp_{c,i} \in P_c, \exp_i \in P, \ \exp_{c,i} = \exp_i \mid u_j \text{ with } u_{in,j} \text{ for every } u_j \in U_{tb}
\]

\[
\therefore [\text{syn}(u_{in,j}) \text{ in } M^c] \iff [\text{syn}(u_j) \text{ in } M] \land [\text{syn}(u_k) \text{ in } M^c] \iff [\text{syn}(u_k) \text{ in } M]
\]

\[
\rightarrow [\text{syn}(\exp_{c,i})] \iff [\text{syn}(\exp_c)] \text{ where } u_k \in U_{tn}
\]

- Base and recursive cases for the synthesis equivalence proof of \(u_{in}\) and \(u\) in \(M\) and \(M'\)

\[
\text{syn}(u_{in,j}) = \begin{cases} 
\text{flipflop corresponding to } u_i & \text{base case} \\
\text{output wire of } \text{syn}(\exp_{c,j}) & \text{recursive case}
\end{cases}
\]

\[
\text{syn}(u_k) = \text{flipflop of value} \begin{cases} 
\text{initial register value (input)} & \text{base case} \\
\text{syn}(\exp_{c,k}) & \text{recursive case}
\end{cases}
\]

- Equivalence of output functions in \(M'\) to output and transition functions in \(M\)

\[
l^c = \text{syn}(\exp_c) \ \forall l^c \in L^c
\]

Therefore: \(L^{c,nsf} \iff \delta\) and \(L^{c,0} \iff L\)
Transformation Equivalence

Original RTL Module

Transformed RTL Module

Combinational

Sequential

Synthesis

Since \( \{ L_1, \ldots, m \equiv L_{1, \ldots, m}^0 \} \& \{ \delta_{1, \ldots, k} \equiv L_{1, \ldots, k}^{nsf} \} \) then \( M' \equiv M \)

\[ \Leftrightarrow \text{synthesis}(P') = M' \equiv M = \text{synthesis}(P) \]

\[ \Leftrightarrow P' \equiv P \]
Implementation and Results

• Developed and deployed an implementation for Verilog RTL
• The implementation supports:
  – Automatic insertion of design concerns at RTL
  – Mapping to proprietary technology at RTL
  – Lifting verification to the RTL level
• Reduced verification time by 40% in the last design cycle
• No drawbacks on synthesis optimizations
Conclusion

• We introduce the concept of aspect oriented design into hardware by a methodology for automatic concern insertion into RTL programs.

• The methodology reduces verification time by allowing it to take place at the RTL level.

• The methodology can be generalized to different hardware languages, and is proved to maintain original program functionality.

• It is currently used in an industrial settings and has produced significant design cycle time savings for state-of-the-art SoC designs.
Thank You
Synthesis Based Transformation Example

```
module mor1 (clk, en, d1, d2, q);
  input wire clk, en;
  input wire d1, d2;
  output reg q;

  reg tmp;

  always @(posedge clk) begin
    if (en)
      tmp = d1 & d2;
    else
      tmp <= !d1;
    q <= tmp;
  end
endmodule
```

```
module mor1 (clk, en, d1, d2, q);
  input wire clk, en;
  input wire d1, d2;
  output wire q;

  reg tmp;
  wire [0:0] $0;
  wire [0:0] $1;
  wire $5;
  wire [0:0] $2;

dff dff_q(.d($4), .clk(clk), .q(q));
dff dff_tmp(.d($5), .clk(clk), .q(tmp));

  assign $0  = (d1 & d2);
  assign $1  = (~en);
  assign $2  = (d1 == 1'b0);
  assign $5  = (en ? $0 : $2);
  assign $4  = (en ? $0 : tmp);

endmodule
```