Methodology for Separation of Design Concerns Using Conservative RTL Flipflop Inference

Maya H. Safieddine, Fadi A. Zaraket, Rouwaida Kanj
American University of Beirut, Lebanon,email: {mhs36, fz11,rk105}@aub.edu.lb

Ali Elzein, Wolfgang Roezner
IBM Systems and Technology Group, Austin TX, email: {elzein, wolfgang}@us.ibm.com

***Abstract*- Design-for-test, LBIST, memory technology mapping and clocking concerns require team-months of verification time as they traditionally happen at gate-level. We present a novel concern-oriented methodology that enables automatic insertion of these concerns at the register-transfer-level where verification is easier. At the heart of our methodology is a conservative flipflop inference transformation that takes entry RTL and outputs RTL where memory is separated from functionality. We prove the transformation correct by construction. We implemented the methodology in a tool that also automatically weaves the different concerns into the output RTL. The tool reduces verification time by 40% when used in industry.**

I. Introduction

Recent research and industry reports suggest that hardware design and verification schemes incorporate separation of concerns and aspect oriented design to shorten verification debug time and meet the short time-to-market requirements [1]. System-on-Chip (SoC) and high-performance processor design flows use VHDL, Verilog, SystemVerilog, and less often SystemC as the Register Transfer Level (RTL) description in their design and verification platforms [2]. Existing RTL logic exhibits IP modules with high coupling between several concerns such as Design-for-Test (DFT), Logic Built-in Self-Test (LBIST), memory technology mapping, power-gating and clock divisions [3-6]; i.e. all these concerns exist in the same module. Logic designers are solely concerned with functionality of the code, timing engineers are concerned with the delay characteristics and models for purposes of timing verification, and power engineers are concerned with power grid, clock gating, and power-gating logic for purposes of power verification. Coupling all of these concerns in one module renders the RTL code too difficult to handle. Manual DFT Logic placement techniques [7], for example, require inspection of the whole RTL as well as explicit instantiation of the DFT elements. On the other hand, automated insertion of the DFT memory elements traditionally happens post-synthesis at the gate-level resulting in a significant overhead in the verification debug time. The designers long for a methodology that enables automatic separation of concerns at the RTL level, drives more automation, and improves design and verification efficiency.

Separation of concerns in software applications is supported by means of Aspect-Oriented Programming (AOP) where functionality is explicitly separated from other concerns. These concerns are specified in *aspects* that are later automatically interwoven into the program. AspectVHDL considers concepts from AOP in HDL and proposes an aspect-oriented extension of VHDL [8]. It provides an approach for easily plugging extensions with aspects that crosscut a VHDL code base. It introduces to the VHDL language *joint points* of *procedures, types, architectures*, and *process triggers*, and related *pointcut expressions* and *advices*. However, the implementation is preliminary and the approach cannot capture logic design concerns such as DFT, LBIST and clock-gating.

Frameworks for embedded systems design, such as Metropolis [9, 10, 11], and Behavior Interaction Priority (BIP) [12,13], explicitly separate concerns. Metropolis separates (1) communication from computation, (2) functionality from architecture, and (3) behavior from performance. BIP separates behavior from synchronization and data transfer structure through a hierarchy of prioritized interactions. A SystemC analog of BIP separates functionality, timing, synchronization, and transactional concerns [14]. These approaches address concern separation in heterogeneous systems by defining their own syntax and semantics.

More recently, commercial solutions are being developed to render RTL scan-compliant [15,16]. Yet up to our knowledge, no solution exists that presents all design flow concerns separately at the RTL level.

In this paper, we present a novel methodology for separation of logic design concerns at the RTL level focusing on improving design and verification efficiency. Our contributions are: 1) A flipflop inference based transformation that is conservative in that it instantiates a flipflop for every variable that could be a memory element. 2) The transformation results in the separation of the original RTL code into separate combinational and sequential programs. 3) A proof of correctness of the transformation. 4) A supporting tool that implements the methodology for Verilog systems.

This methodology has the following advantages:

1. It enables verification of the concerns at RTL,
2. It allows automatic insertion of design elements relevant to DFT, LBIST, flipflop technology mapping and clocking concerns at the RTL level with *reduced coupling*, and
3. It reduces verification time and thus enhances design cycle efficiency.

The remainder of this paper is organized as follows. Section II provides a review of existing concern separation tools. Section III describes DFT insertion in current design flows in terms of automatic gate-level or manual RTL approaches. Section IV presents the proposed methodology along with an example program transformation. Section V provides a proof of correctness for the transformation. Section VI presents the experimental setup and results. Finally, section VII presents the conclusions.

II. Related Work

Metropolis is an embedded system design platform based on formal modeling and separation of concerns for an effective design process. It structurally separates communication from computation, behavior from performance, and function from architecture. Process and media structures ensure the communication-computation separation. The quantity manager structure, along with the *annotator* and the *scheduler* objects [11], manages energy, time and event priorities and thus separates behavior from performance. Networks are composed of processes and media objects that are connected via ports and interfaces. One mapping network separates functionality from architecture and maps the hierarchical networks to architecture specifying software or hardware implementations of each network.

BIP is a component-based framework for embedded systems design with behavior, interaction, and priority layers that separate computation from communication and prioritization and enhance reusability, scheduling analysis, and re-configurability. Components in BIP are composed by the superposition of the three layers. A BIP extension of Petri-nets with data elements defines *atomic* component sand specifies the behavior layer. The *connector* objects specify interactions and allow communication among atomics. The scheduling of these interactions forms the priority layer [12,13].

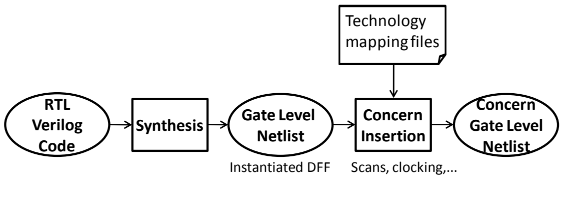
Transaction-level modeling is proposed as a methodology for separating timing and functionality in SoC [14]. A component is separated into a programmer view (PV) model and a PV with timing (PVT) model. PV models target functionality and are essential for programmers who are not concerned with micro-architecture and timing details. PVT models include realistic timing and are built on top of PV models. PV and PVT models are weaved dynamically by an alternating execution model that depends on System Synchronization Points (SSPs) explicitly declared in the PV model.

These approaches address concern separation, however, by devising their own syntax and semantics. Furthermore, none of these approaches tackle the separation of concerns at the logic level like DFT, LBIST, memory technology mapping and clocking. We would like for our approach to target these concerns and to be applicable to industrial RTL such as Verilog and VHDL without enforcing any language restrictions.

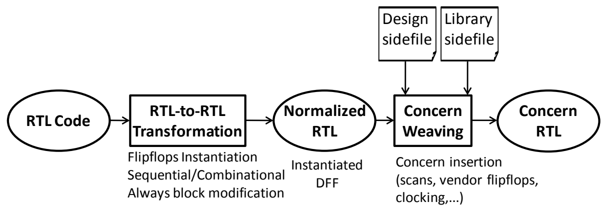
SpyGlassfrom Atrenta [15] creates a hardware virtual prototype of the design to enable early analysis and optimization of the area, timing and power constraints. SpyGlassDFT [16] wraps around the synthesis phase to ensure that RTL is scan-compliant and yields highest coverage. Up to our knowledge, no further description of their methodology is available in literature. Our approach, on the other hand, will provide a solution for concern separation solely at the RTL level with minor modifications to the original input RTL code.

III. DFT And Memory Related Concern Insertion For Traditional Design Flows

Currently, concerns are inserted in the design flow either manually at the RTL level at the expense of added design complexity, or automatically at the gate-level at the expense of additional verification time. In automatic gate-level concern insertion, the concern functionality (e.g., DFT), which is not coded as part of the RTL, is inserted automatically by synthesis tools at the gate-level [17] as illustrated in the design flow of Figure 1(a). The advantage of this approach is that it does not couple the design with a specific concern mechanism, and it frees logic designers from the concern functionality. However, this prohibits verification from occurring early at RTL and forces it at the



(a)



(b)

Figure 1 A traditional gate level flow (a) and the proposed RTL concern insertion flow (b).

gate-level where it is more complicated and requires more design flow iterations. This results in significant increase in verification time.

Concern insertion at the RTL level is therefore highly desired as it allows early verification of the concern functionality and reduces the number of design flow iterations between RTL and gate-level. Currently, this is enabled via manual concern insertion, where DFT scan chains, for example, are explicitly instantiated by the designers at the RTL.

However, this is an exhaustive approach and the designers must abide by *strict code style restrictions on RTL variables*; hence, they need to ensure that all variables that are not explicitly instantiated as flipflops are purely combinational. Writing the RTL under these restrictions becomes challenging and time consuming. This is because the designer, who should be concerned about algorithms, now also needs to worry about the different variable definitions, blocking or non-blocking statement modifications, the instantiation, wiring, and implementation of the flipflop; thereby, *guessing what the synthesis would have done*. The designer also needs to handle the weaving of the different concerns properly. This includes sophisticated modules (hierarchically nested) that accommodate for multiple concerns as scan design, frequency division, clock-gating and DFT, as well as sensitivity lists handling. Similarly, other concern specific vendor-based or proprietary industrial modules may be inserted. Other complexities may also require the designer to add local clock buffers (LCB), LCB control cells, and clock divisions as well as understand their semantics and implications.

In addition, the manually processed RTL will be coupled with a specific mechanism. Switching to other mechanisms or concerns requires RTL rewriting. Hence, there is a clear necessity for automating the process to leverage the advantages of RTL concern insertion.

Our proposed methodology offers a solution for concern functionality integration that alleviates the disadvantages of approaches described above.

IV. Proposed Methodology

Concerns that relate to design memory elements, represented as flipflops in the synthesized circuits, require the identification of these elements prior to integrating the concerns with the legacy code at either RTL or gate-level. *Key to enabling the integration of these concerns at RTL is the capability of converting RTL with flipflop inference into RTL with flipflop instantiation*. We propose a methodology that explicitly identifies memory elements, and automatically converts input RTL into output RTL where combinational logic is separated from sequential logic, and where concern insertion is a straight forward flipflop replacement at the RTL level within the sequential logic. The changes our method introduces in the output RTL are small such that a designer who understands the input RTL can easily understand the output RTL. Figure 1(b) presents the overall methodology flow diagram. Two major tools are involved to enable the automatic concern insertion at RTL.

1. **RTL-to-RTL transformation tool**: Infers memory elements and transforms the original RTL into RTL with generic instantiated memory elements.
   - **a.** Identifies RTL memory elements that can directly be mapped to flipflops.
   - **b.** Maps original RTL blocks into separate (1) combinational RTL blocks and (2) sequential RTL blocks with generic RTL flipflop instantiations.
   - **c.** Outputs separated and "normalized" RTL code where inferred memory elements are replaced with generic flipflop gates and Verilog formatting including comments, attributes and pragmas are preserved.
2. **Concern insertion tool**: A tool to insert concern structures into a "normalized" RTL code.
   - **a.** Re-maps the instantiated generic flipflops into library components that support the concern. The components route the additional clocking and the asynchronous logic signals across all the relevant module declarations.
   - **b.** Instantiates LCBs, scan control cells …etc.
   - **c.** Hooks up and balances scan chains.
   - **d.** Is controlled by two sidefiles:
     - *Library Side file:* Specifies flipflop/LCB library and connection rules

- *Design Side file:* Permits some overriding of Library Side file, and specifies flipflop properties, LCB types, Scan chain types, and test ports …etc.

e. Output RTL with concern insertion.

This provides a complete design with weaved separated concerns and elevates concern verification to the RTL level. Translation of event traces all the way to the RTL will still be supported by the methodology when hardware synthesis optimizations are included in the design flow. This is because downstream hardware synthesis optimizations typically preserve a two-way mapping between the original and the optimized modules. In the following section, we describe the RTL-to-RTL transformation.

A. RTL-to-RTL Transformation

The proposed transformation applies to all synthesizable Verilog and VHDL constructs. We illustrate it for a sufficient subset of Verilog including: conditionals, assignments, arithmetic, and Boolean constructs. The steps we present below are for Verilog RTL. We rely on the example module in Figure 2 for illustration. For VHDL implementation, the main difference is in the flipflop inference step (step 1 below).

Step1: Any variable assigned inside an *always* block with *posedge* or *negedge* is inferred as a flipflop.

- In '*Before'* of Figure 2, *r, r2, r3,* and *x* are identified to be flipflops.

Step2: For every flipflop u as determined in step 1, introduce two declarations '*wire* u*'* and '*reg* u_in' in '*After'* (in Figure 2) to represent the output and the next state (input) of the flipflop, respectively.

- In '*Before'*, we declare *wires r, r2, r3, and x* and *regs r_in, r2_in, r3_in, and x_in* for the inferred flipflops in step 1.

Step3: For every flipflop u as determined in step 1, instantiate a library flipflop *ff* in '*After'* where 'u' is the output and 'u_in' is the next state function. *ff* is a generic flipflop that will later be mapped to the technology that supports the target concern. Extract clocking and asynchronous logic from '*Before'* and pass them to the instantiated library flipflops in '*After'*. The flipflop instantiations constitute the sequential block.

- In '*Before'*, we instantiate generic flipflops *ff_r, ff_r2, ff_r3,* and *ff_x.*

Step4: In '*After'* create a combinational always@(*) block. Copy all the assignments and their control logic from '*Before'* to '*After'*. Modify them as follows. Step 4 is formally defined in Table 1.

For every flipflop u as determined in step 1:

a. If it is assigned with non-blocking assignments '<=':
  i. Replace it with the next state input 'u_in' of the flipflop wherever it appears as a target (left hand-side) of an assignment.
  ii. Keep it as 'u' wherever it appears as a reference (on the right hand-side).

b. If it is assigned with blocking assignments '=':
  i. Replace it with the next state input 'u_in' of the flipflop wherever it appears as a target of an assignment.
  ii. Replace it with the next state input 'u_in' of the flipflop wherever it appears as a reference in an assignment.

- For instance in '*Before'*, given the following non-blocking and blocking assignment statements (where *r3*, *r2* and *x* are flipflops determined in step 1):

  'r3 <= a;' *'r2 <= r3 && x;'* *'x = a || b;'*

  We obtain in '*After'* the following assignment statements.

  'r3_in <= a;' *'r2 _in<= r3 && x_in;'* *'x _in = a || b;'*

Before

```
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
```

After

```
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;
```

Sequential Block

```
// instantiating flip flop with the ff(d, q, i, clk,…)
// where d is the next state, i is the initial
// value, q is the output, clk is the clock, and
// '…' denotes wires for other concerns

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 …);
```

Combinational Block

```
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
```

endmodule

Figure 2. Example module before and after transformation.

Thus, *r3* being the target of a non-blocking assignment is replaced by *r3_in* only when it appears on the left hand-side of the assignment statements. *x*, on the other hand, being the target of a blocking assignment is replaced by *x_in* wherever it appears in the assignment statements.

Step5: For every flipflop u as determined in step 1, add to the beginning of the combinational always@(*) block in '*After'* a blocking assignment 'u_in=u;' assigning the output of the flipflop to its input by default. This ensures that no additional flipflops are inferred inside the combination always block during synthesis, and alleviates understanding flipflop inference algorithms.

- For instance in '*Before'*, we add *"r_in = r;"* for *r* and similar statements for *r2, r3,* and *x*.

Figure 3 illustrates the corresponding circuit for *'After'*. While the transformation separated the combinational logic from the sequential logic at the Verilog level, it is clear that it preserved logical functionality. In the following section, we provide a formal proof of correctness for the transformation.

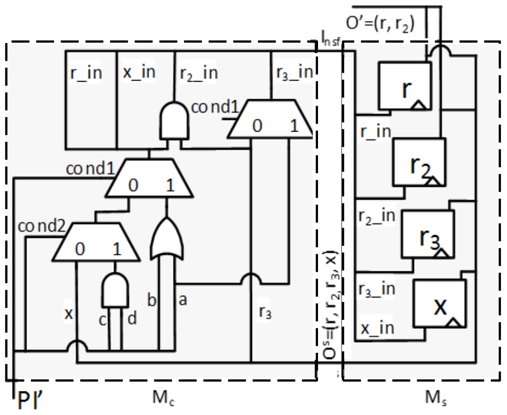


Figure 3. Circuit of the 'After' generated code of Figure 2.M^c and M^s represent the circuits corresponding to the combinational and sequential blocks of 'After' in Figure 2.

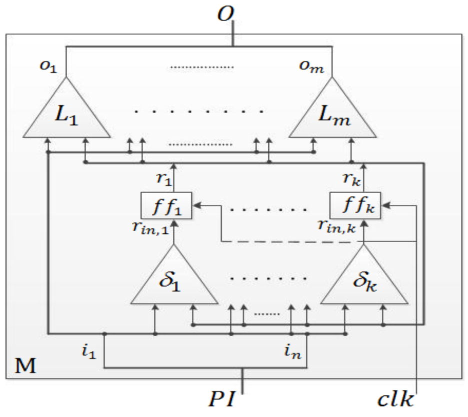


Figure 4. Mealy Machine structure. PI are the primary inputs. <ff1, ff2,..,ffk> denote the flipflops, L_i denotes the output function of output o_i, and δ_i denotes the next state function of flipflop ff_i.

V. RTL-TO-RTL CORRECTNESS

Definition 1.Flipflop: A *flipflop* is a tuple *ff*=<r_{in}, *r*, r_0, *clk*, …> where r_{in} is the next state input, *r* is the output, r_0 is the initial value of *r*, "*clk*, …" denote the clocking and asynchronous logic inputs of the flipflop and might vary depending on the target concern. The basic semantics of a flipflop is defined in terms of the value of *r* across time. At time 0, *r* takes the value of r_0,at time 't+1', *r* takes the value of r_{in} at time 't'.

The time is a concept that is controlled by the clock and asynchronous logic inputs and might vary depending on the target concern. For example, a flipflop connected to a scan chain might pause logical time for the flipflop until all the values of the flipflops in the chains have been scanned.

Definition 2.Mealy Machine: A *Mealy Machine* is a tuple *M*=$\langle \boldsymbol{ff}, \boldsymbol{RI}, \boldsymbol{I}, \boldsymbol{O}, \boldsymbol{\delta}, \boldsymbol{L}\rangle$, where ***ff*** is a vector of flipflops, ***RI*** is a vector of initial values of the flipflops, ***I*** is a vector of primary inputs, ***O*** is a vector of outputs, $\boldsymbol{\delta}$ and ***L*** are vectors of next state and output functions, respectively, ranging over the outputs of ***ff*** and ***I***. The semantics of *M*, as illustrated in Figure 4, are given in terms of traces of values across time where the values of the outputs of ***ff*** at time 0 are given by ***RI***, the values of ***I*** are non-deterministic, the values of the outputs of ***ff*** at time 't+1' are given by the values of the next state inputs of ***ff*** at time 't' which are connected directly to $\boldsymbol{\delta}$; i.e., ($r_{in,i}$=$\delta_i(r_1, \dots, r_k, \boldsymbol{I})$). Finally, the values of $\boldsymbol{O}(t)$are given by ***L***; i.e., $o_i = L_i(r_1, \dots, r_k, \boldsymbol{I})$.

Definition 3.Module: A *module P*(***V***, ***S***) consists of variables ***V*** and statements ***S***. A variable is declared to be an input, an output, a reg, or a wire. An expression ranges over variables and Boolean and arithmetic operators. A blocking assignment statement is of the form "*v=exp*" where *v* is a variable and *exp* is a well formed expression. A non-blocking assignment statement is of the form "*v<=exp*". Let ***BS***$\subseteq$***S***, and ***NBS***$\subseteq$***S*** be the set of blocking and non-blocking assignment statements, respectively. A flipflop instantiation statement is of the form "*ff(d,q,i,clk,…)*" where *d* is the next state input, *q* is the output, *i* is the initial value, and "*clk,…*" denote the clocking and asynchronous logic that can vary depending on the target concern. The semantics of a module follow the regular Verilog execution semantics and are well represented in the synthesis transformation below.

For simplicity, we assume all statements of entry level P to be assignment statements inside an always block with a sensitivity list. For programs with conditional statements, preprocessing can embed the condition predicates into the assignment statements in a fashion similar to multiplexers. We denote by $\boldsymbol{R}\subseteq\boldsymbol{V}$ the set of assigned variables inside an always block with *posedge* or *negedge*, $\boldsymbol{O}\subseteq\boldsymbol{V}$ is the set of outputs$\{o_1, \dots, o_m\}$, $\boldsymbol{I}\subseteq\boldsymbol{V}$ is the set of inputs $\{i_1, \dots, i_n\}$, and $\boldsymbol{W}\subseteq\boldsymbol{V}$ is the remaining set of variables $\{w_1, \dots, w_l\}$.

Definition 4.Flipflop inference: Every variable assigned in an always block with a posedge or a negedge is inferred as a flipflop. This is exactly similar to flipflop inference step 1 in Section IV.A.

Definition 5.Synthesis: Given a module $P(\boldsymbol{V}, \boldsymbol{S})$ we construct a Mealy Machine $M=\langle \boldsymbol{ff}, \boldsymbol{RI}, \boldsymbol{I}, \boldsymbol{O}, \boldsymbol{\delta}, \boldsymbol{L}\rangle$=Synthesis($P$) as formally defined in [20] and as shown in Table 1 that implements P and defines its semantics. In brief, Synthesis(P) performs the following steps:

1. For every input and output in P there is a corresponding input and output wire in M respectively.
2. For every $r$in $\boldsymbol{R}$ in P, let $S_r = \{s_1, s_2, \dots, s_{|Sr|}\}$ be the set of assignment statements in $\boldsymbol{S}$ with target $r$and under Boolean conditional statements $\{c_1, c_2, \dots, c_{|Sr|}\}\subseteq S$ respectively. A single non-blocking assignment is generated such that it assigns r to either of the expressions $\{exp_1, exp_2, {}_{\dots,} exp_{|Sr|}\}$ corresponding to right hand-side of the statements $s_1, s_2, \dots, s_{|Sr|}$ according to values of $c_1, c_2, \dots, c_{|Sr|}$. The resulting assignment is of the form:

   $r <= c_1?\ exp_1 : [c_2?\ exp_2: [\dots\dots: r]]]$

   For variables that are more than one bit wide, bit blasting is performed. Each generated assignment statement is synthesized to a flipflop *ff* in $\boldsymbol{ff}$ in M. The corresponding next state function δ is composed of a sequence of multiplexers fed by the synthesized logic of the expressions and *ff*, and controlled by the synthesized logic of the conditions.
3. For every expression setting a variable o in $\boldsymbol{O}$ in P, we construct a corresponding output function L that encodes the logic of the expression and connects it to the corresponding output in M. L ranges over the inputs and flipflop outputs in M corresponding to the expression variables.

Theorem 1: Given a module P; let $P' = T(P)$ where T is the transformation of Section IV.A, P and P' are equivalent. That is for the same initial values and trace of input values both P and P' produce the same trace of output values.

Sketch of proof: *Theorem 1* is correct by construction. P' consists of the flipflop instantiation block P^s and the list of assignment statements housed in a combinational always@(*) block P^c *as* illustrated in Table 1. P^c and P^s are formally defined in Table 1.

TABLE 1. SEPARATION AT RTL LEVEL OF COMBINATIONAL AND SEQUENTIAL BLOCKS AND THEIR CORRESPONDING MEALY MACHINES.

Sequential Machine	**Combinational Machine**
$M^s = \langle \boldsymbol{ff^s}, \boldsymbol{RI^s}, \boldsymbol{I^s}, \boldsymbol{O^s}, \boldsymbol{\delta^s}, \boldsymbol{L^s}\rangle$	$M^c = \langle \boldsymbol{ff^c}, \boldsymbol{RI^c}, \boldsymbol{I^c}, \boldsymbol{O^c}, \boldsymbol{\delta^c}, \boldsymbol{L^c}\rangle$
$\boldsymbol{I^s} =< \boldsymbol{I^{nsf}}, clk >$	$\boldsymbol{I^c} =< \boldsymbol{PI'}, \boldsymbol{I^{ff}} >$
$\boldsymbol{RI^s} = \boldsymbol{RI}$	$\boldsymbol{RI^c} = \{\}$
$\boldsymbol{\delta^s} = \boldsymbol{I^{nsf}}$	$\boldsymbol{\delta^c} = \{\ \}$
$\boldsymbol{ff^s} = \{ff_i^s : i = 1, \dots, k\}$ $ff_i^s =< r_{in,i}^s, r_i^s, ri_i^s, clk >$ $r_{in,i}^s = \delta_i^s(I^s, ff^s)$	$\boldsymbol{ff^c} = \{\ \}$
$\boldsymbol{L^s} = \{L_i^s : i = 1, \dots, k\}$ $L_i^s = r_i^s$	$\boldsymbol{L^c} = \{L_i^c : i = 1, \dots, m + k\}$ $L_i^c = \begin{cases} L_i\,(\boldsymbol{PI'}, \boldsymbol{I^{ff}}) & i = 1 \dots m \\ \delta_{i-m}(\boldsymbol{PI'}, \boldsymbol{I^{ff}}) & i = m+1 \dots m+k \end{cases}$
$\boldsymbol{O^s} = \{o_i^s :\ i = 1, \dots, k\}$ $o_i^s = L_i^s(\boldsymbol{I^s}, \boldsymbol{ff^s})$	$\boldsymbol{O^c} =< \boldsymbol{O'}, \boldsymbol{O^{nsf}} >$ $\boldsymbol{O'} = \{o_i' = L_i^c(\boldsymbol{PI'}, \boldsymbol{I^{ff}}) : i = 1 \dots m\}$ $\boldsymbol{O^{nsf}} = \begin{Bmatrix} o_i^{nsf} = L_{i+m}^c(\boldsymbol{PI'}, \boldsymbol{I^{ff}}): \\ i = 1 \dots k \end{Bmatrix}$

$\boldsymbol{P^s}(\boldsymbol{V^s}, \boldsymbol{S^s})$
$V^s = V \cup \{u\_in : u \in R\}$ $BS^s = \{\ \},\quad NBS^s = \{\}$ $S^s = \{s_i^s$:*ff* ff_u(u_in, u, 0,clk, ...) $, where\ u \in R$

$\boldsymbol{P^c}(\boldsymbol{V^c}, \boldsymbol{S^c})$
$V^c = V \cup \{u_{in} : u_{in}\ is\ a\ fresh\ variable\ \ for\ all\ u \in R\}$ $\boldsymbol{U_{tb}} = \{u : u \in R, u\ is\ \ target\ of\ blocking\ statement\}$ $\boldsymbol{U_{tn}} = \{u : u \in R,$ $u\ is\ target\ of\ nonblocking\ statement\}$ $S^c$is defined iteratively as follows   Let $S_{(0)}^c = S$, $h_n = \lvert \boldsymbol{U_{tn}} \rvert$and $h_b = \lvert \boldsymbol{U_{tb}} \rvert$ $S_{(i)}^c = \begin{cases} S_{(i-1)}^c \rvert_{u_i : u_i\_in} & i = 1 \dots\dots h_b \\ S_{(i-1)}^c \rvert_{u_i : u_i\_in} & i = h_b + 1 \dots\dots h_b + h_n \\ & and\ u_i\ is\ a\ target \end{cases}$ Where $u_i : u_i\_in$ means substitute $u_i$by u_i_in.   Then$S^c = S_{(h_b+h_n)}^c$

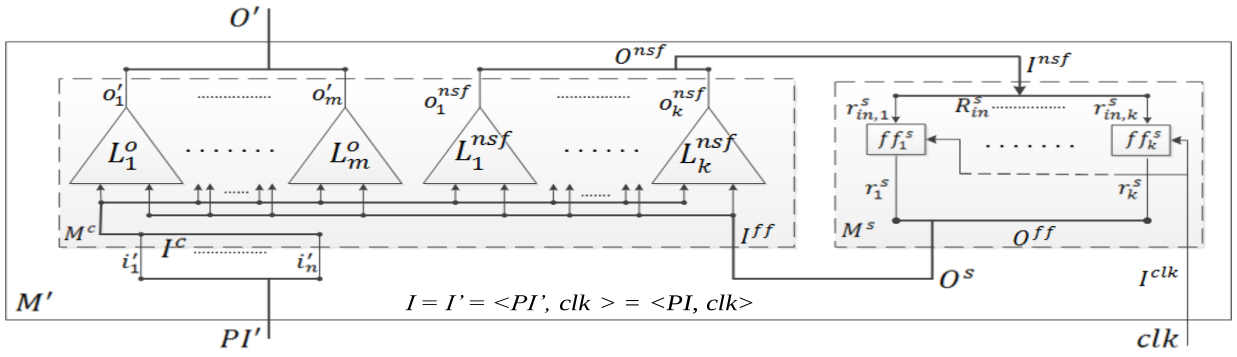


Figure 5. Mealy machine M′ constructed from connecting Mc and Ms.

Consider the Mealy machines $M=\langle \boldsymbol{ff}, \boldsymbol{RI}, \boldsymbol{I}, \boldsymbol{O}, \delta, \boldsymbol{L}\rangle$=*Synthesis*(*P*), $M^s = \langle \boldsymbol{ff^s}, \boldsymbol{RI^s}, \boldsymbol{I^s}, \boldsymbol{O^s}, \delta^s, \boldsymbol{L^s}\rangle$= *Synthesis*($P^s$) and $M^c = \langle \boldsymbol{ff^c}, \boldsymbol{RI^c}, \boldsymbol{I^c}, \boldsymbol{O^c}, \delta^c, \boldsymbol{L^c}\rangle$=*Synthesis*($P^c$) resulting from the synthesis of *P*, P^s and P^c, respectively. *P* is equivalent to *P'* since *M* is equivalent to *M';* the Mealy machine resulting from the composition of M^c and M^s as illustrated in Figure 5. The proof follows by structural induction.

VI. Implementation And Experimental Results

We implemented our tool in an industrial design framework for Verilog RTL. The tool takes a Verilog RTL as input on which it applies the transformation and outputs a concern-RTL Verilog as was described in Figure 1(b). Typical verification methodologies and techniques are then applied such as simulation, model checking, and debugging tools. We use verification time per designer team and number of lines of code as metrics to analyze the transformation efficiency and the design complexity, respectively.

In previous design cycles, designers in our teams dropped verification at the gate-level due to the large verification time and adopted the manual RTL insertion methodology (Method 1) which is at least an order of magnitude faster. In the current design cycle, the designers used our proposed methodology (Method 2) to further improve verification time. We report the advantages of Method 2 in comparison with Method 1. Both methods were effectively executed by the same team. The reported results are for the following two Verilog designs.

1) *Embedded PowerPC core:* a 32-bit RISC CPU for use in custom logic applications.
2) *Embedded PCIE PCS:* The Physical Coding Sublayer of the Peripheral Component Interconnect Express [21], a high-speed serial computer expansion bus standard.

Table 2 compares the RTL implementation size of the two designs before and after the transformation of Method 2. The advantages of the concern specific RTL in terms of verification time improvement compared to gate-level verification outweigh the linear increase in the implementation size. It's worth noting that the original RTL is still used directly for pure functional verification where nonfunctional design concerns are not relevant; thus, our methodology has no negative effect on that. Table 2 also lists the resulting gate-level netlist size for illustration purposes; the gate-level netlist was not used for the verification of concerns covered at RTL verification.

Table 2. Implementation size for industrial Embedded power PC and PCIE PCS (a highly sequential design) using Method 2.

# Lines of Code (thousands)			
Type of Code \ Design		Embedded PowerPC	Embedded PCIE PCS
RTL	w/ Inference	94.0	15.0
	w/ Instantiation	98.0	15.5
	w/ Concern Insertion	99.0	49.0
Gate-Level	w/ Concern Insertion	398.0	340.0

Table 3. Verification time comparison for industrial designs.

Verification Time (team weeks)			
Concerns Insertion		Embedded PowerPC	Embedded PCIE PCS
RTL	Method 1 (Manual)	12.9	8.6
	Method 2 (Proposed)	7.0	5.0
Gate-Level	Post-Synthesis	NA: Large	NA: Large

For the current design cycle, these designs were converted by Method 2 to Verilog RTL with the memory elements explicitly extracted. The mapping to proprietary technology was carried out at the RTL level. This involves replacing generic flipflops with technology flipflops as well as weaving of scan chains insertion and proper clocking. This further improved verification time by 40% compared to Method 1 (RTL manual insertion) as illustrated in Table 3.

Concern insertion automation frees the designer from the complexities that are concern-specific. For example, DFT pervasive verification and scan chain debugging require a lot of team months mainly due to time overhead for flipflop instantiation and concern weaving.

VII. CONCLUSION

We present a conservative flipflop inference based design methodology that leverages the separation of memory from functionality concerns in custom logic designs to reduce verification overhead. Our method enables activities such as Design-for-Test, LBIST, memory technology mapping and clocking to happen at the RTL where design and verification engineers are more comfortable. The methodology takes input RTL and automatically generates output RTL with separate sequential and combinational blocks. The different concerns are then automatically weaved as instantiated concern specific flipflops, in place of the generic flipflops, in the sequential blocks of the output RTL. We present a formal proof of correctness. Our method is currently used in an industrial setting and has produced significant design cycle time savings for state-of-the-art SoC designs where verification time was reduced by 40%.

ACKNOWLEDGMENT

The authors would like to thank the Faculty of Engineering at the American University of Beirut for supporting this work.

REFERENCES

[1] Y. Abarbanel et al., "Validation of SoC Firmware-Hardware Flows: Challenges and Solution Directions," DAC 2014.

[2] S. Hassoun, and T. Sasao. Editors, R. Brayton, Consulting Editor. *Logic Synthesis and Verification*. Kluwer Aacdemic Publishers. 2002.

[3] N. Tenderlot et al., "Test methodology for Freescale's high performance e600 core based on PowerPC/spl reg/ instruction set architecture," IEEE, ITC 2005.

[4] H. Singh, et al., "Enhanced Leakage Reduction Techniques Using Intermediate Strength Power Gating," IEEE TVLSI, 2007. pp. 1215-1224.

[5] S. Kim et al., "A Multi-Mode Power Gating Structure for Low-Voltage Deep-Submicron CMOS ICs." IEEE TCAS II. pp. 586-590. 2007.

[6] L. Benini, et al., "Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers," ACM Trans. on Design Automation of Electronic, Oct. 1999. Pp. 351-375.

[7] L. C. Bening et al., "Physical Design of 0.35-um Gate Arrays for Symmetric Multiprocessing Servers," Hewlett-Packard Journal (April 1997).

[8] M. Meier, et al., "AspectVHDL Stage 1: The Prototype of an Aspect-Oriented Hardware Description Language," in Proceedings of the 2012 workshop on Modularity in Systems Software (MISS'12), March, 2012.

[9] F. Balarin, et al., "Metropolis: An integrated electronic system design environment," in IEEE Computer Society, pp. 45-52, Vol. 36, No. 4, (April 2003).

[10] A. Sangiovanni-Vincentelli, "Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design," Proceedings of the IEEE 2007.

[11] A. Davare, et al., "A Next-Generation Design Framework for Platform-based Design". DVCon 2007.

[12] A Basu et al., " Heterogeneous Real-time Components in BIP," in IEEE International Conference on Software Engineering and Formal Methods, p.3-12, September, 2006.

[13] A. Basu et al., "Rigorous component-based design using the BIP framework," IEEE Software 2011

[14] J. Cornet, et al., "A method for the Efficient Development of Timed and Untimed Transaction-Level Models of Systems-on-Chip", IEEE DATE 2008.

[15] Atrenta Inc., Spyglass, http://www.atrenta.com

[16] SpyGlassDFT DataSheet, " http: //www.europractice.stfc.ac.uk/vendors/atrenta_spyglass_dft.pdf"

[17] Mentor Graphics Corporation, Tessent Scan, http://www.mentor.com/products/silicon-yield/products/scan

[18] L. Wang, C. Stroud, and N. Tuba. System-on-Chip Test Architectures: Nanometer . Morgan Kauffman, 2007.

[19] M. Shaker et al., "Novel Clock Gating Techniques for Low Power Flip-flops and Its Applications," MWSCAS 2013.

[20] P. M. Nyasulu. *Introduction to Verilog*. Carleton University. 2003.

[21] R. Buduk et al., PCI Express System Architecture. Addison Wesley Publications, September 2003.