Verification at RTL Using Separation of Design Concerns

Maya H. Safieddine, Fadi A. Zaraket, Rouwaida Kanj, *Senior Member, IEEE*, Ali El-Zein, and Wolfgang Roesner

***Abstract*—Design-for-test, logic built-in self-test, 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. The methodology involves three main phases: 1) flipflop inference and instantiation algorithms that handle parametric register transfer level (RTL) modules; 2) transformations that take entry RTL and produce RTL modules where memory elements are separated from functionality; and 3) a concern weaving tool that automatically inserts memory related design concerns implemented in recipe files into the RTL modules. The transformation is sound as proven and validated by equivalence checking using formal verification. We implemented the methodology in a tool that is currently used in an industrial setting wherein it reduced design verification time by more than 40%. The methodology is also effective with open source embedded system frameworks.**

***Index Terms*—Concern insertion, design-for-test (DFT), register-transfer level (RTL), verification.**

I. Introduction

With the advance of communication and computational systems, traditional hardware flows no longer meet the short time to market requirements. State-of-the-art system on a chip (SoC) and high performance microprocessor designs include several intellectual property (IP) modules with logic that serves the functionality of the IP highly coupled with logic that supports design *concerns*. Concerns are design requirements that are: 1) independent of the functionality of each IP module and 2) *pervasive* in their nature such that they are supported across almost all the design modules.

Manuscript received September 6, 2017; revised January 17, 2018 and April 6, 2018; accepted May 23, 2018. Date of publication June 18, 2018; date of current version July 17, 2019. This work was supported in part by the Maroun Semaan Faculty of Engineering and Architecture, in part by the AUB University Research Board, and in part by the EDA Team at IBM. The work of M. Safieddine was supported by the Lebanese National Council for Scientific Research. This paper was recommended by Associate Editor A. K. Coskun. *(Corresponding author: Maya H. Safieddine.)*

M. H. Safieddine, A. El-Zein, and W. Roesner are with the Department of Electronic Design Automation, IBM, Austin, TX 78758 USA (e-mail: mhs36@mail.aub.edu; alzein@us.ibm.com; wolfgang@us.ibm.com).

F. A. Zaraket and R. Kanj are with the Department of Electrical and Computer Engineering, Maroun Semaan Faculty of Engineering and Architecture, American University of Beirut, Beirut 1107, Lebanon (e-mail: fz11@aub.edu.lb; rk105@aub.edu.lb).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the author.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2848589

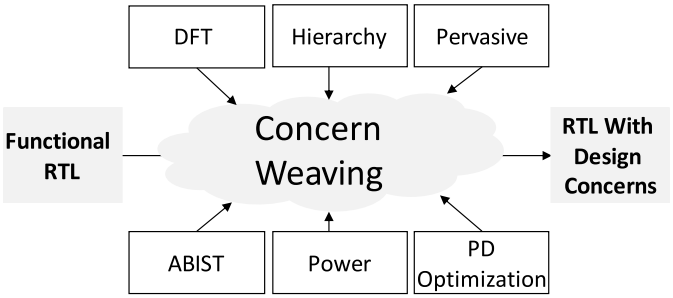

Fig. 1. Concern weaving in hardware design flow.

Concern engineers build their concern supporting modules. Functional engineers build the functional logic modules and then together with concern engineers integrate the concerns therein. The concern integration requires routing signals and injecting concern logic across IP. This leads to the presence of a variety of concern elements that are not part of the primary functionality of an IP module within the IP module. For example, a complete IP module typically requires the insertion of scan chains for automatic built-in self-test, design-for-test (DFT), clocking hierarchy structures, and other *pervasive logic* elements across the functional IP modules [2]. Fig. 1 shows several concerns involved in the design flow such as memory technology mapping, physical design (PD) optimizations, and clock divisions.

In addition, the design is mapped to a technology library and perturbed due to PD floor planning and routing constraints. The concern signal routing, the concern logic injection, and the technology mapping changes introduce high cross-coupling at the register transfer level (RTL) and render RTL code too difficult to handle. The architectural complexity of these flows and concerns implies a high cost associated with fixing defects, especially when found late in the design cycle. The cross-coupling in IP modules makes their verification hard to manage and accomplish. Traditional hardware flows no longer suffice to meet the short time to market requirements.

Several researchers addressed the concern coupling problem and its effects on the design cycle [3]–[5], [7], [8]. Section VIII reviews and discusses their work. *Separation of concerns* across the design IP modules, the design flow and even within each individual design IP module is necessary for fast hardware validation [1]. We consider lifting verification to RTL and decoupling the different concerns to enhance the overall design cycle. Key to decoupling is the ability to insert the concerns one separate from the other.

In this paper, we present a novel concern-oriented methodology (COM) with supporting tools and techniques. COM enables automatic separation of concerns at RTL, drives more automation, and improves design and verification efficiency.

0278-0070 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

COM significantly extends our previous work where we separate *memory related design concerns* by separating IP modules into combinational and sequential blocks. Combinational blocks house functional logic. Sequential blocks are pure memory element instantiations. Memory element refinement with concern interfaces enables concern insertion. The novelties and contributions of our previous work for industrial designs [9] and embedded systems [10] were as follows.

1) A flipflop inference-based transformation detects variables that could be memory elements and instantiates generic flipflops (GFFs) for them.
2) The transformation separates each original RTL module into a composition of two separate combinational and sequential RTL ($\mathcal{C}/\mathcal{S}$-RTL) modules. We highlight the novelty in preserving the structure of the functional logic, and performing the restructuring at RTL in comparison with existing synthesis to gate level techniques.
3) Novel automatic weaving of concerns replaces the GFF in $\mathcal{C}/\mathcal{S}$-RTL with concern-specific flipflops using recipe files provided by concern engineers.
4) Supporting tools implement the methodology for Verilog and embedded systems.

This paper provides the following additional novelties.

1) A novel *iterative engine* transforms RTL code into $\mathcal{C}/\mathcal{S}$-RTL.
2) The engine constitutes of three: a) structural; b) algorithmic; and c) synthesis-based transforms. Each transform handles a category of Verilog modules. Two nonprevailing categories of modules with explicit memory element instantiation and with conflicting assignment types are discussed in an online supplementary file.
3) A novel *instantiation path condition* detection algorithm computes conditional memory elements based on symbolic conditions involving *reconfigurable* parameters.
4) The transforms support reconfigurable RTL, with *generic* parameters therein, using the instantiation path condition detection algorithm.
5) Supporting tools implement the methodology for Verilog designs using Verific as a front end [11]. Supporting tools along with supplemental material are available online (http://research-fadi.aub.edu.lb/concerns).
6) Extensive analysis to evaluate the proposed method on eight new industrial macros and ten open source designs.

COM has the following advantages.

1) It enables concern-specific verification at RTL.
2) It allows automatic insertion of design concerns related to memory elements at RTL with *reduced coupling*.
3) It reduces verification time, increases designer productivity, and thus enhances design cycle efficiency.

We evaluated COM with industrial and open source designs. The open source designs included IP controller, processor, and GPU implementations. The industrial experiments conducted at IBM covered industrial embedded PowerPC and peripheral component interconnect express (PCIE) design macros. COM led to approximately 40% verification time reduction for the PowerPC macros. It contributed 30%–40% of the 1–1.5 years verification time reduction for the PCIE macros.

The remainder of this paper is organized as follows. Section II defines concern concepts, flipflops, and the Verilog modules, Section III presents concern insertion in traditional design flows and a motivational example. Section IV presents COM with an example module transformation. Section V presents the code transformations that enable COM.

```
type: 'input'|'output'|'reg'|'wire'|'param';
decl: type v=ID | type [NUM:NUM] a=ID;
expr: ID|'('UNOP expr')'|'('expr BINOP expr')'|
  '('expr'?'expr':'expr')'|//ternary select
  ID '[' ai=expr ']'; //array indexing
flptype: 'ff' | 'gff';
// signals in a flipflop instantiation
flpsigs:d=ID','q=ID','ri=ID','clk=ID',σ=ID[','ID]*;
stmt: flptype ID'('flpsigs');'|//ff instantiation
//instantiation condition flipflop instantiation
  'gicff' ID'('ic=expr')('flpsigs');'|
  'assign' w=ID'='rhs=expr';'|//continuous assign
   r=ID'='rhs=expr';'|//blocking assignment
   r=ID'<='rhs=expr';'|//non-blocking assignment
//conditional statement
   'if''('cond=expr')'t-block=stmt+
   'else' e-block=stmt+ 'end'|
//loop statement
   'for''('i=ID'='init=expr';'cond=expr';'
     i=ID'='inc=expr')' lp-block=stmt+'end';
module: 'module' P=ID'('//module name followed by
   io-list=ID[','ID]*,'clk',σ=ID[','ID]*')'//lists
//of input/output variables, and concern signals
   decl+ //list of declarations
   s-block=stmt+ //list of statements
   'always'@''('ID [, ID]*')'//always block
   al-block=stmt+           //with variable
   'end' 'end' 'module'    //sensitivity list
```

Fig. 2. ANTLR-like [6] grammar of a Verilog program P.

Section VI presents concern weaving. Section VII presents experimental setup and results. Section VIII reviews related work. Section IX concludes and discusses future work.

II. Background

Definition 1 (Concern Interface): A *concern interface* is a set of signals σ that vary based on the specific concern logic.

Definition 2 (Flipflop): A flipflop *ff* is a tuple $\langle r_{in}, r, r_0, \text{clk}, \sigma\rangle$, where r_{in}, r, and r_0 denote the next state, the output, and the initial value of *ff*, respectively. Signal clk specifies the *ff* clocking and σ denotes the concern interface. When σ is empty, *ff* is a *concern-free flipflop*.

Fig. 2 defines a Verilog module using an ANTLR [6] like grammar. For simplicity, module instantiation statements are not allowed except for flipflops. Token ID starts with an alphabetical character, may contain alphanumeric and underscore characters and denotes an identifier. Identifiers are used to denote program, variable, parameter, and flipflop instantiation names. Tokens UNOP and BINOP denote unary arithmetic/Boolean operators ($-$ and !) and binary arithmetic/Boolean operators ($+, -, *, /, <, \leq, ==, >, \geq, \&\&$, and $||$), respectively.

Rule `expr` recursively denotes expressions used in assignments, conditionals and loop statements. Rules `decl` and `stmt` define declarations and statements, respectively. Statements are *ff* and instantiation condition flipflop (gicff) instantiation, assignment, conditional, and loop statements. An *gicff*(ic) gets instantiated only when expression *ic* evaluates to true.

A module P is a pair $P(V, S)$, where V and S denote all variables and statements in P, respectively. Sets R, W, I, and $O \subseteq V$ denote the reg, wire, input, and output variables, respectively. Lists B, NB, C, and $L \subseteq S$ denote the blocking assignments, nonblocking assignments, conditional, and loop statements, respectively. Set $A = B \cup \text{NB}$ denotes the procedural assignments. Lists can be empty. The always block may be omitted. Continuous assignments are only allowed in

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

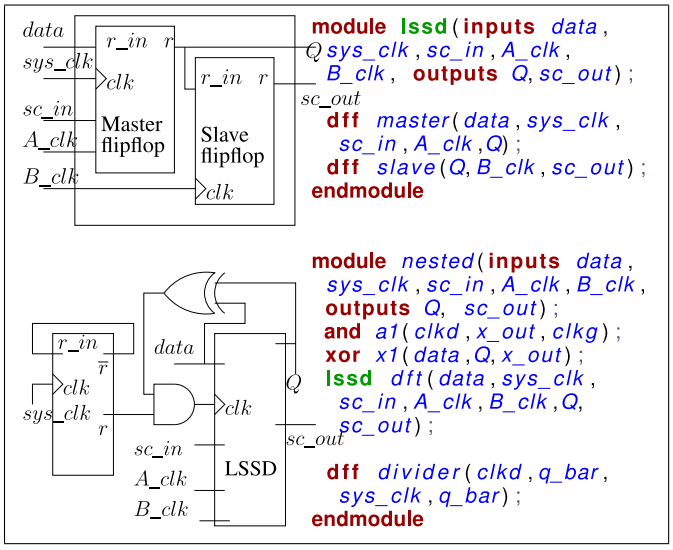

Fig. 3. Schematics for LSSD [20] (DFT) and nested clock-gating/frequency-division [21] flipflops with their modules.

s-block and not allowed in al-block. The module has the typical semantics as defined in [22]. All modules must list concern interface σ such that it is replaced by concern signals and routed to *ff* instantiations in the concern weaving phase.

Definition 3 (Memory Related Concern): A *memory related concern* is composed of: 1) a memory related concern interface σ and 2) RTL Verilog modules that define the concern logic. Each of the modules should have σ as part of its io-list. Hereon, a concern denotes the underlying signals, wires, and logic necessary to implement a pervasive requirement across a design. Fig. 3(left) shows concern-free master and slave flipflops. It also shows level sensitive scan design (LSSD) [20] and nested clock-gating/frequency division [21] flipflops with concern signals $\sigma = sc_in, A_\text{clk}, B_\text{clk}$, and *sc*_out with the corresponding Verilog interfaces (right).

A. Helper Functions

In the following, we introduce helper functions needed in presenting COM transforms and correctness in Section V. Function VARS(e) returns the set of variables referenced by e when e is an expression and returns empty otherwise. Similarly, EXPRSN(s) returns *rhs* when s is an assignment statement, *cond* when s is a conditional or loop statement and returns empty otherwise. Function INDEX(e) returns *ai* when e is an array indexing expression, TARGET(s) returns w when s is a continuous statement and r when s is a blocking or a nonblocking assignment, ASGNMTTYPE(s) returns whether s is continuous, blocking or nonblocking when s is an assignment statement, and STMTTYPE(s) returns whether s is an assignment, flipflop instantiation, conditional, or a loop statement when s is a statement; they return empty otherwise.

When s is a conditional, THENB(e) and ELSEB(e) return t-block and e-block, respectively, and return empty otherwise. When s is a loop statement, LOOPB(s), ITERATOR(s), INIT(s), INITEXP(s), INCREMENT(s), and INCREMENTEXP(e) return lp-block, i, $i = init$, $init$, $i = inc$, and inc, respectively, and return empty otherwise. Function ALWAYSB(s) returns al-block.

III. Traditional Design Flows

In traditional design flows, concerns are inserted into the logic: 1) manually at RTL or 2) automatically at gate-level.

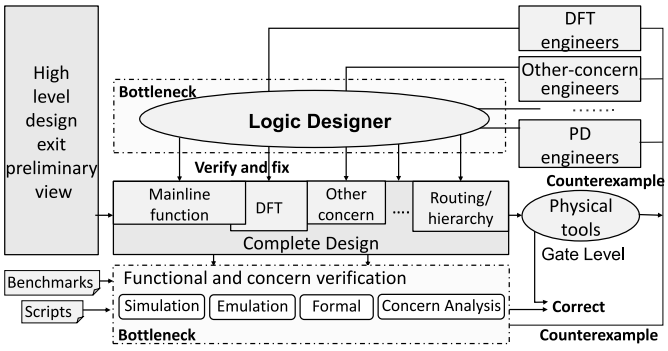

Fig. 4. Traditional design flow with the logic designer at bottleneck.

Fig. 4 shows a typical design flow. Verification processes such as simulation, emulation and formal take input RTL and report either correctness or a counterexample. In case a counterexample is reported, the engineers interpret it and fix it at the RTL. In manual concern insertion interpretation happens directly at RTL. In gate level concern insertion, engineers have to interpret the counterexample at the level it was generated, lift it to RTL, identify responsibilities and fix the problem.

Setting up the verification environment includes defining the tool chain support, the reporting mechanisms, grouping the modules and their interfaces, building the benchmarks, and preparing the glue logic (scripts) that drive the verification processes as illustrated in Fig. 4.

Manual concern insertion at RTL enables verification to happen early in the design cycle. However, it requires the direct involvement of logic designers in the concern insertion as well as the verification processes of both functionality and concerns. Fig. 4 shows that logic designers are at bottleneck working with engineers of different backgrounds. The logic designers start verification with a preliminary physical hierarchy mapping. They manually insert the concerns at RTL with strict code restrictions enforced by style and linting tools.

They manually map the logical hierarchy to the physical hierarchy which are grouped according to functionality and physical constraints, respectively. This mapping process has several drawbacks. First, it is challenging and time consuming. Second, the hierarchy mapping is subject to change throughout the verification process. Third, the cross-coupling between functional RTL and concern specific RTL makes the verification environment excessively large and unstable. Thus, concern-specific verification is slow and requires huge setup overhead. A change in concerns requires revalidation of the whole verification environment and thus the design verification environment is at a bottleneck as illustrated in Fig. 4.

Automatic concern insertion at gate-level relieves logic designers from the pressure. Yet, it forces verification to happen late at the design cycle. Verification is harder when done at the gate level and requires more iterations to lift the counterexamples and identify responsibilities. This is impractical for modern designs with time-to-market limitations.

A. Motivational Example

We use concern free Verilog module example of Fig. 5(left) to illustrate the typical steps involved in manually inserting concerns at RTL. The module consists of blocking and nonblocking assignment, conditional, and parameterized loop statements. It conditionally assigns register variables

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

```
module example(cond1,cond2,a,
    b,c,d,in,r,r2,out,clk,σ);
 parameter p;
inputs cond1,cond2,a,b,c,d,
       clk;
 input [0:p-1] in;
 outputs reg r, r2;
output reg [0:p-1] out;
 reg r3, x;
 integer i;

 always @(posedge clk,σ)begin
  if(cond1==1) begin
    r3 <= a;
    x = a || b;
  end
 r <= x;
  if(cond2==1)
    x = c && d;
r2 <= r3 && x;
ℓ: for (i=0; i<p; i=i+1) begin
  if(i % 2)
    if (i != 5)
      out[i] <= in[i];
 end
  out[2] <= in[2];
  if(p > 7)
    out[7] <= in[7] && in[5];
 end
endmodule
```

```
module example-m(cond1,cond2,a,b,c,d,in,r_q,r2_q,
 out_q,sc3_q,sc_qx,sc_q,sc2_q,clk,sc3_d,sc_dx,sc_d,
  sc2_d,sc_A,sc_B);
 parameter p; //same as in example
inputs cond1,cond2,a,b,c,d,clk;//same as in example
 input [0:p-1] in; //same as in example
 //new input/output scan, and flipflop in/out wires
inputs sc3_d,sc_dx,sc_d,sc2_d,sc_A,sc_B;//DFT concern
 outputs r_q,r2_q,sc3_q,sc_qx,sc_q,sc2_q;//DFT concern
 output [0:p-1] out_q;
reg r_d, r2_d, r3_d, x_d;
 reg [0:p-1] out_d;
 wire r_q, r2_q, r3_q, x_q;
integer i; //same as in example
 //LSSD instantiations – DFT concern
 lssd r_lssd (r_d, clk, sc_d, sc_A, sc_B, r_q, sc_q);
lssd r2_lssd (r2_d, clk, sc2_d,sc_A,sc_B,r2_q,sc2_q);
 lssd r3_lssd (r3_d, clk, sc3_d,sc_A,sc_B,r3_q,sc3_q);
 lssd x_lssd (x_d, clk, sc_dx, sc_A, sc_B, x_q,sc_qx);
generate
 for(i=0;i<p;i=i+1)begin // conditional LSSD instantiations
  if((i % 2) && (i != 5) &&(i!=3) && (i!=7 || !(p>7)))
   lssd out_lssd[i](out_d[i],clk,sc_dout[i],sc_A,sc_B,
              out_q[i], sc_qout[i]);
 end
lssd out_lssd[2](out_d[2],clk,sc_dout[2],sc_A,sc_B,
              out_q[2], sc_qout[2]);
 if(p>7) lssd out_lssd[7](out_d[7],clk,sc_dout[7],
              sc_A,sc_B,out_q[7],sc_qout[7]);
end generate
```

```
 always@(posedge clk,
       sc_A,sc_B) begin
  //Vars updated in stmts
  x_d = x_q;
 if (cond1==1) begin
  r3_d <= a;
  x_d = a || b;
end
 r_d <= x_q;
 if (cond2==1) begin
  x_d = c && d;
 end
 r2_d <= r3_q &&x_d;

 for(i=0;i<p;i=i+1)
 begin
  if(i % 2)
   if (i != 5)
    out_d[i] <= in[i];
end
 out_d[2] <= in[2];
 if(p > 7)
  out_d[7] <= in[7] &&
              in[5];
 end//end always
endmodule
```

Fig. 5. Left: Verilog module `example`. Middle, right: `example-m` with manual LSSD-DFT (Fig. 3) concern insertion to `example`.

(r, $r2$, $r3$, and x) based on variables cond1 and cond2, and vector variable out in a parameterized `for` loop. Parametric variable p bounds loop iterator i. Conditionals in the loop specify that out[i] is assigned when i is odd and $\neq 5$.

The designer: 1) critically studies the declaration statements and the assignment statements in the always block; 2) decides which variables should be flipflops; 3) follows conditionals and loop parameters to compute instantiation conditions for flipflops within parameterized loops; and 4) manually instantiates inferred flipflops as concern specific flipflops. The designer ensures that variables that are not inferred or explicitly defined as flipflops are combinational. Additionally, she has to handle the flipflop wiring and implementation and correctly route the corresponding concern signals, thereby guessing what the synthesis tool would do. These are all nontrivial tasks, error prone, and subject to change upon modifications of the concern support and the functionality later on. Module `example-m` in Fig. 5(middle, right) illustrates a possible outcome of the manual concern insertion process.

Manual insertion also requires the designers to properly handle the weaving of the concerns. For example, module `example-m` uses instances of LSSD flipflop, presented in Fig. 3. In case more sophisticated technologies, vendor-based and proprietary industrial flipflops were adopted, the designer has to understand the semantics and implications and reflect the change in `example-m`. Fig. 3(bottom) shows an example of a hierarchically nested module that utilizes LSSD flipflops and routes proper logic to accommodate for frequency division, clock gating, and DFT. Moreover, designers reuse IPs from legacy RTL that they may not be familiar with. This requires additional efforts to identify memory elements therein prior to integrating the concerns.

IV. Concern-Oriented Methodology

We propose COM, that: 1) resolves the concern coupling issue by separating memory related concerns from functional logic; 2) automates the integration of memory related design concerns at RTL; and 3) enables verification early at RTL.

Fig. 6 illustrates the flow of COM where logic designers prepare the base functionality RTL, concern designers prepare concern supporting RTL. Concern engineers prepare recipe files that integrate memory related concerns with functional and concern RTL. Recipe files are *hidden from the functional engineers* and are similar to aspects in software with flipflop instantiation join points. The concern recipe file is prepared once per project and may be modified as needed.

Fig. 6 also illustrates how the bottleneck in the verification environment is resolved. Functional and concern verification engineers prepare the verification environment with benchmarks that validate functional and concern correctness. Decoupling allows each verification engineer to only worry about setting up inputs and miters for her specific RTL. This renders the verification environment more stable.

Upon a new feature request or a defect with a counterexample reported by the verification or physical tools, the responsible engineer is identified and she works directly in the corresponding functional or concern block with *no coupling*. In case the defect was a routing problem, a PD engineer manages and fixes the corresponding recipe files. In case it was a timing problem, the timing logic engineer intervenes. This saves iterations in the design flow and verification cycles and reduces the high interaction cost between functional and concern engineers that is required in both manual and automated gate level concern insertion. COM also transforms parametric designs into parametric $\mathcal{C}/\mathcal{S}$-RTL. Fixes apply to the parametric $\mathcal{C}/\mathcal{S}$-RTL and are valid for all values of parameters.

Finally, Fig. 6 refers to the transformations supporting COM. Fig. 8 presents the detailed flow of the transformations. Key to the transformations are a flipflop inference step that returns the memory elements of the design and hands them to the sequential block separation transformations. In case of parametric designs, a path condition computation that treats

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

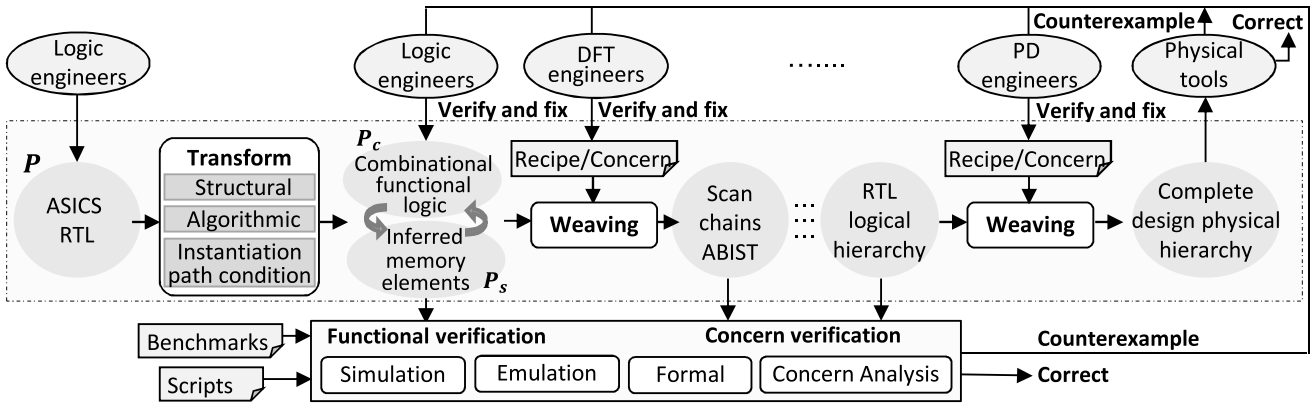

Fig. 6. COM: transforms separate memory related design concerns from functionality and enable verification at RTL.

Module definition — similar to `example`

```
//module header from I' and O'
module example-a (cond1, cond2, a, b, c, d,
    in, r, r2, out, clk,σ);
//declarations
 parameter p;
inputs cond1,cond2,a,b,c,d,clk;
 input [0:p-1] in;
 outputs wire r, r2;
output wire [0:p-1] out;
 wire r3, x;
 reg r_in,r2_in,r3_in,x_in;//next state of FF
reg [0:p-1] out_in;
 integer i;
```

Sequential block – Ps

```
 gff ff_r(r_in,r,0,clk,σ);
 gff ff_r2(r2_in, r2, 0,clk,σ);
gff ff_r3(r3_in, r3, 0, clk,σ);
 gff ff_x(x_in, x,0, clk,σ);
 generate //Parametrized sequential block
for(i=0; i<p; i=i+1) begin
  gicff ff_out[i]((i>=0 && i<p) && (i %2) &&(i!=5)
          &&(i!=2)&&!(i==7 && p>7))
        (out_in[i],out[i],0,clk,σ);
end
 gicff ff_out[2](true)(out_in[3],out[3],0,clk,σ);
//≡gff ff-out[2](out_in[3],out[3],0,clk,σ);
 gicff ff_out[7](p>7)(out_in[7],out[7],0,clk,σ);
end generate
```

Combinational block – Pc

```
always @(*) begin
 r_in = r; // RWBlk starts here
r2_in = r2;
 r3_in = r3;
 x_in = x ;
//Parametrized RWBlk block
for (i=0; i < p; i=i+1) begin
 if((i % 2) &&(i!=5)&&(i!=2)&&!(i==7 && p>7))
  out_in[i] = out[i];
end
 out_in[2] = out[2];
if(p>7) out_in[7] = out[7];
 if(cond1==1) begin // FLBlk starts here
   r3_in <= a;
   x_in = a || b;
end
 r_in <= x_in;
if(cond2==1)  x_in = c && d;
 r2_in <= r3 && x_in;

for (i=0; i < p;i=i+1) begin
   if(i % 2)
     if (i != 5)
       out_in[i] <= in[i];
end
 out_in[2] <= in[2];
if(p > 7) out_in[7] <= in[7] && in[5];
end //end of always block
end module
```

Fig. 7. Module `example-a` is the result of automatically transforming `example` of Fig. 5. The shaded code corresponds to the parameterized code.

the parameters as symbolic values computes an instantiation condition for each inferred memory element. The instantiation condition computation differentiates between variables declared in parameterized loops, outside parameterized loops, and in both and makes sure the instantiation condition for one of the latter is exclusive. The inferred memory elements and instantiation condition memory elements constitute the resulting sequential block. The logic of the next state functions of the memory elements constitute the combinational block. Blocks in Fig. 8 refer to algorithms that implement them which are discussed in Section V.

A. Decoupling and Separation of Concern Automation

Key to COM is the automated transformation of RTL that requires inference of memory elements into RTL with memory element instantiation. Fig. 7 shows module `example-a` the result of applying COM on module `example` from Fig. 5(left). The definition of `example-a` adds new variables with a *_in* suffix that COM uses as next state functions for the memory elements and uses the original variables to connect the separated sequential block *Ps* to the combinational block *Pc* of `example-a`. Block *Ps* contains *gff* and generic instantiation condition flipflop gicff instantiations corresponding to instantiated memory elements in the nonparametric and parametric statements of `example`, respectively. Block *Pc* is highly similar to the functional code in `example` with slight modifications to refer to the introduced next state variables of the inferred flipflops where necessary. COM applies concern weaving over *Ps* and functional logic designers work over *Pc* after the separation takes place.

The instantiated memory elements (*gff* and gicff) can be concern free which enables pure functional verification, or concern specific which enables concern specific verification.

COM involves three main automated processes.

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

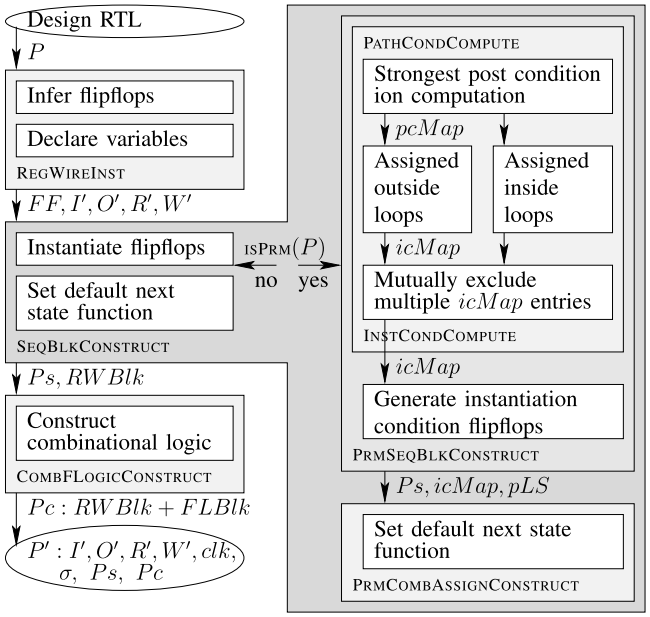

Fig. 8. Flow diagram of the memory related concern separation transformations annotated with algorithm, function, and return object names.

1) Automated Memory Element Inference and Isolation: This identifies memory elements and separates them from functional logic. In effect, memory element separation is memory related concern separation from functionality.

2) RTL-to-$\mathcal{C}/\mathcal{S}$-RTL Transformation: Automatically separates RTL into combinational and sequential blocks. It uses the output of the memory element inference to instantiate technology- and concern-independent flipflops in the sequential block. Our RTL-to-$\mathcal{C}/\mathcal{S}$-RTL transformation is *novel* in that it:

1) produces memory element instantiations at the RTL level;
2) preserves variable declarations;
3) constructs combinational logic around the boundaries of the right hand side expressions of assignment statements thus preserving functional logic structure;
4) only introduces named wires that directly relate to the next state function inputs of an inferred memory element with minimal name preserving changes.

Therefore, a designer who understands the input RTL should be able to readily understand the resulting $\mathcal{C}/\mathcal{S}$-RTL: in particular the combinational blocks. Synthesis tools infer memory elements at gate level without preserving structure.

3) Concern Weaving (Integration): Only visits the module declaration and the sequential block to integrate concerns. It involves three steps.

1) Substitution of generic with concern-aware flipflops.
2) Routing the concern signals to the concern-aware flipflops.
3) Connecting the concern signals to the concern supporting modules.

The concern-aware flipflops are RTL flipflop modules with concern interfaces. Intuitively, concern weaving is an automatic memory element refinement that may be applied in sequence to support several concerns. For example, nested refinements allow the following.

1) Concern weaving takes the $\mathcal{C}/\mathcal{S}$-RTL, a technology recipe file created for a specific technology and the technology mapping RTL as input and produces a technology specific memory and clock mapped RTL.
2) Mapping to a different technology (from 14 to 7 nm) only requires pointing to a different technology mapping RTL and may not need to invoke concern weaving again.
3) Different concerns that were not part of the base design can be weaved in sequence by relying on concern-specific recipe files. For example, recipe files that specify the interconnections of scan chains and their corresponding scan structures are used for the DFT concern. Other examples of memory related concerns that can be automatically inserted due to identification of memory elements include:
   a) conversion of scan memory elements to nonscan;
   b) mapping to memory elements of certain hardware characteristics (soft error rate);
   c) conversion of functional clock gating for power savings and retiming optimizations.

   The concern weaving applies the recipe files in sequence for the different concerns and produces concern specific RTL for the next step in the design flow.

In the sequel, COM maps the logical hierarchy to the physical hierarchy in order to satisfy physical constraints.

B. Efficient Concern Separated Verification at RTL

COM enables separated functional and concern verification at RTL where functional and concern engineers are familiar with their code and thus more productive. Particularly for external IPs, debugging problems with third party is easily done at RTL where the structure is preserved. All that is possible with manual concern insertion. However, manual insertion entails overhead and is error prone whereas COM concern integration is correct by construction.

The same verification environment works on $\mathcal{C}/\mathcal{S}$-RTL for functional and concern verification. For concern verification an augmented verification environment was needed with manual insertion. Verification engineers had to adhere to concern wire and signal naming conventions to save time. With COM, the concern signals are normalized and preserved across all modules and therefore the setup is automated.

The time required for setting up the environment can grow significantly with *parameterized* designs as the setup needs to be constructed for each parameter value in the manual insertion approach. With COM, the process is automated. The only overhead is preparing the concern specific recipe files once by the concern specific verification engineers.

COM separated concerns allow for small debugging scopes. Automation of concern integration requires no setup overhead and consequently makes partial concern verification practical. Designers can simulate the base functionality with concern-free flipflops. Incorporating different concerns can be on as needed. This renders design sizes smaller and consequently renders simulation and emulation faster, designer productivity higher, requires less resources from emulation and makes formal verification techniques that might not scale to large designs feasible. Furthermore, the COM verification environment is *concise* since it only includes the target concerns. When a defect is fixed, COM automatically weaves back the fixed recipe files into the design.

V. MEMORY-CONCERN SEPARATION TRANSFORMATIONS

Fig. 8 presents a high level flow diagram of algorithm RTLTRANSFORM presented in Fig. 9. It starts with program

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

```
P'=RTLTRANSFORM(P, clk, σ)
 //Infer flipflops. Instantiate registers and wires
 [FF,I',O',R',W',V']=REGWIREINST(P, clk, σ);
 [RWBlk,Ps]=SEQBLKCONSTRUCT(P,FF,clk,σ);
 FLBlk=COMBFLOGICCONSTRUCT(FF,S);//functional logic
 Pc='always@(*)begin'+RWBlk+FLBlk+'end';
 P'= Ps+Pc //Seq/Comb RTL

[FF,I',O',R',W',V'] = REGWIREINST (P, clk, σ)
 FF= {v. v∈V, ∃s∈S. v=TARGET(s) and CLKEDGED(s)};
 //Fix variables and declarations
 I'= I; O'= O; R'=R–FF; W'=W;
 foreach u in FF//Instantiate registers and wires
  u_in = CREATEWIRE(u,V);
  W'= W' ∪ {u,u_in};
 endfor
 V'= I' + O' + R' + W'; //Form all variables of P'

e=HANDLEEXPRSN(FF,es)
 e = es;
 foreach v in VARS(es) and v in FF
  if(∃ s ∈ S. TARGET(s)=v, ASGNMTTYPE(s)=BS)
   e=REPLACEVAR(e, v, LOOKUPWIRE(v));
  endif
 endfor

[RWBlk,Ps]=SEQBLKCONSTRUCT (FF,B)
 if (not ISPRM(P))//P is non–parametric
  foreach u in FF //Construct sequential block Ps
   Ps.append(GFFINST(ff_u,LOOKUPWIRE(u),u,0,clk,σ));
   RWBlk.append(COMPASTMT(LOOKUPWIRE(u),u));//u_in=u
  endfor
 else//P is parametric: construct parametric Ps
  [Ps,icMap,pLS]=PRMSEQBLKCONSTRUCT(P,FF,clk,σ);
  RWBlk= PRMCOMBASGNCONSTRUCT(FF,pLS,icMap); endif

FLBlk=COMBFLOGICCONSTRUCT (FF,B)
 foreach s in B //B block of statements
  e = HANDLEEXPRSN(FF,EXPRSN(s));
  if s ∈ A //assignment statement
   t=TARGET(s)∈FF? LOOKUPWIRE(TARGET(s)): TARGET(s);
   FLBlk.append(COMPASTMT(t,e));
  elseif s ∈ C //conditional statement
   tblk= COMBFLOGICCONSTRUCT(FF, THENB(s));
   eblk = COMBFLOGICCONSTRUCT(FF, ELSEB(s));
   FLBlk.append(COMPCSTMT(e, tblk, eblk));
  elseif s ∈ L //loop statement
   si= COMBFLOGICCONSTRUCT(FF, INIT(s));
   sinc= COMBFLOGICCONSTRUCT(FF, INCREMENT(s));
   lbk= COMBFLOGICCONSTRUCT(FF, LOOPB(s));
   FLBlk.append(COMPLSTMT(si,e,sinc,lblk));
  endif endfor
```

Fig. 9. RTLTRANSFORM algorithm and its service functions.

P, calls REGWIREINST to compute the variables of P'. Then it invokes SEQBLKCONSTRUCT to compute sequential block Ps with flipflop instantiations and their next state default assignments $RWBlk$. In case P is parametric, SEQBLKCONSTRUCT calls PRMSEQBLKCONSTRUCT and PRMCOMBASSIGNCONSTRUCT to compute instantiation conditions, $RWBlk$ and Ps with instantiation condition flipflops. Key to the instantiation condition computation in INSTCONDCOMPUTE is an expensive path condition computation. The computation is tractable due to the existential quantification simplification of the instantiation conditions so that these only depend on parameter related variables. Then, RTLTRANSFORM computes the combinational block Pc using COMBFLOGICCONSTRUCT . Sections V-A–V-C discuss the intuition, detail the steps, and present effects of each function on `example-a` (Fig. 7) that is the result of RTLTRANSFORM(`example`) (Fig. 5). Sections V-D and VI present proofs for correctness of RTLTRANSFORM and discuss the CONCERNWEAVE algorithm, respectively.

A. Nonparametric Module Transformation

First, RTLTRANSFORM calls REGWIREINST to infer the flipflops FF and instantiate the registers and wires of P'. The inferred flipflops are targets of assignment statements in a clock-edged always block context. FF is $\{r, r2, r3, x\}$ and P' has the same inputs I'=I and outputs O'=O as P for module `example`. Function REGWIREINST also computes the registers R' and wires W' of P' based on FF. Every variable $u \in FF$ corresponds to an instantiated flipflop with wires $u, u_in \in W'$ corresponding to its output, and next state function, respectively. Module `example-a` declares r, $r2$, $r3$, and x as `wires` and r_in, $r2$_in, $r3$_in, and x_in as `regs`. Verilog requires `reg` for procedural assignment targets.

Next, RTLTRANSFORM calls SEQBLKCONSTRUCT to construct the sequential block Ps and a block of assignments $RWBlock$. that ensures no additional flipflops are inferred from the combinational `always` block during synthesis. When P is nonparametric SEQBLKCONSTRUCT uses helper function GFFINST for each flipflop in FF and appends the resulting *gff* to sequential block Ps. GFFINST(ff, u, i, clk, σ) creates a GFF instantiation of the form "*gff* $ff(u_in, u$, init, clk, $\sigma)$." Module `example-a` instantiates flipflops ff_r, ff_r2, ff_r3, and ff_x.

SEQBLKCONSTRUCT uses helper function call COMPASTMT(LOOKUPWIRE(u), u)) that composes an assignment statement of the form "u_in = u;" to construct $RWBlk$. Each assignment in $RWBlk$ sets the next state function u_in of a flipflop $u \in FF$ to its default value. In this case, that is the output of the corresponding flipflop. $RWBlk$ in `example-a` of Fig. 7 is in lines 29–32.

Function COMBFLOGICCONSTRUCT (FF, B) in Fig. 9(right) constructs functional logic block $FLBlk$. It processes each statement $s \in B$ by calling HANDLEEXPRSN and passing to it the right hand side expression of s when s is an assignment statement, and the conditional expression when s is a conditional or a loop statement. HANDLEEXPRSN takes expression es and uses REPLACEVAR(exp, v, v_in) to replace variable v therein by the next state function v_in of its corresponding flipflop in case v referred to an inferred flipflop and is itself a target of a blocking assignment. The base case is where s is an assignment with target t = TARGET(s). COMBFLOGICCONSTRUCT constructs an assignment where the right hand side is e the result of HANDLEEXPRSN. The target is t_in in in case $t \in FF$, otherwise the target is t itself.

COMBFLOGICCONSTRUCT recursively handles the then and else blocks of a conditional statement, loop block, and initialization and increment statements of a loop statement. For that it uses two helper functions. Function calls COMPCSTMT(exp, tb, eb) and COMPLSTMNT($init, exp, inc, lb$) take a conditional expression exp, blocks tb, eb, and lb, initialization statement init, and increment statement inc, and composes conditional and loop statements of the form "if(exp) then tb else ebend" and "for(init; ex; inc) lbend," respectively. For `example-a` COMBFLOGICCONSTRUCT produced lines 34–40. Notice how x_in replaced x on line 40 since x is a target of a blocking assignment.

Finally, RTLTRANSFORM concatenates $RWBlk$ and $FLBlk$ in a combinational always block Pc and concatenates the

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

sequential *Ps* and combinational *Pc* blocks into a Verilog module P'. P' has I' and O' and concern interface σ as `io-list` and I', O', R', and W' as declarations.

B. Parametric Module Transformation

Parameterized designs such as libraries use parameter variables in conditional expressions and loop bounds. These parameters may express concepts such as bit-width, adder-size, precision, and delay count and remain to be decided upon full system integration. The verification problem is trivial if the values of the parameter variables are determined. However, designers and design houses desire to verify their library IPs before the values are determined.

COM uses symbolic execution to compute instantiation conditions for inferred flipflops and transform parameterized RTL into parameterized $\mathcal{C}/\mathcal{S}$-RTL. The transformation enables symbolic and bounded model checking techniques for parameterized $\mathcal{C}/\mathcal{S}$-RTL. Other verification techniques require passing parameter values. If the input design is parametric, the method proceeds as follows.

1) It considers symbolic values for undecided parameters, and for each assignment statement a, computes *path condition* pc_a using the *strongest postcondition* technique. Intuitively, assignment a executes if and only if pc_a evaluates to *true*. Path condition pc_a ranges over parameter-related variables. A variable v is parameter-related to parameter p if there exists an expression e such that $v, p \in vars(e)$. Typically these are parameter and iterator variables. Other variables are existentially quantified to simplify path conditions. This simplification along with the bounded nature of HDL modules renders the typically expensive strongest postcondition computation tractable.
2) COM uses instantiation condition flipflops gicff where the instantiation condition *ic* is computed appropriately from the path conditions of all assignments that affect the corresponding flipflops.
3) COM leverages the Verilog `generate` statement to use gicff within loops.

Lines 17–27 of `example-a` in Fig. 7 illustrate a `generate` loop with gicff instantiations that solves the parameterized loop problem of `example` in Fig. 5. Array variable out belongs to *FF* and its elements are assigned using a parameterized loop in `example`. Elements of gicff array ff_{out} are instantiated on condition that expression $(0 <= i)\&\&(i\%2)\&\&(i = 5)\&\&(i! = 2)\&\&(i = 7||!(p > 7))$ evaluates to true. The expression is the path conditions that annotate the assignment statements with elements of out as target in `example`.

When P is parametric, function SEQBLKCONSTRUCT of Fig. 9 calls functions PRMSEQBLKCONSTRUCT and PRMCOMBASSIGNCONSTRUCT illustrated in Fig. 10(top left and right) to compute the sequential block *Ps* and the next state function initialization block *RWBlk*, respectively.

Function PRMSEQBLKCONSTRUCT calls INSTCONDCOMPUTE of Fig. 11 to compute *icMap*. Section V-C discusses function INSTCONDCOMPUTE . The dynamic map *icMap* associates parameterized statement and flipflop pairs with Boolean expressions: $ic = icMap(pls, u)$ represents the instantiation condition of $gicff ff_u(ic)$ in parameterized loop *pls*. For convenience, when $u \in FF$ is inferred in a nonparameterized statement, notation $icMap(\text{null}, u)$ refers to the corresponding instantiation condition.

Then, PRMSEQBLKCONSTRUCT computes parameterized loop statements *pLS*. It iterates over *pLS* and uses the recursive function GNRTLPCONSTRUCT [Fig. 10(left bottom)] to compute generate loop statements of gicff instantiations. PRMSEQBLKCONSTRUCT calls helper function GICFFINST with parameters $(ff, ic, u, \text{init}, \text{clk}, \sigma)$ to instantiate a gicff flipflop of the form "$gff\ ff_u(u_in, u, \text{init}, \text{clk}, \sigma)$." Then PRMSEQBLKCONSTRUCT iterates over *icMap* tuples with null parameterized statements and constructs gicff instantiations of flipflops that occur outside the parameterized loops. PRMSEQBLKCONSTRUCT returns sequential block *Ps*, *icMap*, and *pLS*. For example, array variable out is assigned in the body of a parameterized loop "`for(i=0;i<p;i=i+1)`" in `example`. PRMSEQBLKCONSTRUCT creates a corresponding loop (lines 17–27 of `example-a`, Fig. 7) with instantiations of flipflop out, in a `generate` loop. Since *r*, *r*2, *r*3, *x*, out[3], and out[7] $\in$ *FF* are assigned outside the parameterized loop in `example`, PRMSEQBLKCONSTRUCT instantiates their corresponding flipflops outside the loop in `example-a`. A *gff* is used for the of IC flipflops *r*, *r*2, *r*3, and *x* since they have an IC of true.

Function GNRTLPCONSTRUCT takes a parameterized loop *pls* and recursively constructs `generate` loop statement *gls*.

Statement *gls* contains instantiations $gicff\ ff_u(ic)$ for flipflops corresponding to variables $u \in FF$ assigned in LOOPB(*pls*) with $ic = icMap(u, pls)$.

Function PRMCOMBASSIGNCONSTRUCT returns *RWBlk* that sets default initial values for flipflop next state functions. It first iterates over *pLS* and for each $pls \in pLS$ it calls LPASSIGNCONSTRUCT [Fig. 10(right bottom)] to create loops that handle default initial values for variables assigned in the loop body LOOPB (*pls*). Then it composes conditional assignment statements for the flipflops inferred outside *pLS* where the conditional expression is the corresponding instantiation condition. PRMCOMBASSIGNCONSTRUCT produced lines 33–39 of `example-a` in Fig. 7. It conditionally initialized out_in[*i*] and out_in[7] based on the instantiation condition of their corresponding flipflops out[*i*] and out[7].

Function LPASSIGNCONSTRUCT creates initializing statements for variables assigned in LOOPB (*pls*) that correspond to inferred flipflops. In Fig. 5, out[*i*] is assigned in a parameterized loop. Hence, LPASSIGNCONSTRUCT constructed a parameterized loop to set the initial values of out_in[*i*] on line 33 of Fig. 7. The function is recursive to handle nested parameterized loops.

C. Instantiation Condition Computation

Algorithm INSTCONDCOMPUTE listed in Fig. 11 computes dynamic map *icMap* that maps pairs (s, u) of statements and inferred flipflops to instantiation condition $ic = icMap(s, u)$. Condition *ic* ranges over parameter and parameter related variables. A variable is *parameter related* if it was used with a parameter in a Boolean expression. An example in `example` of Fig. 5 is iterator *i* in the loop statement of line 20 that is bounded by parameter *p*. Intuitively, when *ic* evaluates to *true*, statement *s* refers to an inferred flipflop. Otherwise, *s* does not imply a flipflop instantiation.

Intuitively, INSTCONDCOMPUTE first computes path conditions for each statement and stores the path conditions in dynamic map *pcMap*. It aggregates the path conditions into instantiation conditions for each inferred flipflop variable *u*

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

```
[Ps,icMap,pLS]=PrmSeqBlkConstruct(P,FF,clk,σ)
 pLS={s. s∈ L, s is parametrized, clkEdged(s)};
 icMap = InstCondCompute(S,FF);
 foreach pls in pLS//Handle paramertrized loops
  Ps.append(//Append generate statement to Ps
    GnrtLpConstruct(pls,icMap,clk,σ) );
 endfor
 foreach (null,u,ic) in icMap//assignments ∉ pLS
  Ps.append(
    GicffInst(ff_u,ic,lookUpWire(u),u,0,clk,σ) );
 endfor
 Ps='generate'+Ps+'end generate';

gls=GnrtLpConstruct(pls,icMap,clk,σ)//Recursive
 foreach pls' in loopB(pls) and isPrmLoop(pls')
  Body+=GnrtLpConstruct(pls',icMap,clk,σ);
 endfor
 foreach (pls,u,ic) in icMap
  Body+=GicffInst(ff_u,ic,lookUpWire(u),u,0,clk,σ);
 endfor
 gls=compLStmt(
  initExp(pls),exprsn(pls),incrementExp(pls),Body);

RWBlk=PrmCombAssignConstruct(FF,pLS,icMap)
 foreach pls in pLS //inferred in pLS
  LpAsgnmtStmt=LpAssignConstruct(pls,icMap);
  RWBlk.append(LpAsgnmtStmt);
 endfor
 foreach (null,u,ic) in icMap//inferred outside pLS
  AStmt=compAStmt(lookUpWire(u),u);//u_in=u
  CStmt=compCStmt(ic,AStmt);//if(ic) u_in=u
  RWBlk.append(CStmt);
 endfor

LpAsgnmtStmt=LpAssignConstruct(pls,icMap)//Recursive
 foreach pls' in LoopB(pls) and isPrmLoop(pls')
  Body+=LpAssignConstruct(pls',icMap);
 endfor
 foreach(pls,u,ic) in icMap //inferred in LoopB(pls)
   AStmt=compAStmt(lookUpWire(u),u);
   Body+=compCStmt(ic,AStmt);
 endfor
 LpAsgnmtStmt = compLStmt(
  initExp(pls),exprsn(pls),incrementExp(pls),Body);
```

Fig. 10. Service functions for computing the parameterized sequential block *Ps* and the next state function default value.

```
icMap=InstCondCompute(S,FF)
 PathCondCompute(S,pcMap,true);//Compute pcMap
 plsAll=nestedPrmLps(S);//All parametrized loops in S
 foreach u in FF//u−dataflow outside plsAll
  icMap(null,u)=⋁_{s∈A ∧ s∉plsAll ∧ u=target(s)} pcMap(s);
  foreach pls in plsAll //u−dataflow in plsAll
   icMap(pls,u)=⋁_{s∈A∩loopB(pls) ∧ u=target(s)} pcMap(s);
  endfor
 endfor
 ICMutex(S,FF,plsAll,icMap);//Exclusive conditions

PathCondCompute(S,cond,pcMap)//Recursive
 Process(cond);
 foreach s in S//Propagate cond to blocks of s
  pcMap(s)=cond;
  if s in L//Compose a guard grd of loop s
   grd=(iterator(s)>=exprsn(initExp(s)))∧exprsn(s);
   PathCondCompute(loopB(s),grd∧cond,pcMap);
  elseif s in C //Conditional statement
   PathCondCompute(Then(s),exprsn(s)∧cond,pcMap);
   PathCondCompute(Else(s),(¬exprsn(s))∧cond,pcMap);
  endif
 endfor

ICMutex(S,FF,plsAll,icMap)
 foreach it=iterator(pls) where pls in plsAll
  foreach u in FF where icMap(pls,u)
   //Consider parametrized loops that follow pls
   pLSF=followingPrmLps(pls,parentBlock(pls));
   foreach plsf in pLSF where icMap(pls,u)
    ic_f = icMap(plsf,u);
    //Substitute iterators to exclude ic_f in ic
    ic_f= replaceVar(ic_f,iterator(plsf),it);
    icMap(pls,u)= icMap(pls,u) ∧ ¬ic_f;
   endfor
   pLSN=nestedPrmLps(LoopB(pls));
   foreach plsn in pLSN where icMap(plsn,u)
    ic_n= icMap(plsn,u);
    //Substitute iterators to exclude ic_b in ic
    ic_n = replaceVar(ic_n,iterator(pls_n),it);
    icMap(pls,u)=icMap(pls,u) ∧¬ic_n;
   endfor
   //Consider instantiations outside plsAll
   Au={s. s ∈ A, s ∉ plsAll and target(s)= u};
   foreach s in A_u
    ic_a= (it==index(u,s))∧icMap(0,u);
    icMap(pls,u)= icMap(pls,u)∧¬ic_a;
   endfor endfor endfor
```

Fig. 11. InstCondCompute Algorithm and its service functions: PathCondCompute and ICMutex.

that is assigned: 1) outside and 2) inside parameterized loops *plsAll*. Then it calls ICMutex to make sure the instantiation conditions are mutually exclusive per inferred flipflop u.

1) Function PathCondCompute starts with a block of statements S and a condition *cond*. First, PathCondCompute simplifies *cond* using function Process. Function Process removes *cond* dependency on nonparameter related variables by existentially quantifying these variables. Then, it updates $pcMap(s)$ with *cond* for each statement $s \in S$. When s is a loop statement, PathCondCompute constructs a loop guard *grd* and recursively propagates the conjunction of cond with *grd* to loop body loopB(s). For simplicity of presentation, we compute *grd* assuming all loops are monotonically increasing with a step of 1. When s is a conditional, PathCondCompute recursively propagates the conjunction of *cond*: a) with condition exprsn(s) to the then block and b) with the $\neg$exprsn(s) to the else block. For the shaded code of module `example` in Fig. 5, PathCondCompute computes the path condition of statements in S (lines 20, 25, and 26) as *true* since PathCondCompute is first called with true. The recursive part of PathCondCompute sets the path conditions for statements within conditional or loop blocks. For example, assignment statement at line 23 has a path condition: $(i >= 0 \&\& i < p) \&\& i\%2 \&\& i! = 5$.
2) Next, PathCondCompute calls helper function nestedPrmLps to compute all nested parametric loops and stores the result in *plsAll*. Function nestedPrmLps(B) takes block B and recursively returns the set of parameterized loop statements in B. Then PathCondCompute iterates over inferred flipflops $u \in FF$ to compute the following.
   a) Instantiation condition $icMap(null, u)$ of u outside *plsAll* is the disjunction of the path condition $pcMap(s)$ of each assignment statement s with target u that is not part of a parameterized loop. Intuitively, this is the dataflow of u outside parameterized loops. The instantiation conditions of variables out[2] and out[7] in `example` of Fig. 5 are true and $p > 7$, respectively.

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

b) Instantiation condition *icMap*(*pls*, *u*) of *u* in parameterized loop statement *pls* ∈ *plsAll* is the disjunction of the path condition *pcMap*(*s*) of each assignment statement *s* with target *u* that is part of LOOPB(*pls*). Intuitively, this is the dataflow of *u* in parameterized loop *pls*. The instantiation condition of inferred flipflop out[*i*] in parameterized loop ℓ line 20 of Fig. 5 is the path condition of the assigning statement at line 23.

2) Finally, INSTCONDCOMPUTE calls ICMUTEX to make sure that instantiation conditions of the same inferred flipflop *u* are mutually exclusive. Intuitively, ICMUTEX checks whether a flipflop inferred variable $u \in FF$ has more than one instantiation condition in *icMap*. If so, it makes sure that the conditions are mutually exclusive and that the statements that appear last have priority in instantiation. The exclusion happens by conjunction with the negation of other conditions. For each parameterized loop statement *pls* and inferred flipflop *u* that have an instantiation condition *icMap*(*pls*, *u*), ICMUTEX checks the parameterized loops following *pls*, the loops nested within LOOPB(*pls*), and the assignment statements outside parameterized loops. It uses helper function call FOLLOWINGPRMLPS(*pls*, PARENTBLOCK(*pls*)) to compute the statements that follow *pls* within its parent block. In case of multiple *icMap* entries for an inferred flipflop *u*, ICMUTEX finds the corresponding instantiation conditions *ic_f* in the following statements, *ic_n* in the nested statements and *ic_a* in the nonparameterized loop statements and excludes each of them from the *icMap*(*pls*, *u*) condition. For loop iterator consistency, it replaces the loop iterators in *ic_f* and *ic_n* with *it* the iterator of *pls* using REPLACEVAR. In `example` of Fig. 5, array variable out is assigned in parameterized loop ℓ and in assignments at lines 25 and 27. ICMUTEX updates *icMap*(ℓ, out[*i*]) to ($i >= 0 \&\& i < p) \&\& i\%2 \&\& i! = 5 \&\& !(i == 2 \&\& \text{true}) \&\& !(i == 7 \&\& p > 7)$.

D. COM Correctness

Theorem 1: Given a nonparameterized program *P* with no loops and with only assignment statements in `t-block` and `e-block`, $P' = \text{RTLTRANSFORM}(P)$ is equivalent to *P*.

Proof: P' is equivalent to *P* by construction. For each o' is O' there exists an $o \in O$ with the same name, assignments $s' \in S'$ and $s \in S$ such that $o = \text{TARGET}(s)$ and $o' = \text{TARGET}(s')$. Statements *s* and s' are equivalent by structural induction.

1) The structural induction has base case with variables that are the same in *s* and $s' : i \in I$ and $r \in R$ in *P* with corresponding $i \in I'$ and $r \in R' \cup W'$ in P', respectively. These are equivalent since $I' = I$, and if $r' \in R'$ then it is equivalent to *r* as it was not modified. The case where $r' \in W'$ remains and is similar to the following inductive case since r' is connected to the output of the instantiated GFF *ff_r* whose next state function is *r_in*.
2) Statements *s* and s' differ in case EXPRSN (*s*) contained a variable $r \in R$, $r = \text{TARGET}(bs)$ and *bs* was a blocking statement. Without loss of generality, let $r \in \text{VARS}(\text{EXPRSN}(s))$ be such a variable, it will be replaced by $r_in \in W'$ in s'. However, *r_in*, denoting the next state function of *ff_r*, is guarded by an assignment `r_in = r;` in P' that guarantees a default value, and also guarded by an assignment `r_in = EXPRSN (bs);` in P'. By substitution of *r* with EXPRSN(*bs*) in *s* and *r_in* with EXPRSN(*bs*) in s', both statements are equivalent.

The same applies to conditional expressions in conditional statements and wires in *W* and W'. ■

Theorem 1 generalizes to all nonparameterized programs as follows. A loop can be unrolled into a sequence of conditional statements. Nested conditional statements can also be factored into a sequence of conditional statements.

Theorem 2: Given program *P* with a parameter variable *p*, RTLTRANSFORM(*P*) returns a program P'. Let $P|_{p=b}$ be *P* with parameter *p* substituted with an arbitrary value *b*. RTLTRANSFORM $(P|_{p=b})$ is equivalent to $P'|_{p=b}$.

Proof: We unroll all loops of RTLTRANSFORM($P|_{p=b}$) and $P'|_{p=b}$ for the given value *b* to form equivalent programs P_1 and P'_1. We evaluate all flipflop instantiation conditions to either true or false in P'_1. We replace all occurrences of *gicff*(*true*)(*args*) with its equivalent *gff*(*args*) and *gicff*(*false*)(*args*) with its equivalent empty statement and form P'_2. P'_2 is equivalent to P_1 by structural induction. ■

VI. CONCERN WEAVING

Algorithm CONCERNWEAVE presented in Fig. 12 takes as input a $\mathcal{C}/\mathcal{S}$-RTL program P', inferred flipflops *FF* in P', concern supporting RTL PCONC, and recipe file RCP and returns *Pw* and *Pa*. *Pw* is P' with concerns weaved in and *Pa* is the auxiliary concern interconnect.

The recipe file *RCP* contains the following.

1) A SELECTRULE that takes *FF* and returns a subset $FFSel \subseteq FF$. The concern engineers specify the rule either explicitly by flipflop names or implicitly using regular expressions and other flipflop element features.
2) A list of common concern interface signals σ_{common} that connects to all flipflops.
3) A list of flipflop specific interface signals σ_{ff}. CONCERNWEAVE clones and connects σ_{ff} to each *ff* in *FFSel*.
4) The type of the concern flipflop FFTYPE.
5) A template for an interconnect RTL that takes as parameter the cardinality of *FFSel*.
6) An interconnect rule that specifies connections among the concern support RTL wires and σ. Executing INTERCONNECTRULE constructs the connectivity within *Pa*. Concern engineers specify this rule by providing tuples of connecting signals with simple logic expressions that are simple wires most of the time.

CONCERNWEAVE starts *Pw* as a clone of P' and applies *RCP*.SELECTRULE to compute *FFSel* the flipflops selected for concern weaving. For each *u* in *FFSel*, it clones σ_{ff} into $u.\sigma_{ff}$, appends it to σ_c and injects σ_{common}, σ_{ff} as the concern interface of flipflop *RCP*.FFTYPE*ff*_*u*. CONCERNWEAVE replaces σ with σ_c in *Pw*, instantiates *Pa* to be auxiliary interconnect RTL from template *RCP.interconnectRTL*. Finally, CONCERNWEAVE applies *RCP*.INTERCONNECTRULE to construct the interconnection logic within *Pa*.

The interconnect template and the concern interconnect rule are typically simple wiring with care to the cardinality of the concern flipflops. For example, the DFT concern interconnect connects the flipflop scan chain concern wires in sequence

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

```
[Pw, Pa]=ConcernWeave(P', FF, PCONC, RCP)
Pw=clone(P');  //Start with clone
σc= RCP.σcommon; //Common interface signals
FFSel= RCP.selectRule(FF); //Select the flipflops
foreach u in FFSel
 u.σ=clone(RCP.σff); //Clone flipflop interface
 σc+=u.σ; //Add it module's interface
 u.σ= RCP.σcommon+u.σ; //Complete flipflop interface
 u.ffType = RCP.ffType;
 Pw.replaceInst(ff_u, instantiate(u));
endfor
Pw.replace(σ, σc);
//Construct auxiliary interconnect RTL
Pa=instantiate(RCP.interconnectRTL, |FFSel|);
Pa=RCP.interconnectRule(Pa, PCONC, Pw.σc, |FFSel|);
```

Fig. 12. Algorithm ConcernWeave.

in *Pa*. The clocking interconnect, on the other hand, creates in *Pa* local clock buffers and control cells, and connects their output to the flipflop clock control concern wires.

COM invokes ConcernWeave with one design concern at a time. It uses the $\mathcal{C}/\mathcal{S}$-RTL concern specific output in the next steps of the design flow. COM supports library recipe files with default rules and design recipe files that may override ffType, selectRule, and interconnectRule. The design recipe files can be used to specify additional flipflop properties such as local clock buffer types, scan chain types and test ports. In practice, the design recipe files are minimally used.

VII. Implementation and Results

The IBM electronic design automation group implemented and applied COM in the current design cycle on industrial Verilog RTL. The supporting tools are also used to import legacy and third party IP into IBM's methodology. A version of the tools that uses Verific as a front end [11] is available online for reviewers.

COM automatically validates the output of the implementation within the SixthSense transformation-based verification framework [23]. It uses combinational equivalence checking for transformations that preserve memory elements such as RTLTransform . It uses sequential equivalence checking for ConcernWeave when concerns come with service memory elements in supporting RTL,

The transformation and concern weaving flows are fully automatic independent of the module. The resulting $\mathcal{C}/\mathcal{S}$-RTL is comparable in to the manual concern insertion RTL as illustrated by `example-a` of Fig. 7 and `example-m` of Fig. 5, respectively. The $\mathcal{C}/\mathcal{S}$-RTL output of COM was used for functional verification including simulation, model checking and debugging tools. We report on the verification time per designer team and we report the number of lines of code as metrics to illustrate the transformation efficiency and the design complexity, respectively. We attribute the verification time reduction to: 1) verification setup; 2) faster simulation and emulation; 3) faster debugging; 4) defect reduction; and 5) a boost in designer productivity.

In the following, we report on the results for the IBM industrial designs. We then report on applying the methodology to open source Verilog applications.

A. Industrial Verilog Applications

In previous design cycles, IBM designers dropped verification at the gate-level due to the large verification time and adopted the manual RTL insertion methodology which was at least an order of magnitude faster. Concern insertion automation frees the designer from the complexities that are concern-specific. For example, DFT verification and scan chain debugging require a lot of team months mainly due to time overhead for flipflop instantiation and concern weaving.

In the current design cycle, the designers use COM to further improve verification time. Both methods were effectively executed by the same team. We report the advantages of COM in comparison with RTL manual concern insertion. The reported results are for the following Verilog designs.

Application 1 (Embedded PowerPC Core): A 32-bit RISC CPU for use in custom logic applications.

Application 2 (Embedded PCIE PCS): The physical coding sublayer of the PCIE [21], a high-speed serial computer expansion bus standard.

Application 3 (Embedded PCIE Macros): Eight different PCIE macros that cover timing, phase locked loop controller logic, receiver and transmitter logic and clocking, reference clock selection, and calibration.

Table I compares the RTL implementation size of the designs before and after the COM transformation and lists the number of DFT concern flipflop instantiations for both bit and vector flipflops. Entries PCIE M1 to PCIE M8 in the table represent eight design macros of the embedded PCIE on which COM was applied, and which are referred to as Application 3. The transformation introduced a negligible increase in code size for most macros. For example, the LOC of PowerPC increased from 90.7K to 91.9K. The rounding of LOC hides the increase for other macros. In case of heavily sequential RTL, the transformation results in a $\mathcal{C}/\mathcal{S}$-RTL composed mostly of flipflop instantiations and hence the LOC drops. This is the case for PCIE M3, M4, and M6, where the LOC dropped from 4.8K to 4.7K, 34K to 33.9K, and 69.7K to 69.2K, respectively. The increase in LOC due to concern weaving varies according to the number of GFFs in the $\mathcal{C}/\mathcal{S}$-RTL that match the selection rules and their dimensions. For example, in Table I, the LOC increased from 1K to 2.5K for PCIE M2, and from 359.4K to only 405.9K for PCIE M5. 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 is worth noting that the original RTL is still used directly for pure functional verification where nonfunctional design concerns are not relevant; thus, COM has no negative effect on that. Table I also lists the resulting gate-level netlist size for illustration purposes; the gate-level netlist was not used for the verification of memory related concerns covered at RTL verification.

For the current design cycle, COM converted these designs to $\mathcal{C}/\mathcal{S}$-RTL using an efficient implementation of RTLTransform . Concern weaving mapped $\mathcal{C}/\mathcal{S}$-RTL to proprietary technology also at RTL using an efficient implementation of ConcernWeave. This involved replacing GFFs with technology flipflops as well as weaving of scan chains insertion and proper clocking. This further improved verification time by 40% compared to RTL manual concern insertion as indicated in Table II for the embedded PowerPC and embedded PCIE PCS applications.

COM was applied automatically on 72% of PCIE macro components (Application 3) where a total of 300 000 lines of code were added to insert the concerns. COM, along other optimization techniques, introduced 1–1.5-year savings in the verification time. The contribution of COM to these savings is estimated to be up to 40%. We report the following.

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

TABLE I
IMPLEMENTATION SIZE IN THOUSANDS OF LINES OF CODE FOR INDUSTRIAL EMBEDDED POWERPC AND PCIE MACROS USING COM. [1]RESULTS REPORTED IN [9]. [2]NEW RESULTS FOR OTHER MACROS OF PCIE. [3]NUMBER OF INSTANTIATED DFT FLIPFLOPS (VECTORED AND SINGLE BITS)

Type of Code		**PowerPC**[1]	**PCIE PCS**[1]	**PCIE M1**[2]	**PCIE M2**[2]	**PCIE M3**[2]	**PCIE M4**[2]	**PCIE M5**[2]	**PCIE M6**[2]	**PCIE M7**[2]	**PCIE M8**[2]
RTL	**Entry**	90.7	15.0	16.3	1.0	4.8	34.0	359.4	69.7	4.4	0.5
	$\mathcal{C}/\mathcal{S}$-*RTL*	91.9	15.5	16.3	1.0	4.7	33.9	359.4	69.2	4.4	0.5
	Concern weaved $\mathcal{C}/\mathcal{S}$-*RTL*	140.2	18.7	19.8	2.5	10.6	46.0	405.9	96.8	6.2	1.3
Gate level concern insertion		1,547.2	340.0	369	24	109	771	8,148	1,581	99	12
#DFT-*ff* Instances[3]		1,140	195	72	34	168	365	1,345	894	53	13

TABLE II
MANUAL TO COM VERIFICATION TIME COMPARISON. [1]RESULTS SHOWN IN [9]. PROPOSED METHOD CONTRIBUTES IN 30%–40% OF THE TOTAL 1–1.5-YEAR VERIFICATION TIME SAVINGS ACHIEVED BY OPTIMIZATION TECHNIQUES FOR OTHER PCIE MACROS LISTED IN TABLE I

Verification time for embedded designs (team weeks)			
Concern insertion		**PowerPC**[1]	**PCIE PCS**[1]
RTL	**Manual**	12.9	8.6
	COM	7.0	5.0
Gate level: post-synthesis		NA: Large	NA: Large

1) We note 40%–50% reduction in the overall verification time, where 10%–20% are saved due to faster debugging and change management time, and 30%–40% are saved due to faster verification tool running in terms of simulation and emulation times as indicated in Table II. Verification time decreased from 12.9 to 7 team-weeks for PowerPC when we moved from manual concern insertion to COM. It decreased from 8.6 to 6 team-weeks for PCIE PCS.
2) We note that verification environment setup effort was reduced by 30%.
3) In addition, we note significant boost in designer productivity with at least 40% less time spent on RTL composition and management due to automated generation of $\mathcal{C}/\mathcal{S}$-RTL and concern-weaved $\mathcal{C}/\mathcal{S}$-RTL.

Automatically mapping external application specific integrated circuits IPs obtained from third party vendors to the design flow with proprietary technology cut the verification effort by more than 40% in most cases. This is due to the fact that entry IP was preserved which reduced unnecessary communication about proprietary technology and concern specific knowledge with third party.

Pure functional verification is now possible in the same verification setup with concern free flipflops. Partial concern verification that is now practical at no overhead to verification engineers cut the RTL presented to simulation and emulation by half in many cases.

In several cases, concern verification defects were reduced in half due to *correct by construction* insertion of concerns enabled by COM, versus error prone manual insertion.

The runtime of code transforms and concern weaving is negligible overhead compared to time needed: 1) for manual insertion and 2) debugging which requires several weeks. For example, it took less than 1 min for the PowerPC.

B. Open-Source Verilog Applications

In this section, we demonstrate COM on general non-IBM designs. We applied COM on ten parametric modules extracted from open source projects. The modules involve parameter variables used in conditional and loop statements. Hence, the instantiation of flipflops corresponding to variables assigned within such parametric statements depends directly on the parameter value. The modules are instantiated several times in different projects with different values of the parameters. This well reflects the need for generic RTL verification of hardware modules. Tables III and IV present a summary of the results in terms of post-transformation lines of code and instantiation condition complexity.

TABLE III
PARAMETRIC MODULES FROM PID, ZET, AND MOR1KX APPLICATIONS

Application	**PID** [24]	**Zet** [26]		**Mor1kx** [30]	
Module	**synth_reg**	**div_uu**	**div_su**	**dcache**	**icache**
LOC before	52	135	92	439	282
#Parameters	2	2	2	18	15
LOC after	57	242	155	516	318
#GICFF	1	11	9	11	7
IC complexity	(4, 15)	(1-4, 0-14)	(1-3, 0-16)	(3-19, 4-149)	(0-13, 0-28)
Simplified IC complexity	(2, 11)	(0-3, 0-12)	(0-2,0-14)	(0-4, 0-5)	(0-5, 0-7)

TABLE IV
PARAMETRIC MODULES FROM THE MIAOW APPLICATION

Application	**MIAOW** [29]				
Module	**Instr Buffer**	**Dispatch Soft**	**Cu Handler**	**Resource Update Buffer**	**Vgpr Conten Tracker**
LOC before	37	141	220	74	116
#Parameters	1	1	4	3	2
LOC after	79	279	406	109	194
#GICFF	6	30	27	5	11
IC complexity	(4-4, 6-14)	(1-5, 0-30)	(1-9, 0-30)	(1, 2)	(1-15, 0-42)
Simplified IC complexity	(2-2, 3-7)	(0-3, 0-17)	(0-5, 0-21)	(0, 0)	(0-42, 0-41)

In the absence of a parameter value, and without instantiation condition flipflops, manual insertion of concerns is not feasible. Comparing the time it takes to insert concerns in $\mathcal{C}/\mathcal{S}$-RTL generated from parameterized RTL, against the time it takes to insert concerns manually for *several values of a parameter* is incomplete and will obviously show results in favor of COM. So no direct comparison is possible.

The modules listed below demonstrate the effectiveness of COM supporting tools with online open source IPs.

Application 1 (IP Proportional-Integral-Derivative Controller (PID) [24], [25]): An IP core library of PID controller by Analog Devices Inc. The library consists of 2568 lines of code, and includes 48 modules 65% of which are parametric. We considered the *synth_reg_reg* latency

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

register module whose parameters specify input width and introduced latency. COM allows to instantiate conditional flipflops corresponding to internal latency register variables.

Application 2 (Zet Processor [26], [27]): An x86 processor. The implementation consists of 2858 lines of code, and includes 29 modules 17% of which are parametric. We considered the *zet_div_uu* and *zet_div_su* divider modules. The parameters specify input widths and thereby determine the number of flipflops needed for the division implementation.

Application 3 (Many-Core Integrated Accelerator of Wisconsin (MIAOW) [28], [29]): An RTL implementation of the AMD Southern Islands GPGPU ISA. The project consists of 33 041 lines of code, and includes 235 modules 29% of which are parametric. We considered the *instr_buffer*, *dispatcher_soft_tb*, *cu_handler*, *resource_update_buffer_tb*, and *vgpr_contention_tracker* modules. The parameters specify width of architecture units, as for example the *wavefront* (collection of threads) size and the slot number for wavefronts. In addition, modules often have local parameters that similarly specify architectural characteristics.

Application 4 (mor1kx Processor [30]): An OpenRISC 1000 compliant processor IP core. The implementation consists of 15 942 lines of code, and includes 46 modules 83% of which are parametric. We considered the *mor1kx_dcache* and *mor1kx_icache* cache modules. The parameters specify block width, set width, and number of cache ways. These parameters affect the instantiation of flipflops necessary to hold the least recently used tag information.

Tables IV and III report the number of parameters and lines of code of each module in Application 2 and in the rest of the applications, respectively. They also present the lines of code and the number of GFF instantiations in each transformed module. For example, in Table III, the lines of code of *dcache* increased from 439 to 516 and the number of instantiated flipflops is 11. Finally, the tables present the complexity of the flipflop instantiation conditions in the output modules before and after simplification with existential quantification and constant propagation. The complexity is given in terms of the number of variables and operators in the conditions. For example, module *Instr Buffer* of the MIAOW project has 6 instantiation condition flipflops each with a condition composed of two variables and a number of operators ranging from 3 to 7. In some cases, the instantiation condition resolved to *true* as in the case of the flipflop instantiated in module resource update buffer of the MIAOW project.

VIII. Related Work

Aspect oriented programming promotes the explicit separation of functionality from other concerns in software artifacts. The concerns are specified in *aspects* that are later automatically woven into the software. The importance of aspect oriented programming for hardware verification purposes is highlighted in [13] as a method that supports cross-functional iterations as well as integration of legacy verification IP. AspectVHDL [12] is an aspect-oriented extension of VHDL. AspectVHDL introduces procedure, type, architecture, and process trigger *jointpoints* and relates *pointcut expressions* and *advices* to them. However, these jointpoints cannot capture concerns such as DFT, LBIST, and clock-gating [21].

The *successive refinement* method [3] uses the IEEE standard unified power format (UPF) [4] for early specification of power intent. It allows verification of power management to happen with the design functionality verification. The method frees the designer from writing technology-specific implementation UPF [5] by using constraint, configuration, and implementation UPF side files that vary in their description and detail level. The constraint files are provided by the IP provider, and the configuration and implementation files are provided by the IP consumer/end user. Khan *et al.* [3] discussed the challenge of manual concern separation for the successive refinement method.

While successive refinement method separates the power concern from the functionality, COM separates the sequential and combinational logic, and hence separates all memory-related concerns from functionality. COM automatically performs concern separation and weaving at the RTL level. This allows high level verification and debugging without the need for special simulator and tool support. For COM, the logic designer only deals with his original functional code. The weaving tool replaces the GFFs with concern specific flipflops using concern specific recipe files provided by concern specific engineers. This fully separates responsibilities of functional engineers and concern engineers.

Metropolis [14]–[16] is an embedded design platform based on formal modeling and separation of concerns. Process and media structures ensure communication is separate from computation. Quantity manager structures manage energy, time, and event priorities and thus separate performance from behavior. Netlists help separate functionality from architecture.

Behavior interaction priority (BIP) [17], [18] is a component-based design framework for embedded systems where behavior, interaction, and priority concerns are separate. BIP components are labeled transition systems with data elements. Interactions connect the components through ports, and perform data transfer between component data elements. Priority constraints annotate interactions. A scheduler executes the BIP system and resolves nondeterministic behavior.

Transaction-level modeling (TLM) [19] separates functionality and timing. Programmer view (PV) models specify functionality and shield designer engineers from micro-architecture and timing concerns. PV with time (PVT) models extend PV models with realistic timing. An alternating execution model dynamically weaves the PV and PVT models.

Metropolis, BIP and TLM address concern separation by devising their own syntax and semantics. Furthermore, none of them tackle the separation of concerns at the RTL like DFT, LBIST, and clocking. COM targets these concerns, and is applicable to industrial RTL such as Verilog and VHDL including existing legacy RTL with no language restrictions.

Atrenta SpyGlass [7] creates a virtual hardware design prototype to enable early analysis and optimization of area, timing and power constraints. SpyGlassDFT [8] wraps around the synthesis phase to ensure that RTL is scan-compliant and yields desired coverage. Up to our knowledge, no further description of their methodology is available in literature. COM provides a solution for concern separation solely at the RTL level with minor modifications to the original RTL code.

IX. Conclusion

This paper presented COM, that separates functionality from memory related design concerns to enable verification early at RTL. It takes input RTL and automatically generates output RTL with separate sequential and combinational blocks. The methodology allows the implementation of memory related concerns such as DFT and clocking by substituting GFF instantiations in the sequential blocks with concern specific

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.

library flipflops. It is sound and its outputs is validated equivalent to the input RTL using formal verification equivalence checking. The methodology 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%.

REFERENCES

[1] Y. Abarbanel, E. Singerman, and M. Y. Vardi, "Validation of SoC firmware-hardware flows: Challenges and solution directions," in *Proc. DAC*, San Francisco, CA, USA, 2014, pp. 1–4.

[2] T. Glokler *et al.*, "Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning," in *Proc. Formal Methods Comput. Aided Design (FMCAD)*, San Jose, CA, USA, 2006, pp. 3–10.

[3] A. Khan *et al.*, "Successive refinement: A methodology for incremental specification of power intent," in *Proc. Design Verification Conf.*, Mar. 2015.

[4] *Standard for Design and Verification of Low Power Integrated Circuits*, IEEE Standard 1801 TM-2009, Mar. 27, 2009.

[5] *IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems*, IEEE Standard 1801-2015, pp. 1–515, Mar. 25, 2016, doi: 10.1109/IEEESTD.2016.7445797.

[6] T. Parr, *The Definitive ANTLR4 Reference: Building Domain Specific Language*, Pragmatic Bookshelf, 2007.

[7] Atrenta Inc. *Spyglass*. Accessed: Jun. 27, 2018. [Online]. Available: http://www.atrenta.com

[8] *SpyGlass DFT ADV RTL Testability Analysis and Improvement*. Accessed: Jun. 27, 2018. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/spyglass-dft-adv-ds.pdf

[9] M. Safieddine *et al.*, "Methodology for separation of design concerns using conservative RTL Flipflop inference," in *Proc. Design Verification Conf. Exhibit.*, Mar. 2015.

[10] M. H. Safieddine, R. Kanj, F. A. Zaraket, A. Elzein, and M, Jaber, "Separation of concerns for hardware components of embedded systems in BIP," in *Proc. Int. Symp. Qual. Elect. Design*, Santa Clara, CA, USA, Mar. 2015, pp. 337–344.

[11] *Verific Design Automation*. Accessed: Jun. 27, 2018. [Online]. Available: http://www.verific.com

[12] M. Meier, S. Hanenberg, and O. Spinczyk, "AspectVHDL stage 1: The prototype of an aspect-oriented hardware description language," in *Proc. Workshop Modular. Syst. Softw. (MISS)*, 2012, pp. 3–8.

[13] J. Strober and C. Goss, "What ever happened to AOP?" in *Proc. Design Verification Conf. Exhibit.*, Mar. 2015.

[14] F. Balarin *et al.*, "Metropolis: An integrated electronic system design environment," *IEEE Comput. Soc.*, vol. 36, no. 4, pp. 45–52, Apr. 2003.

[15] A. Sangiovanni-Vincentelli, "Quo Vadis SLD? Reasoning about trends and challenges of system-level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.

[16] A. Davare *et al.*, "A next-generation design framework for platform-based design," in *Proc. DVCon*, 2007.

[17] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Proc. IEEE Int. Conf. Softw. Eng. Formal Methods*, Sep. 2006, pp. 3–12.

[18] A. Basu *et al.*, "Rigorous component-based system design using the BIP framework," *IEEE Softw.*, vol. 28, no. 3, pp. 41–48, May/Jun. 2011.

[19] J. Cornet, F. Maraninchi, and L. Maillet-Contoz, "A method for the efficient development of timed and untimed transaction-level models of systems-on-chip," in *Proc. IEEE Design Autom. Test Europe*, Munich, Germany, 2008, pp. 9–14.

[20] L. Wang, C. Stroud, and N. Tuba, *System-on-Chip Test Architectures: Nanometer Design for Testability*. San Francisco, CA, USA: Morgan Kauffman, 2007.

[21] M. Shaker and M. Bayoumi, "Novel clock gating techniques for low power flip-flops and its applications," in *Proc. MWSCAS*, Columbus, OH, USA, 2013, pp. 420–424.

[22] P. M. Nyasulu, *Introduction to Verilog*, Carleton Univ., Ottawa, ON, Canada, 2003.

[23] P. K. Nalla *et al.*, "The art of semi-formal bug hunting," in *Proc. 35th Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, 2016, pp. 1–8.

[24] *IP PID Controller*. Accessed: Jun. 27, 2018. [Online]. Available: https://github.com/analogdevicesinc/hdl/tree/5e082be9637888fd0c024f4fde16647e97da4627/library/ip_pid_controller

[25] *Analog Devices Inc*. Accessed: Jun. 27, 2018. [Online]. Available: http://www.analog.com

[26] *Zet Processor*. Accessed: Jun. 27, 2018. [Online]. Available: http://zet.aluzina.org/index.php/Zet_processor

[27] *Zet Processor*. Accessed: Jun. 27, 2018. [Online]. Available: https://github.com/marmolejo/zet

[28] *MIAOW*. Accessed: Jun. 27, 2018. [Online]. Available: https://github.com/VerticalResearchGroup/miaow

[29] R. Balasubramanian *et al.*, "MIAOW—An open source RTL implementation of a GPGPU," in *Proc. IEEE COOL Chips XVIII*, Yokohama, Japan, 2015, pp. 1–3.

[30] *Mor1kx*. Accessed: Jun. 27, 2018. [Online]. Available: https://github.com/openrisc/mor1kx

Maya H. Safieddine received the B.Eng. and Ph.D. degrees from the American University of Beirut, Beirut, Lebanon, in 2012 and 2018, respectively.

In 2016, she did a three month internship at Defacto Technologies, Grenoble, France. In 2017, she joined the IBM Systems and Technology Group, Austin, TX, USA, as a Software Engineer.

Dr. Safieddine was a recipient of the CNRS-L/AUB Doctoral Scholarship Award in 2016 from the Lebanese National Council for Scientific Research in recognition of her research work.

Fadi A. Zaraket received the bachelor's and master's degrees in CCE from the American University of Beirut (AUB), Beirut, Lebanon, in 1996 and 2001, respectively, and the Ph.D. degree in ECE from the University of Texas at Austin, Austin, TX, USA, in 2007.

He is currently an Associate Professor with the ECE Department, AUB. From 2001 to 2008, he was with IBM, Austin, researching on logic verification and debugging tools. He was also with Sun Microsystems, Santa Clara, CA, USA, and Santa Cruz Operations, Santa Cruz, CA, USA, from 1999 to 2001.

Rouwaida Kanj (SM'12) received the Ph.D. degree from the University of Illinois Urbana–Champaign, Champaign, IL, USA, in 2004.

She is currently an Assistant Professor with the American University of Beirut, Beirut, Lebanon. From 2004 to 2012, she was a Research Staff Member with IBM Research Laboratories, Austin, TX, USA. She has authored over 60 technical papers, 30 issued patents, and several pending patents.

Dr. Kanj was a recipient of three IBM Ph.D. fellowships.

Ali El-Zein received the bachelor's degree in computer and communication engineering from the American University of Beirut, Beirut, Lebanon, in 1990, and the Ph.D. degree in ECE from the University of Iowa, Iowa City, IA, USA, in 1993.

Since then, he has been with the Design and Verification Automation Tools Group, IBM, Austin, TX, USA, where he was the Technical Team Lead for a newly design process and tools introduced, related to aspect oriented design, for the last five years.

Wolfgang Roesner received the Dipl.Ing. and Ph.D. degrees from the University of Kaiserslautern, Kaiserslautern, Germany, in 1980 and 1983, respectively.

He has been a Fellow of Functional Verification and Verification Tools with IBM, Austin, TX, USA, since 2011. He has a long history developing foundational technologies across the whole presilicon verification flow. During the last 12 years, he has also led several server microprocessor verification projects and is responsible overall for IBM System's verification methodology.

Authorized licensed use limited to: American University of Beirut. Downloaded on April 29,2024 at 11:49:36 UTC from IEEE Xplore. Restrictions apply.