Exercising Symbolic Discrete Control for Designing Low-power Hardware Circuits: an Application to Clock-gating [⋆]

Mete Özbaltan [*] **Nicolas Berthier** [*]

[*] *Department of Computer Science, University of Liverpool, UK*

Abstract: We devise a tool-supported framework for achieving power-efficiency of hardware chips from high-level designs described using the popular hardware description language Verilog. We consider digital circuits as hierarchical compositions of sub-circuits, and achieve power-efficiency by switching-off the clock of each sub-circuit according to some clock-gating logic. We encode the computation of the latter as several small symbolic discrete controller synthesis problems, and use the resulting controllers to derive power-efficient versions from original circuit designs. We detail and illustrate our approach using a running example, and validate it experimentally by deriving a low-power version of an actual Reed-Solomon decoder.

Keywords: Symbolic Discrete Controller Synthesis, Digital Synchronous Circuits, Power-efficiency

1. INTRODUCTION

Power efficiency of digital circuits is nowadays of paramount importance for constructing embedded electronic devices, and various mechanisms can be used to reduce power consumption in hardware chips. At the technological level, these include clock-gating, multi-supply voltage, and power-gating for instance (Hariharan and Jaya Kumar, 2012). In synchronous circuits in particular, clock-gating is used to selectively cut off the clock of components with the aim of reducing the power dissipation induced by the switching activity it incurs. This technique mostly consists in computing the Clock-Gating Logic (CGL) for sub-circuits, and then translating each CGL itself into a piece of circuit whose output wires can be used to switch-off (to gate) the clocks that drive the sub-circuits.

In this paper, we observe that CGL computation is actually a feedback control problem, where the sub-circuit constitutes the system to control, and the objective is to switch-off its clock whenever possible. This new perspective constitutes a first step towards employing other control techniques for producing self-adaptive power efficient digital circuits, *e.g.,* that would automatically adapt to the remaining capacity of some battery according to an objective power/performance trade-off.

Discrete Controller Synthesis (DCS) The control theory of Discrete Event Systems (DES — Ramadge and Wonham (1989); Cassandras and Lafortune (2007)) allows the use of constructive methods ensuring, *a priori* and by means of control, required properties on a system's behavior. Usually, the starting point of these theories is: given a model for the system and control objectives, a *controller* must be derived by various means such that the resulting behavior of the closed-loop system meets the *control objectives*. A typical example is the *safety control problem* for *symbolic* systems (*i.e.,* described using state and input variables with associated dynamics), where the desired objective is the enforcement of some *invariant a priori* not satisfied by the system: a controller is to be computed that restricts the admissible values for a subset of the input variables (referred to as *controllable variables*) so that the resulting *controlled system* satisfies the invariant. Finding DCS algorithms computing such controllers in the case of finite-state symbolic systems (*i.e.,* where the state and input variables are Booleans only) has been the objective of several studies, and led to several implementations, *e.g.,* by Marchand et al. (2000); Marchand and Samaan (2000). Berthier and Marchand (2014) later extended these studies for infinite-state systems, featuring numerical state and input variables. Meanwhile, other studies considered *optimization* objectives, given partial order relations (Marchand and Le Borgne, 1998; Marchand and Samaan, 2000), or cost functions (Dumitrescu et al., 2010). Most of these solutions adapt Bellman's algorithm for the computation of optimal strategies using dynamic programming (Bellman, 1957). In this work, we exploit DCS principles through the use of the BZR environment (Delaval et al., 2013), that integrates symbolic DCS within the reactive data-flow language Heptagon; we give more background on Heptagon in Section 2.2.

Contributions We present a framework involving symbolic DCS to achieve energy efficiency of hardware chips. We consider circuits described using the Hardware Description Languages (HDL—*e.g.,* Verilog) at the Register-transfer Level (RTL) abstraction. RTL descriptions are high-level hierarchical compositions of components, registers, and logical operators, linked using wires. They can be converted into equivalent digital chip designs for Application-specific Integrated Circuits (ASICs) or Field-programmable Gate Arrays (FPGAs).

The framework broadly comprises the following steps: First, the RTL description of the original circuit is translated into a set of synchronous models where controllable variables represent output wires of CGLs. These models are associated with control objectives whose enforcement guarantee the correct behavior of the CGLs. Second, the latter are obtained using a symbolic DCS algorithm. Last, the CGLs are translated into pieces of circuits that are then integrated into a new, clock-gated circuit design. Our translation algorithm is parametrized with a set of variables to be picked from the HDL description, and that is used

[⋆] Authors' version.

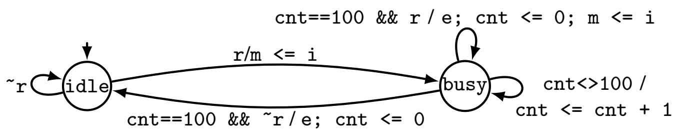

cnt==100 && r / e; cnt <= 0; m <= i
r/m <= i
~r
idle
busy
cnt<>100 /
cnt <= cnt + 1
cnt==100 && ~r / e; cnt <= 0

Fig. 1. Mealy machine symbolically encoded by register `state` in Verilog module `m` of List. 1, decorated with operations on `cnt` and `m`. The initial value of `cnt` is `0`.

to abstract away most of the circuit in order to: (i) focus on the portion of its sequential logic that is relevant for expressing the CGLs; and thus (ii) restrict the size of the DCS problems. Our algorithm automatically generates interpreted non-controllable inputs called *oracles* to model the non-determinism introduced by the abstractions and allow the computation of deterministic, hence implementable, CGLs. We give a running example along with a description of the Verilog HDL and BZR in Section 2, and use it to describe and illustrate the framework in Section 3. We exercise our technique on a realistic case study in Section 4, and give related works in Section 5, and conclude in Section 6.

2. BACKGROUND & RUNNING EXAMPLE

We now introduce the Verilog HDL using a running example, and then describe the fragment of the Heptagon language that we use for the modeling and computation of CGL for RTL circuits.

2.1 The Verilog Hardware Description Language

Verilog is an HDL dedicated to the design of electronic systems. In particular, it can be used to specify *synchronous circuits*. The description of such a circuit in Verilog consists of a *main module* made of an assemblage of *registers*, *wires*, *gates* and/or *sub-modules*. Each of the latter components features an interface that comprises *input* or *output wires* or *registers*. Verilog provides several constructs to program modules, such as conditional and case statements, wire/register declarations and assignments, and event detection (*e.g.,* positive edge detection, triggered when the value carried by a wire transitions from 0 to 1). One input wire, usually called `clk`, carries a clock that is used to trigger changes in the values of registers.

We give in List. 1 an example specification of a module "`m`". It starts with the declaration of its interface, here comprising basic wires (*e.g.,* `clk`, `r`, and `e`) or wire arrays (*e.g.,* `i` and `o`, here used to carry data). A declaration of constants used internally (`idle` and `busy`) follows, along with internal registers. Assignments at lines 7 and 8 describe the logical values taken by the corresponding wires at any instant by means of logical expressions. The code following "`always @(posedge clk)`" consists in conditional "clock-triggered" assignments to internal or output registers, denoted by "<=". For instance, the statement "`state <= busy`" at line 11 states that, at every instant t where a positive edge of `clk` occurs, the value memorized by the register `state` takes the value `busy` if the condition `r` holds (*i.e.,* `r` carries the value `1` at the very instant where `clk` becomes `1`); notice the value of `state` would actually become `busy` at a subsequent instant $t+\varepsilon$. The internal registers in `m` *symbolically* encode a finite-state automaton, that we represent in the form of a Mealy machine with variables in Fig. 1.

We show in List. 2 an example module `main` making use of `m`. Sub-module instantiations `m1` and `m2` are at lines 8 and 10. `main` features an internal working mode memorized using the register `cfg` taking its values in $\{\texttt{LQ},\texttt{HQ}\}$: as can be seen on line 18, upon a positive edge of `clk`, if the request signal `r` holds and neither

```
 1 module m (input clk, input r, input [7:0] i,
             output [7:0] o, output e);
     parameter idle = 0, busy = 1;
     reg state = idle;
 5   reg [6:0] cnt = 0;        // to fake lengthy computations
     reg [7:0] m;              // internal memory
     assign o = m;             // always output memorized value
     assign e = (cnt == 100); // raise e when counter reaches 100
     always @(posedge clk)
10     case (state)
         idle: if (r) begin m <= i; state <= busy; end
         busy: if (e) begin
                 cnt <= 0;             // reset counter
                 if (r) m <= i;        // restart immediately
15               else state <= idle;
               end else cnt <= cnt + 1;
       endcase
   endmodule
```

List. 1. Verilog module `m`, faking computations on data given on input wires `i` upon request `r`. Output wires `o` carry its result, available when `e` becomes `1`.

```
 1 module main (input clk, input r, input mode, input [7:0] i,
                output [7:0] o, output e, output error);
     parameter LQ = 1, HQ = 0;     // Low/High Quality modes
     reg cfg = LQ;
 5   reg wait_m1 = 0, wait_m2 = 0; // sub-module's working statuses
     wire [7:0] o1, o2;            // output data of sub-modules
     wire r1 = r & ~error, e1;     // request & end wires for m1
     m m1 (.clk(clk), .r(r1), .i(i), .o(o1), .e(e1));
     wire r2 = e1 & (cfg == HQ), e2;// request & end wires for m2
10   m m2 (.clk(clk), .r(r2), .i(o1), .o(o2), .e(e2));
     // raise error upon request before end of m1 or m2
     assign error = r & (wait_m1 | wait_m2);
     // select data output and end signal based on configuration
     assign o = (cfg == LQ) ? o1 : o2;
15   assign e = (cfg == LQ) ? e1 : e2;
     always @(posedge clk) begin   // behavioral assignments
       if (r1) begin               // accept new request
         cfg <= mode;              // change mode
         wait_m1 <= 1;             // wait for end of m1
20     end
       if (e1) wait_m1 <= 0;
       if (cfg == HQ) begin        // HQ Mode
         if (r2) wait_m2 <= 1;     // wait for end of m2
         if (e2) wait_m2 <= 0;
25     end
     end
   endmodule
```

List. 2. Verilog module `main`, instantiating `m` twice.

`m1` nor `m2` is currently computing (`r1 & ~error`), `cfg` takes the current value of the input wire `mode`. The selection of values for output data `o` and end signal `e`, along with the triggering of computations by sub-module instance `m2`, depend on `cfg`: in mode `LQ` an input data `i` is only processed by `m1`, whereas in mode `HQ` this data is serially processed by `m1` and then `m2`.

In the remainder of the paper, we consider Verilog circuits given as directed acyclic graphs whose nodes are modules, where arcs describe the "instantiates" relation, and with a single source node representing the module that describes the whole circuit. In turn, every Verilog module `M` is considered as a tuple $\texttt{M} = (I_{\texttt{M}}, O_{\texttt{M}}, R_{\texttt{M}}, Sub_{\texttt{M}})$ where: $I_{\texttt{M}}$ denotes input wires; $O_{\texttt{M}}$ denotes output wires; $R_{\texttt{M}}$ are internal or output registers; $Sub_{\texttt{M}}$ is a set of sub-module instantiations. A Verilog module with a non-empty set of sub-module instantiations is called a *super-module*.

Clock-gating in Verilog Circuit Specifications Consider a module instance `mi`. We say that $\eta_{\texttt{m}i}$ is a *Clock-inhibition Predicate* (CIP) for `m`i if, upon an edge of the clock of `m`i (*e.g.,* `clk`), $\eta_{\texttt{m}i}$ holds if the values of every registers and output wires of `m`i are strictly equivalent before and after the edge of the clock. If translated into a CGL, $\eta_{\texttt{m}i}$ can then be used to save dynamic power by gating the clock of `m`i by preventing flip-flop switches. Considering our example Verilog module `main` again, a piece

```
type state_t = Idle | Busy
node m (r: bool; i: int) returns (e: bool; o: int)
var last state: state_t = Idle;
    last cnt: int = 0; last m: int = 0;
let
  state = if last state = Idle & r then Busy else
               if last state = Busy & e & not r then Idle else
                 last state;
  cnt = if last state = Busy & e then 0 else
          if last state = Busy & not e then last cnt + 1 else
            last cnt;
  m = if last state = Idle & r or last state = Busy & e & r
        then i else last m;
  o = m;
  e = (cnt = 100);
tel
```

List. 3. Heptagon node encoding the machine of Fig. 1.

of circuit encoding the CGL for sub-module instances `m1` and `m2` would typically output two wires, say $\eta_{\mathtt{m1}}$ and $\eta_{\mathtt{m2}}$, used to filter each of their respective input clocks `clk`. The extract of Verilog code instantiating the clock-gated instance of `m1` (replacing the beginning of line 8 in List. 2) would then be "`m m1 (.clk(clk & ~`$\eta_{\mathtt{m1}}$`),`".

2.2 A Fragment of Heptagon

Heptagon (Delaval et al., 2013) is a reactive data-flow language where programs are built as parallel and hierarchical compositions of data-flow *nodes*, each having *input*, *local*, and *output flows*. The body of a node describes how input flows are transformed into output flows, in the form of a set of *equations*. These equations define the values of outputs (and possible local flows), using the current values of inputs, and the current *state* of the node: the latter is made of memorized values expressed by "`last`" values of flows. New values for input flows are given at each *execution step*, where equations are then evaluated all together, and values of output flows are updated accordingly.
We give an example Heptagon node in List. 3; this node symbolically encodes the Mealy machine given in Fig. 1 by using "`last`" flows to memorize its state.
One can compose Heptagon nodes using instantiations; *e.g.,* like "`(e1, o1) = inlined m (r1, i1);`" for `m`.

An *invariant* and *controllable flows* (each taking its value in the Boolean domain $\mathtt{bool} = \{\mathtt{false}, \mathtt{true}\}$) can be specified for Heptagon nodes using *contracts*. When it encounters a node featuring a contract, the BZR compiler involves a symbolic DCS algorithm to automatically produce a *controller* constraining the values of the controllable flows so as to guarantee that the resulting *controlled node* satisfies the invariant. The controllers produced take the form of as many predicates as controllable flows, that implement the following behavior: considering every controllable flow c in turn (according to their order of declaration), the controller tries to assign c to `true` unless this could lead to a potential violation of the desired invariant in subsequent execution steps.
Given a Boolean output `o`, a contract enforcing that `o` holds using controllable flows `c1` and `c2` for a node is declared as "`contract enforce o with (c1, c2: bool)`".

2.3 Variables & Further Notations

In Verilog terms, a set of variables V represents wires and outputs of registers; equivalently in Heptagon terms, variables in V represent flows, including state ones ("`last`" flows). $\mathcal{P}_V$ is the set of propositional predicates expressed using variables in V. Given an instantiation $\mathtt{M}i$ of a Verilog module `M` and a set of variables $V_{\mathtt{M}}$ pertained to `M`, we denote by $V_{\mathtt{M}i}$ the set of variables substituted according to the instantiation $\mathtt{M}i$. By extension, we write $V_{Sub_{\mathtt{M}}}$ to denote $\biguplus_{\mathtt{m}i \in Sub_{\mathtt{M}}} V_{\mathtt{m}i}$, where $\uplus$ is the disjoint union.

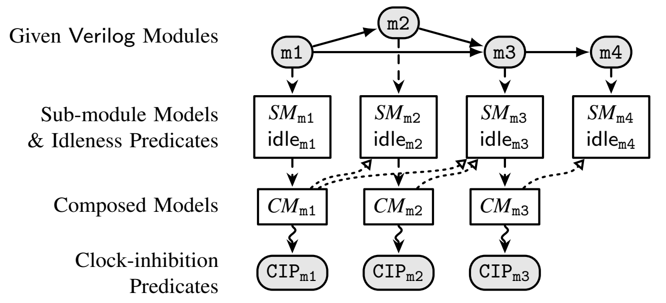


Fig. 2. Example Verilog module instantiation graph, associated models, and resulting CIPs. Arrows $\rightarrow$ (resp. $\dashrightarrow$) represents Verilog sub-module (resp. Heptagon node) instantiation relations. In turn, $\dashrightarrow$ (resp. $\rightsquigarrow$) denotes modeling (resp. symbolic DCS) steps.

3. COMPUTING CGLS USING SYMBOLIC DCS

We now describe our technique for computing CGLs. We give an overview of the modeling principles and how we eventually integrate the resulting CGLs into the original circuit. We then detail the models and results we obtain from our example.

3.1 Overview of the Modeling Technique

Our translation algorithm produces two families of models (represented using Heptagon nodes) that each fit two distinct purposes:

SM the first family of models, called *Sub-module Models*, aims at representing generic sub-modules (*i.e.,* not yet instantiated). They model their behavior using one internal state flow per marked register, and abstract away any sub-module instantiation;

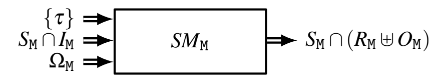
CM the second kind of models, referred to as *Composed Models*, is derived from the Sub-module Models of every Verilog super-module `M`. CMs instantiate Sub-module Models, and encode the computation of CIPs as symbolic DCS problems.

Our algorithm for computing CIPs works by visiting every Verilog module in the instantiation graph according to an inverse topological order. Every module `M` is first translated into a Sub-module Model $SM_{\mathtt{M}}$, accompanied with an *idleness predicate* $\mathsf{idle}_{\mathtt{M}}$ that expresses a condition on which the registers' values of any instance of `M` do not change. Then, every node $SM_{\mathtt{M}}$ of a super-module `M` is further transformed into a Composed Model $CM_{\mathtt{M}}$ that instantiates Sub-module models. Each node $CM_{\mathtt{M}}$ features a contract that involves control objectives (*i.e.,* at least an invariant involving idleness predicates of sub-module instances) and controllable flows that represent the CIPs of each sub-module instantiated by `M`: the enforcement of this contract by using a symbolic DCS algorithm results in correct CIPs.
We sketch this process using an example circuit specification in Fig. 2. In this example, three symbolic DCS problems are solved, leading to as many sets of CIPs. Note that $SM_{\mathtt{m1}}$ and $\mathsf{idle}_{\mathtt{m1}}$ are never instantiated: $SM_{\mathtt{m1}}$ is only used to derive $CM_{\mathtt{m1}}$. During the modeling process, one CGL-enabled Verilog module $\mathtt{m}i'$ is derived from the each original one $\mathtt{m}i$. Each $\mathtt{CIP}_{\mathtt{m}i}$ is eventually translated into Verilog code, and then integrated into $\mathtt{m}i'$ to derive a clock-gated Verilog design.

Tackling Complexity Issues Consider a Verilog module `M` with sub-module instantiations $Sub_{\mathtt{M}}$, and assume a perfect knowledge of the values of all its input wires $I_{\mathtt{M}}$, its registers $R_{\mathtt{M}}$, and the registers of all its direct and indirect sub-module instances. The optimal CIP for each of its sub-module instances $\mathtt{m}i \in Sub_{\mathtt{M}}$ is $\eta_{\mathtt{m}i}^{optim} \in \mathcal{P}_{I_{\mathtt{M}} \uplus R_{\mathtt{M}} \uplus I_{Sub_{\mathtt{M}}} \uplus R_{Sub^*_{\mathtt{M}}}}$, where $Sub^*_{\mathtt{M}}$ denotes every direct and indirect sub-module instances within `M`. In principle, one can then build the CGL computing a value for $\eta_{\mathtt{m}i}^{optim}$ at each clock cycle, and use it to inhibit the clock of $\mathtt{m}i$ within `M`. However, the size of today's circuit designs make the exact computation of optimal CIPs generally intractable. To tackle this problem, we compute under-approximations of CIPs by: (i) using a layered approach, where CIPs are computed separately for each super-module and only their direct sub-module instances are taken into account; and (ii) devising a parametrized abstraction technique.

Marking Variables To drive the abstractions, we parametrize our algorithm with a set of variables to be taken into account when modeling the circuits. This key aspect of our approach allows designers to exploit the knowledge they have on their designs. In particular, the usual distinction between command parts and operational parts of hardware circuits permits a quick identification of registers and wires that are relevant for the computation of CIPs that would otherwise be hard to compute. Referring to our example Verilog module `m` in List. 1, one can observe that the computations on the input data given using wire array `i` and output using wire array `o`, are driven by the values held in registers `state` and `cnt`, plus input wire `r`. The output wire `e` is also relevant *w.r.t.* the behaviors of any circuit instantiating `m` as it indicates the termination of its computations. Regarding module `main` of List. 2, relevant wires and registers include `r`, `mode`, `e`, `error`, `cfg`, `wait_m1`, and `wait_m2`.
Marked wires and registers shall be specified as a union S of sets $S_{\mathtt{M}}$ of variables pertained to a Verilog module `M` instantiated in the circuit. Note that our modeling algorithm is sound *w.r.t.* the set of marked variables, meaning that, although some sets S give better results than others (*e.g.,* in terms of dynamic power savings), it always produces functionally equivalent results. In the worst case ($S_{\mathtt{M}} = \varnothing$), the resulting model for `M` symbolically describes a single-state automaton, and it is only likely to result in a module that is never considered idle.
Abstracting behaviors of the Verilog modules leads to potentially non-deterministic models. Consider for instance an explicit automaton with an input wire i and two transitions from the same source and distinct destinations, respectively guarded with i and $\neg i$; abstracting away i would lead to a non-deterministic automaton. To still construct Heptagon nodes (that are deterministic by definition), we automatically generate *oracles* to replace sub-expressions whose values are abstracted away, thereby explicitly modeling the non-determinism.

Introducing Oracles Given an expression e on any set of variables, the *oracle* ω_e is an interpreted input that proxies e. In particular, ω_e takes its values in the domain of e (*e.g.,* the Booleans if e is a predicate), and can thus be used to model behaviors where e itself is abstracted away and can non-deterministically take any value in its domain. Every knowledge about the modeled behaviors is not lost however. Indeed, assuming that e and e' admit the same canonical representation e'', every occurrence of e and e' can be replaced with the same oracle $\omega_{e''}$, and the equality of valuations for e and e' can still be represented. For instance, given the expression $x+y$, where x and y are Integer variables, the oracle ω_{x+y} can non-deterministically take any Integer value: the addition operation, x, and y are abstracted away, replaced by some undetermined Integer. Additionally, if $y+1+x-1$ admits the same canonical representation as $x+y$, then it can also be modeled with ω_{x+y}.



$\{\tau\}$
$S_{\mathtt{M}} \cap I_{\mathtt{M}}$
$\Omega_{\mathtt{M}}$
$SM_{\mathtt{M}}$
$S_{\mathtt{M}} \cap (R_{\mathtt{M}} \uplus O_{\mathtt{M}})$

Fig. 3. Interface of a Sub-module Model $SM_{\mathtt{M}}$.

When constructing $SM_{\mathtt{M}}$ or $CM_{\mathtt{M}}$ from a Verilog module `M`, we introduce a set of oracles $\Omega_{\mathtt{M}}$ to handle expressions within `M` that involve non-marked wires and registers (not belonging to $S_{\mathtt{M}}$): *i.e.,* the actual values of these expressions are abstracted away in the resulting models. Yet, our goal is to actually generate circuits that encode CGLs, and that can thus be used to inhibit the clock of instances of `M`: the actual values of marked registers and abstracted expressions computed within `M` are hence required when translating the resulting CIPs into Verilog code. As a result, while oracles $\Omega_{\mathtt{M}}$ are *inputs* to the models of `M`, we also build a *CGL-enabled* version `M'` that features one additional *output wire* per oracle in $\Omega_{\mathtt{M}}$ that does not represent expressions only involving inputs of `M`; these additional wires carry the actual values of the corresponding expressions, and are thus used to feed the CGL of super-modules. Additional output wires of `M'` also carry the value of marked registers belonging to $S_{\mathtt{M}}$.

Bottom-up Clock-inhibition Allowance Of course, the clock of any instance of `M` also drives sub-module instances $\mathtt{m}i \in Sub_{\mathtt{M}}$. As a result the clock of `M` should not be inhibited whenever any sub-module instance $\mathtt{m}i$ must not be inhibited. However, $SM_{\mathtt{M}}$ does not model the behavior of any of its sub-module instances. Hence, we choose to add a *bottom-up clock-inhibition allowance* output wire $\mathsf{allow}\eta_{\mathtt{M}}$ to `M'`, built as the conjunction of every CIPs of sub-modules instantiated by `M`, or `1` if there is none.

Resulting CGLs Eventually, the CGL to be integrated within a Verilog super-module `M` consists of one CIP $\tilde{\eta}_{\mathtt{m}i} \in \mathcal{P}_{S_{\mathtt{M}} \uplus \Omega_{\mathtt{M}} \uplus S_{Sub_{\mathtt{M}}} \uplus \Omega_{Sub_{\mathtt{M}}} \uplus \biguplus_{\mathtt{m}i \in Sub_{\mathtt{M}}} \mathsf{allow}\eta_{\mathtt{m}i}}$ per sub-module instance $\mathtt{m}i \in Sub_{\mathtt{M}}$. $\tilde{\eta}_{\mathtt{m}i}$ under-approximates the condition upon which the clock of sub-module instance $\mathtt{m}i$ can be inhibited: *i.e.,* it is such that $\tilde{\eta}'_{\mathtt{m}i} \Rightarrow \eta_{\mathtt{m}i}^{optim}$, $\tilde{\eta}'_{\mathtt{m}i}$ being $\tilde{\eta}_{\mathtt{m}i}$ where every oracle ω_e is substituted by e. We rely on a symbolic DCS algorithm to compute such CIPs.

3.2 Building Sub-module Models & Idleness Predicates

We outline in Fig. 3 the interface of a Sub-module Model $SM_{\mathtt{M}}$ for a Verilog module `M`. Its inputs include (i) an enable flow τ; (ii) flows mirroring marked input wires selected for this module ($S_{\mathtt{M}} \cap I_{\mathtt{M}}$); and (iii) a set of input oracles $\Omega_{\mathtt{M}}$ that are used to model undetermined behaviors of instances of `M`. The outputs of $SM_{\mathtt{M}}$ comprise flows mirroring the marked registers and output wires selected for `M` ($S_{\mathtt{M}} \cap (R_{\mathtt{M}} \uplus O_{\mathtt{M}})$).
We further associate each Sub-module Model $SM_{\mathtt{M}}$ with an idleness predicate $\mathsf{idle}_{\mathtt{M}} \in \mathcal{P}_{S_{\mathtt{M}} \uplus \Omega_{\mathtt{M}}}$, that under-approximates the condition on which the registers' values of any instance of `M` do not change; *i.e.,* given values for marked input wires, marked internal registers, and for the oracles, if $\mathsf{idle}_{\mathtt{M}}$ holds then any assignment to any (both marked and non-marked) internal and output registers of `M` would not change the value it memorizes.
One can easily build the Heptagon node $SM_{\mathtt{M}}$ and the associated condition $\mathsf{idle}_{\mathtt{M}}$ from a Verilog module `M` where every internal wire is substituted with the expression it is assigned to. A traversal of clock-triggered assignments to marked registers clocked

```
 1 type state_t = Idle | Busy
   node SM_m (τ, r, ω_{cnt==100}: bool)
   returns (last state: state_t = Idle; e: bool)
   let
 5   state = if not τ then last state else
               if last state = Idle & r then Busy else
                 if last state = Busy & ω_{cnt==100} & not r then Idle
                 else last state;
     e = ω_{cnt==100};
10 tel
```

List. 4. Heptagon node $SM_{\mathtt{m}}$, obtained from the Verilog module m of List. 1 using marked variables $S_{\mathtt{m}} = \{\mathtt{state}, \mathtt{r}, \mathtt{e}\}$.

```
module m’ (input clk, input r, input [7:0] i, output [7:0] o,
           output e, // additional outputs:
           output state’, output ω_{cnt==100}, output allowη_m);
  ...
  assign state’ = state;
  assign ω_{cnt==100} = (cnt == 100);
  assign allowη_m = 1; // no clock-gated sub-module instance
endmodule
```

List. 5. Excerpts of the CGL-enabled Verilog module m’ obtained from the module m of List. 1.

```
type cfg_t = LQ | HQ
node CM_main (ω_r, ω_{m1_cnt==100}, ω_{m2_cnt==100}: bool; ω_mode: cfg_t)
returns (last wait_m1: bool = false; last wait_m2: bool = false;
         last cfg: cfg_t = LQ; φ_M: bool)
contract enforce φ_M with (η̃_m1, η̃_m2: bool)
var m1_state, m2_state: m_state_t; e1, e2: bool;
let
  (m1_state, e1) = inlined SM_m (not η̃_m1, ω_r & not last wait_m1 &
        not last wait_m2, ω_{m1_cnt==100});
  (m2_state, e2) = inlined SM_m (not η̃_m2, e1 & not last wait_m2 &
        last cfg = HQ, ω_{m2_cnt==100});
  cfg = if ω_r & not last wait_m1 & not last wait_m2
        then ω_mode else last cfg;
  wait_m1 = if ω_r & not last wait_m1 & not last wait_m2
            then true else if e1 then false else last wait_m1;
  wait_m2 = if last cfg = HQ & e1 & not last wait_m2
            then true else if e2 then false else last wait_m2;
  φ_M = (η̃_m1 => (m1_state = Idle &
                  not (ω_r & not last wait_m1 & not last wait_m2)))
     & (η̃_m2 => (m2_state = Idle &
                  not (e1 & not last wait_m2 & last cfg = HQ)));
tel
```

List. 6. Heptagon node $CM_{\mathtt{main}}$ obtained from the Verilog module main of List. 2 using marked variables $S_{\mathtt{main}} = \{\mathtt{cfg}, \mathtt{wait\_m1}, \mathtt{wait\_m2}\}$ and $S_{\mathtt{m}}$ as used in List. 4.

using clk allows the construction of cascading conditional statements for the assignments to flows encoding the state within $SM_{\mathtt{M}}$ ("last" flows). A similar traversal can be used to construct $\mathsf{idle}_{\mathtt{M}}$ as the conjunction of the negation of every guard leading to the assignment of a register. An efficient introduction of oracles can be performed by building a canonical representation of every expression using (basic and/or multi-terminal) binary decision diagrams for instance. Then, every canonical expression e that involves a non-marked variable becomes an oracles $\omega_e \in \Omega_{\mathtt{M}}$.
Note that the constructions above do not necessarily traverse every expression, register or wire of a module declaration, hence only a limited number of oracles might be required even for large modules. This claim is supported by the application of our technique on a realistic case study detailed in Section 4.
When applied to the Verilog module m of List. 1 with marked variables $S_{\mathtt{m}} = \{\mathtt{state}, \mathtt{r}, \mathtt{e}\}$, our construction technique for Sub-module Models builds the Heptagon node of List. 4. The assignment to e on line 9 corresponds to the assignment on line 8 in List. 1: the value of "cnt == 100" is abstracted away using an oracle as cnt does not belong to $S_{\mathtt{m}}$. In turn, the predicate that describes the idleness condition of m is $\mathsf{idle}_{\mathtt{m}} = (\mathtt{state} = \mathtt{Idle} \ \& \ \mathtt{not}\ \mathtt{r})$. Finally, we show in List. 5 the additions to module m that are necessary to construct the corresponding CGL-enabled Verilog module m’. m’ features three additional output wires (one for as many oracles in $\Omega_{\mathtt{m}}$, one per variable in $S_{\mathtt{m}} \cap R_{\mathtt{m}}$, plus $\mathsf{allow}\eta_{\mathtt{m}}$). The bottom-up clock-inhibition allowance output $\mathsf{allow}\eta_{\mathtt{m}}$ is assigned to 1 as no sub-module instantiation exists within m to prevent the clock of m from being inhibited.

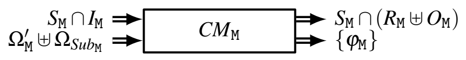

$S_{\mathtt{M}} \cap I_{\mathtt{M}}$
$\Omega'_{\mathtt{M}} \uplus \Omega_{Sub_{\mathtt{M}}}$
$CM_{\mathtt{M}}$
$S_{\mathtt{M}} \cap (R_{\mathtt{M}} \uplus O_{\mathtt{M}})$
$\{\varphi_{\mathtt{M}}\}$

Fig. 4. Interface of a Composed Model $CM_{\mathtt{M}}$.

3.3 Building Composed Module Models

A Composed Model $CM_{\mathtt{M}}$ is derived from $SM_{\mathtt{M}}$ by taking sub-module instantiations mi into account, and formulating the computation of $\tilde{\eta}_{\mathtt{m}i}$'s as a symbolic DCS problem.
Basically, the instantiation of a sub-module m by M translates within $CM_{\mathtt{M}}$ into the instantiation (say, $SM_{\mathtt{m}i}$) of the Heptagon node $SM_{\mathtt{m}}$. The input τ of each Heptagon node instantiation $SM_{\mathtt{m}i}$ is assigned to the negation of the corresponding CIP "not $\tilde{\eta}_{\mathtt{m}i}$". CIPs, in turn, are the controllable flows as they represent the CGL outputs. (The input flow τ modeling the clocks in Sub-module Models is no longer required, and can be substituted with true everywhere else in $CM_{\mathtt{M}}$.) Further, we build $\Omega_{\mathtt{m}i}$ and $\mathsf{idle}_{\mathtt{m}i}$ according to the appropriate renaming in $\Omega_{\mathtt{m}}$ and substitutions in $\mathsf{idle}_{\mathtt{m}}$. Note that, with the additional output flows from Sub-module Models, some oracles in $\Omega_{\mathtt{M}}$ may represent expressions that are now fully determined. A substitution of such oracles with their respective expressions is thus necessary in $CM_{\mathtt{M}}$ so that marked outputs of sub-modules are taken into account. Let $\Omega'_{\mathtt{M}}$ be $\Omega_{\mathtt{M}}$ pruned from the latter oracles. At last, the invariant to enforce by control $\varphi_{\mathtt{M}}$ states that a CIP $\tilde{\eta}_{\mathtt{m}i}$ for a sub-module instance mi should not hold unless $\mathsf{idle}_{\mathtt{m}i}$ holds:

$$\varphi_{\mathtt{M}} = \bigwedge\nolimits_{\mathtt{m}i \in Sub_{\mathtt{M}}} (\tilde{\eta}_{\mathtt{m}i} \Rightarrow \mathsf{idle}_{\mathtt{m}i}).$$

We sketch the interface of a Heptagon node $CM_{\mathtt{M}}$ in Fig. 4; note that it also admits as inputs the oracles of every instantiated sub-module. We give in List. 6 the result we obtain for $CM_{\mathtt{main}}$.

3.4 Computing & Integrating the CGLs

As stated in Section 2.2, the compilation of a Heptagon node that features a contract (as Composed Models do), involves a symbolic DCS computation step that produces a controller made of one predicate per controllable flow (*i.e.,* CIPs). By virtue of the semantics assigned to such flows by the Heptagon compiler (*i.e.,* assigning them to true whenever possible), one can eventually translate the controller into some Verilog code encoding a CGL that inhibits the clock of sub-module instances whenever possible. We show in List. 7 excerpts of the end result that we obtain for our running example. The assignments to registers[1] holding $\tilde{\eta}_{\mathtt{m1}}$ and $\tilde{\eta}_{\mathtt{m2}}$ are clocked using clk: their respective input value consists in the conjunction between their respective bottom-up clock-inhibition allowance ($\mathsf{allow}\eta_{\mathtt{m1}}$ and $\mathsf{allow}\eta_{\mathtt{m2}}$), and their respective CIPs as computed by using symbolic DCS. The clocks of sub-module instances m1 and m2 are now filtered according to $\tilde{\eta}_{\mathtt{m1}}$ and $\tilde{\eta}_{\mathtt{m2}}$. As a side note, remark that $\omega_{\mathtt{e1}}$ and $\omega_{\mathtt{e2}}$ are output wires of main’ since these outputs of sub-module instances are required to construct $SM_{\mathtt{main}}$ and $\mathsf{idle}_{\mathtt{main}}$. However, $\omega_{\mathtt{r}}$ and $\omega_{\mathtt{mode}}$ are not part of these outputs as they represent input wires only.

[1] Although using wires for CIPs would seem sufficient from a functional point of view, registers are required to avoid glitches (Benini et al., 1999).

```
module main' (input clk, input r, input mode, input [7:0] i,
              output [7:0] o, output e, output error,
              // additional outputs:
              output wait_m1', output wait_m2', output cfg',
              output ω_e1, output ω_e2, output allowη_main);
  reg  η̃_m1, η̃_m2;
  always @(posedge clk) begin
     η̃_m1 <= allowη_m1 &
      (m1_state == idle) & (wait_m1 | wait_m2 | ~r);
     η̃_m2 <= allowη_m2 &
      (m2_state == idle) & ((cfg == LQ) | wait_m2 | ~ω_m1_cnt==100);
  end
  wire allowη_m1, m1_state, ω_m1_cnt==100;
  m' m1 (.clk(clk & ~η̃_m1), .allowη_m(allowη_m1), ...
         .state'(m1_state), .ω_cnt==100(ω_m1_cnt==100));
  wire allowη_m2, m2_state, ω_m2_cnt==100;
  m' m2 (.clk(clk & ~η̃_m2), .allowη_m(allowη_m2), ...
         .state'(m2_state), .ω_cnt==100(ω_m2_cnt==100));
  ...
  // assignments to outputs to be CGL-enabled:
  assign wait_m1' = wait_m1, wait_m2' = wait_m2;
  assign cfg' = cfg; assign ω_e1 = e1, ω_e2 = e2;
  assign allowη_main = η̃_m1 & η̃_m2; // <- clock-inhibition allowance
endmodule
```

List. 7. Excerpts of resulting Verilog module obtained from module `main` of List. 2.

	Original	CGL-enabled	Saving (%)
Cyclone IV (100MHz)	69.98	58.55	16.33
Stratix III (100MHz)	86.99	74.16	14.75
HardCopy IV (100MHz)	15.48	13.84	10.59
Cyclone IV (1GHz)	699.80	585.48	16.34
Stratix III (1GHz)	869.94	741.60	14.75
HardCopy IV (1GHz)	154.85	138.46	10.58

Table 1. Estimated mean power dissipation (in mW) of original and resulting RS decoders.

4. APPLICATION ON A CASE STUDY

To experimentally validate our approach, we have manually applied our clock-gating synthesis technique on a real hardware component. To this end, we chose to use a Reed-Solomon (RS) decoder[2]. RS codes are a group of error-correcting codes, that have wide range of applications in digital communications and storage (Reed and Solomon, 1960). Basically, this decoder takes coded words of 204 bytes as inputs, and outputs decoded words of 188 bytes. The original decoder is made of 23 modules that build up a circuit with around 52,000 gates and 3,000 registers (flip-flops). Among them, 6 modules drive the operations to be performed on the data: they feature two easily identifiable families of wires and registers that we marked for our modeling: (i) register arrays named `state` or `step`, that take their values into discrete domains made of a few constants (similarly to `state` in List. 1); these registers are typically used to encode some command automaton that drives operations on data; and (ii) input and output wires named `*_ready` or `*_done`, that signal end of computations. We then produced a "CGL-enabled" circuit including CIPs produced using symbolic DCS.

To experimentally assess the functional correctness and compare the respective dynamic power dissipation of each of the designs at hand (original and CGL-enabled), we first performed logic synthesis on both of them using the Altera Quartus synthesizer. We then used the Altera ModelSim simulation tool to perform functional simulations using the same benchmark (provided with the source code of the original decoder) for the two circuits, and checked that the resulting traces were strictly equivalent. To assess actual dynamic power savings, we have carried out estimations of mean power dissipation on simulations of the benchmarks, for various target technologies and main clock frequencies as these factors have a great impact on dynamic power. The Altera PowerPlay Power Analyzer tool offers several pre-configured target technologies, among which we chose the Cyclone IV (dedicated to low-power FPGA designs), the Stratix III (for high-performance FPGA designs), and the HardCopy IV (ASICs) families. We also setup the main clock frequencies to be either 100Mz or 1GHz.

We show the resulting estimations of power dissipation and respective dynamic power savings in Table 1. We consider that these results are promising when put in perspective with the relative simplicity of our approach. Indeed, we generated effective CIPs by imposing invariants only, as our technique do not even incorporate control techniques towards any sort of optimization yet.

[2] `https://opencores.org/project,reed_solomon_decoder`.

5. RELATED WORKS

Low-power Chip Design Several families of design methods permit the (semi-)automated use of power-saving technologies: they can be integrated into high-level (*aka* system level) or RTL descriptions, or further down the implementation process, during "synthesis" (*i.e.,* translation of RTL descriptions into a network of gates and wires), or placement and routing steps. Nonetheless, Dale (2008) found that considering higher levels of abstraction generally leads to more power savings. Designers most commonly rely on the RTL code itself to implement clock-gating, yet a few approaches automatically generate RTL code with integrated clock gating form higher-level descriptions. Among them, Agarwal and Dimopoulos (2008) developed an environment for high-level design with their own procedural language. Ahuja et al. (2010) also provide a solution to design circuits directly using the C language. In these approaches, designers are responsible for the selection of gated components. One can distinguish three classes of RTL clock-gating algorithms based on the hierarchical level at which they consider the circuit: combinatorial or sequential ones focus on individual registers (Sudhakar et al., 2015; Liu et al., 2015), system- or module-level (Bhutada and Manoli, 2007) focus on clock-gating whole modules or blocks of clock-triggered assignments. We rely on the latter level for the layered abstractions that it allows. Several commercial and academic tools already target automated clock-gating from RTL code. Raghavan et al. (1999) developed an algorithm that automatically insterts CGL into RTL descriptions of circuits. They focus on the exact computation of idlness conditions for individual registers within a single module, hence their solution suffers from scalability issues. To partially overcome these issues, Babighian et al. (2005) suggest an algorithm that automatically tries to approximate idleness conditions. Later, Chang et al. (2007) used a control-based adaptive clock-gating algorithm to shut down IP cores based on given explicit finite-state models. In an approach that targets the conditional activation of individual hardware components using their "enable" signal (an approach similar to clock-gating), Benini et al. (1994) tried to detect idleness conditions by using explicit finite-state machines. At last, Raghavan et al. (1999) exploited conditional statements and case structures within blocks of clock-triggered assignments in HDL languages to determine such conditions. Our approach draws from the latter ones in the sense that we also operate at the HDL level, and build symbolic finite-state machines from conditional clock-triggered assignments. We additionally bring layered and semi-automated abstractions for the sake of scalability.

Applying DCS for Low-power Hardware Design Few hardware design techniques rely on DCS for saving power. Quadri et al. (2010) present a high-level design flow for reconfigurable FPGA-based System-on-Chip (SoC); they model potential reconfiguration behaviors and manually derive a "controller" that automatically takes reconfiguration decisions. Later, Guillet et al. (2012) and An et al. (2013, 2016) were among the firsts to apply DCS algorithms for reconfiguration management in SoC design. Doing so, they could automate the generation of controllers, thereby exploiting the formal correctness and guarantees that DCS techniques provide. In particular, An et al. (2013, 2016) model the applications' behaviors and the needed resources (area in hardware—*i.e.,* regions of the FPGA) using explicit automata; they then use a symbolic DCS algorithm to automatically compute a reconfiguration manager for the system.

6. CONCLUSIONS & FUTURE WORKS

In this paper, we have described a systematic approach for computing the CGL of synchronous circuits described using the Verilog hardware description language. This approach exercises symbolic DCS algorithms by means of a semi-automated modeling in Heptagon of each individual Verilog modules. We have demonstrated its principles using an example, and have reported on its manual application on a realistic case study.

The next steps involve a formalization of our modeling algorithm to validate its correctness, along with the development of an implementation in a tool. Our approach can also be extended to compute CGLs for individual registers within modules. Although we exercised our technique to implement clock-gating as it currently offers the best trade-off between extra occupied circuit area and power savings (Kathuria et al., 2011), it is also applicable to other low-power design mechanisms such as power-gating (especially for computation-intensive modules that can be shut down for long periods of time). Also, the abstractions induced by our modeling approach make it a good candidate for constructing models suitable for the application of control algorithms that do not scale up to exact whole-circuit models. In particular, the adaptation of our algorithm for the computation of "suspendability" predicates would allow to suspend the computations of sub-module instances by control. Combined with the recent advances in control algorithms for symbolic infinite-state systems with applications to quantitative models (Berthier and Marchand, 2014; Berthier et al., 2015), our framework could permit the application of optimal control techniques towards the minimization of peak dynamic power or energy dissipation over several clock cycles. Similarly, incorporating stochastic models (*e.g.,* inferred from simulation traces) would provide interesting cases for developing new optimal control algorithms targeting such goals. Also, automatically identifying "good" sets of marked variables constitutes an interesting challenge. At last, the support of black-box sub-modules with simple user-provided models can also be considered.

REFERENCES

Agarwal, N. and Dimopoulos, N. (2008). High-level FSMD design and automated clock gating with CoDeL. *Canadian Journal of Electrical and Computer Engineering*, 33(1).

Ahuja, S., Zhang, W., Lakshminarayana, A., and Shukla, S.K. (2010). A methodology for power aware high-level synthesis of co-processors from software algorithms. In *Proceedings of the 23rd International Conference on VLSI Design*, VLSID '10, 282–287. IEEE.

An, X., Rutten, É., Diguet, J.P., and Gamatié, A. (2016). Model-based design of correct controllers for dynamically reconfigurable architectures. *ACM Trans. Emb. Comp. Syst.*

An, X., Rutten, É., Diguet, J., Le Griguer, N., and Gamatié, A. (2013). Discrete Control for Reconfigurable FPGA-based Embedded Systems. In *4th IFAC Workshop on Dependable Control of Discrete Systems*, DCDS '13.

Babighian, P., Benini, L., and Macii, E. (2005). A scalable algorithm for RTL insertion of gated clocks based on ODCs computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1), 29–42.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition.

Benini, L., De Micheli, G., Macii, E., Poncino, M., and Scarsi, R. (1999). Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers. *ACM Trans. Des. Autom. Electron. Syst.*, 4(4), 351–375.

Benini, L., Siegel, P., and De Micheli, G. (1994). Saving power by synthesizing gated clocks for sequential circuits. *IEEE Design & Test of Computers*, 11(4), 32–41.

Berthier, N., An, X., and Marchand, H. (2015). Towards Applying Logico-numerical Control to Dynamically Partially Reconfigurable Architectures. In *5th Int. Workshop on Dependable Control of Discrete Systems*, DCDS '15, 132–138. IFAC.

Berthier, N. and Marchand, H. (2014). Discrete Controller Synthesis for Infinite State Systems with ReaX. In *12th Int. Workshop on Discrete Event Systems*, WODES '14, 46–53. IFAC.

Bhutada, R. and Manoli, Y. (2007). Complex clock gating with integrated clock gating logic cell. In *Design & Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on*, 164–169. IEEE.

Cassandras, C. and Lafortune, S. (2007). *Introduction to Discrete Event Systems*. Springer.

Chang, X., Zhang, M., Zhang, G., Zhang, Z., and Wang, J. (2007). Adaptive clock gating technique for low power IP core in SoC design. In *IEEE International Symposium on Circuits and Systems*, ISCAS '07, 2120–2123.

Dale, M. (2008). Utilizing clock-gating efficiency to reduce power. *EE Times, India*.

Delaval, G., Rutten, É., and Marchand, H. (2013). Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler. *Discrete Event Dynamic Systems*, 23(4), 385–418.

Dumitrescu, E., Girault, A., Marchand, H., and Rutten, É. (2010). Multicriteria optimal discrete controller synthesis for fault-tolerant tasks. In *10th Int. Workshop on Discrete Event Systems*, WODES '10, 356–363. IFAC.

Guillet, S., de Lamotte, F., Le Griguer, N., Rutten, É., Gogniat, G., and Diguet, J.P. (2012). Designing formal reconfiguration control using UML/MARTE. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 1–8. IEEE.

Hariharan, K. and Jaya Kumar, C. (2012). Clock gating for low power circuit design by merge and split methods. *IOSR Journal of Engineering*, 2(4), 1–5.

Kathuria, J., Ayoubkhan, M., and Noor, A. (2011). A review of clock gating techniques. *MIT International Journal of Electronics and Communication Engineering*, 1(2), 106–114.

Liu, J., Hong, M.S., Do, K., Choi, J.Y., Park, J., Kumar, M., Kumar, M., Tripathi, N., and Ranjan, A. (2015). Clock domain crossing aware sequential clock gating. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, 1–6. EDA Consortium.

Marchand, H., Bournai, P., Le Borgne, M., and Le Guernic, P. (2000). Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4), 325–346.

Marchand, H. and Le Borgne, M. (1998). On the optimal control of polynomial dynamical systems over Z/pZ. In *4th International Workshop on Discrete Event Systems*, 385–390.

Marchand, H. and Samaan, M. (2000). Incremental Design of a Power Transformer Station Controller Using a Controller Synthesis Methodology. *IEEE Trans. Softw. Eng.*, 26, 729–741.

Quadri, I.R., Yu, H., Gamatié, A., Meftali, S., Dekeyser, J.L., and Rutten, É. (2010). Targeting Reconfigurable FPGA-based SoCs using the MARTE UML profile: from high abstraction levels to code generation. *International Journal of Embedded Systems*.

Raghavan, N., Akella, V., and Bakshi, S. (1999). Automatic insertion of gated clocks at register transfer level. In *Proceedings of the 12th International Conference on VLSI Design*, VLSID '99, 48–54. IEEE.

Ramadge, P. and Wonham, W. (1989). The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1), 81–98.

Reed, I.S. and Solomon, G. (1960). Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2), 300–304.

Sudhakar, J., Prasad, A.M., and Panda, A.K. (2015). GFCG: Glitch free combinational clock gating approach in nanometer VLSI circuits. In *2nd International Conference on Electronics and Communication Systems (ICECS)*, 146–150. IEEE.