Non-cycle-accurate Sequential Equivalence Checking

Pankaj Chauhan, Deepak Goyal, Gagan Hasteer, Anmol Mathur, Nikhil Sharma
Calypto Design Systems, Inc., Santa Clara, CA 95054.
{pchauhan, dgoyal, ghasteer, amathur, nsharma}@calypto.com

ABSTRACT

We present a novel technique for Sequential Equivalence Checking (SEC) between non-cycle-accurate designs. The problem is routinely encountered in verifying the correctness of a system-level model versus an RTL design which has been derived from the former either manually or through high-level synthesis. The existing state-of-the-art in formal verification/SEC does not provide an efficient mechanism to perform such an equivalence check. Our technique reduces the SEC problem to a cycle-accurate equivalence-checking problem by constructing a pair of normalized cycle-accurate designs from the original designs, on which standard equivalence-checking techniques can then be deployed. We report the results of deploying our techniques on several industrial examples.

Categories and Subject Descriptors

B.2.2 [**Performance Analysis and Design Aids**]: Verification; B.5.2, B.6.3, B.7.2 [**Design Aids**]: Verification

General Terms

Verification, Algorithms

Keywords

Sequential Equivalence Checking, Model Checking, Formal Verification, Unit Product Machine, High Level Synthesis

1. INTRODUCTION

The problem of Sequential Equivalence Checking (SEC) [10, 18, 20] between a pair of designs with different cycle behavior is of growing importance because of the increased popularity of High Level Synthesis (HLS, also known as Behavioral synthesis) tools which can transform untimed or partially-timed high-level specifications into timed RTL models [14]. For example, an HLS tool may take as input, a partially-timed hardware model of an image processor specified in SystemC, which, takes in an array of 256 inputs, processes them, and produces an array of 256 outputs, all in a single time step. The output RTL produced by HLS may schedule the inputs to be read serially over a period of 256 clock cycles, may process the computation on these inputs over a period of 512 cycles and produce the outputs serially over another period of 256 cycles. Furthermore, the RTL model may be able to start reading the next set of 256 inputs as soon as the previous computation is finished, *i.e.* in parallel with the outputs of the previous computation. Thus, the first set of inputs would be read during cycles 0 to 255, the second set of inputs during cycles 768 to 1023, and so on. The first set of outputs would be produced during cycles 768 to 1023, the next set of outputs during cycles 1536 to 1791, and so on. In steady state, the original design completes a computation and produces 256 new outputs every cycle, whereas the synthesized RTL model does the same every 768 cycles. We refer to such pairs of designs as non-cycle-accurate designs. In contrast, cycle-accurate designs have inputs and outputs matching every cycle.

In this paper, we present the concept of input, output and state maps with latency which can be used to precisely formulate an equivalence checking problem between non-cycle-accurate designs. We present an algorithm for doing SEC between such designs by transforming them into a product machine of a cycle-accurate pair of designs (called the normalized product machine) such that the transformed designs are equivalent *iff* the original designs are equivalent. We also give an algorithm for converting any counterexample on the normalized product machine into a counterexample on the original pair of designs. Thus, the problem of SEC between non-cycle-accurate designs is reduced to the problem of model checking [5] on the normalized product machine.

Our method is not just relevant in the context of HLS but also in the context of cycle-accurate RTL-RTL verification. If the RTL designs perform a full computation (called a transaction) over a period of a fixed number of cycles, it may be easier to verify the one-transaction normalized machines against each other. This is because the process of creating the normalized machine can sometimes eliminate the entire state machine that schedules the computation and may reduce a sequential verification problem to a combinational verification problem.

2. DEFINING NON-CYCLE-ACCURATE SEQUENTIAL EQUIVALENCE

In this section we define non-cycle-accurate sequential equivalence, and some terminology used in the rest of the paper.

DEFINITION 1. *A* sequential machine *is a tuple* $M = (I, S, R, Y, T)$, *where* $I = \{i_0, i_1, \ldots, i_{n-1}\}$ *is the set of* inputs, $S = \{s_0, s_1, \ldots, s_{m-1}\}$ *is the set of* state variables (flops), $R = \{r_0, r_1, \ldots, r_{m-1}\}$ *is the set of* reset values *for the state variables,* $Y = \{y_0, y_1, \ldots, y_{k-1}\}$ *is the set of* outputs, *and* $T = \{t_0, t_1, \ldots, t_{m-1}\}$ *is the set of* transition functions. *A transition function* $t_i(I, S)$ *de-*

DAC'09, July 26-31, 2009, San Francisco, California, USA

scribes the next state value of the state variable s_i. Each output is a function of the present state and the inputs. We use y_i to denote both the i^{th} output variable, as well as the function that computes that output.

The reset value for a state variable may either be a constant or unspecified (symbolic). A state is an initial state, *iff* the values of the state variables are consistent with the partially specified reset values.

The concept of *period* of a sequential machine is central to our notion of equivalence. This denotes the period in terms of number of cycles, or clock-ticks, after which the equivalence assumptions and checks are repeated. The period naturally corresponds to the notion of the length of a *transaction*, in which a finite state system finishes a computation. For the image processor example mentioned earlier, 768 is the period.The algorithm we describe in the next section accelerates a pair of sequential machines by their period, to get a unit-period product sequential machine. In this paper, we only consider sequential machines with a fixed period. Note that in many cases, SEC between two variable period designs can be cast into a fixed period SEC problem.

Fig. 1 shows two examples of sequential machines, one with period 1, and the other with period 4, both of which compute the sum of 4 inputs. The output of the parallel and serial machines are comparable at cycles $0, 1, 2, \ldots$ and $3, 7, 11, \ldots$ respectively which correspond to transactions $1, 2, 3 \ldots$ for each machine. Note that we always start counting cycles from 0 and transactions are counted starting from 1.

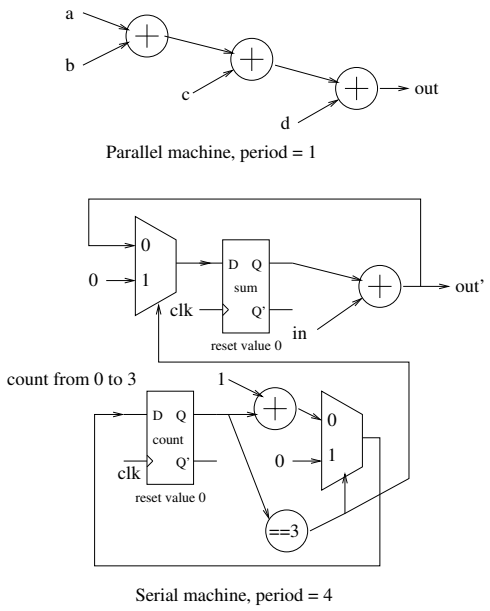

Figure 1: Two example sequential machines, one sums up four inputs in a single cycle, while the other takes four cycles.

Our model of a sequential machine is similar to a Mealy machine, except that it does not have a generalized transition relation, but a set of transition functions for state variables. Most synchronous digital hardware system descriptions fit this model.

We consider equivalence between two sequential machines, M_S and M_I (*specification* and *implementation*), along with their periods P_S and P_I. The variables corresponding to the two machines have the subscripts S or I, e.g., i_S and y_I. We use the $@t$ notation to indicate the cycle (or clock-tick), starting with cycle 0, e.g., $y_S@3$ denotes the output y_S at cycle 3. The $@t$ notation is also used in the specification of mapping (input/output/state) relations, described next.

Our system allows the specification of a variety of input, output, and state maps which together capture the assumptions to be made and the equalities to be proven. The maps are meant to be repeatable every *period* number of cycles. In our system, every equality (assumed or derived) is effected by a process called *mitering*, in which one signal in the equality (miter winner) is used to drive the fan out of the other signal (miter loser). The term *speculative reduction* [2] has also been used to refer to mitering when applied to assumptions.

Input Maps: Input maps describe assumptions to be made about a pair of inputs being equal at certain times. An input map is a tuple $(i_S@l_1, i_I@l_2)$. Here, the specification input i_S at cycles $l_1, l_1 + P_S, l_1 + 2 \cdot P_S, \ldots$ is to be assumed equal to the implementation input i_I at cycles $l_2, l_2 + P_I, l_2 + 2 \cdot P_I, \ldots$. Typically, $0 \leq l_1 < P_S$ and $0 \leq l_2 < P_I$. We also use the term *latency* for the cycle offsets l_1 and l_2. After all the input assumptions are specified, timed inputs that are not constrained by any assumption are assumed to be independent free symbolic variables. For the example in Fig. 1, $(a@0, in@0)$, $(b@0, in@1)$, $(c@0, in@2)$, and $(d@0, in@3)$ are reasonable input maps.

Our system also allows for the specification of constants, partial constants, ranges of constants, symbolic constants, etc. for input assumptions, but since these are not central to the main ideas in this paper, we do not discuss them further. Moreover, any such constraints can be easily modeled by adding suitable functionality to the designs themselves.

Output Maps: Output maps describe proof obligations about a pair of outputs being equal at certain times. An output map is a tuple $(y_S@l_1, y_I@l_2)$. Here, the specification output y_S at cycles $l_1, l_1 + P_S, l_1 + 2 \cdot P_S, \ldots$ is to be proven equal to the implementation output y_I at cycles $l_2, l_2 + P_I, l_2 + 2 \cdot P_I, \ldots$. For the running example, $(out@0, out'@3)$ is a reasonable output map. Note that the latencies l_1 and l_2 may be different, and in some cases, much larger, than the periods. E.g., a deep pipeline may take a long time to fill up, but once it does, an output is produced every cycle.

State Maps (Flop Maps): Maps between state variables of the designs help reduce the complexity of verification by allowing a divide and conquer approach. A state map takes the form of a tuple $(s_S@l_1, s_I@l_2)$. A state map generates an *assume-guarantee* pair. The *assume* part of the state map corresponds to the assumption that $s_S@(l_1 + (j-1) \cdot P_S)$ is equal to $s_I@(l_2 + (j-1) \cdot P_I)$ for transaction j. The *guarantee* part of the state map corresponds to the proof obligation that $s_S@(l_1 + j \cdot P_S)$ and $s_I@(l_2 + j \cdot P_I)$ are equal for transaction j. The state maps also introduce a one-time initial check $s_S@(l_1) = s_I@(l_2)$ starting from the reset sets of states R_S and R_I. This *assume-guarantee* setup ensures the soundness of state maps by ensuring that the *assume* part of transaction j has been proven through the *guarantee* part of transaction $j-1$ for $j > 1$. For transaction 1, the *assume* part has been proven by the one-time initial check. By implementing equality assumptions through mitering, a state map may help replace two state variables in the product machine by one state variable. An extension of the state maps is that of *cutting* the state maps, in which we remove both state variables altogether, and drive the fanout of the present state value of both state variables by a fresh primary input. Note that the proof obligation still remains the same irrespective of whether the state map is cut or not. The *cut* is a form of safe abstraction, under which a proof remains a valid proof, but a falsification may be spurious. Combinational equivalence checking corresponds to the special case of SEC in which all state elements

are required to be mapped and cut. However, for general SEC, a partial state map or no map at all still allows one to make progress.

Sequential machines, their periods, input maps, output maps, and state maps thus form the core concepts that allow one to precisely formulate a non-cycle-accurate SEC problem between designs with arbitrary timing differences. Except for state maps, all other components need to be provided by the user to formulate the problem. State maps are hints that, if available, simplify the verification task. Techniques for finding state maps automatically [21, 19] are not further discussed in this paper. In the next section, we describe how we take such a problem setup, and create an idealized, period-1, cycle-accurate product sequential machine (or a unit product machine), with latency 0 on all the outputs, on which further analysis, *e.g.*, bounded or unbounded model checking, can be done.

3. CONSTRUCTING UNIT PRODUCT MACHINE

Our system provides an on the fly, lazy unrolling scheme for sequential netlists, where the fanin of a signal x at cycle t ($x@t$) is created only when needed. Sequential machines are stored as word-level (bit-vector) netlists containing combinational operators (such as arithmetic), signals, flops, and input/output ports. The outputs and next state functions of flops are computed by combinational netlists, that realize the respective functions.

The unrolled netlists are stored in a database, where each netlist entity has a cycle counter and attributes correlating the entity to the one in the original netlist for book-keeping purposes. Unrolling through combinational operators keeps the same cycle but when unrolling across a flop (from the output Q-port to the next state input D-port), we decrement the cycle counter, as long as the cycle counter does not become negative. Otherwise, the unrolling stops at the initial state value of the flop, and a primary input is created in the unrolled netlist. These inputs may be replaced by the reset values, as and when needed. Facilities exist to interpret unrolled netlists as regular netlists, with enough book-keeping to allow for finding the original signal, and the cycle. Our system also provides combinational solvers that can provide answers to equality queries at the level of abstraction (word-level or bit-level) that the original sequential machines are in.

We are given two sequential machines, M_S and M_I, their periods P_S and P_I, a set of input maps Φ, a set of output maps Ψ, and a set of flop maps Θ. Thus, the equivalence check problem is $\langle M_S, P_S, M_I, P_I, \Phi, \Psi, \Theta \rangle$. The output of the algorithm is a single sequential machine M_U, along with a set of output maps Ψ_U. All the signals in Ψ_U are drawn from M_U, and have their latencies as zero. The set of output maps Ψ_U may be seen as the union of two disjoint sets of maps Ψ_{1_U} and Ψ_{2_U}, where the former corresponds to the original output maps Ψ and the latter corresponds to the *guarantee* part of the state maps Θ. The period P_U of M_U is 1. The *assume* part of the state maps Θ are factored into the construction of $M_U = (I_U, S_U, R_U, Y_U, T_U)$ through mitering.

In the following algorithm, the period of a flop refers to the period of the design that the flop belongs to. The high-level algorithm is described below in two phases.

Preprocessing:

1. Process the input maps Φ to obtain equivalence classes of timed inputs. Unconstrained timed inputs end up in an equivalence class of size 1. Recall that the maps repeat every period number of cycles of the two sequential machines. For each equivalence class, choose one as the *leader*. Miter all members of each equivalence class choosing the *leader* as the miter-winner.
2. **Initial State Checks:** For each state map $(s_S@l_1, s_I@l_2) \in \Theta$, add $s_S@l_1$ and $s_I@l_2$ to a joint unrolling frontier F_0. Unroll F_0, and apply reset values whenever unrolling reaches a flop output at cycle 0. Use combinational solvers, to compare $s_S@l_1$ and $s_I@l_2$ (if both latencies are 0, and both reset values are constants, then it is merely a comparison of constants). If they are not equal, report a counterexample and quit.
3. In order to reduce the number of flops in the product machine, we use the procedure OPTIMIZESTATEVARS (Fig. 2) on both M_S and M_I independently. It is a 1-step induction to find the set S_{ind} of state variables that are equal to their reset values every period cycles, in other words a fixed-point of the constant state variables. The term phase abstraction is used in [3, 16], for a similar process. For each flop $s \in S_{\text{ind}}$ the flop output at cycle 0 obtained during the unrolling process is replaced by the respective constant reset value. This step is very critical to our algorithm as it can help reduce the number of flops in the product machine significantly. For example, this step followed by constant propagation converts the serial machine of Fig. 1 to one with no flops, and 3 adders. In the absence of this optimization, the serial machine would have resulted in 2 flops, 8 multiplexors, and 8 adders.

Algorithm CONSTRUCTUNITPRODUCTMACHINE

1. For each state map $(s_S@l_1, s_I@l_2) \in \Theta$, add $s_S@l_1$ and $s_I@l_2$ to a joint unrolling frontier F_U. Since the initial state assumptions are proven, miter $s_S@l_1$ and $s_I@l_2$. This corresponds to the *assume* part of the flop maps. If the miter winner flop is part of S_{ind} and has a latency 0, then replace it by the constant reset value, otherwise leave it as a primary input to the unrolled netlist. If the state map is specified to be cut, then remember the miter winner state variable in a set S_c.
2. For each output map $(y_S@l_1, y_I@l_2) \in \Psi$, add $y_S@l_1$ and $y_I@l_2$ to F_U, and to Y_U. Also, add $((y_S@l_1)@0, (y_I@l_2)@0)$ to Ψ_U, the new set of output maps. Note the overloaded @ notation, the first referring to the realized output in M_U, and the second @ for the actual latency specification.
3. For each flop map $(s_S@l_1, s_I@l_2) \in \Theta$, add $s_S@(l_1 + P_S)$ and $s_I@(l_2 + P_I)$ to F_U, and to Y_U. Also, add $((s_S@(l_1 + P_S))@0, (s_I@(l_2+P_I))@0)$ to Ψ_U. This corresponds to the *guarantee* part of state maps.
4. Unroll from frontier F_U until we reach either primary inputs or flop outputs at cycle 0. Let S_r be the set of flops whose cycle-0 outputs are hit during unrolling. For each flop $s \in S_r$ with a period P, add $s@P$ to the frontier F_U and unroll again. The unrolling from the new frontier may find more flops whose cycle-0 outputs are hit. Add these flops to S_r, and keep repeating this step until convergence.
5. The next state function of each $s \in S_r$ is just $s@P$. Replace the primary input corresponding to s@0 with the output of a new flop s'. Add s' to S_U. Connect the input D-port of flop s' by the signal corresponding to $s@P$. The set of reset values R_U is made up of the reset values in the original sequential machines.
6. Optionally, apply combinational optimizations to the unrolled netlist, such as constant propagation, structural hashing, functional sweeps [12, 13], etc.

Fig. 3 shows the unit product machine obtained from the above algorithm for the example in Fig. 1. The unit machine contains the

Procedure OPTIMIZESTATEVARS

1 Let S_{ind} be the set of all relevant flops, e.g., the ones in the fanin of outputs being compared.
2 Remove from S_{ind} all the flops whose reset values are not constants.
3 Repeat the following 3 steps until S_{ind} does not change.
4 For each flop $s \in S_{\text{ind}}$, add $s@0$ and $s@P$ (where P is its period) to a frontier F, and unroll.
5 Replace cycle-0 values of the flops in S_{ind} in the unrolled netlist by the respective reset values.
6 Use combinational solver to find out for each flop $s \in S_{\text{ind}}$, if $s@P$ is equal to its reset value. If not, remove s from S_{ind}.
7 Return S_{ind}.

Figure 2: Procedure to optimize the number of state variables in the unit machine.

output at time 0 of the parallel machine, and the output at time 3 of the serial machine. The *count* and the *sum* state variables are always equal to their reset values (0) every period number of cycles, so they get eliminated by the OPTIMIZESTATEVARS procedure. A simple structural hashing on the unrolled machine causes the two outputs to be trivially mitered in the unit product machine, and hence, a trivial SEC proof is obtained.

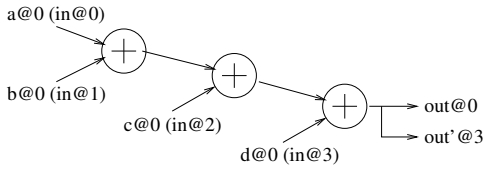

Figure 3: Unit product machine for the example in Fig. 1.

4. TIME CORRELATIONS

The joint unrolling based algorithm described in the previous section misses important time correlations between related input signals. Consider just a single machine in Fig. 4(a) with a latency of 1 on the output signal out. On applying the above algorithm to produce a unit machine with latency 0, we get the machine in Fig. 4(b). The output sequence of the unit machine $out'@0, out'@1, \ldots$ and of the original machine $out@1, out@2, \ldots$ are supposed to match. However, the unit machine clearly does not capture the computation of the original machine. To start with, it consumes two inputs every clock cycle, whereas the original machine consumes just one. In essence, the temporal correlation that the input x (corr. to $in@1$) is really just input y (corr. to $in@0$) one cycle later is lost. This lost correlation can be captured by inserting a flop between x and y as shown in Fig. 4(c). In general, if the unrolled inputs corresponding to two inputs on the unit machine are $m-$multiples of their period apart, there need to be m flops between them. Moreover, the reset values of the flops should be symbolic, to allow for fully symbolic inputs in the earlier cycles.

The situation is exacerbated with two designs unrolled together, even more so when the input maps are such that the first few values of some input are unconstrained, and they get constrained in future. Consider the example of Fig. 5. To construct the unrolled machines, we start with the frontier $out1@0$ and $out2@1$, giving us $G1@0$, z (corr. to $b@0$) and $F2@0$ as the inputs of the unrolled netlist. Based on the above algorithm, we add $G1@2$ and $F2@2$ to the

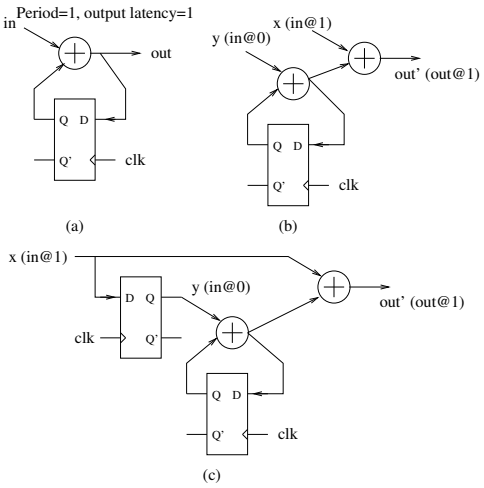

Figure 4: (a) A simple sequential machine with output latency 1, (b) corresponding unit machine, with output latency 0, and (c) Correct unit machine with correlated inputs.

frontier and unroll again, giving us y (corr. to $a@0$ and $b@1$), and x (corr. to $a@1$) as more inputs. After inserting flops $G1'$ and $F2'$ corresponding to the original flops $G1$ and $F2$, we get the unit machine in Fig. 6.

The outputs of the unit machine at cycle 0 i.e. $out1'@0$, and $out2'@0$, are $G1'@0$ and $F2'@0 \times z@0$, both of which evaluate to 0 since the reset values of both flops $G1'$ and $F2'$ are zero. However, at cycle 1 we get a falsification. The outputs at cycle 1, i.e. $out1'@1$, and $out2'@1$ (corr. to $out1@2$ and $out2@3$ on the original machines) evaluate to $x@0 \times y@0$ and $y@0 \times z@1$ respectively which results in a falsification. On the original machine, $x@0 \times y@0$ corresponds to $a@1 \times a@0$ and $y@0 \times z@1$ corresponds to $a@0 \times b@2$. Moreover, from the input constraints (as seen in Fig. 5), $b@2$ is equal to $a@1$, and hence, the counterexample is invalid on the original machines. The problem is that we are missing the correlation between $x@0$ and $z@1$ on the unit product machine in Fig. 6. The fix is to have input z (corr. to $b@0$) of the unit product machine be fed by a flop whose next state input is fed by input x (corr to $a@1$ and $b@2$). It is also important to leave the reset value of this inserted flop unspecified, as it allows a fully unconstrained symbolic value for $b@0$. Also note that $a@0$ and $a@1$ are not 2 cycles (i.e. period) apart, so they remain independent. Fig. 7 shows the fixed unit machine on which we get a proof of equality on the outputs $out1'$ and $out2'$.

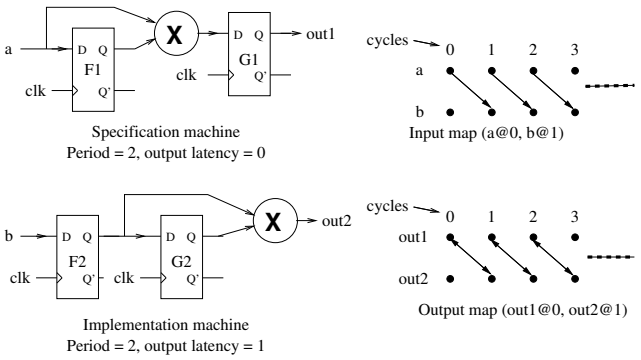

Figure 5: Two period 2 sequential machines, with input and output maps having a non-zero latency. The reset value of all flops is 0.

The lost correlations can be fixed by analyzing the inputs of the

unit machine, and if they are a m-multiples of their respective period apart, inserting m flops between them. The reset values of these flops are left symbolic. For a set of correlated inputs, the only one that remains after flop insertion is the one with the highest transaction counter (furthest in the future). We also need to properly account for signals such as $b@0$, which become miter losers in the future. Due to the lack of space, we will skip the detailed algorithm for correlating unit machine inputs.

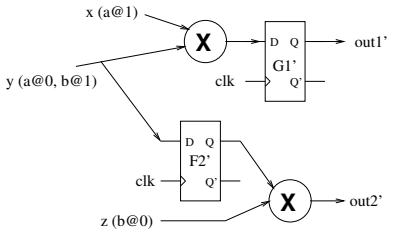

Figure 6: Unit product machine for the example in Fig. 5 with uncorrelated inputs.

Constructing the unit product machine, and applying the time correlations captures all the input/state assumptions, and proof obligations. Moreover, each primary input and state variable of the unit machine has enough book-keeping information to provide the cycle and the original signal/state variable that it came from. This book-keeping enables us to translate any trace on the unit machine to the pair of traces on the original sequential machines. For example, if a signal $c@l$ (input, output, state) is assigned value v in cycle k of the unit machine, it corresponds to the assignment of value v to the original signal c at cycle $l + k \cdot \mathit{period}(c)$. Moreover, if $d@m$ is a miter loser to $c@l$, it corresponds to the value assignment of v to signal d at cycle $m + k \cdot \mathit{period}(d)$.

Note that the cutting at state-maps is a user specified abstraction, so a counterexample on the unit machine may not necessarily reproduce on the original sequential machines. However, it faithfully reproduces on the original machines, if the state variables are abstracted correspondingly in the original machines.

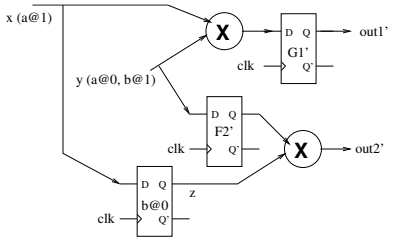

Figure 7: Correct unit machine for the example in Fig. 5 with correlated inputs.

We thus state the following theorem (proof omitted for lack of space) to capture the correctness of the unit machine.

THEOREM 1. *Given a sequential equivalence checking problem $\langle M_S, P_S, M_I, P_I, \Phi, \Psi, \Theta \rangle$, let M_U, with proof obligations Ψ_U be constructed using the algorithm* CONSTRUCTUNITPRODUCTMACHINE *followed by input correlation capture. Then the original proof obligations Ψ and Θ are valid if and only if Ψ_U on M_U are. Moreover, any counterexample on M_U can be translated to an equivalent counterexample on the pair of machines M_S and M_U.*

5. EXPERIMENTS

We present experiments to show the variety and scale of non-cycle-accurate SEC problems to which our technique has been successfully applied. Since we are not aware of any other system which does non-cycle-accurate SEC by the way of constructing cycle accurate unit machine, we cannot compare our approach to other approaches. Other systems that deal with non-cycle-accurate SEC are either proprietory, or limited in their applicability. Table 1 shows characteristics of a small sample of real industrial examples, on which SEC was successfully done using our system SLEC [4]. These designs come from a variety of problem domains, such as video/audio processing, wireless, encryption and were obtained using various approaches such as HLS, manual C v/s RTL, C v/s C, and RTL-RTL.

It should be obvious to the reader that the unit machine construction is only a part of a solution to SEC. Our system provides cycle accurate bounded/unbounded equivalence checking using word-level state-abstraction, induction, reachability, model checking, etc. Word-level solvers, word-level and bit-level netlist optimizations, efficient unrolling, are critical components of our system.

In Table 1, for each SEC problem, we report the sizes of the input, output and state map sets. We report the period, gate-count and number of word-level flops in each design M_S and M_I. Multiplying design sizes by their periods provides an upper bound on the size of the netlists for pure unrolling based bounded proofs, such as [6]. Next, we report the size of the unit machine relevant to each design. The size of the combined unit machine may be smaller than the sum of the individual sizes because of the sharing induced by mitering. Finally, we report the verification times and memory usage.

It is noteworthy that the size of the unit product machine is much smaller than the upper bound given by pure unrolling for most cases. The dramatic reductions are largely attributed to state variable optimization, and aggressive combinational optimizations. Many problems in the table are comparisons between untimed, period-1 specifications to implementations with large periods and a large number of flops. There are also some cases where periods of both machines are similar. In the case of FIR, even though a cycle-accurate, period-1 problem setup was available, lack of an effective state mapping made the verification infeasible. For the setup with period 350 reported here, state variable optimization eliminated the scheduling state machines which helped simplify the problem significantly.

6. RELATED WORK

Sequential equivalence is an important, and well studied problem. Theoretically, bisimulation equivalence [17] of transition systems is a close representation of sequential equivalence in the presence of resets. Pixley [18] provided a comprehensive theoretical framework for hardware equivalence, and introduced the concept of *alignability* for bringing two state machines into corresponding states. Singhal et al. [19] explored the notion of *safe-replaceability* and *delayed safe-replaceability* for non-reset based sequential equivalence. Authors in [9, 10] further formalized various approaches to equivalence, including compositional equivalence checks, and presented a SAT-based algorithm for alignability. van Eijk et al. [20, 21] presented methods for detecting equivalent state variables, and structural similarities for sequential equivalence. In addition to these, various forms of state mappings in different contexts have been described in the past, e.g., [1, 11].

The state machine reduction algorithm we proposed is similar to phase abstraction introduced in [3]. Lu et al. [15], Mishchenko et al. [16] use inductive techniques to determine state variables that follow repeatable patterns. However, their techniques are demonstrated for boolean representations only, which we are not restricted to. Finally, [2, 8, 16] present sequential equivalence checking sys-

Name	$\|\Phi\|$	$\|\Psi\|$	$\|\Theta\|$	M_S			M_I			unit M_S	unit M_I	verification	
				P_S	GC	$\|S\|$	P_I	GC	$\|S\|$	GC	GC	time (s)	mem (MB)
video1	73	64	192	1	1.623 M	256	147	83 K	1121	26 K	33 K	17988	7750
video2	53	6	19	34	25 K	56	36	26 K	60	77 K	80 K	12381	988
video3*	195	64	64	1	5.638 M	128	36	18 K	260	186 K	187 K	10693	644
video4*	50	1	1	1	217 K	33	30	29 K	459	35 K	65 K	1025	435
video5*	3	1	1	181	24 K	1324	182	21 K	260	6 K	6 K	433	405
FIR	740	2162	2156	350	593 K	2198	350	606 K	2198	42 K	41 K	9375	9966
FFT256	31	82	0	1091	12 K	263	1075	12 K	195	3.358 M	3.358 M	11272	1175
wireless1	3	2	0	1	25 K	3	128	10 K	791	16 K	18 K	8811	325
wireless2	8	2	0	1	168 K	2	132	163 K	2639	158 K	185 K	3306	1216
DES	6	1	0	1	171 K	2	129	112 K	461	97 K	74 K	3658	923
AES	19	1	8	1	1.164 M	8	424	19 K	87	73 K	135 K	6494	994

Table 1: SEC for large industrial designs. $|\Phi|/|\Psi|/|\Theta|$-number of input/output/state maps, GC-optimized gate count, $|S|$-number of word-level flops. A falsification was found for starred designs.

tems, that incorporate various techniques. In [6], bounded model checking was used for SEC between C programs and RTL in Verilog. Extraction of verification models from high-level descriptions, such as C++, is separately described in [7].

None of the approaches above address the problem of non-cycle-accurate SEC, especially while preserving the abstraction-level of the netlists. Our normalization method can be used get a cycle-accurate SEC problem, to which any of the above approaches can be applied.

7. CONCLUSIONS AND FUTURE WORK

We have presented a definition of non-cycle-accurate SEC, and a novel method for reducing a non-cycle-accurate SEC problem to a cycle-accurate SEC problem. Our normalization method provides many benefits. All state variables that go through a set pattern in every transaction completely disappear from sequential analysis. Unit machine construction puts the computations in a normal form, enabling meaningful intermediate equivalences, crucial for divide and conquer approaches, such as cuts and heaps, or structural similarities [13, 20]. Moreover, our approach does not restrict us to just bit-level representations. We have successfully demonstrated our method on a variety of industrial designs.

There are a few limitations of our approach, which we are working towards lifting, such as variable periods, and more complex mappings.

8. REFERENCES

[1] M. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for microprocessor correctness statements. In *CHARME*, pp. 433–448, 2001.

[2] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *ICCD*, pp. 259–266, 2006.

[3] P. Bjesse and J. Kukula. Automatic generalized phase abstraction for formal verification. In *ICCAD*, pp. 1076–1082, 2005.

[4] Calypto Design Systems. Sequential Equivalence Checking: A new approach to functional verification of datapath and control logic changes. `http://www.calypto.com/wp_request.php?paper=sequential`, 2007.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[6] E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pp. 368–371, 2003.

[7] M. Haldar, G. Singh, S. Prabhakar, B. Dwivedi, and A. Ghosh. Construction of concrete verification models from C++. In *DAC*, pp. 942–947, 2008.

[8] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, C.-Y. Huang, and F. Brewer. AQUILA: An equivalence checking system for large sequential designs. *IEEE TOC*, 49(5):443–464, 2000.

[9] D. Kaiss, M. Skaba, Z. Hanna, and Z. Khasidashvili. Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In *FMCAD*, pp. 20–26, 2007.

[10] Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna. Post-reboot equivalence and compositional verification of hardware. In *FMCAD*, pp. 11–18, 2006.

[11] A. Koelbl, J. R. Burch, and C. Pixley. Memory modeling in ESL-RTL equivalence checking. In *DAC*, pp. 205–209, 2007.

[12] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *ICCAD*, pp. 50–57, 2004.

[13] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21:1377–1394, 2002.

[14] Y.-L. Lin. Recent developments in high-level synthesis. *ACM TODAES*, 2(1):2–21, 1997.

[15] F. Lu and K.-T. Cheng. Sequential equivalence checking based on k^{th} invariants and circuit sat solving. In *HLDVT*, pp. 45–51, 2005.

[16] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang. Scalable and scalably-verifiable sequential synthesis. In *ICCAD*, 2008.

[17] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI-Conference*, volume 104 of *Theoretical Computer Science*, pp. 167–183, Karlsruhe, 1981. Springer-Verlag.

[18] C. Pixley. A theory and implementation of sequential hardware equivalence. *IEEE TCAD*, 11(12):1469–1478, 1992.

[19] V. Singhal, C. Pixley, A. Aziz, S. Qadeer, and R. K. Brayton. Sequential optimization in the absence of global reset. *ACM TODAES*, 8(2):222–251, 2003.

[20] C. A. J. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE TCAD*, 19(7):814–819, 2000.

[21] C. A. J. van Eijk and J. A. G. Jess. Exploiting functional dependencies in finite state machine verification. In *DATE*, pp. 266–271, 1996.