

Sequential Equivalence Checking between System Level and RTL Descriptions

Shobha Vasudevan
Computer Engineering Research Center
University of Texas at Austin
shobha@cerc.utexas.edu

Jacob A. Abraham
Computer Engineering Research Center
University of Texas at Austin
jaa@cerc.utexas.edu

Vinod Viswanath
Intel Corporation
Austin, TX
vinod.viswanath@intel.com

Jiajin Tu
Computer Engineering Research Center
University of Texas at Austin
tujiajin@cerc.utexas.edu

Abstract

Sequential equivalence checking between system level descriptions of designs and their Register Transfer Level(RTL) implementations is a very challenging and important problem in the context of Systems on a Chip (SoCs). We propose a technique to alleviate the complexity of the equivalence checking problem, by efficiently decomposing it using compare points. Traditionally, equivalence checking techniques use nominal or functional mapping of latches as compare points. Since we operate at a level where design descriptions are in System Level Languages or Hardware Description Languages, we leverage the information available to us at this level in deducing sequential compare points. Sequential compare points encapsulate the sequential behavior of designs and are obtained by statically analyzing the design descriptions. We decompose the design using sequential compare points and represent the design behavior at these compare points by symbolic expressions. We use a SAT solver to check the equivalence of the symbolic expressions. In order to demonstrate our technique, we present results on a non-trivial case study. We show an equivalence check between a System C description and two different Verilog RTL implementations of a Viterbi decoder, that is a component of the DRM SoC.

1 Introduction

System-on-a-chip (SoC) designs contain unprecedented levels of functional and structural complexity in a single system, making their verification a daunting challenge to known verification methodologies. Inordinate amounts of time and effort are spent in the SoC industry, validating a

chip for functional and timing requirements. Although simulation based verification is the most widely used technique for validating SoCs, the degree of confidence in these simulations is low for these highly complex, monolithic designs. Formal verification is desirable due to its high quality assurance, but the known techniques of hardware or software verification are not equipped to handle the size, or the heterogeneity of SoCs. Futuristic verification research then, involves new formal methods that are capable of scaling to the SoC domain.

The problem of checking sequential equivalence of a system level model (SLM) with respect to its implementation in Register Transfer Level (RTL) is a relatively novel domain. An equivalence check at this level, or even at the RTL to RTL level is desirable due to a number of reasons [13]. Optimizations for power, speed, area or other design parameters is usually done during this stage of design. These optimizations could range from shifting logic between flip-flops, or using latches in place of flip-flops in certain portions, or changing the algorithm of a certain portion of the design to a faster one. In all these cases, an equivalence check at this level would enhance the confidence in the optimized implementation, as well as be more economically viable than detecting bugs at a later stage in the design cycle.

We present an equivalence checking technique to verify system level design descriptions against their implementations in RTL. Our technique involves the efficient decomposition of the equivalence checking problem, in order to make it more tractable. We present an automatic technique to compute high level *sequential compare points*, to compare variables of interest (observables) in the candidate design descriptions. Our compare points are defined as co-

ordinates on the space-time axis of the design, denoted by their relative position with respect to the time domain (clock cycles), and their position in the space domain (data variables). This aligns with the sequential behavior of the designs being compared, and provides an easy, intuitive abstraction of the equivalence checking problem space. We start the two design state machines at the same initial (or reset) state, and step the machines through every cycle, until we reach a sequential compare point.

At the sequential compare points, we construct symbolic expressions for the observables that encapsulate the sequential behavior of the designs, until the cycle of comparison. At each sequential compare point, we prove the equivalence of the two state machines. using a lower (Boolean) level engine, which in this work, is a Boolean satisfiability (SAT) solver. The principal gain in our technique is that we are leveraging the expressiveness and information available to us at the register transfer and system levels. Although significant amount of research has been done on compare points for gate level equivalence checkers, these algorithms and heuristics are limited by their domain. On the other hand, since we operate at the higher, source code level, our sequential compare points are more intuitive and easier to detect. Also, they capture the notion of design progression through time, which is useful in meaningful decomposition of the equivalence checking state space.

We present the results of our technique on a System C [3] description and two different Verilog RTL descriptions of a Viterbi decoder [19] module that is a part of the Digital Radio Mondiale (DRM) SoC [2]. Our results show the performance benefits of using our technique to verify real designs.

The principal contributions of this work are the following.

- We present a theoretically sound sequential equivalence checking method between system level design descriptions and their RTL implementations
- We present an automatic decomposition technique for splitting the equivalence checking problem space, by introducing a notion of sequential compare points that exactly model the sequential behavior of designs.
- We leverage the expressive power and relative simplicity of high level descriptions in our equivalence checking, by reasoning entirely at that level.
- Our technique can statically be used to analyze and decompose the source code, in order to assist Boolean level engines to overcome capacity issues.
- We demonstrate the effectiveness of our technique by checking the equivalence of the implementation of a real SoC component against its specification.

The outline of the paper is as follows. Section 2 provides an overview of the related work in this, and allied areas. Section 3 details the technique, the algorithm for automatic decomposition, and explains it with a simple example. Subsection 3.4 gives the proof of correctness of our technique. In Section 4, a case-study of the Viterbi decoder is presented. Subsection 4.1 and Subsection 4.2 describe the two implementations of the Viterbi decoder and the verification process. The result of our experiments are presented in Subsection 4.3. Section 5 contains a brief discussion of the merits and demerits of the technique.

2 Related Work

We provide a brief background that explores the related work in the entire spectrum of topics covered by this work.

Formal verification, especially equivalence checking, has achieved considerable success in the context of hardware. Combinational equivalence checking checks two acyclic, gate-level circuits. Combinational equivalence checkers can also be used to check equivalence of two sequential designs, provided the state encodings of the two designs are the same. Although this technique has widespread use in many commercial tools, the real challenge of sequential verification is in verifying two designs with different state encodings. Sequential satisfiability engines [11], [14] and sequential ATPG engines [4], [9] solve this problem to a large extent by unrolling the circuit until a given time frame. Considerable research has been done to find compare points for latch mapping [5], [6], [17]. However, these techniques operate at the gate level, where they reason in the Boolean domain.

Fewer attempts have been made to apply sequential equivalence checking to the behavioral RTL descriptions of designs. In [15] a methodology for checking the combinational equivalence between C and RTL is described. The C source code is converted to a Hardware Description Language (HDL) and commercial RTL to RTL equivalence checkers are used thereafter. The C code is very similar to the RTL, in order for the translation to be achieved, which might not be a scalable solution.

Clarke and Kroenig [7], [10] proposed a solution with CBMC, a C-based bounded model checking engine that takes a C program and a Verilog implementation. The two programs are unwound together, and converted into a Boolean satisfiability checking problem. The Verilog code is converted to Boolean formulas by a synthesis-like procedure, and an innovative technique is described to convert the C-code into Boolean formulas, including pointers and nested loops. However, the capacity of CBMC is limited by space and time considerations. This is due to the fact that the reasoning done by this tool is entirely in the Boolean domain. On the other hand, our technique reasons at the system and register transfer level, splitting the equivalence

```

main (M: System level model, V: RTL model, O: Set of Observables)
  C =  $\phi$ 
  while O is not empty
    for every cycle (transition) in the state transition graphs of M and V
      if a set of variables  $S \subseteq O$  is assigned in cycle  $t_i$  in the state-transition graph of M
        check the state-transition graph of V
        if  $o \subseteq S$  assigned in cycle  $t_j, j \leq i$  in V
           $C = C \cup \{t_i, o\}$ 
           $O = O \setminus o$ 
          result = compare ( $t_i, o$ )
          if (result == true)
            move to the next state in M and V
          else
            go to error state

compare (t: Time cycle, d: Set of variables)
  for every variable v in d in M, V
     $E^M = 1, E^V = 1$ 
    while ( $t \geq 0$ )
       $E_t^M = \text{symbolic}(v, t, M)$ 
       $E_t^V = \text{symbolic}(v, t, V)$ 
       $E^M = E^M \wedge E_t^M, E^V = E^V \wedge E_t^V$ 
      decrement t
    ans = check ( $E^M, E^V$ )
    return ans

symbolic (L: Variable, t: Time cycle, Z: Model)
  do
    for every assignment  $L = R$  under control signals X in the current cycle t
       $E = f(Z[L/R], X, t)$ 
       $R = L$ 
    while R is not an input, or  $R \notin O$ 
    return E

```

Figure 1. Algorithm for proving equivalence between a C-like system and its RTL implementation

checking problem into smaller problems that can be handled by the lower level engines. This static analysis of the source code, before running the problem through Boolean level engines, is the principal contribution of our technique.

Another approach to equivalence checking between C descriptions, that could be extensible to C vs RTL descriptions, is described in [12]. This approach detects and extracts the textual differences in the two target programs, and then does a dependence analysis using program slicing, to check for the actual differences in the two programs. It then symbolically simulates this difference and reports the equivalence checking results. This technique, however, is most effective when the two target programs being compared are very similar to each other, in function as well as structure. Since this process uses syntactic information entirely, the similarity of the target descriptions is very essential to its application. Our technique does a semantic comparison of the two target programs, with respect to their functionality, and is therefore wider in its scope.

A few commercial tool vendors [1] also aim at solving the sequential equivalence checking problem between SLM and RTL. However, this area still presents a major opportunity for further research.

In previous work, [18] we presented a technique for RTL to RTL equivalence checking of complex combinational circuits, including multipliers. We extend our technique to sequential equivalence checking, and address the problem in the realm of SLM vs RTL.

3 Technique for System level vs RTL equivalence checking

We present a technique for equivalence checking of two high level design descriptions. Our technique involves the automatic decomposition of the equivalence checking state space, from the source code of two candidate designs. We introduce the notion of sequential compare points, that encapsulate the sequential behavior of designs, with respect to time as well as data. In the rest of the paper, we will refer to sequential compare points, simply as compare points.

3.1 Selecting Comparison Points

In a previous version of the work, we selected the observables by visual inspection from a block diagram of the systems. These observables are observed by stepping the state-transition graphs for the two systems. When an observable is assigned in one (specification) system, the other (implementation) system is stepped until the time step where the same observable is assigned in it. This time step would then constitute a compare point. Selecting observables from a block diagram can be prone to poor conjectures, that can lead to omission of some temporary variables worthy of observation. If only the outputs or other prominent signals are observed in a large design, the symbolic expressions might get intractable. Since the efficiency of the technique largely depends on the decomposition strategy, the selection of observables is an important step. Once the observables are selected, depending on the size of the systems, building the state-transition graphs of both systems might get arbitrarily complex. This is potentially a cause for inefficiency in the technique. In order to address this inefficiency, we propose an alternative semi-formal solution to select the comparison points.

Timing diagrams represent the temporal behavior of signals in a design with reference to the system clock, as well as other causal signals. When described as a part of the specification document, the timing diagrams can provide valuable insight about the most relevant signals in the design. We can obtain the information about when to observe these relevant signals by simulating the SLM and the RTL model with respect to the same clock. Simulation over a few cycles can indicate the relative time latency at which the observables are available in both the models. We do not need a very high precision simulation model for our simulations. The simulations can be cycle accurate (or coarser grained accuracy) for each model. Also, since we are not trying to simulate all possible input values, but only a small sample set of values to identify relative latencies, we do not require a fast simulation model. If timing diagrams descriptions of signals are not provided in the specification, we will not be able to make educated guesses about the observables. In this case, we would have to rely on the information from the block diagram. However, once identified, the observables can be simulated to find the relative times at which they are computed in both the SLM as well as the RTL models.

An inherent limitation of this method of selecting comparison points is that the information about the cycle of comparison is obtained from the RTL implementation model itself. The state-transition graph or the simulation of the RTL provide accurate information about the time at which an observable is available for comparison according to the design. However, if the design has a flaw that computes the “correct” value of an observable at an “incorrect” cycle, we cannot detect the flaw with our technique.

3.2 Algorithm

An algorithm for our technique is presented in Figure 1.

Let M and V denote the (C-like) specification and the RTL implementation respectively. The Kripke structure or the state-transition graphs for these systems at the source code level is constructed. A system state consists of symbolic values of all state variables in the system, and a transition corresponds to progression of time, with respect to the explicit clock in the system. The signals that are interesting for observation are a part of the list of observables, O . Typically, this list is obtained from the primary outputs, or a block diagram of the C and the RTL designs. This list is the same for both the systems, as it is assumed that a name mapping is provided for all the primary inputs and observable signals in the designs. The systems are assumed to start in the same initial states. C denotes the list of all compare points. A compare point has a two-tuple description, one for the time cycle, and another for the set of variables that are being compared. This list is empty initially.

The state transition graphs of the two systems are traversed step-by-step. This can be thought of as stepping both the systems in time. After every transition (cycle), the two systems are checked to see if any of the observable variables are assigned. If a variable from the list of observables is assigned in one of the systems (say M), the other system (say V) is checked to see if the same variable has been assigned in the current, or some previous cycle. If the variable was assigned in V before M , the two systems are compared at the current cycle, *i.e.* the current cycle becomes a compare point. If, however, the variable has not yet been assigned in V , the two systems are transitioned until the next observable is assigned in either.

The *compare()* function compares the observables at a given cycle. Since there may be more than one observable that is being compared at a cycle, this function takes a set of variables. For each of these variables, a symbolic expression is computed at the “current” time cycle, t . The symbolic expressions computed in every previous cycle, until $t = 0$ are iteratively concatenated in both the systems, to obtain E^M and E^V . The two expressions E^M and E^V are now checked for equivalence using a SAT solver. The *check()* function corresponds to the SAT solver call in our algorithm. It may be noted, however, that other equation solving engines may also be employed to check the equivalence of expressions at this stage.

The *symbolic()* function computes the symbolic expression at a given cycle. For every assignment to a given variable, it substitutes the right hand side of the definition for the variable (denoted by $Z[L/R]$ for any model Z) along with the control signal information X , required for the substitution. X is a Boolean expression that represents the constraints on the control signals. The substitutions $M[L/R]$, $V[L/R]$ are valid when X is true. The sym-

```

SC_MODULE(bcd_to_binary)
{
  sc_in_clk clk_i;
  sc_in <bool> rst_i;
  sc_in <sc_uint<5>> dat_bcd_i;
  sc_out <sc_uint<5>> dat_binary_o;

  sc_uint<4> msb;
  sc_uint<4> ls_byte;

  void convert_code() {
    msb = dat_bcd_i[4].read();
    ls_byte = dat_bcd_i[3:0].read();
    dat_binary_o.write ( (msb * 10) + ls_byte );
  }

  SC_CTOR(bcd_to_binary) {
    SC_METHOD(convert_code);
    sensitive << dat_bcd_i << clk_i.pos();
  }
};

```

Figure 2. Example SystemC code for BCD to binary conversion.

bolic expression also identifies the cycle t in which it holds true. This expression is computed by $f()$ in the algorithm.

If during this substitution, a primary input or a previously “observed” observable is reached, the substitution stops. This is valid because the two systems have been proved equivalent with respect to this observable in a previous compare point. From that cycle onwards, the two systems are always equivalent with respect to the observables at that compare point. In a sequential design, the data from the previous cycle progressively gets used in future time cycles. The symbolic expression of the observables at a given compare point remains the same for all future compare points. Therefore, these observables need not be proved equivalent, at every reference to the corresponding compare point.

If they are found equivalent, the proof proceeds. The comparison process is repeated, until all observables have been “observed”. If they are not found equivalent, an error trace is generated at that compare point. This is very useful, since the equivalence proof can be assumed to hold until the compare point where it fails. If n is the number of observables, T the total number of cycles, and P the size of the program graph on any timeslice, then the time complexity of running this algorithm would be $O(T^2 \times n \times P)$.

3.3 Example to show equivalence between SystemC and Verilog

We illustrate our technique using an example that converts a given number in Binary Coded Decimal (BCD) for-

```

module bcd_to_binary (clk_i, dat_bcd_i, dat_binary_o, done_o)
input clk_i;
input [4:0] dat_bcd_i;
output [3:0] dat_binary_o;
output done_o;

reg [3:0] dat_binary_o;

reg [3:0] tens_digit;
reg [3:0] no_tens_digit;
reg tens_select;

always @(posedge clk_i)
begin
  tens_digit <= dat_bcd_i[3:0] + 4'd10;
  no_tens_digit <= dat_bcd_i[3:0];
  tens_select <= dat_bcd_i[4];
end

always @(posedge clk_i)
begin
  if (tens_select)
    dat_binary_o = tens_digit;
  else
    dat_binary_o = no_tens_digit;
end

endmodule

```

Figure 3. Example Verilog RTL code for BCD to binary conversion.

mat into binary format. Figure 2 shows a code segment written in SystemC. Figure 3 shows its implementation in Verilog RTL. The clock signal clk_i and input BCD data (dat_bcd_i) are inputs for both the systems. From the timing diagram obtained by simulating these pieces of code, we can see that the output dat_binary_o is computed in a single cycle in the System C, and after two cycles in the Verilog, once the input arrives. Therefore, the first comparison point $C_1 = (t = 2, d = dat_binary_o)$. Since this example design is quite small, there is only one comparison point. In a larger design, there would have been comparison points where the observables are internal variables. Let both the designs start at time t . The symbolic expression s_1 for dat_binary_o after all the substitutions as explained in Figure 1 in the SystemC is $dat_binary_o(t + 1) = (dat_bcd_i[4](t + 1) * 10) + dat_bcd_i[3 : 0](t + 1)$. This denotes that the value of the observable gets computed at $t + 1$. Every variable is annotated with the corresponding value of time. The symbolic expression s_2 from the Verilog code at cycle for dat_binary_o is $(dat_binary_o(t + 2) = (dat_bcd_i[4](t + 1) ? dat_bcd_i[3 :$

$0](t) + 4'd10) : \text{dat_bcd_i}[3 : 0](t))$. It is clear that when s_1 and s_2 are compared in a SAT solver, the symbolic values of `dat_binary_o` will be equal. The SAT solver checks for the functional equality of the two expressions. However, the annotation with respect to t is not a part of the SAT expression. Therefore, we cannot check if the variables are being computed at the “correct” time cycle. We can, however, find a functional mismatch if the observables have been checked at a wrong time cycle.

3.4 Justification of our notion of equivalence

Many notions of sequential equivalence have been proposed in the literature. Most of them adhere to the broad classification of equivalence with respect to a set of initial states \mathcal{I} or alignability equivalence that can demonstrate resetability across all states \mathcal{I} , ?inghal01, ?hashidashvali03. All these notions of sequential equivalence are at the gate level, and deal with retiming and synthesis based optimizations. They also build the state-transition relation in order to reason about sequential equivalence. Since we are dealing with a different level of abstraction, we need to define our own notion of sequential equivalence. However, in philosophy, our notion of equivalence is more along the lines of [?] than the alignability notion of equivalence. We state our theories of correctness with respect to a set of initial states. This is because, as mentioned in ?, this paradigm has higher scalability, due to the potential of leveraging many existing algorithms. In order to

We present here, a theoretical basis to justify our technique.

Let M be the design specification system (model). Let V be its implementation. Let $PI(X)$ and $PO(X)$ denote the primary inputs and primary outputs of system X , such that $PI(M) = PI(V)$ and $PO(M) = PO(V)$. We assume that a signal name mapping is provided between the two systems. Let $\sigma_X(s, t)$ be the function in a system X that takes in a signal s at a point t in time, and returns the symbolic expression for s in terms of $PI(X)$ over all times until t . We define the simulation relation, $\sim_{p,t}$ where p is a set of signals, as the function between two systems X and Y , such that $\forall i \in p, \sigma_X(i, t) \equiv \sigma_Y(i, t)$. Our notion of equivalence is to show that $V \sim_{PO} M$.

We define a compare point $C = (t, d)$ as a co-ordinate on the space-time axis of the design, denoted by its relative position with respect to the time domain t , and its position in the space or data domain d . In the context of designs, t would be in terms of cycles, and d would be a set of observable signals.

Lemma 1 *At a given $t, \forall i \in d$, where $C = (t, d), V \sim_{i,t} M \Rightarrow V \sim_C M$.*

Proof outline: For any observable signal i at a given time step, $q = \sigma_V(i, t)$ is equal to the expression obtained by

iteratively expanding i for every time step until t , such that the symbolic values are obtained in terms of the primary inputs, $PI(V)$. $r = \sigma_M(i, t)$ can also be similarly interpreted. In order to prove that $q \equiv r$, we use a SAT solver or another engine whose functionality we assume is correct. Once $q \equiv r$ is proved, it implies that for all input values, at cycle t , the two systems will have the same value for i . If this result is proved for every $i \in d$, since t is a given constant, we can conclude that a simulation relation holds between the two systems.

Theorem 1 *Let the two systems M and V be described with $PI(M) = PI(V)$ and $PO(M) = PO(V) = P(O)$. Let n be the longest cycle length (time step) taken to obtain all primary outputs in both systems. Let M and V be compared at every compare point $C = (t, d)$, such that $0 \leq t \leq n$. Then, $\forall C, V \sim_C M \Rightarrow V \sim_{PO} M$.*

Proof outline: The proof follows from induction, where the base case is at time $t = 0$, when the systems start from the same initial state. The induction hypothesis is relieved by using Lemma 1 and substituting equals for equals in the entire symbolic expression obtained. If all the primary outputs are generated by cycle n , at $C = (n, PO)$, the desired simulation relation will hold between the two systems.

3.5 Error Detection

An inherent limitation of our method of selecting comparison points is that the information about the cycle of comparison is obtained from the RTL implementation model itself. The state-transition graph or the simulation of the RTL provide accurate information about the time at which an observable is available for comparison according to the design. Since our technique attempts to capture the design progression in time as well as in data space, we present a brief discussion about the functional and temporal error scenarios in our domain, and how our technique performs in these scenarios.

There are four possible outcomes of the `compare()` function in Figure 1 when comparing SLM and the RTL model at any compare point $C = (t, v)$, such that t is the time at which the observable v is computed in the RTL model.

- **Functionally and temporally correct.**
In this case, `check()` returns **true** and t is the correct cycle of computation. This is the case when the algorithm will return a **true** value. This means that the symbolic function of v in the RTL is implemented functionally as per specification and at the correct time cycle.
- **Functionally incorrect and temporally correct.**
In this case, `check()` returns **false** and t is the correct cycle of computation. This is the case when the algorithm will return a **false** value. This means that the

symbolic function of v is not implemented correctly in the RTL. This scenario is detected by our technique. An error trace is provided between the past compare point and the current compare point.

- Functionally incorrect and temporally incorrect. In this case, the $check()$ procedure returns a **false**, which provides a functional error trace, but not a temporal error trace. In other words, there is a possibility of obtaining false negatives in this scenario, since a mismatch does not indicate if there is an error in the functionality or timing.
- Functionally correct and temporally incorrect. In this case, t is not the correct cycle of computation. If t then

4 Equivalence Checking of System C vs RTL of Viterbi Decoder

We perform our experiments on a Viterbi decoder, that is a part of the Digital Radio Mondiale (DRM), implemented in System C. Since the Viterbi decoder module (embedded in the MLC decoder) took an inordinately long number of simulation cycles, the DRM SoC design was partitioned to implement the Viterbi decoder in hardware. This hardware accelerator was implemented in Verilog RTL, from the initial System C description of the Viterbi module. More details on this process can be obtained from [16].

Optimizations for speed, like pipelining, are typical applications where an equivalence checking between the system level design and the RTL are desired. Our experiments use such optimized implementations to show the efficacy of our technique.

The System C specification of the Viterbi decoder is a very basic model, that implements the Viterbi decoding algorithm, but has no optimizations for speed, area or power (Figure 4). The first RTL design we compared against, is a pipelined implementation of the Viterbi decoder, optimized for speed. The second implementation is optimized for area. We show the results of doing equivalence checking using our technique on these Verilog designs, with respect to the specification in System C.

Figure 4 shows the basic block diagram of a Viterbi decoder [19]. There are two major stages to the functionality of the Viterbi decoder. One is collecting the inputs depending on the Puncture Pattern and storing them in an buffer (FF Buffer). The other stage is the Trellis computation. The next state values of the Trellis matrix are computed by a function (Butterfly network) of current state values of the Trellis matrix and the inputs stored in the FF Buffer.

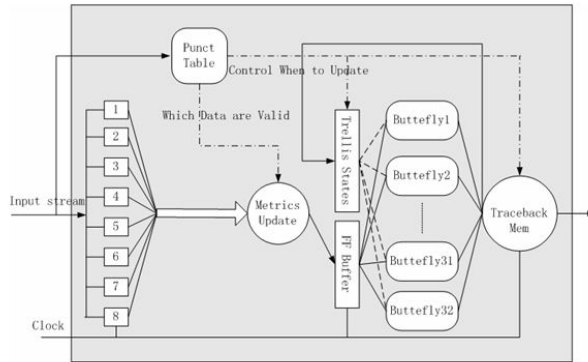


Figure 4. Viterbi decoder- System C design specification

4.1 Equivalence Checking of a Pipelined Viterbi Design

We started with a System C description, as well as the pipelined Verilog RTL implementation of the Viterbi decoder design. Figure 5(a) shows the block diagram of the pipelined implementation. From the block diagram, we arrive at the following observables in the experiment.

- 8 FIFO entries, each 32-bits wide: $FF[7:0][31:0]$
- 64 Trellis Matrix entries, each 32-bits wide: $TM[63:0][31:0]$
- 2 entries in the MatDec, each 32-bits wide: $MD[1:0][31:0]$
- Decoded output, 32-bits wide: $Out[31:0]$

The signal (variable) mapping between the two designs is provided. For the sake of readability, we denote the observables in the System C design with a subscript s , and the observables in the Verilog design with a subscript v . We outline the proof methodology using our technique. We start both the designs at the reset state initially. We step the two designs in tandem. From the state-transition graph of the System C design, we observe that the output is computed at every cycle. From the state-transition graph of the Verilog design, however, we observe that the output is computed in the 10^{th} cycle after the reset state. In accordance with our algorithm in Figure 1, we need to step the Verilog design more than the System C specification, to arrive at compare points. Figure 4 is a pictorial representation of the decomposition of the equivalence checking proof, on the basis of compare points. The horizontal axis represents the data (observables), and the vertical axis shows the number of systems being compared (in our case, two). Time is represented along the axis normal to the plane of the paper.

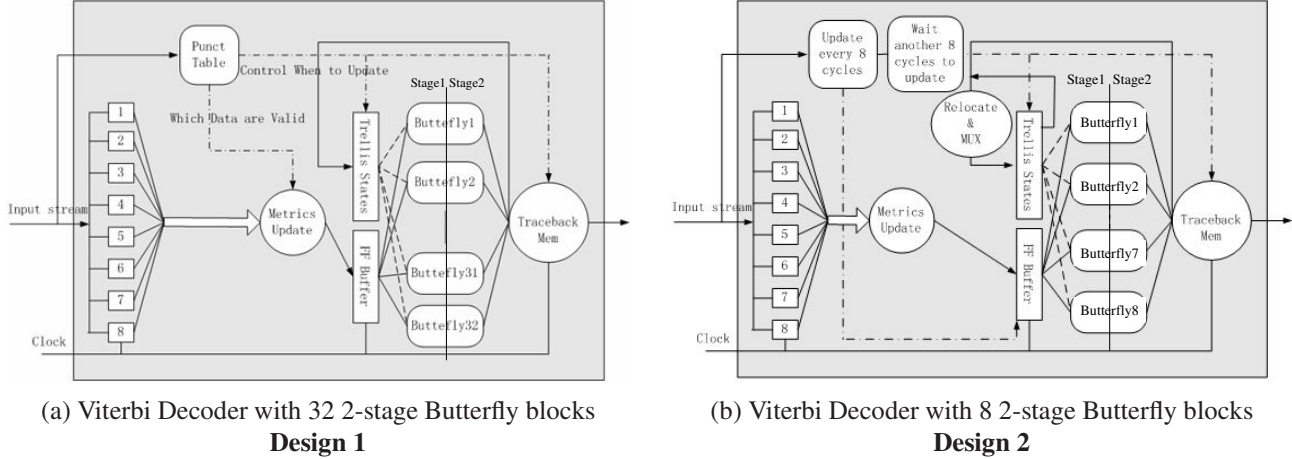


Figure 5. Viterbi Decoder block diagram. The decoder on the left is a pipelined Viterbi decoder with a 2-stage Butterfly, with 32 parallel butterfly blocks. The decoder on the right is further optimized for area with only 8 parallel butterfly blocks. The area-optimized design will run 4 times slower on the Trellis computation.

The first set of observables $FF[7:0][31:0]$ is available after 8 cycles, at the output of the FF Buffer. The first compare point, is therefore $C_1 = (t = 8, d = FF[7:0][31:0])$.

For each entry i in the FIFO buffer, the FIFO variables are $FF_s[i][31:0]$ and $FF_v[i][31:0]$. We call the *compare()* and *symbolic()* functions at the compare point, and obtain the expressions for the FF variables.

In both the designs, the FF Buffer gets updated by the function *GetMetricSet()*. Therefore, the symbolic expressions correspond to an expansion using this function. The two symbolic expressions for $FF_s[i][31:0]$ and $FF_v[i][31:0]$ are checked using a SAT solver. This procedure is repeated 8 times, for every entry in the FF Buffer, since each of them has a unique symbolic expression.

The next comparison point is obtained by stepping the two state machines of the designs after the 8th cycle. Although the System C assigns to an observable every cycle, the Verilog design assigns to the next observable at the 10th cycle. The next $v \in d$ is the Trellis Matrix, $TM[63:0][31:0]$. All the entries in this 64×32 matrix need to be checked, since the entire table is updated every 10th cycle. The values of the MatDec decision table, $MD[1:0][31:0]$ is also updated in this cycle, as is the decoded output, $Out[31:0]$. The intermediate variable every 9th cycle, btm which is not an observable is shown in lower case in Figure 4.

The second compare point, is therefore, $C_2 = (t = 10, d = TM[63:0][31:0], MD[1:0][31:0], Out[31:0])$.

The Trellis Matrix table gets its values from the 32 butterfly blocks in the design, each of which output 2 entries.

The symbolic expression from the RTL, therefore, is a function of the butterfly blocks. For every 2 entries in the Trellis Matrix, the corresponding symbolic expression can be obtained from the butterfly. For instance,

$$TM_v[0], TM_v[1] = \text{Butterfly}(TM_v[0], TM_v[2], FF_v[0][31:0], FF_v[7][31:0])$$

Similarly, in the System C design,

$$TM_s[0], TM_s[1] = \text{Butterfly}(TM_s[0], TM_s[2], FF_s[0][31:0], FF_s[7][31:0])$$

Since $FF_v[0] = FF_s[0]$ from a previous comparison point C_1 , the symbolic expression for these signals are not expanded any further. The symbolic expressions for $TM_v[0]$, $TM_v[1]$ and $TM_s[0]$, $TM_s[1]$ are checked for equivalence by the SAT solver. This procedure is repeated 32 times, for every pair of entries in the Trellis Metric that need to be checked.

The other observables $MD[1:0][31:0]$ and $Out[31:0]$ are similarly checked for equivalence. The proof of $Out[31:0]$ is not shown in the figure. The results of the *check()* function are discussed in the next section.

4.2 Equivalence Checking of a Pipelined Viterbi Design Optimized for Area

We used our technique to perform equivalence checking of a pipelined Viterbi design, that is further optimized for area. In this design, the 32 butterfly units are split into 4 stages, each stage having 8 butterfly units. Figure 7 shows the decomposition of the proof using compare points.

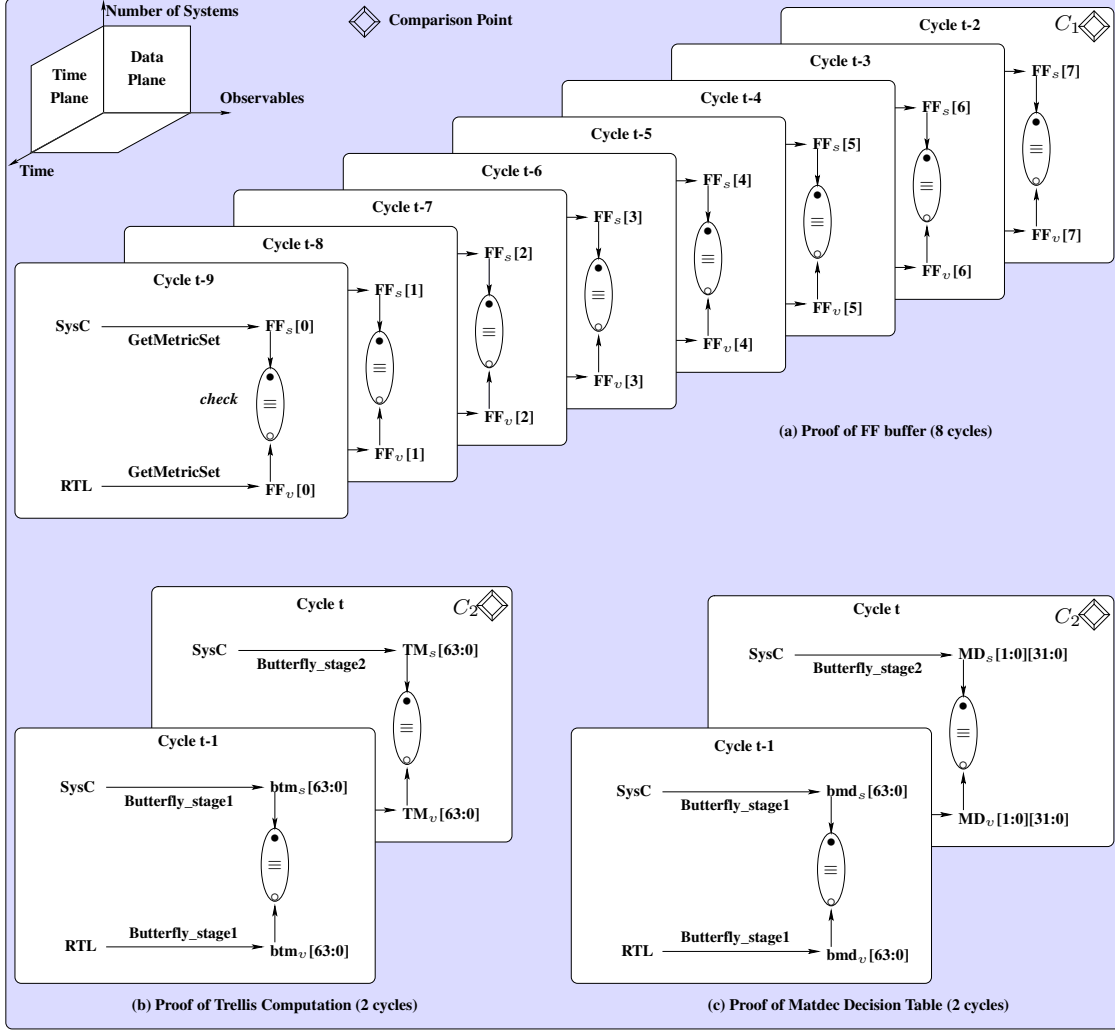


Figure 6. Proof of sequential equivalence checking of pipelined Verilog Viterbi design against System C design

Figure 7a shows the same proof progression as discussed in Subsection 4.1 with respect to the outputs of the FIFO buffers. As in the previous example, the first compare point is $C_1 = (t = 8, d = FF[7 : 0][31 : 0])$. Thereafter, the two state machines are stepped in tandem. The next observables, namely the Trellis Matrix and the Mat-Dec decision table values are computed at the end of the 10th cycle. However, since the butterfly unit has been divided into 4 stages, only 16 values of the Trellis Matrix and 8 values of the MatDec table are obtained at the end of this cycle. The second compare point, therefore, is $C_2 = (t = 10, d = TM[15 : 0][31 : 0], MD[1 : 0][7 : 0])$.

Similarly, the other compare points are $C_3 = (t = 12, d = TM[31 : 16][31 : 0], MD[1 : 0][15 : 8])$,

$C_4 = (t = 14, d = TM[32 : 47][31 : 0], MD[1 : 0][16 : 23])$ and $C_5 = (t = 16, d = TM[47 : 63][31 : 0], MD[1 : 0][23 : 31], Out[31 : 0])$
The decoded output gets computed at the end of the 16th cycle.

The symbolic expressions for $TM_s[63:0][31:0]$ and $TM_v[63:0][31:0]$ are similar to those described in the previous proof subsection. Similarly, the values of the other observables can be symbolically computed and checked with a SAT solver.

It hold be noted that this proof has more compare points than the proof shown in Figure 4 and also requires a sequential progress over more time cycles.

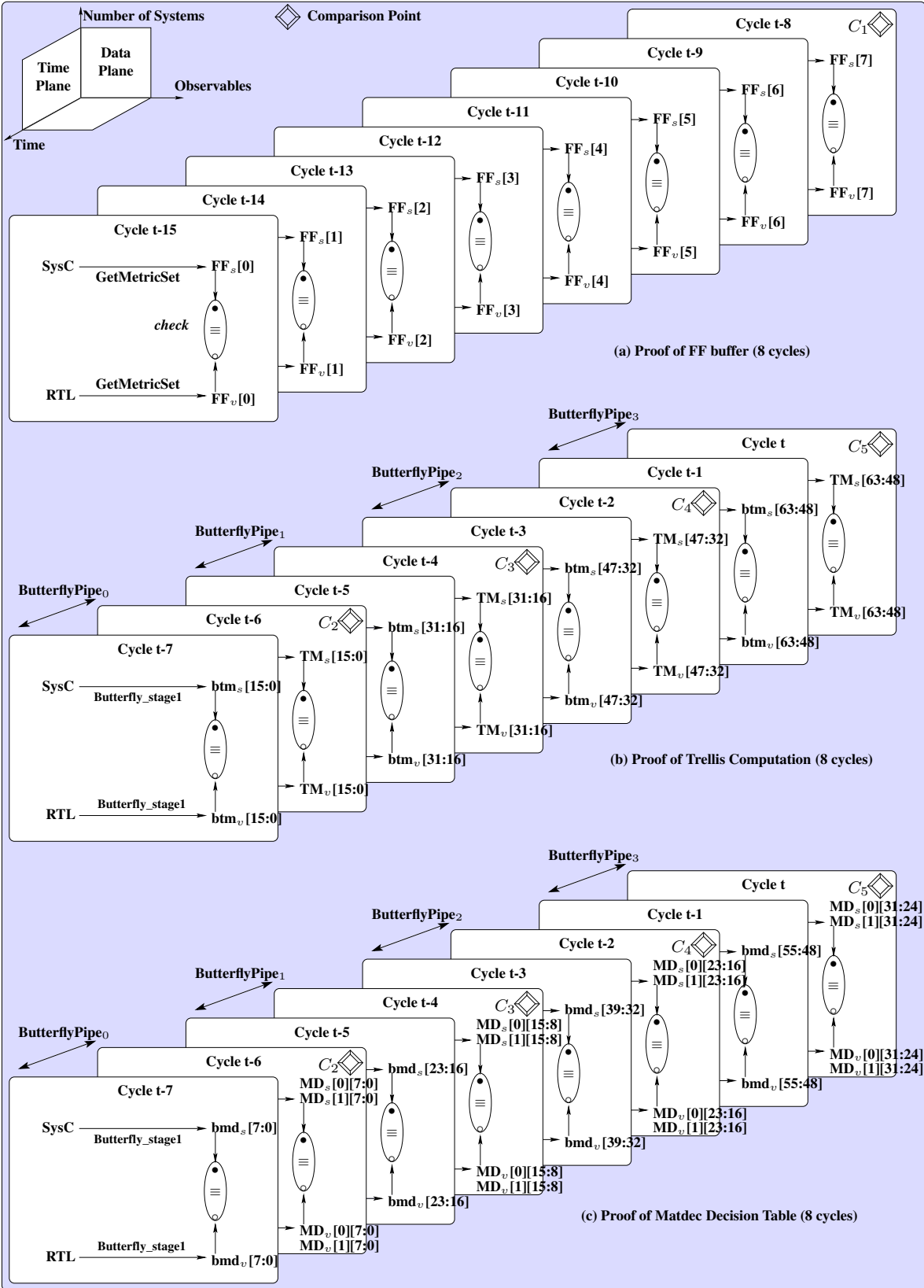


Figure 7. Proof of sequential equivalence checking of pipelined Verilog Viterbi design with area optimizations against System C design

Block/Function	Number of clauses in the CNF formula
PLUS	448
LESSTHAN	32
Trellis Condition in the Butterfly	14336
Trellis computation in each stage of butterfly	28672
Trellis per butterfly	57344
MatDec each stage of butterfly	896
MatDec per butterfly	1792

Table A

Design	Number of clauses in the CNF formula
Monolithic Trellis	1892352
RTL decomposition (Design 1)	59136
RTL decomposition (Design 2)	59136

Table B

Block/Function	Number of variables	Number of Symbolic variables generated
PLUS	64	2
Butterfly	128	66
Trellis (monolithic)	2304	2112
Trellis (decomposed)	128	66

Table C

Figure 8. Breakdown of number of variables and clauses in the CNF input to zChaff for different blocks.

4.3 Experimental Results

We use zChaff [8] as the SAT solver to implement the *check()* function. In order to pass the equivalence checking through zChaff, we had to model the symbolic expressions as a Boolean satisfiability problem. We used the XNOR operation to combine the two target symbolic expressions. In order to provide a glimpse into the complexity and size of our design, we present some relevant statistics in Figure 8. Table A gives a breakdown of number of clauses in the CNF formula for various blocks. PLUS and LESSTHAN were two primary functions used to synthesize the RTL into symbolic boolean expressions. We can observe the benefits of our technique of splitting monolithic equivalence functions into smaller functions. Using our decomposition technique, we created 32 independent CNF formulas, that were input to zChaff. Each of these formulas had 59136 clauses and 128 variables. Without this decomposition, the monolithic Trellis computation would generate a CNF with nearly 1.9 million clauses. An interesting observation is, due to our decomposition methodology, the size of the CNF for the pipelined version is exactly the same as the non-pipelined design. This shows that the equivalence checking problem can be greatly reduced using a high level decomposition, in order to make it easier for lower level engines. These observations are captured in tables Table B and Table C.

5 Discussion and Conclusions

We have shown a novel technique for sequential equivalence checking of a system level specification and its implementation in RTL. Our technique decomposes the equivalence checking problem using automatically computed compare points. We demonstrate the efficiency of our technique using a non-trivial example. In principle, the tech-

nique can be also be applied to RTL to RTL equivalence checking, since it is effective for source to source checking at the higher level. One of the limitations of this technique is that it requires the high level model to be synchronized by a clock. Also, the technique is not scalable in the number of cycles. As the number of cycles gets larger, the size of the expression grows quadratically, causing capacity problems for the lower level SAT engine.

We show our technique using System C as our system level modeling language, due to its recent IEEE standardization. Our technique, however, can be applied to other system level languages also. In its current form, this technique cannot handle pointers, nested loops or other software specific constructs. In future, we plan to extend this work to handle a larger subset of the C-like description language.

Since this work primarily seeks to decompose the problem into more tractable sub-problems, it might be usable in conjunction with existing tools and techniques. The information content at the source code level can be exploited to assist existing technologies to tackle the SoC verification problem. This work motivates the necessity to reason at the higher levels in the design cycle, in order to work toward scalable verification solutions.

References

- [1] Calypto Design Systems <http://www.calypto.com>.
- [2] Digital Radio Mondiale <http://www.drm.org/>.
- [3] System C Reference Manual http://homes.dsi.unimi.it/pedersin/AD/SystemC_v201_LRM.pdf.
- [4] J. A. Abraham, V. M. Vedula, and D. G. Saab. Verifying properties using sequential atpg. *International Test Conference*, pages 194–200, 2002.
- [5] D. Anastasakis, R. Damiano, H.-K. T. Ma, and T. Stanion. A practical and efficient method for compare-point matching.

In *Proceedings of the 39th conference on Design automation*, pages 305–310, 2002.

- [6] J. R. Burch and V. Singhal. Robust latch mapping for combinational equivalence checking. In *ICCAD*, pages 563–569, 1998.
- [7] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311, 2003.
- [8] Z. Fu, Y. Mahajan, and S. Malik. zChaff Solver. In <http://www.princeton.edu/~zchaff/zchaff.html>.
- [9] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen. Verifying sequential equivalence using atpg techniques. *ACM Trans. Des. Autom. Electron. Syst.*, pages 244–275, 2001.
- [10] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371, 2003.
- [11] F. Lu, M. K. Iyer, G. Parthasarathy, L.-C. Wang, K.-T. Cheng, and K.-C. Chen. An efficient sequential sat solver with improved search strategies. In *DATE*, pages 1102–1107, 2005.
- [12] T. Matsumoto, H. Saito, and M. Fujita. An equivalence checking method for c descriptions based on symbolic simulation with textual differences. In *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, pages 3315–3323, 2005.
- [13] C. Pixley. Formal verification of commercial integrated circuits. In *IEEE Design and Test of Computers*, 2001.
- [14] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.
- [15] L. Smria, R. Mehra, B. Pangrle, A. Ekanayake, A. Swright, and D. Ng. Rtl c-based methodology for designing and verifying a multi-threaded processor. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 123–128, 2002.
- [16] J. Tu. Viterbi decoder coprocessor in the drm application soc. http://www.cerc.utexas.edu/~shobha/master_report.pdf 2006.
- [17] C. van Eijk and J. Jess. Detection of equivalent state variables in finite state machine verification, 1995.
- [18] S. Vasudevan, V. Viswanath, R. Sumners, and J. Abraham. Automatic verification of arithmetic circuits in rtl using stepwise refinement of term rewriting systems. In *Accepted in IEEE Transactions on Computers*, To appear in Fall 2006.
- [19] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transactions on Information Theory*, pages 260–269, 1967.
@inproceedings lenk94, author = Stefan Lenk, title = Extended timing diagrams as a specification language, book-title = EURO-DAC '94: Proceedings of the conference on European design automation, year = 1994, pages = 28–33, location = Grenoble, France,

@articlekhordoc98, author = K. Khordoc and E. Cerny, title = Semantics and verification of action diagrams with linear timing, journal = ACM Transactions on Design and Automation of Electronic Systems, volume = 3, number = 1, year = 1998, pages = 21–50, doi = <http://doi.acm.org/10.1145/270580.270582>,