A Compositional Approach to the Combination of Combinational and Sequential Equivalence Checking of Circuits Without Known Reset States

In-Ho Moon Per Bjesse Carl Pixley

Advanced Technology Group
Synopsys Inc.

Abstract

As the pressure to produce smaller and faster designs increases, the need for formal verification of sequential transformations increases proportionally. In this paper we describe a framework that attempts to extend the set of designs that can be equivalence checked. Our focus lies in integrating sequential equivalence checking into a standard design flow that relies on combinational equivalence checking today. In order to do so, we can not make use of reset state or reset sequence information (as this is not given in combinational equivalence checking), and we need to mitigate the complexity inherent in the traditional sequential equivalence checking algorithms. Our solution integrates combinational and sequential equivalence checking in such a way that the individual analyses benefit from each other. The experimental results show that our framework can verify designs which are out of range for pure sequential equivalence checking methods aimed designs with unknown reset states.

1. Introduction

Formal equivalence checking is the standard way of checking whether two designs (for example an RTL description, and a synthesized gate level netlist) have exactly the same input-output behavior. The reduction of this problem to checking whether the designs correspond cone by cone, *combinational equivalence checking*, has been the defacto standard way of deciding this problem for more than a decade. However, due to the increasing pressures to synthesize smaller and faster designs, there is a large incentive to move to a synthesis flow where transformations are utilized that do not preserve a cone-by-cone mapping between the pre- and post-synthesis designs.

In order to make this move, it is necessary to have tools available to check sequential equivalence. While this problem could be solved by model checking [1], there are two difficulties. First of all, the reset states of the pre- and post-synthesis designs are generally not known in the early phases of the design cycle. Second, since the equivalence checking is performed between whole designs, the inherent complexity of formal verification makes it vital to not ignore the known problem structure. In fact, the general sequential equivalence checking problem is so hard that, with the exception of some large semiconductor companies that have developed in-house tools tailored to their particular design styles [4, 6], the industry standard is to either not use sequential optimizations, or to not verify them formally.

In this paper, we present an attempt to integrate sequential equivalence checking into a combinational equivalence checking environment in such a way that we get maximum synergy between the different approaches, while not relying on the presence of reset states. In order to cope with the complexity of the verification problem we attempt to solve as large parts of the designs as possible without doing a full sequential analysis. The use of mature combinational verification engines makes it possible for us to cope with datapaths and other hard to verify parts of the design as long as we do not need to include them in the analysis of the sequential transformations. This is often the case, as many optimizations such as finite state machine (FSM) re-encoding and optimization are local in nature.

We further mitigate the capacity problems by using an abstraction refinement scheme that applies sequential verification on local neighborhoods only. By doing this we make our capacity dependent on how big a portion of the design is left unproven due to sequential optimizations and how much surrounding logic is needed to prove each sequentially optimized point, rather than the design size.

The experimental results show that the resulting combined framework increases the capacity significantly, and that we are able to verify a large number of sequentially optimized industrial designs. All but the simplest of the designs we check are not verifiable using naive sequential equivalence checking due to capacity problems. Our combined verification approach, however, can both prove designs equivalent and diagnose bugs.

The contribution in this paper is twofold. First of all,

978-3-9810801-2-4/DATE07 © 2007 EDAA

we define a notion of equivalence that is compositional, and that allows us to combine results from sequential and combinational analysis. Our equivalence notion is essentially a simpler version of the theoretical framework in [4] that does not rely on the use of manually added properties. We also discuss precisely in what sense our notion of equivalence implies correctness. Second, we describe completely automated tool that leverages our theory to prove equivalence between sequentially optimized designs.

2. Preliminaries

In the remainder of this paper, we will concern ourselves with the equivalence checking of standard synchronous circuits. In particular, we assume that all circuits are free of combinational loops. We use the notation $C(s_0, \pi) = s_1$ to denote that s_1 is the state reached when starting the circuit C in the state s_0, and applying the input sequence π. We use the notation $Output(C, s_0, \pi_i) = \pi_o$ to denote that π_o is the output sequence resulting when starting C in the state s_0, and applying the input sequence π_i.

Let us define an *equivalent state pair* (ESP) for two circuits C_1 and C_2 to be a pair (s_0, s_1) such that $Output(C_0, s_0, \pi) = Output(C_1, s_1, \pi)$ for all input sequences π. We will say that two circuits are *I/O equivalent* from some starting point if they will provide identical output patterns forever, assuming that they are given the same inputs.

We will be concerned with compositional use of equivalence checking. We define a *well partitioned design* to be a partitioning of a particular design's gates and registers into a finite number of sets such that every gate and register belongs to a unique set (the restriction of gates to belong to a unique partition is not strictly necessary as shared gates could be duplicated without changing the semantics of a given design). Inputs of a partition that are not inputs of the overall design are referred to as *pseudo inputs*.

Without loss of generality, we define a *partitioned equivalence checking problem* to consist of two well partitioned circuits together with (1) a one-to-one matching between the partitions, and (2) one-to-one matchings of all partition inputs and outputs. For technical reasons that will become clear in Section 5, we require that matched partition outputs depend combinationally on precisely the same pseudo inputs.

3. Initialization of circuits

When power is connected to a circuit, its state holding elements will have random contents. However, for a circuit to function, it is likely that a number of internal state machines and data registers have to start in particular states, and have known initial contents, respectively. Most circuits will therefore have a means to force the circuit into a known region of operation from the nondeterministic initial state.

One popular notion of initialization is *synchronizability*: A circuit is said to be synchronizable if there is an input sequence π that brings the circuit from all system states to some particular state s_0 (formally, $C(s_i, \pi) = s_0$ for all system states s_i). We will refer to such a state s_0 as an *initial state*. There are numerous relatively cheap heuristics that allow a verification engineer to show that a design is synchronizable; a popular choice is to use three valued simulation to find a sequence that takes the all-X value to some completely specified binary state.

While requiring designers to produce synchronizable circuits would seem attractive, the cost would be excessive in many cases. For example, it would require a large amount of on-chip wiring to force the memory bank in a FIFO into a deterministic state at bootup. Moreover, the implementation could easily be engineered so that the particular values in the empty slots post-reset do not matter. However, if the design let the empty slots contain uninitialized values after reset, it can clearly not be synchronizable. There would hence be advantages to a notion of initialization that allows a given design to be forced into a *set* of states, all of which must guarantee equivalent behavior. This generalization of synchronizability is called *weak synchronizability*. Formally a design is weakly synchronizable if there exists some input sequence π that will take it from an arbitrary power up state to some group of states $\{s_0 \dots s_k\}$ that are equivalent in the sense that $Output(C, s_0, \pi) = Output(C, s_1, \pi) \dots = Output(C, s_k, \pi)$ for all π. Deciding whether a design is weakly synchronizable is more involved than checking synchronizability; the state-of-the-art combines the power of modern SAT technology with the use of heuristics [9].

Weak synchronization sequences have the following interesting property that we will make use of: If two machines both are weakly synchronizable, then there exists an input trace that weakly synchronizes both of them. To see this, assume that π_1 weakly synchronizes C_1 and π_2 weakly synchronizes C_2. Then the concatenation of π_1 and π_2 weakly synchronizes both C_1 and C_2.

4. Combinational Equivalence

If one design has been transformed into another using operations that only change the functionality of combinational logic, it is possible to perform *combinational equivalence checking*:

1. Use heuristics to construct a one-to-one match between inputs, state elements and outputs of the two designs. If inputs, state elements and outputs can not be matched, then fail.

2. For each cone of logic feeding an output or a register in the implementation machine, check whether the

matched cone outputs have the same values under the assumption that the inputs to the cones have the same values. If this is true, we are done, otherwise fail.

If two circuits C_1 and C_2 are declared equivalent by the above (naive) combinational equivalence checking algorithm, then they have at least one equivalent state pair (s_1,s_2). To see this, just construct s_1 and s_2 by assigning zero to every state entry. This choice of s_1 and s_2 clearly respect the state mapping. Moreover, the combinational equivalence check guarantees that states that respect the state mapping produce the same outputs and only transitions to states that respect the state mapping.

The combinational equivalence checking algorithm is correct in the following sense: If (1) the two designs are weakly synchronizable and (2) the algorithm classifies the designs as equivalent, then there exists an input sequence whose application to both machines will guarantee that their subsequent behavior is I/O equivalent forever. The reason for this is simple: We have just showed that there exists an equivalent state pair (s_0, s_1). Since the two machines are weakly synchronizable, then there exists a sequence π that weakly synchronizes both of them. Since (s_0, s_1) is an equivalent state pair, then the state pair (s'_0, s'_1) reached by applying π from (s_0, s_1) is an equivalent state pair. Moreover, since s'_0 and s'_1 are weak syncronization states, any other pair that can be formed from weak syncronization states will be an equivalent state pair.

It is important to realize that the correctness statement for combinational equivalence checking has two antecedents; in particular, a positive result from the equivalence checking tool alone does not guarantee I/O equivalence. If both designs lack a weak synchronizing sequence, then there is no way to force the circuits into states where they behave the same way. As the equivalence checking method that we will present in Section 6 relies on combinational equivalence checking at its heart, it will inherit this trait.

5. Sequential equivalence

Many powerful synthesis transformations such as retiming, FSM reencoding, and FSM optimization will produce a transformed design that can not be checked by combinational equivalence checking—specifically, there will be no matching possible between the pre- and post-optimization netlists. It is then necessary to apply more complex sequential equivalence checking algorithms. However, (1) sequential optimizations are often only applied to local portions of the design, and the rest of the design may be combinationally equivalent, and (2) there may exists a pairing of combinationally failing regions that allows us to show each pair sequentially equivalent.

Our starting point is thus that the user has not provided us with reset state information (just like in combinational equivalence checking), and that we need to come up with a framework that allows us to combine results from combinational verification with sequential analysis results in a compositional way.

Unfortunately, there are problems with the standard approaches to defining sequential equivalence in the absence of reset states. Some notions, like alignability [8], are not compositional in their nature—equivalence of parts does not imply equivalence of the top-level designs. Other notions of equivalence such as safe replacement [10] are compositional, but much more expensive to compute, as every state of the two systems has to be examined. Moreover, combinational equivalence does not imply alignability or safe replacement.

Our solution is to define our notion of sequential equivalence in the following way:

Definition 1. *Two circuits are sequentially equivalent precisely when they have a nonempty set of equivalent state pairs (nonempty ESP).*

This equivalence definition has several nice properties that make it preferable to previous attempts at defining equivalence in the absence of reset states.

First, we can compute the equivalent state pairs of two designs using BDD-based techniques exactly in the same way as it is done in the first step of alignability computation [8]. Our complexity is hence guaranteed to be bounded by the complexity of checking alignability.

Second, combinational equivalence implies our notion of sequential equivalence: As shown in Section 4, if a combinational equivalence checking algorithms classifies two designs as equivalent, they have at least one equivalent state pair.

Third, we will now show that if we have a well-partitioned equivalence checking problem between circuits C_1 and C_2 and each partition has a nonempty set of equivalent state pairs, then the overall designs have at least one equivalent state pair.

Lemma 1. *The partition outputs of a well-partitioned design can be sorted so that there are no combinational paths between later and earlier entries in the ordering.*

Proof. Take $O_1 \prec O_2$ to mean that the partition output O_2 is combinationally dependent on O_1. It is clear that $\prec$ is a partial order, as a mutual combinational dependency between two partition outputs would imply that the circuit has a combinational cycle. Any finite partial order can be extended to a total order by a topological sort, so there exists a total order $<$ such that $O_i < O_j$ implies that O_i is not combinationally dependent on O_j. □

By definition, matched partition outputs in our partitioned equivalence checking problems depend combinationally on precisely the same matched pseudo inputs. We

can thus lift the ordering to pairs of matched partition outputs.

Theorem 1. *If all partitions in a partitioned equivalence checking problem for C_1 and C_2 have nonempty set of equivalent state pairs, then C_1 and C_2 have an equivalent state pair.*

Proof. Pick an equivalent state pair for each matched partition. Construct our candidate state pair (s_0, s_1) so that the projection of this state onto each partition pair gives us the picked partition state pair. This is possible as all partitions are nonoverlapping.

Assume that C_1 and C_2 are in a state pair that is comprised by equivalent state pairs for all partitions, and that the matched inputs to C_1 and C_2 have the same values. Also assume that there exists a smallest partition output pair (O_1, O_2) that does not have matching values. The partition output pair will be combinationally dependent on some real circuit inputs, some state elements, and some pseudo inputs. As the circuit inputs match and the partitions are in an equivalent state pair, some pseudo input pair that O_1 and O_2 are combinationally dependent on must not match. But this is impossible as (O_1, O_2) is the smallest mismatched partition output pair. All partition output pairs hence have matching values (and in particular, all *circuit* outputs match). Moreover, all pseudo inputs to all partitions match, so the next state of each partition is an equivalent state pair. □

Note that Theorem 1 does not require that partitions can be ordered so that later partitions do not feed earlier partitions; we only make use of the fact that the designs have no combinational loops. Also note that our notion of sequential equivalence implies circuit correctness in precisely the same sense (and using the same argument) as combinational equivalence checking: If two designs are weakly synchronizable, a classification of them as equivalent entails that there exists an input sequence that can be used to force I/O equivalence of the two circuits. As a consequence, a positive verification result needs to be accompanied by separate checks of weak synchronizability for the two designs, just like in the case of pure combinational equivalence checking. When restricted to weakly synchronizable designs, our notion of equivalence is a reflexive, transitive and symmetric relation that implies alignability (this follows from Theorem 3 and Theorem 9 of [8]).

6. Implementation

We have implemented a general framework that extends a state-of-the-art industrial combinational equivalence checker to handle sequential optimizations. Figure 1 shows the general structure of our algorithm. The upper dashed region performs combinational equivalence checking and the lower dashed region is concerned with sequential analysis. We make use of design information such as the location of counters, FSMs, datapath registers, and memory registers. This information is extracted in the front end of our tool. Our heuristic for performing sequential analysis on combinationally failing regions is very naive (and we are working on more elaborate schemes); however, it has worked surprisingly well in practice.

Our overall algorithm alternates between cheap combinational verification and a more expensive sequential analysis that checks whether the region around the matched but currently failing compare points has a nonempty set of equivalent state pairs. If we have successfully analyzed all output compare points, we know that the designs as a whole have a nonempty set of equivalent state pairs, due to the results in Section 5. We refer to the high level structure of our computation as the *outer loop*.

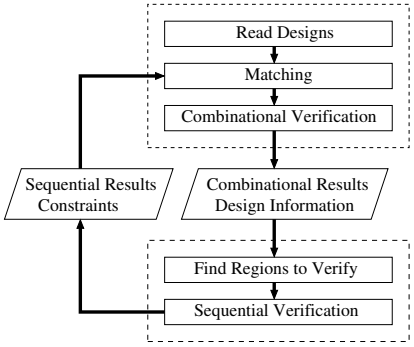

Figure 1. Outer loop.

6.1. Sequential analysis

Our implementation uses a modified BDD-based equivalent state pair computation algorithm that can deal with don't care constraints encoded in the design description [7]. This modification is necessary to be able to process many of the designs that we have encountered.

Our sequential analysis makes use of a simple abstraction refinement scheme. We form an initial verification output boundary by making our output pair set contain the current set of matched and combinationally failing compare points. Given the output boundary, we form an abstraction by including all the fanin logic and registers up to the closest boundary containing only verified compare points.

Next, we attempt to compute the set of equivalent state pairs.

If the resulting set of equivalent state pairs is nonempty, we mark all the output compare points of the region as verified.

Alternatively, if the computation results in an empty set of equivalent state pairs, we grow the current region back-

wards to include more context and repeat the equivalent state pair computation. Our strategy for growing a given region is coarse: If a matched FSM or counter register is feeding the current region, we include the complete FSM or counter immediately; we also expand the current inputs to the region by the inclusion of all fanin logic and registers up to the next excluded matched and verified compare points. However, we do not expand a current input backwards if doing so would include datapath registers or memory registers.

Finally, if the equivalent state pair computation times out, we see if we can move our output boundary forward by including the next set of compare points, and then form a new abstraction and start the process over. In order to curb the computation time, we only attempt to move our output boundary forward a single time in each sequential verification pass.

6.2. Use of sequential information

After each sequential verification pass, we construct a new combinational verification problem as follows. For every pair of sequentially equivalent compare points (either primary outputs or registers), we compute the set of unmatched or combinationally failing registers that transitively drive only sequentially equivalent compare points, and remove these registers from consideration. Compare points that were diagnosed as sequentially failing could be artifacts of a design bug, but is more likely to be the symptom of an incorrect matching. We thus use this information to refine our matching of registers.

If FSMs or counters have been identified and matched in both designs by the combinational equivalence checking tool, then we use the symbolic characterization of their equivalent state pairs to compute combinational don't care constraints over the partition outputs. The constraints are then used in the next iteration of combinational verification (the extension of our theory to handle this is straight forward, but we omit it for space reasons).

7. Experimental Results

Table 1 shows our experimental results, generated on a 1.4 Ghz Intel processor machine with 4 Gb memory running Red Hat Linux 7.2. In the table, the columns R_s and R_i show the number of registers in specification and implementation designs. The two columns (I and O) show the number of inputs and outputs in the design. Next, the columns C_f and C_t contain the number of combinationally failing compare points in the first round of combinational verification, and the number of total compare points. Note that the total number of compare points is not necessarily the same as the sum of primary outputs and registers in the design since we can have compare points at hierarchical boundaries and some registers could be left unmatched. Finally, the two columns T_c and T_s shows the total time in seconds spent during combinational equivalence checking and sequential equivalence checking respectively.

D	R_s	R_i	I	O	C_f	C_t	T_c	T_s
D1	28	21	17	8	21	39	4	3
D2	33	55	12	9	23	29	5	886
D3	53	50	29	31	18	79	8	3
D4	58	106	25	28	17	77	3	192
D5	104	98	9	30	68	102	20	82
D6	175	176	17	10	3	189	12	3
D7	216	216	31	5	14	247	8	1
D8	228	227	59	66	1	186	9	81
D9	259	259	118	78	1	375	34	20
D10	296	288	216	35	33	489	21	7
D11	6840	6840	399	312	558	7239	776	4654
D12	7265	7260	262	252	16	7490	213	120

Table 1. Experimental results.

Among the 12 designs, we have found two real bugs in $D2$ and $D12$, and successfully verified all others. As indicated by the number of initially failing compare points, combinational equivalence checking alone cannot handle any of the designs. All designs except $D11$ are proved using a single iteration of the outer loop; $D11$ needs two iterations. Pure top-level sequential analysis through alignability checking without any combinational verification could only verify $D1$ and $D3$ within 24 hours. However, $D1$ was easy enough that pure sequential checking took less time than running our combination method, and our method was about 18X faster than a pure sequential analysis in the case of $D3$. We also tried our implementation of the SAT-based alignability checking from [3] for the 10 harder designs, but we could not solve any of them within 24 hours.

8. Related work

The most related work to ours is Khasidashvili and coworker's framework at Intel for alignability-based compositional equivalence checking [4]. However, there are a number of differences between our approach and theirs. First of all, the work in [4] is centered around the use of manually added verification properties, whereas we focus on a flow that is completely automatic. Second, the framework in [4] uses computationally expensive alignability computations to check all subparts, whereas we apply combinational techniques wherever possible. Moreover, our notion of equivalence for a partition is easier to compute than alignability as we do not need the aligning sequence computation. Our theory is essentially a simplified version of the theory in [4] that allows the use of combinational results, while taking away the use of verification

properties.

In independent research to the work presented here, Khasidashvili and coworkers recently presented a revised equivalence checking method that aim to integrate combinational and sequential equivalence checking in a compositional manor [5]. In contrast to our approach, the work in [5] presupposes knowledge of a sequence that is used to force a design into a set of post-reboot "good" states. As our method is aimed at integration in an traditional combinational equivalence checking environment, we do not require this information (but we could use it to prove weak synchronizability or I/O equivalence in a compositional manor, if we had it available). However, if a reboot sequence is known we believe that the most computationally efficient course of action is to compute reset states and use standard model checking algorithms, rather than relying on modifications of equivalence notions aimed at designs with unknown reset states.

There are other notions of sequential equivalence in the absence of reset states, such as variations of safe replacability [10]. However, as our notion of equivalence is easier to compute than alignability, it is likely to be easier to compute than safe replacibility for the reasons outlined in [4]. Moreover, as top-level alignability implies nonempty ESP, and safe replacement for weakly synchronizable designs implies alignability [3], we can show at least as many weakly synchronizable designs equivalent as safe replacement checking. Finally, combinational equivalence is compositional with our notion of equivalence. This is not the case for safe replacement.

Sequential equivalence checking of industrial designs have been addressed in a number of other papers. Specifically, Mony and coworkers at IBM have developed a framework for equivalence checking based on applying verification transformations [6], Huang and coworkers have constructed a sequential equivalence checker called AQUILA that uses ATPG and BDD techniques [2], and Stoffel et al. have implemented a system that uses a representation of state spaces based on the queue of transformations that merge points [11]. Both AQUILA and the IBM equivalence checker use abstraction refinement and make use of combinational verification. However, all of the three previous approaches rely on the knowledge of the reset states of the checked designs, and are hence not applicable to the problem that we are trying to solve.

9. Conclusions

We have presented a general framework for sequential equivalence checking of designs without a known reset state that integrates combinational and sequential analysis seamlessly. Our focus is to be able to show as many designs as possible equivalent without having to resort to manual intervention from the designer.

The experimental results showed that our proposed framework was able to verify many industrial designs that are not combinationally equivalent, while being out of range for pure sequential equivalence checking methods that do not rely on reset information.

As future work, we are investigating more efficient refinement strategies since our current strategy is excessively greedy. We are also looking into ways of coping with the result of global sequential transformations such as retiming.

References

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

[2] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, F. Brewer, and C.-Y. Huang. AQUILA: An equivalence checking system for large sequential designs. *IEEE Transactions on Computers*, 2000.

[3] Z. Khasidashvili and Z. Hanna. SAT-based methods for sequential hardware equivalence verification without synchronization. *Electronic Notes in Theoretical Computer Science*, 89(4):593–607, 2003.

[4] Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints. In *Proceedings of the International Conference on Computer-Aided Design*, pages 58–65, San Jose, CA, Nov. 2004.

[5] Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna. Post-reboot equivalence and compositional verification of hardware. In *Formal Methods in Computer Aided Design*, 2006.

[6] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. Exploiting suspected redundancy without proving it. In *Design Automation Conference*, 2005.

[7] I.-H. Moon, P. Bjesse, and C. Pixley. Practical issues in sequential equivalence checking through alignability: Handling don't cares and generating debug traces. In *IEEE International High Level Design Validation and Test Workshop*, 2006.

[8] C. Pixley. A theory and implementation of sequential hardware equivalence. *IEEE Transactions on Computer-Aided Design*, 11(12):1469–1478, Dec. 1992.

[9] A. Rosenmann and Z. Hanna. Alignability equivalence of synchronous sequential circuits. In *International Workshop on High Level Design Validation and Test*, pages 111–114, Cannes, France, Oct. 2002.

[10] V. Singhal, C. Pixley, A. Aziz, and R. K. Brayton. Theory of safe replacements for sequential circuits. *IEEE Transactions on Computer-Aided Design*, 20(2):249–265, Feb. 2001.

[11] D. Stoffel, M. Wedler, P. Warkentin, and W. Kunz. Structural FSM traversal. *IEEE Transactions on Computer-Aided Design*, 23(5):598–619, May 2004.