Robust Latch Mapping for Combinational Equivalence Checking

Jerry R. Burch and Vigyan Singhal
Cadence Berkeley Labs

Abstract

Existing literature on combinational equivalence checking concentrates on comparing combinational blocks and assumes that a latch mapping (register mapping) has already been constructed. We describe an algorithm for automatically constructing a latch mapping. It is based on the functionality of the circuits being compared rather than on heuristics. As a result, if two circuits are combinationally equivalent, then our algorithm is guaranteed to find a latch mapping. Our empirical results show that the method is practical on large circuits.

1. Introduction

When applied to a pair of sequential circuits, combinational equivalence checking typically consists of two steps. The first step is to construct a latch mapping (also known as a register mapping). This identifies corresponding latches in the two designs to be compared. It is then possible to break the circuits into corresponding combinational blocks. The second step is to verify whether the corresponding combinational blocks are equivalent. If so, then the circuits are functionally equivalent.

The research literature has many papers on the second step. Kuehlmann and Krohm [5] and Pradhan et al. [6] are recent examples. Their lists of references contain many more examples. However, there is little in the literature on the first step, where a latch mapping is constructed [1, 2, 4].

Existing commercial tools use heuristics based on signal names or circuit structure to construct a latch mapping. If two combinational blocks are found to be inequivalent, it may be because of an incorrect latch mapping rather than a bug in the circuit. This complicates debugging. Tools that do design transformations, such as synthesis or clock tree insertion, often do not preserve signal names, especially when a design is being flattened as part of the transformation.

We describe an algorithm based on the functionality of the circuits being compared rather than on heuristics. The method produces a legal latch mapping if and only if one exists. If a circuit bug exists, our method provides a predictable, interactive procedure for isolating a combinational block which has a bug.

Our empirical results show that the method is practical on large circuits. The total CPU time required for latch mapping and equivalence checking is similar to that required by a combinational equivalence checking algorithm that assumes that a latch mapping has already been provided.

We also show how the method can be generalized to ternary circuit models. This allows an implementation to be compared to a specification with don't cares. Finally, we describe the mathematical relationship between our latch mapping algorithm and sequential equivalence checking. This relationship can be helpful for developing formal verification methods that combine some of the strengths of both combinational and sequential equivalence checking.

1.1 Related Work

The latch mapping algorithm described by van Eijk [2] is quite similar to the fixed point computation we have developed (sections 3 and 5). However, van Eijk does not show how bugs can be localized in incorrect circuits. In fact, the behavior of the algorithm on inequivalent circuits is not discussed at all. Finally, van Eijk does not consider the case where don't care conditions must be used to show equivalence.

van Eijk [3] described a method that combines some of the strengths of combinational and sequential equivalence checking.

1.2 Organization

Section 2 describes the circuit models we use. Section 3 describes how we represent latch mappings and gives a high-level description of our fixed-point algorithm for constructing latch mappings. The relationship between latch mappings and reset states is discussed in section 4. Section 5 gives more details about the implementation of our latch mapping algorithm while section 6 shows how circuit bugs can be localized when automatic latch mapping is used. Section 7 gives empirical results. Sections 2 through 7 are based on a Boolean circuit model. Section 8 shows how our techniques can be extended to handle ternary models and don't care conditions. Finally, section 9 discusses the relationship between combinational and sequential equivalence checking and section 10 concludes.

2. Circuit Models

We consider two circuits, the implementation circuit and the specification circuit. To simplify the exposition, we assume that the circuits have a single clock, and have the same inputs I and outputs O. Let n_I and n_O be the number of inputs and outputs, respectively.

We refer to the state holding elements of the circuits as latches. Let L_{Impl} and L_{Spec} be the set of latches of the implementation and the specification, respectively. Let $L = L_{Impl} \cup L_{Spec}$ be the combined set of latches. Let S be a combined state, i.e. $S: L \rightarrow \{0, 1\}$. We write S_{Impl} and S_{Spec} to denote the projection of S onto L_{Impl} and L_{Spec}, respectively. Thus, S_{Impl} and S_{Spec} represent the states of the individual circuits while S represents their combined state. We write $S_{Impl} \cup S_{Spec}$ to denote the recombining of S_{Impl} and S_{Spec} to form the combined state S.

Let δ_{Impl} be the transition function of the implementation, i.e., given an input vector I and a state S_{Impl}, $\delta_{Impl}(S_{Impl}, I)$ is the next state. The transition function of the specification, δ_{Spec}, is analogously defined. The combined transition function δ is the result of combining δ_{Impl} and δ_{Spec}: given an input vector I and a combined state S, $\delta(S, I)$ is the next combined state.

Let γ_{Impl} be the output function of the implementation: $\gamma_{Impl}(S_{Impl}, I)$ is the output vector O for the implementation state S_{Impl} and input vector I. The output function γ_{Spec} of the specification is analogously defined.

Our latch mapping algorithm does not require that the user provide a reset state (or states) for the circuits being compared. However, reasoning about the relationship between combinational and sequential equivalence checking requires considering the reset states of the circuits, even if all circuit states are considered possible resets states. Let R_{Impl} and R_{Spec} be the set of reset states of the implementation and the specification, respectively (we do not distinguish between a predicate and the set of all objects that satisfy the predicate).

It is standard to define a predicate P_{Impl} on implementation states to be *inductive* if

$$\forall S_{Impl} [R_{Impl}(S_{Impl}) \Rightarrow P_{Impl}(S_{Impl})]$$

and

$$\forall S_{Impl} \forall I [P_{Impl}(S_{Impl}) \Rightarrow P_{Impl}(\delta_{Impl}(S_{Impl}, I))].$$

It is easy to show that if P_{Impl} is inductive, then it contains all reachable implementation states.

In this paper, we often use predicates on combined states rather than on implementation states. Thus, we must consider the reset states of both the implementation and the specification. As a result, the notion of an inductive predicate (which considers only the reset states of one state machine) is not adequate. We address this issue in two steps. First, we define a predicate P on combined states to be *semi-inductive* if

$$\forall S \forall I [P(S) \Rightarrow P(\delta(S, I))].$$

This corresponds to the second condition in the definition of an inductive predicate. Second, we define the following *reset condition* on a predicate P on combined states:

$$\forall S_{Impl} \in R_{Impl} [\exists S_{Spec} \in R_{Spec} [P(S_{Impl} \cup S_{Spec})]].$$

Typically, combinational verification tools determine whether the implementation is equivalent to the specification. However, the specification may have more reset states (and, therefore, more possible execution traces) than the implementation. In this case, the implementation could be safely substituted for the specification, although the two circuits are not equivalent. To handle this situation, we formalize what it means for the implementation to *conform to* the specification. First, however, an auxiliary definition is required, as follows.

The implementation and the specification are *sequentially equivalent for the initial combined state* S_0 iff for all input vectors I and all combined states S reachable from S_0

$$\gamma_{Impl}(S_{Impl}, I) = \gamma_{Spec}(S_{Spec}, I).$$

Let P be the set of all states S_0 such that the implementation and the specification are sequentially equivalent for the initial combined state S_0. The implementation *conforms to* the specification iff P satisfies the reset condition.

Finally, a predicate P on combined states *forces output equality* iff

$$\forall S \forall I [P(S) \Rightarrow \gamma_{Impl}(S_{Impl}, I) = \gamma_{Spec}(S_{Spec}, I)].$$

Theorem 1: The implementation conforms to the specification if and only if there exists a predicate P that is semi-inductive, forces output equality and satisfies the reset condition.

Proof: The forward implication follows by letting P be the set of all combined states S such that the implementation and the specification are sequentially equivalent for the initial combined state S. The reverse implication follows from the definitions using induction on the length of sequences of input vectors. **QED.**

3. Latch Mappings

We represent latch mappings with a predicate M. The latches l_0 and l_1 are mapped together iff $M(l_0, l_1)$ is true. We do not require that M be one-to-one; it can be an arbitrary equivalence relation. This allows, for example, two latches in the implementation to be mapped to a single latch in the specification (see section 4). We say a combined state S *satisfies* M if any two latches that are mapped together by M have the same value in S. Let P_M be the set of combined states S that satisfy M:

$$P_M(S) \Leftrightarrow \forall l_0 \forall l_1 [M(l_0, l_1) \Rightarrow S(l_0) = S(l_1)].$$

Several of the properties we defined for predicates are extended to latch mappings, as follows. A latch mapping M is *semi-inductive* if the predicate P_M is semi-inductive. A

mapping M *forces output equality* if P_M forces output equality. Finally, a mapping M satisfies the *reset condition* if P_M satisfies the reset condition.

We say that M_1 is a *refinement* of M_0 (also written $M_1 \subseteq M_0$) if

$$\forall l_0 \; \forall l_1 \; [M_1(l_0, l_1) \Rightarrow M_0(l_0, l_1)].$$

If $M_1 \subseteq M_0$, then the set of combined states that satisfy M_1 is larger than the set of combined states that satisfy M_0, as in the following theorem.

Theorem 2. If $M_1 \subseteq M_0$, then $\forall S \; [P_{M0}(S) \Rightarrow P_{M1}(S)]$.

We construct a latch mapping by starting with the mapping M_0 that maps every latch to every other latch:

$$\forall l_0 \; \forall l_1 \; M_0(l_0, l_1).$$

We then iteratively refine this mapping until it is semi-inductive. We formalize this process using a function Φ that takes a mapping M_n and returns a mapping M_{n+1} that is a refinement of M_n. The following definition does not completely specify Φ; there are many possible Φ that will do the job. We say Φ is a *refining function* iff both of the following two conditions hold.

1. M is a fixed point of Φ (that is, $M = \Phi(M)$) iff M is semi-inductive.
2. Φ is monotonic, which means that if $M_1 \subseteq M_0$, then $\Phi(M_1) \subseteq \Phi(M_0)$.

We know that the greatest fixed point of Φ exists and can be computed in a finite number of steps because Φ is monotonic and because there is a finite number of latches. The following theorem shows how the latch mapping that is the greatest fixed point of Φ can be used to determine whether the implementation conforms to the specification.

Theorem 3. If M is a fixed point of a refining function Φ, forces output equality and satisfies the reset condition, then the implementation conforms to the specification.

Proof: By condition 1 of the definition of a refining function, M is semi-inductive. Since M is semi-inductive, forces output equality and satisfies the reset condition, we know that P_M is semi-inductive, forces output equality and satisfies the reset condition. Thus, the result follows from Theorem 1. **QED.**

Using Theorem 3, we can verify that the implementation conforms to the specification, as follows. First, compute the greatest fixed point M* of Φ (see section 5 for more details on how this is done). Confirming that M* forces output equality is simply a matter of using combinational equivalence checking on the combinational blocks driving the primary outputs of the implementation and the specification (M* provides the mapping between the latches that are local inputs to these combinational blocks). All that remains is to check that M* satisfies the reset condition, which is discussed in section 4.

The previous theorems are applicable when the implementation conforms to the specification. We must also consider the case where we do not have conformance.

Theorem 4. Let M* be the greatest fixed point of a refining function Φ. Assume that M* does not force output equality. Then there does not exist a latch mapping M that is semi-inductive and forces output equality. Thus, by Theorem 1, the implementation does not conform to the specification.

Proof: We prove the theorem by assuming there is such an M and then showing that this leads to a contradiction. By condition 1 of the definition of a refining function, any semi-inductive mapping must be a refinement of M*. Thus, $M \subseteq M^*$. By Theorem 2, P_M is weaker than P_{M^*}. Clearly P_{M^*} does not force output equality, so P_M also does not force output equality. This contradicts the assumption that M forces output equality. **QED.**

Theorems 3 and 4 give us the following method for checking conformance. First, automatically construct the greatest fixed point M* of some refining function Φ for the circuits. If M* does not force output equality (which is checked using automatic combinational equivalence checking), then conformance cannot be shown using combinational methods. Section 6 describes how to localize bugs in the implementation in this case. If M* does force output equality, then conformance can be shown by showing that M* satisfies the reset condition, as discussed in the following section.

4. Satisfying the Reset Condition

Assume a latch mapping M is a fixed point of a refinement function Φ, and that M forces output equality. We want to check whether M satisfies the reset condition, in order to use Theorem 3 to show conformance.

Typically, our equivalence checking tool is used without the user providing any information about reset states. In this case, the sets of reset states R_{Impl} and R_{Spec} contain all possible states of their respective circuits. It is easy to show that the reset condition is satisfied if there are no two implementation latches that are mapped together by M. Two implementation latches are mapped together by M only if they are redundant (that is, one is functionally a copy of the other).

In the less common case where there are two implementation latches l_0 and l_1 that are mapped together by M, it is possible for the implementation to start in a state S_{Impl} where l_0 and l_1 do not have the same value. Then there is no specification state in R_{Spec} that can be used to satisfy the reset condition. There may also be no start state that results in the same sequential behavior for the specification as S_{Impl} does for the implementation. This would mean that the implementation does not conform to the specification unless the reset states of the implementation are restricted to have l_0 and l_1 with the same value.

In this situation (where the user has provided no information about reset states and there are redundant implementation latches), our latch mapping algorithm can provide constraints on the reset states of the implementation (that is,

pairs of redundant implementation latches that must have the same initial value). If these constraints are satisfied, then conformance is guaranteed (given that the tool has already been used to check the assumptions at the beginning of this section).

Sometimes these constraint are stronger than necessary. They can be weakened by refining M in such a way that it is still a fixed point of Φ and forces output equality, and that l_0 and l_1 are no longer mapped together. The worst case complexity of searching for such refinements appears to be exponential. Although it is beyond the scope of this paper, there are methods that are likely to work well in practice. van Eijk [2] also considered methods for searching for such refinements.

5. Implementation Issues

Our latch mapping algorithm involves computing the greatest fixed point M* of a refining function Φ (see section 3). This section describes how latch mappings are represented and how the function Φ is computed.

A latch mapping M is an equivalence relation between latches. Thus, it can be represented efficiently as a partition of the set of latches. It is easy to design the data structure so that equivalence classes can be split efficiently.

Given a latch mapping M_n, we need to compute $M_{n+1} = \Phi(M_n)$, where Φ is a refining function as defined in section 3. This can be done with the following algorithm.

1. Randomly produce a combined state S that satisfies M_n. This can be done by randomly assigning Boolean values to the equivalence class of M_n, and then constructing S so that every latch l has the value assigned to the equivalence class that l is a member of. Also, randomly produce an input vector I.
2. Compute $S_1 = \delta(S, I)$. If S_1 satisfies M_n, go to step 4.
3. Construct M_{n+1} such that for all l_0 and l_1, $M_{n+1}(l_0, l_1)$ if and only if $M_n(l_0, l_1)$ and $S_1(l_0) = S_1(l_1)$. Clearly $M_{n+1} \subseteq M_n$. Because S_1 does not satisfy M_n, we know that $M_n \neq M_{n+1}$. Exit and return M_{n+1}.
4. For all pairs of latches l_0 and l_1 such that $M_n(l_0, l_1)$
   a. Use combinational equivalence checking to show whether the combinational blocks that drive l_0 and l_1 are equivalent for all combined states that satisfy M_n.
   b. If the blocks are equivalent, continue the loop with the next pair of latches.
   c. Otherwise, the combinational equivalence checker will produce a counter-example: a combined state S and an input vector I such that $\delta(S, I)(l_0) \neq \delta(S, I)(l_1)$.
   d. Compute $S_1 = \delta(S, I)$ and go to step 3.
5. M_n is semi-inductive. Set $M_{n+1} = M_n$. Exit and return M_{n+1}.

Using transitivity of equality, step 4 can be optimized so that the number of iterations is linear in the number of latches, rather than quadratic. In steps 1 and 2, more than one randomly chosen S and I can be tried before continuing to step 4.

The fixed point algorithm for latch mapping repeatedly calls the above algorithm for evaluating a refining function Φ. It records the sequence of the M_n that are produced and also records the sequences of the S and the I that were computed during each evaluation of Φ.

6. Debugging

So far, we have concentrated on the case where there exists a latch mapping M that is semi-inductive and forces output equality. If the implementation conforms to the specification (and if this can be demonstrated using combinational verification techniques), then our algorithm is guaranteed to find such an M. However, if there is no such M, we must give debugging information to the user. This section describes an interactive procedure that can be used to localize a bug in the implementation. If there is more than one incorrect combinational block in the implementation, the procedure will home in on one of the incorrect blocks.

The algorithm described in section 5 produces a length k sequence of latch mappings M_n, each a refinement of its predecessors. It also produces length k-1 sequences of combined states S_n and input vectors I_n. For all n between 0 and k-2 (inclusive) $P_{Mn}(S_n)$ holds and

$$\forall l_0\ \forall l_1\ [(M_n(l_0, l_1) \wedge \neg M_{n+1}(l_0, l_1)) \Rightarrow$$
$$(\delta(S_n, I_n)(l_0) \neq \delta(S_n, I_n)(l_1))].$$

In addition, M_{k-1} must be semi-inductive. Since we are assuming that there is no latch mapping that is semi-inductive and forces output equality, there must exist an output o, an input vector I_{k-1}, and a combined state S that satisfies M_{k-1} such that

$$\gamma_{Impl}(S_{Impl}, I_{k-1})(o) \neq \gamma_{Spec}(S_{Spec}, I_{k-1})(o).$$

Let S_{k-1} be the combined state S, above. The procedure for isolating a bug is as follows

1. The user is shown the combinational blocks that drive o in the implementation and the specification, and shown the latches that are local inputs to those blocks. The user must choose which of these latches should be mapped together. The number of latches being considered at this step is typically much smaller than the total number of latches, so deciding which of them should be mapped together is much easier than manually mapping the full circuits.
2. If all the latches that should be mapped together have the same value in S_{k-1}, then there must be a bug in the combinational block driving o in the implementation. A bug has been localized to a particular combinational block, and the user can continue with standard debugging techniques.

.

	Latches		Redundant			EQC Calls			Time (sec.)		
Name	**Impl**	**Spec**	**Impl**	**Spec**	**Outputs**	**Given**	**Build**	**Diff**	**Given**	**Build**	**Diff**
D1	1368	1435	114	181	96	1392	1731	24%	1166	1373	18%
D2	1697	1697	0	0	202	1899	1925	1%	1453	1887	30%
D3	932	932	0	0	447	1318	1649	25%	3151	4634	47%
D4	1789	2206	50	467	641	2760	3223	17%	2984	3469	16%

Table 1: Empirical results

3. Otherwise, the user chooses latches l_0 and l_1 that should be mapped together but that have different values assigned to them in S_{k-1}.
4. The tool automatically finds the largest integer n such that l_0 and l_1 are mapped together in M_n.
5. Analogous to step 1, the user is shown the latches that are local inputs to the combinational blocks driving l_0 and l_1, and decides which should be mapped together. If all the latches that should be mapped together have the same value in S_n, then there must be a bug in the combinational block driving either l_0 or l_1 (depending on which is in the implementation). A bug has been localized to a particular combinational block, and the user can continue with standard debugging techniques.
6. Otherwise, there are latches l_2 and l_3 that are local inputs to the combinational blocks driving l_0 and l_1, and that should be mapped together, but that have different values in S_n.
7. Assign l_2 and l_3 to l_0 and l_1, respectively. Go to step 4.

The above procedure is guaranteed to terminate since all latches are mapped together in M_0. After it terminates, the user has been guided to a combinational block in the implementation that has a bug. During the procedure, the user must manually determine some of the latches that should be mapped together.

7. Empirical Results

Our empirical results are summarized in Table 1. We did experiments on four proprietary commercial circuits. In each case, the specification is RTL and the implementation is synthesized gates. The table gives the number of latches, redundant latches and outputs. We compare runs where the latch mapping was given before the run vs where the latch mapping was constructed by our algorithm as part of the run. We compare both the CPU time and the number calls to the core combinational equivalence checker. The columns labeled "Diff" give the percent increases from the "Given" run to the "Build" run

For circuit pairs D1 and D2, the implementation conforms to the specification and every latch is mapped to one or more latches in the other design. For circuit pair D3, there is a mismatch. For circuit pair D4, there are specification latches that are not mapped to any implementation latch. However, all of these unmapped latches are unobservable, so the implementation conforms to the specification.

The results show that the increase in CPU time required for using our algorithm to construct the latch mapping is quite reasonable, averaging 28%. We believe this a worthwhile price to pay for the extra convenience and robustness provided by our method.

8. Don't Cares and Ternary Latch Mappings

It is important to be able to take don't care conditions into consideration when comparing two circuits. These don't cares are typically represented by using X's in the specification description. This section describes how our latch mapping algorithm can be extended to work with don't cares and ternary values.

Define a partial order $\subseteq^X$ on {0, 1, X} such that

$$u \subseteq^X v \;\Leftrightarrow\; u = v \vee v = X.$$

The inputs, outputs, latches, transition functions, output functions and reset states of the implementation and the specification are the same as the Boolean case, except they are extended to ternary values. In addition, the reset condition is the same as in the Boolean case.

The implementation conforms to the specification for the initial combined state S_0 iff for all input vectors I and all combined states S reachable from S_0

$$\gamma_{Impl}(S_{Impl}, I) \subseteq^X \gamma_{Spec}(S_{Spec}, I).$$

Let P be the set of all states S_0 such that the implementation conforms to the specification for the initial combined state S_0. The implementation conforms to the specification iff P satisfies the reset condition.

The notion of a semi-inductive predicate is analogous to the Boolean case. A predicate P forces output conformance if

$$\forall S\ \forall I\ [P(S) \Rightarrow \gamma_{Impl}(S_{Impl}, I) \subseteq^X \gamma_{Spec}(S_{Spec}, I)].$$

In the ternary case, a mapping M is a partial order over L. Let P_M be the set of states that satisfy M:

$$P_M(S) \;\Leftrightarrow\; \forall\, l_0, l_1\ [M(l_0, l_1) \;\Rightarrow\; S(l_0) \subseteq^X S(l_1)].$$

The definition of refinement for ternary latch mappings is analogous to the Boolean case. Theorems 1 and 2 are also true in the ternary case. The definition of a refinement function is analogous, and Theorem 3 holds. The statement and proof of Theorem 6 and Theorem 7 (below) are analogous to Theorem 4 and Theorem 5, respectively.

Theorem 6. If M is a fixed point of a refining function Φ, forces output conformance and satisfies the reset condition, then the implementation conforms to the specification.

Theorem 7. Let M* be the greatest fixed point of a refining function Φ. Assume that M* does not force output conformance. Then there does not exist a latch mapping M that is semi-inductive and forces output conformance.

Although much of the mathematical theory for ternary latch mapping is analogous to the Boolean case, the implementation of the algorithm is quite different. A latch mapping is a partial order rather than an equivalence relation. Thus, it can no longer be represented with just a partitioning. The partitioning must be augmented with links between partitions that indicate that all of the latches in one partition are strictly less than the all of the latches in the other partition.

Refining a latch mapping is also more complicated, as illustrated in Figures 2 and 3. Figure 2 shows a latch mapping with two classes A and B. The link from B to A indicates that all of the latches in A are less than all of the latches in B. Consider a combined state S and an input vector I. Let A_0 be the set of latches l in A such that δ(S, I)(l) = 0. We define A_1, A_X, B_0, B_1 and B_X analogously. Assume that A_0, A_1, A_X, B_0, B_1 and B_X are all nonempty. Figure 3 shows the result of using S and I to refine the mapping in Figure 2.

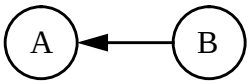

Figure 1: Example ternary latch mapping.

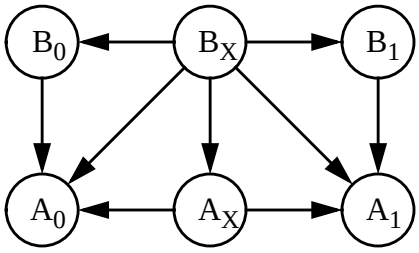

Figure 2: Possible result of refining the latch mapping in Figure 2.

9. Sequential Equivalence Checking

Sequential equivalence checking involves finding the set of reachable combined states P of the implementation and the specification, and then checking whether the two designs have the same output values for all reachable states. The set of states P can be represented either explicitly or symbolically (with BDDs, for example).

More generally, P can be any inductive invariant, not just the set of reachable states. This allows, for example, finding the set of all states Q from which an inequivalent combined state can be reached, and then checking if Q includes a reset state. In this case, the inductive invariant P is equal to the complement of Q. Using the terminology of Theorem 1, we want to find a set of states P that is semi-inductive (that is, closed under the combined transition function, as defined in section 2), forces output equality and satisfies the reset condition.

Like sequential equivalence checking, the latch mapping algorithm of sections 3 and 5 also attempts to find an invariant P. The difference is that latch mapping only considers sets P_M that satisfy some latch mapping M (as defined in section 3). The algorithm starts with the smallest possible P_{M0}, which is where all latches in both designs are mapped together and, therefore, have the same value. If this set is not semi-inductive, then it is increased to a larger set P_{M1}. This process is continued until a fixed point is reached. The resulting fixed point forces output equality iff the two circuits are combinationally equivalent.

If two circuits are sequentially equivalent but not combinationally equivalent, it means there is an inductive invariant that forces output equality, but no such invariant of the form P_M. The performance/accuracy trade-off between combinational and sequential equivalence checking can be viewed in terms of the set of inductive invariants that is being considered. Sequential equivalence checking considers all possible invariants. Combinational equivalence checking only considers those invariants P_M that are the set of all states that satisfy some latch mapping M.

In this regard, combinational and sequential equivalence checking are two extremes. There are intermediate points that can be characterized by the kinds of invariants that are considered. The ternary latch mapping algorithm in section 8 is an example that considers a richer set of invariants than combinational equivalence checking does, and is, therefore, less conservative. Variants on combinational equivalence checking that allow small differences in state encoding (such as allowing a latch to be mapped to the logical negation of another latch) can also be described in this framework. We believe that this framework makes it easier study formal verification methods that might combine some of the strengths of both combinational and sequential equivalence checking.

10. Conclusions and Future Work

We have described an automatic latch mapping algorithm that does not depend on signals names or other heuristics. We have shown how the method can be applied to circuits with reset sequences. In the case that the implementation does not conform to the specification, we described an interactive procedure for identifying a particular combinational block with a bug. We gave empirical results for our implementation that showed that the method is quite efficient in practice, requiring an average of only 28% more CPU time than if a latch mapping was provided. We also

showed how the method can be extended with ternary values to compare an implementation to a specification in the presence of don't care conditions.

Our primary area of future work is to implement and test the ternary version of our algorithm. We would also like to study the relationship of our fixed point algorithm to fixed point algorithms for reachability analysis in sequential equivalence checking, as discussed in section 9. We will explore verification methods that offer a trade-off between the efficiency of combinational equivalence checking and flexibility of sequential equivalence checking.

References

1. H. Cho and C. Pixley. Apparatus and method for deriving correspondence between storage elements of a first circuit model and storage elements of a second circuit model. U. S. Patent 5,638,381. June, 1997.
2. C. van Eijk. Formal Methods for the Verification of Digital Circuits. Ph.D. Thesis, Eindhoven University of Technology, 1997.
3. C. van Eijk. Sequential Equivalence Checking without State Space Traversal. In DATE 1998.
4. T. Filkorn. Symbolische Methoden für die Verifikation endlicher Zustandssysteme. Dissertation Institut für Informatik der Technishen Universität München, 1992.
5. Andreas Kuehlmann and Florian Krohm. Equivalence Checking Using Cuts and Heaps. In DAC 1997.
6. D.K. Pradhan, D. Paul, and M. Chatterjee. VERILAT: Verification Using Logic Augmentation and Transformations. In ICCAD 1996.