Practical Issues in Sequential Equivalence Checking through Alignability: Handling Don't Cares and Generating Debug Traces

In-Ho Moon, Per Bjesse, and Carl Pixley
Advanced Technology Group
Synopsys, Inc.
{mooni,bjesse,pixley}@synopsys.com

Abstract

Reset states are often not known for a given design until late in the design cycle. There is therefore a need for sequential equivalence checking algorithms that work in the absence of this information. One popular choice for such a notion is *alignability*, which has the allure of not necessarily needing a symbolic traversal of every state in the system. However, to use alignability in practice, there are several obstacles that needs to be overcome. First of all, the standard alignability theory does not take into account that the golden design often may specify a range of acceptable implementations by means of designer annotated don't care states and don't care logic. Second, when two designs are unalignable, the fact that there are no specified initial state makes it hard to generate a meaningful counter-example to aid the designer in debugging the designs. We address these issues by extending the standard alignability theory so that it handles don't care information in the absence of reset states, and by devising a heuristic for finding a meaningful initial state for the debug traces. Our experimental results show that our theory extensions are necessary to be able to deal with a large portion of the industrial designs that we have applied our alignability checking tool to. We also show how our debug trace heuristic has allowed us to find a real optimization tool bug.

1 Introduction

In order to be able to optimize hardware designs, it is important that there is some way to check that the pre- and post-optimization designs behave in the same way. The standard way of doing this is to apply combinational equivalence checking, where (1) it is assumed that the two designs have registers that correspond in a one-to-one fashion, and (2) formal techniques are used to prove that the logic optimization did not change the boolean function that each pre-optimization cone implemented.

The ever increasing pressure to generate smaller and faster designs means that the restriction imposed by combinational equivalence checking is becoming counter-productive. As a result, sequential notions of equivalence that allow transformations that change the numbers of register or that use reachable state information to improve the logic optimization are very important.

There are two broad categories of sequential equivalence. The first is equivalence from some known reset state. Here two designs are deemed equivalent if all input sequences give the same output response when the two designs are started in their respective reset state. The second category is equivalence in the absence of a known reset state.

Equivalence from a known reset state can be solved by variations of standard model checking, and is a well-explored subject. The major downside of this notion of equivalence is that it presupposes that the reset states of designs are known, and in many design flows this information is not available until very late in the design cycle. As it is desirable to be able to interleave the verification throughout the design phases, we will focus on equivalence without a reset state.

Several different notions of equivalence have been proposed for designs whose reset state is unknown. The first definition to appear in the literature was Pixley's *sequential hardware equivalence* [8]. In sequential hardware equivalence (also known as *alignability*), it is checked whether the two designs are *alignable* in the sense that there exists some input sequence that will drive the designs from an arbitrary power up state to a state where the implementation and specification behaves the same way forever.

Alignability can be checked by symbolic computation, and while the checking is computationally complex, it does not require the traversal of all specification and implementation states in general. Up until recently, the main impediment to industrial use of alignability is that it is not a compositional notion of equivalence in the sense that block level alignability implies alignability of a complete design. As a result, several alternative notions of equivalence have been proposed that are more suited to compositional verification; for example, *safe replacement* [10] and *3-valued safe replacement* [3]. Unfortunately these notions of equivalence permit fewer optimizations, and may require the inspection of more states while diagnosing a system. As a result, even though several sequential equivalence checkers have appeared in the academic literature [14, 4, 12, 6, 1], there is no significant mainstream adoption as of yet. However, a recent paper [6] introduced a methodology for using compositional alignability in a sound way. This means that the main weakness of alignability has been rendered obsolete, and that there is hope of obtaining a robust tool for sequential equivalence checking in the absence of reset states.

While the new compositionality result means that alignability is ready for prime time in principle, there are two major hurdles that need to be overcome to make the approach usable in practice. First of all, the standard algorithms for alignability assume that both the implementation and specification designs are completely deterministic systems. This is not true in general, as designers often use language features to let the optimization tool know that certain states and logic cones of the specification can be implemented in whatever way that will minimize the size of the resulting design. Such states and cones are called *don't care* states and logic, and if this information is not taken into account the alignability algorithms will provide false negatives [4]. Second, when alignability checking fails, designers have little information with which to debug the failure as there are no known reset state from which to generate a debug trace.

In this paper we address these two weaknesses by presenting an extended alignability theory that can cope with don't care information in the absence of reset states, and by presenting a heuristic for debug trace generation that we have found useful in practice. Our experimental results show that our extended theory is necessary for correct verification of a large portion of the industrial designs that we have investigated. Moreover, we show how our heuristic to generate debug traces has helped diagnose a real optimization tool bug.

Huang *et al.* was the first to address the handling of don't cares in sequential equivalence checking of designs with a reset state [4].

The method in [4] models explicit don't cares as a separate circuit which is added on top of a mitered design. For internal don't cares, symbolic reachability is performed on differently encoded portions of the design to find a subset of unreachable states of the design from a reset state. As a reset state is required to compute don't care information for internal don't cares, the method can not be used in sequential equivalence checking without reset states.

The remainder of this paper is organized as follows. Section 2 recapitulates the alignability theory. Section 3 describes the proposed algorithm to extend the original alignability to deal with don't cares. Section 4 describes how debug traces can be generated during alignability computation. Section 5 explains related work. Experimental results are shown in Section 6, and we conclude with Section 7.

2 Alignability

If the reset states of the two designs that we want to equivalence check is known, then it is very easy to define equivalence: Two systems are equivalent exactly if their input-output behavior is the same when they are started in their respective reset states. However, if the reset states are unknown, a good notion of equivalence is more elusive.

Pixley proposed *alignability* as a good equivalence notion in the absence of reset states, and it has since become a very popular choice [8, 6]. Conceptually, alignability is decided by first checking whether there are any state pairs for the two designs which guarantees that the two designs will behave in the same way for all input sequences. These *equivalent state pairs* are state pairs from which you know you are safe forever if you ever end up in them. If no such state pairs exist, the designs can clearly not be equivalent using any reasonable notion. Second, the question is whether there is some group of equivalent state pairs that every state in the system can reach using some input sequence (the *aligning sequence*). If such a sequence exists, then the system is deemed alignable. The intuition is that the aligning sequence is an input sequence that can be used to bring the systems to some known states which are the good operating states.

Formally, let $x = \{x_1, \ldots, x_n\}$, $y = \{y_1, \ldots, y_n\}$, and $w = \{w_1, \ldots, w_p\}$ be sets of variables ranging over $B = \{0,1\}$ on a finite state machine. The sets x, y, and w are called the present state, next state, and input variables, respectively. Also, let δ_i be the next state function of y_i (the i^{th} y). A state machine can be represented by transition relation T where $T(x,w,y) : B^{2n+p} \rightarrow B$ is 1 if and only if there is a transition from the state encoded by x to the state encoded by y under the input encoded by w.

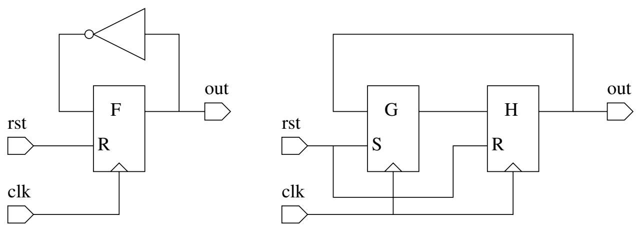

Figure 1: An example for alignability.

The first step in the alignability computation is to build the transition relation of a product machine consisting of specification and implementation designs as in Equation 1. The second step is computing EOP (Equivalent-Output-Pairs) as in Equation 2. EOP is the set of states generating the same values on all outputs with all input values. In Equations 2, S_i and I_i are the i^{th} output in specification and implementation design, respectively. In the example in Figure 1, state pairs are in the form of ([F], [G, H]) on a product machine and the ports R and S represent asynchronous reset and set, respectively. In this example, EOP has four state pairs, which are $\{$ ([0], [1, 0]), ([0], [0, 0]), ([1], [0, 1]), ([1], [1, 1]) $\}$. The third step is computing Equivalent State Pairs (ESP) as in Equation 3. The ESP is the set of states generating the same output values with any transitions forever. In the example, the ESP has only two state pairs, that is $\{$ ([0], [1, 0]), ([1], [0, 1]) $\}$. The fourth step is computing Alignable State Pairs (ASP) as in Equation 4. The ASP of a system is the set of states from which there exists an initialization sequence to reach a state in the ESP. Two designs are said *alignable* if and only if ASP is tautologous, that is, ASP contains all states in the product machine. Our example is alignable since the ASP is tautologous. In Equations 3 and 4, μ and ν are least and greatest fixpoint operators, respectively [13]. *PreImage* computation is finding all predecessors from a given set of states in one step symbolically. *PreImage** is different from *PreImage* in that *PreImage** existentially quantifies only y variables, whereas *PreImage* existentially quantifies y and w variables.

$$T(x,w,y) \quad = \quad \prod_{i=1}^{n} (y_i \equiv \delta_i(x,w)) \tag{1}$$

$$EOP(x) \quad = \quad \forall w. \prod_{i=1}^{n} (S_i(x,w) \equiv I_i(x,w)) \tag{2}$$

$$ESP(x) \quad = \quad \nu Z. EOP(x) \wedge \exists w. PreImage^*(T,Z) \tag{3}$$

$$ASP(x) \quad = \quad \mu Z. ESP(x) \vee PreImage(T,Z) \ , \tag{4}$$

where

$$PreImage^*(T,C) \quad = \quad \exists y. T(x,w,y) \wedge C(y) \ ,$$
$$PreImage(T,C) \quad = \quad \exists w.y. T(x,w,y) \wedge C(y) \ .$$

A reset sequence can easily be computed during the ASP computation [8]. In the example, a reset state is ([0], [1, 0]) which is reached by setting *rst* to 1.

3 Alignability with Don't Cares

When equivalence checking RTL designs, it is necessary to handle don't cares [4, 7, 11]. However, the original alignability theory handles only deterministic designs; that is, designs that do not have don't care states, in other words, completely specified designs. Designers typically introduce don't care information either by explicitly assigning X (unknown) values in the RTL, or implicitly by using case statements that are not full-case. Don't care information is quite common in RTL designs, so it is in issue that we need to cope with if we are to deal with real industrial designs. This section describes how we extend the standard alignability theory to be usable in the presence of don't cares, in other words, for non-deterministic designs or incompletely specified designs.

3.1 Don't Care Modeling

In both combinational and sequential equivalence checking, don't cares can be represented as either a separate circuit as in [4] or interval network by propagating all don't cares through a circuit [11]. The propagation is performed in linear time and space in terms of circuit size. After the propagation, each net in the interval network can have *min* and *max* functions as an interval if they are different.

We found out that the interval network is more appropriate to handle internal don't cares, especially for output don't cares that may differ from an output to another.

3.2 State Don't Cares

We encode don't care transitions by using a non-deterministic transition relation in the following way. The definition of a bit transition relation of y_i can be expressed in the following alternative form

$$\begin{aligned} T_i(x,w,y_i) &= (y_i \wedge \delta_i(x,w)) \vee (\neg y_i \wedge \neg\delta_i(x,w)) \\ &= (\delta_i(x,w) \rightarrow y_i) \wedge (y_i \rightarrow \delta_i(x,w)) \ , \end{aligned}$$

which encodes that y_i becomes 1 when δ_i is 1, and y_i becomes 0 when δ_i is 0. In other words, T_i is 1 iff δ_i implies y_i and y_i implies δ_i.

In the presence of state don't cares, let f be a function and f_{dc} be a don't care set for f, then f can be represented as an interval as below.

$$\begin{aligned} f_{min} &= f \wedge \neg f_{dc} \\ f_{max} &= f \vee f_{dc} . \end{aligned}$$

Now, the next state function, $\delta(x,w)$, can be represented as an interval using $\delta_{min}(x,w)$ and $\delta_{max}(x,w)$ under the don't care function $dc(x,w)$. The bit transition relation is now expressible as

$$\begin{aligned} T_i(x,w,y_i) &= (y_i \wedge \delta_{i\_min}(x,w)) \vee (\neg y_i \wedge \neg\delta_{i\_max}(x,w)) \vee dc(x,w) \\ &= (y_i \wedge (\delta_{i\_min}(x,w) \vee dc(x,w))) \vee \\ &\quad (\neg y_i \wedge (\neg\delta_{i\_max}(x,w) \vee dc(x,w))) \\ &= (y_i \wedge \delta_{i\_max}(x,w)) \vee (\neg y_i \wedge \neg\delta_{i\_min}(x,w)) \\ &= (\neg\delta_{i\_min}(x,w) \vee y_i) \wedge (\neg y_i \vee \delta_{i\_max}(x,w)) \\ &= (\delta_{i\_min}(x,w) \rightarrow y_i) \wedge (y_i \rightarrow \delta_{i\_max}(x,w)) \ . \end{aligned}$$

This means that, in the presence of don't cares, the bit transition relation T_i encodes that y_i becomes 1 when δ_{i_min} is 1, or y_i becomes 0 when the negation of δ_{i_max} is 1, and y_i is both 1 and 0 when dc is 1. This can be re-expressed as T_i is 1 if δ_{i_min} implies y_i and y_i implies δ_{i_max}.

With this non-deterministic expression, the transition relation in Equation 1 can be modified to include don't care information as shown in Equation 5. Similarly, the definition of EOP in Equation 2 can be modified to Equation 6 using the facts that $S_{min} \leq I_{min}$ and $I_{max} \leq S_{max}$.

$$T(x,w,y) = \prod_{i=1}^{n}(\delta_{i\_min}(x,w) \rightarrow y_i) \wedge (y_i \rightarrow \delta_{i\_max}(x,w)) \quad (5)$$

$$EOP(x) = \forall w.(\prod_{i=1}^{n}(S_{i\_min}(x,w) \rightarrow I_{i\_min}(x,w)) \wedge (I_{i\_max}(x,w) \rightarrow S_{i\_max}(x,w))) \quad (6)$$

At this point we have made good progress to the correct handling of don't care information. However, we need to do more: Consider the example in Figure 2. The extended alignability with the non-deterministic transition relations will compute an ESP that contains the state A. This is wrong, as there is a path from the state A to a bad state. The reason for why state A is included in the ESP is due to the fact that there are two transition edges (w=1 and w=0) going to the other states in ESP, regardless of the edge going to a bad state.

We can correct this problem by using *BackImage* [2] rather than $PreImage(T,C)$ when we compute ESP using Equation 7. $BackImage(T,C)$ is different from $PreImage(T,C)$ in that it computes a subset of *PreImage* such that all the successors of the subset are only in C. In Figure 2, *PreImage* of the dashed box region returns the same region, whereas *BackImage* of the dashed region returns the solid box region by excluding the state A since a successor of the state A is a bad state outside the dashed region. *BackImage*

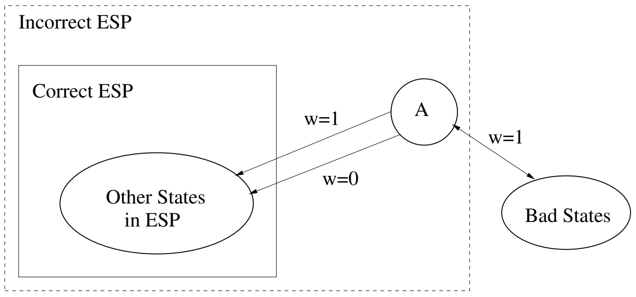

Figure 2: Correct ESP computation with don't cares.

can be computed by *PreImage* by using the identity in Equation 9 that prevents the inclusion of any state into the ESP that has an edge going to a bad state.

$$\begin{aligned} ESP(x) &= \nu Z. EOP(x) \wedge BackImage(T,Z) && (7) \\ ASP(x) &= \mu Z. ESP(x) \vee PreImage(T,Z) \ , && (8) \end{aligned}$$

where

$$BackImage(T,C) = \neg PreImage(T,\neg C) \ . \quad (9)$$

ESP can also be seen as a set of states from which there is no path going to a bad state that generates different values on any corresponding outputs between specification and implementation design. Thus, ESP in Equation 7 can be also computed in the following alternative form of least fixpoint computation

$$\begin{aligned} BS(x) &= \bigcup_{i=1}^{n} \neg((S_{i\_min}(x,w) \rightarrow I_{i\_min}(x,w)) \wedge \\ &\qquad (I_{i\_max}(x,w) \rightarrow S_{i\_max}(x,w))) \ , && (10) \\ BRS(x) &= \mu Z. BS(x) \wedge PreImage(T,Z) \ , && (11) \\ ESP(x) &= \neg BRS(x) \ . && (12) \end{aligned}$$

$BS(x)$ represents *bad states* and is the set of states producing different values on any pair outputs, and $BRS(x)$ represents *backward reachable states* from the bad states and is the set of states that there exists a path going to a bad state. Equation 10 is simplified in the absence of don't cares as below.

$$BS(x) = \bigcup_{i=1}^{n} \neg(S_i(x,w) \equiv I_i(x,w)) \ . \quad (13)$$

3.3 Output Don't Cares

Output don't cares should also be considered properly in sequential equivalence checking. Figure 3 shows an example. In the specification, the output value can have any value when the selector s is low, whereas the output value is assigned to 0 in the implementation. It is obvious that the example is combinationally equivalent since the function of the register input D (that is sF) in the implementation resides in the interval D_{min} and D_{max} of the D (that are sF and $\neg s \vee F$, respectively) in the specification. However, alignability will result in a 'not alignable' diagnose as the standard ESP is empty since there exists at least one path generating different output values from all states.

Figure 4 shows how we can handle output don't cares. The basic idea is that we conjoin all care states with all of the outputs with the transition relation. *HandleOutputDCs* takes a transition relation T

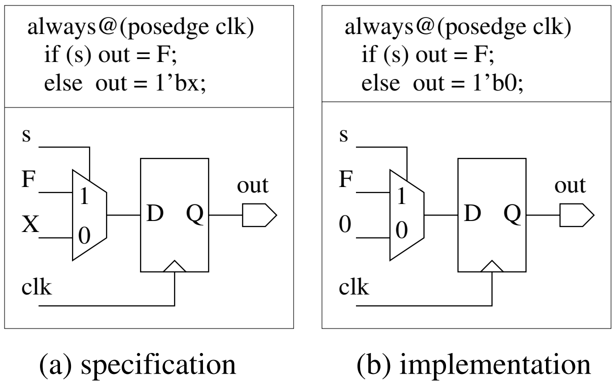

Figure 3: Output don't cares.

and a set of outputs O. F is the set of all fanin registers of O, and each register f in F has f_{min} and f_{max} functions as an interval form [11], as mentioned in Section 3.1. $UnionDC$ is initialized to *empty*, then updated for each register in F. Then, finally T is updated by conjoining with the negation of $UnionDC$.

```
HandleOutputDCs($T$, $O$) {
    $F$ = set of all fanin registers of $O$;
    UnionDC = empty;
    for each ($f \in F$) {
        if ($f_{min} \neq f_{max}$) {
            $DontCares = f_{max} \wedge \neg f_{min}$;
            $UnionDC = UnionDC \vee DontCares$;
        }
    }
    $T = T \wedge \neg UnionDC$;
}
```

Figure 4: Handling output don't cares.

One can argue that the example is sequentially inequivalent, so it should be reported as a discrepancy. However, we decided to take output don't cares into account in such a way that combinational and sequential equivalence checking produce the same answer where they both are applicable.

3.4 ESP and ASP

Alignability is mainly composed of ESP and ASP computations. This can be seen as that ESP computation is to check whether two designs are sequentially equivalent, assuming the two designs start with an initial state that is in the computed ESP. ASP computation is however to guarantee that the initial state exists by finding an aligning sequence that brings any power up state to the initial state. In a sense, this ASP computation is to check whether a design is resettable, in other words, to check the correctness of the design.

In reality, since reset states are often not known for a given design until late in the design cycle, there are cases where two designs are to be verified for sequential equivalence, but one or both of the two designs are not resettable. In these cases, if alignability computation results in a non-empty ESP and non-tautologous ASP, it would be better to say that two designs have non-empty ESP, but not alignable than simply to say that two designs are not alignable.

Notice that combinational equivalence checking does not consider resetability at all. If two designs are not resettable, but combinationally equivalent, then combinational verification says that the designs are equivalent. Combinational equivalence checking is also a kind of sequential equivalence checking when all registers are one-to-one mapped between two designs. Thus, it can be seen as that combinational equivalence checking is a form of sequential equivalence checking without checking resetability.

4 Generating Debug Traces during Alignability

When two designs have been shown to be sequentially inequivalent, it is necessary to provide at least one debug trace. It can be done easily when sequential verification was done from some reset state. However, traditional alignability verification is done without any reset state. Therefore, at a first glance, it seems that generating debug traces is not possible.

However, we have found out that we often can generate a meaningful debug trace on alignability even when we do not have access to a reset state. To do this, we keep all intermediate results in the ESP fixpoint iteration as *onion rings*. Let ESP^i be the i^{th} ESP approximation in the iteration of ESP computation (where ESP^0 is equal to the EOP). Now suppose ESP^k becomes empty at the k^{th} iteration. Even though there is no initial state, we can often provide a meaningful trace by choosing an initial state from ESP^{k-1} as our starting state. This is because any state in ESP^0 requires one or more transitions to get to a bad state, whereas any state in ESP^{k-1} requires at least k transitions. The states in ESP^{k-1} are the farthest from the bad states in terms of how many transitions are required at least to reach a bad state. Thus, the states in ESP^{k-1} is the most likely candidates for valid or meaningful initial states. Once we have chosen a state, we can apply the standard techniques for generating debug trace from the onion rings as in model checking. Note that the initial state from ESP^{k-1} may be an unreachable state that does not provide the user with information on why the designs do not correspond in certain cases. However, we have found that the initial state from ESP^{k-1} often is valid and that the debug trace explains why it goes to a bad state well in practice.

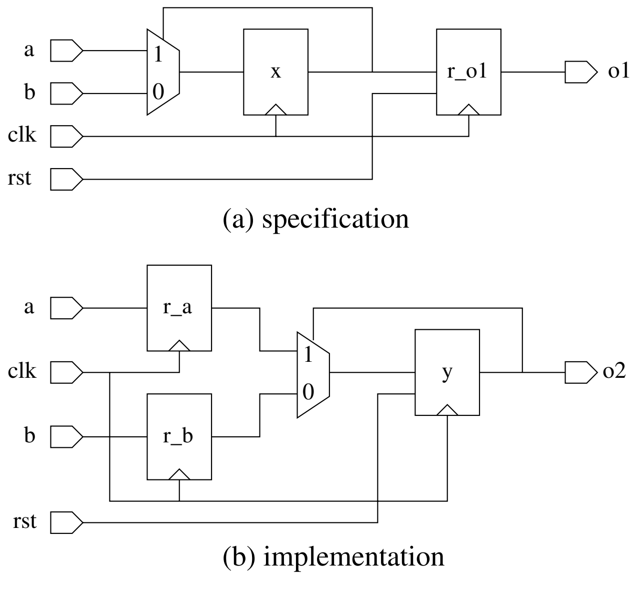

Figure 5: An example for generating debug trace.

As an example, consider Figure 5. In this example, if we set the signal *rst* to a constant 0, the two designs are sequentially equivalent since the register r_o1 in the specification was retimed backward and mapped to the registers r_a and r_b in the implementation. However, once *rst* is high, the two outputs $o1$ and $o2$ become 0 immediately. Now suppose that *rst* becomes low and *clk* becomes high. Then, r_o1 gets the value of x that is the previous value of either a or b in the specification. Whereas, y gets the value of r_b that is the previous value of b in the implementation. On this example, there are total 32 states since there are 5 registers. Alignability computation gives 16 states in EOP, then 8 and 3 states in the first and second iterations in ESP computation. The ESP becomes empty at the third iteration.

Table 1 is the actual debug trace we generate. Step 0 is the initial state picked up from the last non-empty ESP (that is, from the second iteration). In Step 1, *rst* becomes high and *clk* becomes low, then the values of r_o1 and y become 0. In Step 2, r_o1 gets the value of x at Step 1 and y gets the value of r_b at Step 1. This makes the values of $o1$ and $o2$ different.

	Inputs				Spec		Impl			Outputs	
Step	a	b	clk	rst	x	r_o1	r_a	r_b	y	o1	o2
0					1	1	1	0	1	1	1
1			0	1	1	0	1	0	0	0	0
2	1	0	1	0	1	1	1	0	0	1	0

Table 1: A generated debug trace.

The initial state from ESP^{k-1} can be justified by performing backward reachability and if the initial state is reachable from all states in EOP, the initial state is truly valid. In the example in Figure 5, this is the case.

5 Related Work

The first work that attempted to address the problem of verifying the equivalence of circuits before their reset states were known was the work of Pixley [8]. Other notions of equivalence has been proposed, such as safe replacement [10] and 3-valued safe replacement [3] for example, but alignability seems to have been the notion of equivalence that has been most popular and practical to date.

As previously mentioned, Khasidashvili and coworkers have constructed a tool that is centered around Pixley's original alignability verification. In order to deal with the limitations of the original BDD-based algorithms, compositional algorithms that rely on manual annotation of the design with properties are used [6]. The same authors have also investigated the use of SAT-based methods for checking equivalence without a reset state, and the computation of synchronizing sequences [9, 5].

Huang *et al.* was the first to address the handling of don't cares in sequential equivalence checking of designs with a reset state [4]. The method in [4] models explicit don't cares as a separate circuit which is added on top of a mitered design. For internal don't cares, symbolic reachability is performed to find unreachable states that are used as constraint in verifying equivalence. As a reset state is required to compute don't care information for internal don't cares, the method can not be used in our alignability based framework. Another strong point in the proposed model in this paper is that the proposed model treats both external and internal don't cares exactly the same way using interval network by propagating all don't cares through circuit [11], without performing symbolic reachability.

6 Experimental Results

We have developed a sequential equivalence checker and applied it to many industrial designs that have been sequentially optimized. Our sequential equivalence checker uses the extended alignability as its core decision method, but incorporates multiple different engines based on BDDs, SAT, and ATPG.

Table 2 shows the results of applying our sequential equivalence checker on real optimized designs. All examples have been run on a Linux machine with 4G-byte memory. In the table, the first column lists industrial designs from a number of different customers, and the next two columns (L_s and L_i) show the number of registers in specification and implementation designs, respectively. The next two columns (I and O) show the number of inputs and outputs in the design, respectively. The columns named *Time* and *Memory* contain the time in seconds and memory consumptions in megabytes for sequential equivalence checking. The next column (*DCs*) shows whether each design has don't cares. The last column (*Eq*) shows whether each design can be shown sequentially equivalent or not.

There are 10 industrial designs in the table. Conventional combinational equivalence checking can handle none of these designs, as all of the designs have undergone sequential optimizations such as FSM re-encoding, FSM optimizations, register merging and replication, and retiming. Among the 10 designs, five designs have don't cares which clearly shows that handling don't cares is necessary in practice. We have found one real bug in $D2$ and all the other designs were successfully verified.

The implementation design of $D2$ was sequentially optimized by a commercial logic optimization tool. The design was retimed and our sequential equivalence checker found a bug. Our proposed debug trace generation provided a meaningful trace that showed the optimization tool development team why the two designs are not alignable, and helped narrow the problem down to a wrong move made by the logic optimizer. The mismatch has since been fixed in new release of the tool.

Design	L_s	L_i	I	O	*Time*	*Memory*	DCs	Eq
D1	28	21	17	8	7.2	21.8	yes	yes
D2	33	55	12	9	891.3	74.6	no	no
D3	53	50	29	31	10.3	22.7	no	yes
D4	58	106	25	28	195.6	37.9	no	yes
D5	104	98	9	30	102.0	108.7	yes	yes
D6	175	176	17	10	14.9	46.0	yes	yes
D7	216	216	31	5	9.3	44.8	no	yes
D8	228	227	59	66	90.8	40.3	no	yes
D9	259	259	118	78	54.7	108.2	yes	yes
D10	296	288	216	35	27.7	58.1	yes	yes

Table 2: Experimental results.

7 Conclusions

To use alignability in practice, there are a number of obstacles that need to be overcome. We have taken two steps towards practical symbolic equivalence checking by extending the standard alignability theory to handle don't care information and by presenting a heuristic to generate debug traces in the absence of reset states.

The experimental results shows that the correct handling of don't cares is necessary in practice and that our sequential equivalence checker could verify many industrial designs that can not be proven equivalent using combinational equivalence checking. We have also shown that sequential equivalence checking can be a useful tool for finding optimization bugs, and that our heuristic for generating debug traces can provide a useful debugging aid when attempting to diagnose faulty optimization moves.

References

[1] J. Baumgartener and H. Mony. Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies. In *Correct Hardware Design and Verification Methods (CHARME'05)*, pages 222–237, Saarbrucken, Germany, October 2005. Springer-Verlag. LNCS 3725.

[2] A. Hu, G. York, and D. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *Proceedings of the Design Automation Conference*, pages 276–282, San Diego, CA, June 1994.

[3] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and U. Glaeser. An ATPG-based framework for verifying sequential equivalence. In *Proceedings of the International Test Conference*, pages 865–874, Washington, DC, October 1996.

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

[5] 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.

[6] 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, November 2004.

[7] I.-H. Moon and C. Pixley. Non-miter-based combinational equivalence checking by comparing BDDs with different variable orders. In *Formal Methods in Computer Aided Design*, pages 144–158. Springer-Verlag, November 2004. LNCS 3312.

[8] C. Pixley. A theory and implementation of sequential hardware equivalence. *IEEE Transactions on Computer-Aided Design*, 11(12):1469–1478, December 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, October 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, February 2001.

[11] T. Stanion. Circuit synthesis verification method and apparatus. *U.S. Patent 6,056,784*, May 2000.

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

[13] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[14] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 618–622, February 1998.