Matching in the Presence of don’t Cares and Redundant Sequential Elements for Sequential Equivalence Checking

Solaiman Rahim, Bruno Rouzeyre, Lionel Torres, Jerome Rampon

To cite this version:

HAL Id: lirmm-00191957
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00191957
Submitted on 26 Nov 2007
Abstract:
Full sequential equivalence checking by state space traversal has been shown to be unpractical for large designs [10]. To address state space explosion new approaches have been proposed that exploit structural characteristics of a design and make use of multiple analysis engines (e.g. BDDs, Simulation, SAT) to transform the sequential equivalence checking problem into a combinational equivalence checking problem [1][8]. While these approaches, based on induction techniques [1][2][3], have been successful in general, they are not able to reach proof of equivalence in presence of complex transformations between the reference design and its implementation. One of these transformations is redundant Flip-Flops (FFs) removal. FFs may be removed by redundancy removal, or don’t care optimization techniques applied by synthesis tools. Consequently, some FFs in the reference design may have no equivalent FFs in the implementation net-list. Latest researches in this area have proposed specific solutions for particular cases. In [5], matching in the presence of redundant constant input FFs has been addressed and in [11] identification of sequential redundancy is performed. This paper presents an in-depth study of some possible causes of unmatched FFs due to redundancy removal, or don’t care optimization techniques applied by synthesis tools. Consequently, some FFs in the reference design may have no equivalent FFs in the implementation net-list. Latest researches in this area have proposed specific solutions for particular cases. In [5], matching in the presence of redundant constant input FFs has been addressed and in [11] identification of sequential redundancy is performed. This paper presents an in-depth study of some possible causes of unmatched FFs due to redundancy removal, and proposes a generic approach to achieve prove of equivalence in presence of redundant FFs. Our approach is independent from specific synthesis transformations. It is able to achieve matching in presence of complex redundancies, and is able to perform formal equivalence checking in presence of don’t cares. The experimental results show a significant improvement in the matching rates of FFs when compared to industrial equivalence checking tools. This higher matching is directly translated to a higher success rate in proving equivalency.

1. Introduction
Generic equivalence checking methods of two sequential circuits require a state space traversal of the product machine. This method has the capacity to handle sequential optimizations (such as retiming, pruning, state machine re-encoding…), which are performed during synthesis. However, due to computational complexity, those methods cannot be applied to large circuits. Other approaches have been investigated which try to map the sequential equivalence problem of circuits into a combinational equivalence checking problem [8]. They rely on identifying some potential equivalent FFs or nets in the two circuits under verification through a matching/mapping step, and checking the equivalence of those FFs and nets by using combinational verification. The powerful existing matching methods [1][2][3] are functional based method using induction proof. They are applied to try to handle some sequential optimizations done during synthesis as merge, replication, sequential redundancy removal and retiming. These methods consist to find functional equivalent FFs in the two designs. But they cannot handle efficiently all type of sequential optimizations done during synthesis, particularly the optimizations which removed sequential elements such as redundancy removal or don’t care optimizations (only method for constant input FFs was proposed in [5] and [6]). Furthermore, they do not handle unspecified values (which is called don’t care) derived from the circuit (one method was proposed in [2] which consists to do the matching for each possible value of the don’t care, but can not be applied for circuits with large number of don’t care). In this paper, optimizations that remove sequential elements are presented. The problems, they may introduce for the existing matching methods, are illustrated on a set of circuits with redundant FFs. Then a synthesis independent method to complete the existing matching to handle those optimizations is proposed. We demonstrate that it completes the existing matching method efficiency. In section 6, a set of industrial circuits is used to compare our algorithm with some industrial tools.

2. Preliminaries
Here we introduce the notion of redundant FFs and don’t care functions which are used in the following sections.

2.1 Redundant FFs

Definition 1: A FF R of a circuit C is called redundant FF, if the observable input-output behavior of C is invariant after removing R from the circuit.

Definition 2: A constant input FF is an FF with a constant input function (0 or 1) at any clock cycle.
Definition 3: If \( F(t) \) is the input function of an FF \( R \) and \( v \) its output variable, \( R \) is stuck at 0 (resp 1) (An example is given in figure 1) iff:
1) \( \exists t | \forall T \geq t, F(t) = 0 \) (resp 1)
2) \( F(t) \) is a function of \( v \).

Definition 4: An FF \( R \) of a circuit \( C \) is called a non-observable FF, if the observable output behavior of \( C \) is independent of the value of \( R \), for all possible states and all possible input combinations of \( C \). An example is given in Figure 2.

Constant input FF, stuck at FF and non-observable FF are all redundant FFs and can be removed by the synthesis tools.

2.2 Don’t care definitions

Definition 5: A don’t care variable (noted dc-var) is a Boolean function which can be substituted by any Boolean function, in particularly by 0 or 1. Figure 3 described the use of don’t care variables to express the output function of a multiple driver net.

Definition 6: A don’t care function (noted dc-full-func) is a Boolean function with only don’t care variables in its support set. Note that dc-full-func is always reduced to 0 or 1 by affecting don’t care variables to a constant values. (Figure 5)

Let \( \text{Fct} \) be the set of all Boolean functions.
Let \( X = \{Dc1, Dc2\ldots\} \) be the set of dc-var of a Boolean function \( F \).

Definition 7: A dc-var-interpretation is a map \( X \rightarrow \{0,1\} \) which associates a Boolean value to each dc-var.

Definition 8: A dc-func-interpretation of the function \( F \) is a map \( I: \{0,1\}^X \rightarrow \text{Fct} \) which associates a Boolean function to each dc-var-interpretation.

Note that all dc-func-interpretations of the function \( F \) are included to it and there are \( 2^{|X|} \) dc-func interpretations of the function \( F \) possible.

Definition 9: \( R \) is a constant input FFs "modulo don’t care variables" iff \( \exists \) a dc-func-interpretation of \( R / R = 0 \) or \( R = 1 \). (Example in Figure 4)

Reference design
Implementation design

Figure 4: A constant input FF modulo don’t care variables

Definition 10: A zero (resp one) don’t care (noted 0-dc-func) is a function equal to zero (resp one) out of the don’t care set, i.e \( \exists \) a dc-func-interpretation of \( F / F = 0 \) (resp 1). (Example in figure 7). Notice that the example in figure 6 is a 0-dc-func and a 1-dc-func.

Definition 11: A function Impl is contained to a function Ref iff \( \exists \) a dc-func-interpretation I of Ref such as \( I = \text{A} \).

Theorem: A FF \( R \) is a constant input FF modulo don’t care variable iff the input function of \( R \) (noted \( F \)) is a dc-all-func, 0-dc-func or a 1-dc-func.

If the input function of an FF \( R \) is a constant input FF modulo don’t care variable, it may become a redundant FF and may be removed by the synthesis tool depending on which dc-func-interpretation of \( R \) is chosen.
3. Redundancy removal problems in existing matching methods [1,2,3]

3.1 Constant input FFs problems

Let’s consider the example below (Figure 8 (a),(b)). In figure 8a, constant propagation is not performed which leads to Out1 ≠ Out2 (false negative). In figure 8b, constant propagation through FFs was performed and the proof concludes Out1 = Out2.

F = (DC1 ∨ DC2) ∧ DC3

Reference design

BDD representation

Figure 5: dc-all-func

F = (v ∨ DC1) ∧ DC2

Reference design

BDD representation

Figure 6: 0 and 1 - dc-func

F = ¬v ∧ DC2

Reference design

BDD representation

Figure 7: 0-dc-func

The method proposed in [1] consists of finding equivalent nets. This method fails in Figure 9 because there are no equivalent nets.

In [4], the method consists of injecting the input function of the unmatched FFs to its output at the next clock cycle. This method fails in Figure 9 because their method of transforming the circuit with a feedback loop into a loop free circuit requires a positive unate function (otherwise an infinite loop can be created which is the case in Figure 9). This is why we are proposing a method to extend the coverage in such cases.

Figure 9: Example of stuck at 0 FF that cannot be handled in [4]

4. Our redundancy removal engine for matching step

We introduce here our redundancy removal engine and the process of our engine is illustrated in Figure 10, 11, 12, 13.

4.1 Constant input FFs

To handle constant input FFs, the input function of some candidate FFs is computed using BDD representation. Then the constant propagation through the FFs with BDD equal to 0 and 1 is done and the FFs are removed. The candidate FFs come from a random simulation. This reduces the number of BDD computed.

4.2 Stuck at FFs

We propose here to extend the method proposed in [1]. Indeed, an induction technique is used here to find stuck at FFs.

Let F be the input function of a FF R.
- If R is a FF with a reset line, it can be removed if F(t) = 0 ⇒ F(t+1) = 0.
- If R is a FF with a set line, it can be removed if F(t) = 1 ⇒ F(t+1) = 1.

4.3 Non observable FFs

The method is based on the DTPG algorithm proposed in [7] which consists of identifying redundant nodes by searching for undetectable stuck-at-fault. We apply the algorithm to the FFs instead of the circuit nodes. The
FF is removed if no pattern is found to make the output of the FF observable. One way to implement the DTPG algorithm is to check if the output variable of a candidate FF is present in the function of all FFs in its transitive fan out (using Bdds for example). This can be very time consuming because of the Bdds complexity. But the use of simulation techniques on the Bdds shows that the time can be considerably decreased (a average gain of 80% has been noticed).

The final flow is the following:

```c
Void RemoveRedFFs(Netlist *nl){
    while (fixpoint1)
    {
        removeConstFFs(nl);
        removeStuckAtFFs(nl);
    }
    while (fixpoint 2)
    removeNonObsFFs(nl);
}
```

Note that our engine is a fixpoint algorithm because every time a FF is removed from the design, new FFs can become redundant. Two fixpoints is needed because removing constant input or stuck at FFs does not imply new non-observable FFs and removing non-observable FFs does not imply new constant or stuck at FFs.

The process of the algorithm is illustrated on the reference and implementation designs in Figure 10.

![Reference design](image1.png)

**Figure 10: Design to prove**

After first iteration of fixpoint 1, R1 is removed because it is a constant input 0 FF, R3 is removed because it is a stuck-at-0 FF, then the reference design is transformed into the equivalent circuit in figure 11.

![Reference design](image2.png)

**Figure 11: First iteration of fixpoint 1**

After second iteration of fixpoint1, R2 is removed because it is a 0 constant input FF and the circuit in figure 11 is transformed to the equivalent circuit in Figure 12.

![Reference design](image3.png)

**Figure 12: Second iteration of fixpoint 1**

After first iteration of fixpoint1 done after the matching, R6 is removed because it became a stuck at 0 FF after that R4 and R5 have been matched together. Then the circuit in Figure 12 is transformed to the equivalent circuit in Figure 13 and can be proved easily with the implementation design.

![Reference design](image4.png)

**Figure 13: First iteration of fixpoint 1 done after the matching**

5. **How to handle Don’t care?**

Notation: In the following, a function F at next clock cycle (defined as the CBF function in [4]) is noted F'.

5.1 **How to handle dc-full-func**

To handle dc-full-functions, we reintroduce the don’t care input function of the FFs to its output at the next clock cycle (note that a dc-full-func at time t remains a dc-full-func at time t+1). This works because dc-full-function is always reduced to 0 or 1 after don’t care assignment, and 0 and 1 are contained to the new dc-full-func created at time t+1. This technique is illustrated in figure 14.

![Reference design](image5.png)

**Figure 14: Different implementations of a dc-full-func**
5.2 A simulation and assertion method to handle 0-dc-func (resp 1-dc-func).

Let’s consider Figure 15 and 16 that are the results of a different interpretation of the don’t care variables from the same reference design. The method proposed in section 5.1 can not handle the interpretation of figure 15. Indeed the functional matching can result in the following: \{R2(ref), R1(impl), R2(impl)\} are matched, and R1{ref) is unmatched as its input function contains don’t care. Reintroducing the input function of FF R1(ref) to its output results in the following:

\[
\text{Out1ref} = (\overline{DC} \land \overline{(V')})
\]

\[
\text{Out1impl} = V1
\]

\(\Rightarrow\) Out1impl \(\not\subset\) Out1ref (false negative)

The method proposed in [2] which is based on “inclusion matching” can not handle the interpretation of figure 16. Indeed, functional matching using “inclusion checking” can match \{R1(impl), R2(ref), R2(impl)\} because R2(impl) is included in R1(Ref). In this case the proof result is Out1(ref) \(\neq\) Out1(impl) (false negative)

To handle 0-dc-func (1-dc-func), our method combines “inclusion matching” [2] with method presented in section 5.1 to extract assertions for prove of equivalence.

The algorithm is as follow:
1) Reintroduce the input function F(t) of FFs with don’t care variables to its output at next clock cycle.
2) Do the inclusion check.
3) Set all assertion given by inclusion checking (i.e if R is a matched FF with output variable v and it has been checked being included into FF R’ with input function F(t), then the assertion v \(\subset\) F(t) is set).

Applying this method on Figure 15 gives:
1) \(\text{Out1ref} = (\overline{DC} \land \overline{(V')})\)
2) Inclusion check gives \(R1(impl) \subset R1(ref), R2(impl) \subset R1(ref)\)
3) Set the assertion \(V1 \subset (\overline{DC} \land \overline{(V')})\)

\(\Rightarrow\) Out1impl \(\subset\) Out1ref as we made the assertion that V1 is contained in \((\overline{DC} \land \overline{(V')})\)

Applying this method on Figure 16 gives:
1) \(\text{Out1ref} = (\overline{DC} \land \overline{(V')})\)
2) Inclusion check gives \(R2(impl) \subset R1(ref)\)
3) Set the assertion \(V1 \subset (\overline{DC} \land \overline{(V')})\)

\(\Rightarrow\) Out1impl \(\subset\) Out1ref as we have 0 contained in function \((\overline{DC} \land \overline{(V')})\)

5.3 Improvements and Limitations

The method presented in section 5.2 needs to reintroduce the input function of FFs with don’t care variables to its output at next clock cycle. If the FFs have a feedback loop, the method cannot work because of the loop. In this case, methods to transform circuit with feedback loop into loop free circuit can be used as presented in [4]. But this method cannot handle all kind of circuits; this is why we are working to settle a general method in this case.

In this paper, a set of potential optimizations, which removed redundant FFs, have been presented. This set is not exhaustive and new algorithm can be added to our engine to handle more optimizations.

6. Experimental results & Conclusion

We have synthesized a set of industrial designs using Synplify Pro 7.0 (www.synplicity.com) synthesis tool with “prune redundant FFs option” enabled. For constant input and non-observable FFs, both our method and those of industrial tools have the same results. However, for all designs with stuck at and constant FFs, our method is better and is able to complete the proof successfully compared to the industrial tools (the designs T7 which is snippet of the circuit on figure10 in section 4 is proved correct with our method and not with the industrial tools). For circuits with don’t care function, our method is able to perform the proof in all cases. The industrial tools can prove some of them. Our belief is that it randomly assigns a value to the don’t care variables, and thus, this assignment can chose the same value as the synthesis tools (see T10 Figure15 section 5). This is confirmed by the fact that design T11 (Figure16 section 5), a different interpretation of the don’t care variables of the same reference design of T10, is proved different.

Finally, we also notice an increase in run-time with our method. This is due to the “inclusion checking,” which is needed for each possible pair. Our future work is to
use simulation to reduce the number of possible pairs and thus decrease the run-time. (Note: DC case 1: dc-full-func; DC case 2: 0-dc-func or 1-dc-func, SAT0: stuck-at-0 FF, NObs: Non observable FF)

<table>
<thead>
<tr>
<th>Test</th>
<th>Size(K)</th>
<th>Pruning type</th>
<th>Pass</th>
<th>T(s) Pass</th>
<th>T (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>18</td>
<td>NObs</td>
<td>Yes</td>
<td>1.10</td>
<td>Yes</td>
</tr>
<tr>
<td>T2</td>
<td>33</td>
<td>SAT0+const</td>
<td>No</td>
<td>1.89</td>
<td>Yes</td>
</tr>
<tr>
<td>T3</td>
<td>40</td>
<td>Const+NObs</td>
<td>Yes</td>
<td>2.12</td>
<td>Yes</td>
</tr>
<tr>
<td>T4</td>
<td>36</td>
<td>Const+NObs</td>
<td>Yes</td>
<td>1.79</td>
<td>Yes</td>
</tr>
<tr>
<td>T5</td>
<td>15</td>
<td>Const</td>
<td>Yes</td>
<td>1.46</td>
<td>Yes</td>
</tr>
<tr>
<td>T6</td>
<td>45</td>
<td>No red</td>
<td>Yes</td>
<td>24.8</td>
<td>Yes</td>
</tr>
<tr>
<td>T7</td>
<td>90</td>
<td>SAT0+const</td>
<td>No</td>
<td>57.7</td>
<td>Yes</td>
</tr>
<tr>
<td>T8</td>
<td>102</td>
<td>DC case 1</td>
<td>Yes</td>
<td>12.3</td>
<td>Yes</td>
</tr>
<tr>
<td>T9</td>
<td>200</td>
<td>Const+DC case1</td>
<td>No</td>
<td>32.3</td>
<td>Yes</td>
</tr>
<tr>
<td>T10</td>
<td>200</td>
<td>DC case 2</td>
<td>Yes</td>
<td>35.2</td>
<td>Yes</td>
</tr>
<tr>
<td>T11</td>
<td>200</td>
<td>DC case 2</td>
<td>No</td>
<td>40.1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Results on industrial designs

References