A Flip-Flop Matching Engine to Verify Sequential Optimizations

Solaiman Rahim, Bruno Rouzeyre, Lionel Torres

Computing and Informatics, Vol. 23, 2004, 1001–1024, V 2005-Apr-14

A FLIP-FLOP MATCHING ENGINE TO VERIFY SEQUENTIAL OPTIMIZATIONS

Solaiman Rahim

Synplicity
Parc club du millenaire, Bat 3
1025, rue Henri Becquerel
34000 Montpellier
e-mail: rahim@synplicity.com

Bruno Rouzeyre, Lionel Torres

LIRMM
161, rue Ada
34392 Montpellier
e-mail: {rouzeyre, torres@lirmm.fr}@lirmm.fr

Manuscript received 9 January 2005

Abstract. Equivalence checking tools often use a flip-flop matching step to avoid the state space traversal. Due to sequential optimizations performed during synthesis (merge, replication, redundancy removal, ...) and don't care conditions, the matching step can be very complex as well as incomplete. If the matching is incomplete, even the use of a fast and efficient SAT solver during the combinational equivalence-checking step may not prevent the failure of this approach. In this paper, we present a flip-flop matching engine, which is able to verify optimized circuits and handle don't care conditions.

Keywords: Equivalence checking, synthesis, matching, combinational equivalence checking, sequential optimizations, redundancy removal, don't care conditions

1 INTRODUCTION

Equivalence checking proves formally that two given designs are functionally equivalent, e.g., an optimized design is functionally equivalent to its earlier version. Equivalence checking can be applied at different stages during the design process. Figure 1 shows the role of equivalence checking in a typical flow of modern VLSI design. During the design process, checking the equivalence of two designs described at the same or different levels of abstraction is necessary. For example, checking the functional equivalence of the optimized implementation against the RTL specification is critically important in order to guarantee that no error is introduced during the logic synthesis and optimization process. Similarly, checking the equivalence of the gate level implementation versus the gate-level model extracted from the layout can assure that no error is made during the physical design process. In the following text, the circuit before transformation is called the *specification* design, the one after transformation is the *implementation* design.

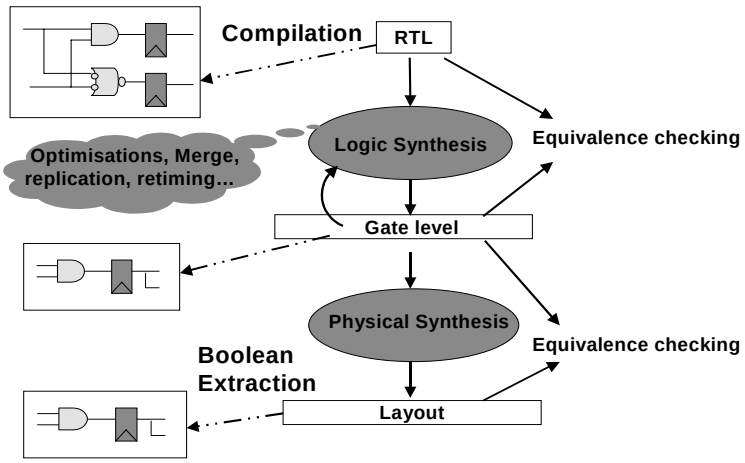


Fig. 1. Equivalence checking in the design flow

The method usually used to solve the equivalence-checking problem is called *state traversal*. This method constructs the *product machine* of the specification and implementation designs (illustrated in Figure 1), computes the reachable states of the product machine from the reset states and checks that the primary output of the product machine is always 0 for any input vector and reachable state.

Equivalence checking may be used to verify combinational and sequential designs. *Combinational equivalence checking* (CEC) is resumed to verify that two Boolean functions are equivalent. This can be done by using canonical representations of Boolean functions as BDDs [1] or non canonical ones as AIGs (And Inverter Graph) [2]. To avoid the problem of BDDs blow-up or the problem of the non canonicity of the AIGs, the state-of-the-art combinational verification methods combine

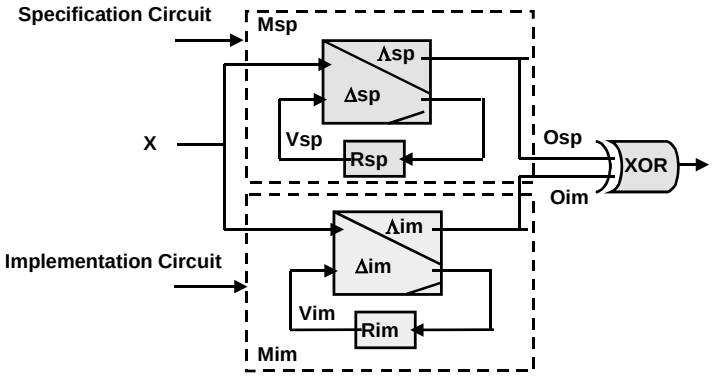


Fig. 2. Example of product machine

a powerful base verification algorithm with techniques to exploit the *structural similarities* of the circuits under verification ([2, 3, 4] or see [5] for an overview). These similarities typically occur in practical problem instances because of the incremental nature of the design process. The techniques to capture them are based on functional equivalences, indirect implications, or permissible functions. The utilization of structural similarities has shown to be very important for the efficient verification of large synthesized circuits. The main problem of these techniques is that they only consider the equivalence of the function and not the *inclusion*. The inclusion checking is needed by the introduction of *don't care conditions* used to represent all the outputs values of a function, which are left unspecified by the designer.

Conventional algorithms for solving the *sequential equivalence-checking* (SEC) problem require a state space traversal of the product machine. Impressive progress has been made in this area by the introduction of so-called symbolic techniques, which are based on the application of BDDs to traverse the state space (see e.g. [6] for an overview). Although these techniques can conceivably handle large circuits and are still being improved, they cannot be expected to scale well with circuit size for many types of circuit because of memory explosion.

To overcome the limitations of the symbolic approaches, an algorithm that performs the combinational verification in stages was proposed by Brand [4] in 1993. This algorithm explores the structural similarity of the two circuits under verification (CUVs) to speed up the verification process and has successfully verified larger circuits. The idea of this approach is to map the sequential equivalence-checking problem into a combinational equivalence checking one. This is a two-step approach. The first step is to find a *matching* between the flip-flops (FFs) in the specification and implementation designs. The aim of this step is to find all the potentially equivalent FFs. The second step often called the *proof* is to check the equivalence of the corresponding combinational blocks resulting from the matching by using combinational formal techniques.

This verification method is efficient and especially applicable when only combinational synthesis techniques are used for optimizations, i.e. when neither the number nor the input function of the FFs are changed by optimization procedures. Indeed, if *sequential optimizations* (e.g. as retiming, replication, merge, redundancy removal, ...) are performed during synthesis, both the number and the input function of FFs may change. Thus, the matching step can be complex and may lead to an *incomplete matching*, as described in [7]. If the matching is incomplete, even the use of a fast and efficient BDD/SAT solver during the combinational equivalence-checking step may not prevent the failure of this approach. Experiments with industrial tools using this method show that if two combinational blocks are found to be different, the cause is often an incomplete matching rather than a real bug in the circuit (*false negative problem*). It is therefore important to have an efficient matching technique.

The matching techniques of industrial CAD tools can be divided into two classes: structural matching and functional one, which are more detailed in Section 3.1.

In this paper, we present a flip-flop matching engine, which is able to produce a complete matching for large sequential optimized circuits. The first advantage of this engine is the structural matching algorithm, which is able to produce a complete matching in more cases than existing techniques. The second advantage is the redundancy removal engine, which considers complex redundant flip-flops. The third advantage is the functional matching, which considers don't care condition.

Our paper proceeds from notations in Section 2. We quickly review the state-of-the-art of sequential equivalence checking in Section 3. In these sections, we discuss particularly the problem introduced by don't care conditions and sequential optimizations to existing methods. Section 4 explains our matching engine, which combines functional and structural matching techniques and a redundancy removal engine. The effectiveness of the proposed method is confirmed by experimental results on retimed and optimized ISCAS'89 and large industrial benchmarks.

2 NOTATIONS

In this paper, the following notations are used to define a finite state machine (FSMs)

- $X = \{X_1, X_2, \ldots, X_n\}$ is the set of primary input.
- $O = \{O_1, O_2, \ldots, O_m\}$ is the set of primary output.
- $R = \{R_1, R_2, \ldots, R_p\}$ is the set of flip-flops.
- $\Delta = \{\Delta_1, \Delta_2, \ldots, \Delta_p\}$ is a set of transition functions where Δ_i is the input function of the flip-flop R_i.
- $V = \{V_1, V_2, \ldots, V_p\}$ is set of variables associated to the output function of the flip-flops where V_i is the variable associated to the output function of the flip-flop R_i.
- $\Lambda = \{\Lambda_1, \Lambda_2, \ldots, \Lambda_m\}$ is a set of output function where Λ_i is the function of the output O_i.

These notations are illustrated in Figure 3.

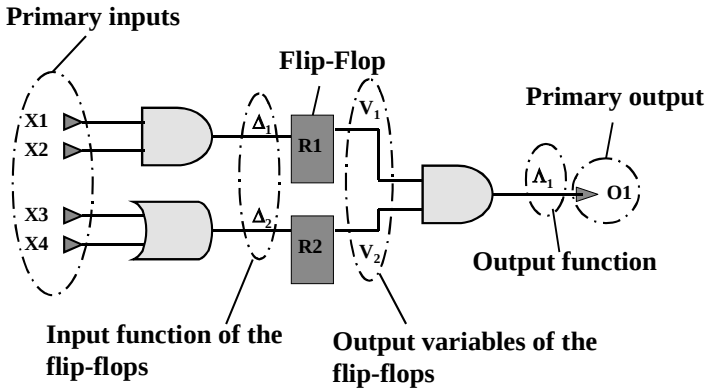


Fig. 3. Notations used to define FSMs

3 SEQUENTIAL EQUIVALENCE CHECKING

3.1 Matching

In order to solve the problem of the explosion of state traversal techniques, industrial equivalence checking tools often use a different method. The aim of this method is to transform the problem of equivalence checking of sequential circuits into a problem of equivalence checking of combinational circuits. For instance, if it can be assumed in Figure 4 that Δ_1 is equivalent to Δ_3 and Δ_2 equivalent to 4 and then, matches the corresponding FFs R_1 with R_3 and R_2 with R_4 by affecting the same output variables V_1 for $\{R_1, R_3\}$ and V_2 for $\{R_2, R_4\}$, the SEC is transformed into a CEC. Indeed checking with combinational techniques that Δ_1 is really equivalent to Δ_3, and Δ_2 equivalent to Δ_4, and finally that Δ_{1sp} is also equivalent to Δ_{1im}, leads to the verification that the two sequential circuits are equivalent.

The matching techniques of industrial CAD tools can be divided into two classes: structural matching and functional matching.

Structural matching consists of finding quickly all the FFs, which are *potentially* equivalent. Then, combinational formal techniques (proof) are used to check all assumptions made by this matching (e.g. check if the input function of two matched FFs is really equivalent). The probability that the assumptions given by structural matching techniques are correct should be as high as possible in order to avoid the false negatives problem. Structural matching is usually performed using FFs name or support comparison. These techniques are able to produce a complete matching when no sequential optimizations have been performed during synthesis (e.g. when there is a one-to-one flip-flop correspondence between the two designs). Therefore,

existing structural matching techniques cannot be applied in cases where sequential optimizations have been performed during synthesis as shown in Section 3.2.

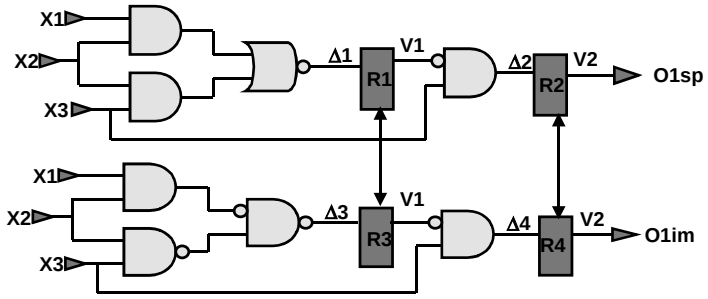


Fig. 4. Example of transformation of the SEC into a CEC

Functional matching consists of finding all FFs, which are equivalent in the specification and implementation designs. Functional matching techniques based on induction as [8] and [9] are able to produce a complete matching for optimizations such as merge, replication or retiming. However due to function representation issues (BDD blow up), these techniques are not able to match all the FFs in the designs. The existing combinations of functional and structural matching may not produce a complete matching, as existing structural techniques are not able to deal with sequential optimizations. Moreover, existing functional techniques do not consider don't care conditions or complex redundancy as stuck at FFs or FFs which become redundant after don't care assignment defined in Section 5.3 of this paper.

3.2 Don't Care and Sequential Optimizations Problems

In this section, we illustrate how don't care condition and sequential optimizations such as merge, replication, redundancy removal are the cause of false negatives problems because existing structural and functional matching methods are not able to deal with them.

3.2.1 Don't Care Problem

The main problem of the previous functional matching and combinational verification approaches is that they only consider the equivalence of the function and not the *inclusion*. To understand why the inclusion checking is needed, let us consider the example in Figure 5. This figure illustrates the specification of a ROM instance, where the output value is not specified by the designer for the input value $\{X_1, X_2\} = \{0, 1\}$ and $\{X_1, X_2\} = \{1, 0\}$. Figure 6 describes a different possible implementation of the ROM, which can be obtained after synthesis. This example clearly shows that checking only the equivalence of Λ_{sp} with Λ_{im} for any implemen-

tation designs leads to a false negative. To solve this problem, it is needed to check that Λ_{im} is included in Λ_{sp}.

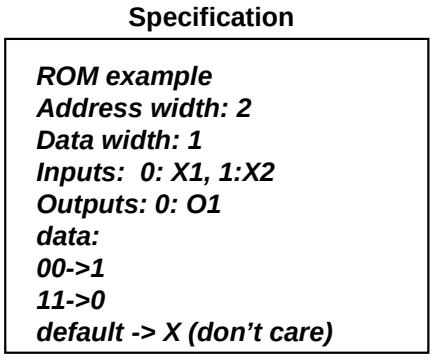


Fig. 5. Description of a ROM instance

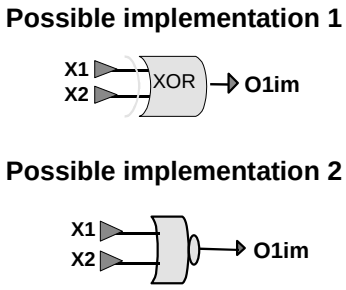


Fig. 6. Different possible implementation of the ROM

3.3 Merge and Replication

FFs in the specification design can be merged into a single one in the implementation design when they have the same input function as long as no timing degradation is observed. Replication of FFs is typically applied when the fanout of a FF is too large. To improve the delay, the FFs are replicated on their fanout network. For equivalence checking tools, these optimizations may introduce the false negatives problem. For instance, in Figure 7, the FF R_1 and R_2 in the specification design have been merged into a single one R_3 in the implementation design. In this case, existing structural matching techniques as name based matching match R_1 with R_3 and leave R_2 unmatched because no FF with the same name is present in the implementation design. This produces automatically a false negative when checking the equivalence of the output O_{2sp} with the output O_{2im}. The same problem may occur for optimization such as replication.

In Figure 7, functional matching can match R_1, R_2 and R_3 together because they have all the same input function. This leads to verify the two designs equivalent after checking that the output functions are also equivalent. However, as functional techniques need to compute the input function of the FFs to match them, they may not be able to produce a complete matching because of function representation issue. Moreover, in Figure 8, where the FF R_1 has been merged with the FF R_2 after the synthesis tool assigned the don't care variable Dc_1 to 0, the existing functional techniques are not able to match R_1, R_2 and R_3 together because they do not consider don't care condition.

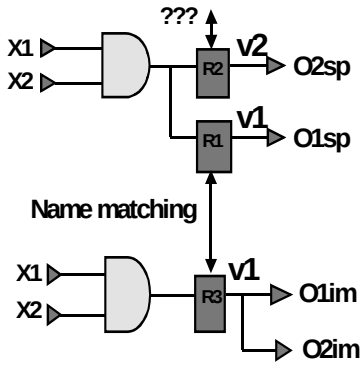


Fig. 7. Example of merge of FFs

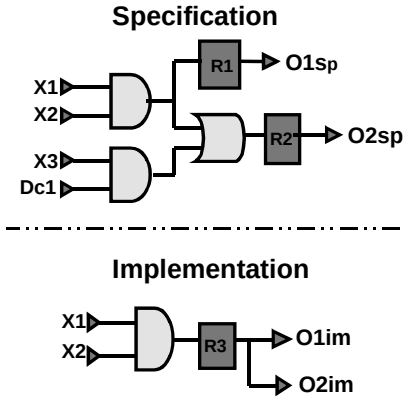


Fig. 8. Example of merge of FF after don't care assignment

3.3.1 Redundancy Removal

It is possible to distinguish several types of redundant FFs, which can be removed by the synthesis tools. A non-exhaustive list could be: constant input FF, "stuck at" FF, non-observable FF, constant input FF after don't care variables assignment.

Definition 1 (Redundant FF)**.** A FF R_i of a circuit C is called a *redundant FF*, if the observable input-output behavior of C is invariant after removing R_i from the circuit.

Definition 2 (Constant input FF)**.** A *constant input FF* is a FF with a constant input function (0 or 1) at any clock cycle.

Definition 3 ("Stuck at" FF)**.** A FF R_i is a *"stuck at"* 0 (resp. 1) iff:

1. $\exists t / \forall T \geq t$, $i(t) = 0$ (resp. 1)
2. $\Delta_i(t)$ is a function of V_i.

Definition 4 (Non-observable FF)**.** An FF R_i of a circuit C is called a *non-observable FF*, if the observable output behavior of C is independent of the value of R_i, for all possible states and all possible input combinations of C.

Examples of "stuck at 0" and non-observable FF are given in Figures 9 and 10, respectively.

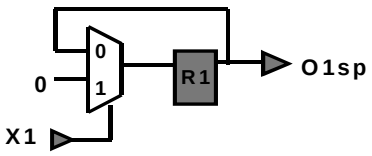


Fig. 9. Example of "stuck at 0" FF

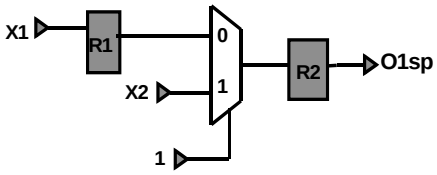


Fig. 10. R_1 is a non-observable FF

Definition 5 (Full don't care FF)**.** A FF R_i is a *full don't care FF* if and only if the support of its input function Δ_i contains only don't care variables.

Definition 6 (Partial don't care FF)**.** A FF R_i is a *partial* 0 *(resp.* 1*) don't care FF* if only if there is at least one assignment of the don't care variables, which makes Δ_i equal to 0 (resp. 1)

An example of *full don't care FF* is given in Figure 11 In this figure, the FF R_1 may become a constant input 1 FF for the following assignment of the don't care variables $\{Dc_1, Dc_2\} = \{1, 1\}$.

An example of *partial* 0 *don't care FF* is given in Figure 12. In this figure, the FF R_1 may become a constant input 0 FF for the following assignment of the don't care variables $\{Dc_1\} = \{0\}$.

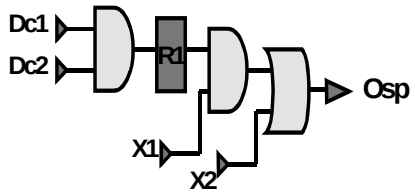


Fig. 11. Example of full don't care FF

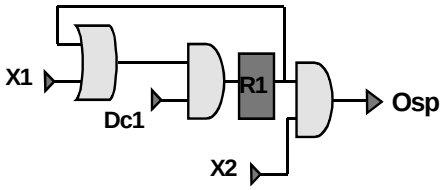


Fig. 12. Example of partial 0 don't care FF

Redundant FFs are removed and the redundancy value is propagated during the synthesis. Therefore, the redundant FF in the specification design does not have any match (functional or structural) in the implementation design. Structural matching techniques are not able to deal with redundant FFs and often produce false negatives problems. Existing functional matching handle only constant input FFs. Therefore, these techniques are not able to deal with *"stuck at"*, *non-observable*, *full or partial don't care FFs* and produce false negatives if these kinds of redundant FFs have been removed during synthesis. In order to solve this problem, we propose a redundancy removal engine, which is part of our matching engine and is able to deal with all redundant FF defined bellow.

3.4 Conclusion

We have seen in the previous section that sequential optimizations may change the number or the input function of FFs and thus are a real bottleneck for existing

structural and functional matching techniques. This is the main reason why industrial equivalence checking tools often produce false negatives if the design has been sequentially optimized. We present in the next sections techniques developed for solving this problem and how they are implemented in our matching engine.

4 OUR MATCHING ENGINE FLOW

Our matching engine flow is presented in Figure 13. The *matching* step is a fix point algorithm. This algorithm combines a functional matching technique with a redundancy removal engine and a structural matching technique. This engine is also detailed in [10, 11] and [12].

The functional matching is based on the induction algorithm proposed in [9]. However, to perform the matching, the method in [9] considers only the equivalence of the input functions of the flip-flops and not the inclusion. Consequently, this method cannot produce a complete matching for designs with don't care conditions. In order to solve this problem, this method has been improved by checking the inclusion of the functions.

The redundancy removal engine proposed in [13] is used to complete the functional matching. To be able to find complex redundancy such as stuck at flip-flops, this engine uses BDD representation and induction techniques.

Finally, our structural matching is used to match the flip-flops, which have not been matched by functional matching because of function representation issues (BDD blow-up). In order to handle sequential optimizations, the structural matching technique is not based on name or support comparison only. This matching uses a combination of techniques using AIG, simulation and support comparison. For instance, if the synthesis tool merges two flip-flops into a single one, then our structural matching can match in most of the case the three flip-flops. It is known that the existing structural techniques are not able to match the flip-flops in this case.

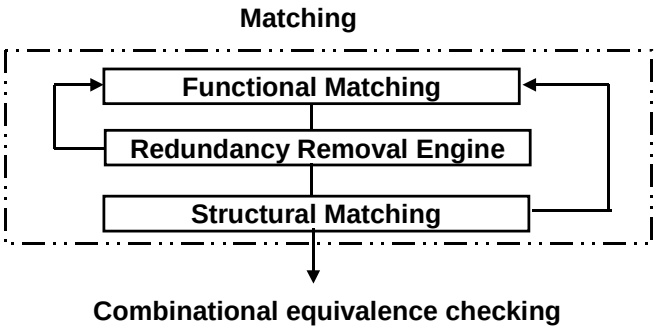


Fig. 13. Our matching engine flow

5 MATCHING STEP

5.1 Inclusion Checking

The first step to check the inclusion of two functions of the designs under verification is to represent the specification designs using don't care variables. The example of Figure 5 can be represented by the circuit in Figure 14. In this circuit, two don't care variables, Dc_1 and Dc_2, are used to specify the output value unspecified for the input value $\{X_1, X_2\} = \{0, 1\}$ and $\{X_1, X_2\} = \{1, 0\}$. The don't care variables are assigned during synthesis to a constant (0 or 1) or even to a function. In the following text, the set of don't care variables is noted $Dc = \{Dc_1, Dc_2, \ldots, Dc_q\}$.

The second step is to check the inclusion of the specification and implementation circuit. For this purpose, the definition 5.1.1 can be used.

Definition 7 (Inclusion of function)**.** A function $G(X_1, X_2, \ldots, X_n)$ of the implementation design is included in a function $F(X_1, X_2, \ldots, X_n, Dc_1, Dc_2, \ldots, Dc_q)$ of the specification design (noted $G \subseteq F$) if and only if $(\forall X_i)(\exists Dc_i)/(F \Leftrightarrow G)$.

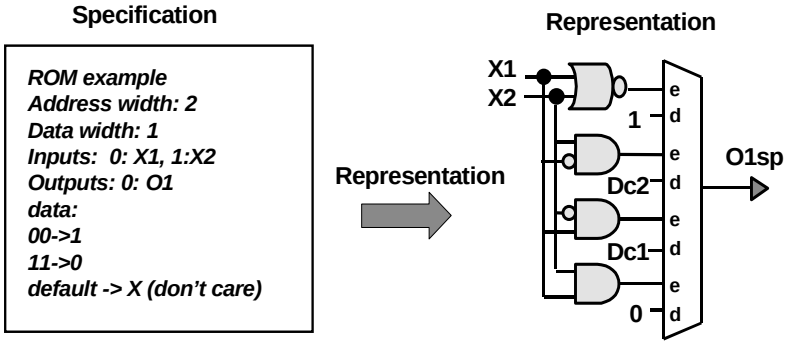


Fig. 14. A possible representation of the ROM of Figure 5 using don't care variables

An algorithm to verify the inclusion of two functions is shown in Figure 15. This algorithm considers the XOR function of the two functions to check. If the XOR function is equal to 0, the two functions are equivalent. Otherwise the smoothing operation regarding the don't care variables (Dc_i) is applied on the BDD representing the XNOR function of the two functions under verification. The smoothing operation $Smooth(F, Dc)$ is equivalent to $(\exists Dc\ F)$ which is equivalent to $F_{Dc} \vee F\neg_{Dc}$ where F_{Dc} and $F\neg_{Dc}$ are the positive and negative cofactor of the function F, respectively, regarding the variables Dc. If the result of the smoothing operation is equal to 1, the two functions are included; in the other case the two functions are checked different.

```
CheckInclusion(BDDXor){
if (BDDXor == 0)
  return EQUIVALENT;
BDDXnor = ¬ BDDXor;
BDD = BDDSmooth(BDDXnor, Dci);
if (BDD == BDD_ONE)
   return (INCLUDED);
else
  return(DIFFERENT); }
```

Fig. 15. Algorithm to check the inclusion of two functions

5.2 Functional Matching

The purpose of functional matching is to find all functional equivalent FFs of the product machine. In this section, we present our partitioning technique used to find all functional equivalents, included or inverted FFs of a design. Our algorithm is outlined in Figure 16. The algorithm starts with one equivalence class (P_0) for the FF matching, containing all FFs with initial state value "1" and "0" in positive and negative phases, respectively (step 1). During each iteration, if the next-state functions of members of an equivalence class differ, the class is partitioned (step 2). The main difference from previous approaches is that two functions Δ_1 and Δ_2 differ in our algorithm if Δ_1 is not included in Δ_2. This process is repeated until a fixed point is reached (step 3). The algorithm terminates after at most p iterations, where p is the total number of FFs of the product machine and thus the maximum number of equivalence classes in any FF correspondence.

Figure 17 gives two sequential circuits under comparison in order to show an example demonstrating our functional FFs matching algorithm. Figures 18 and 19 illustrate the different refinement stages of the equivalence classes. As shown in Figure 18, initially all FFs are assigned to a single equivalence class with phases matching their initial state values. A single new variable v is then created to compute the set of next-state functions in the first iteration. This variable is used positively for $\{R_2, R_3, R_5, R_6\}$ and negatively for $\{R_1, R_4, R_7\}$ reflecting their phase in the class. The resulting next-state functions show $\Delta_1 = \neg\Delta_5$; $\Delta_2 = \Delta_6 \subseteq \Delta_3$ and $\Delta_4 = \Delta_7$. Accordingly, the initial equivalence class is then split into three parts and the process is repeated with new variables v_1, v_2, v_3 for the 3 classes as illustrated in Figure 18. The following iteration does not change the partitioning, which causes the algorithm to terminate and return the resulting functional FFs matching as shown in Figure 19.

One way to speed up the process is to notice that each input function of the FFs does not need to be re-computed at each fixpoint iteration. For example, the input

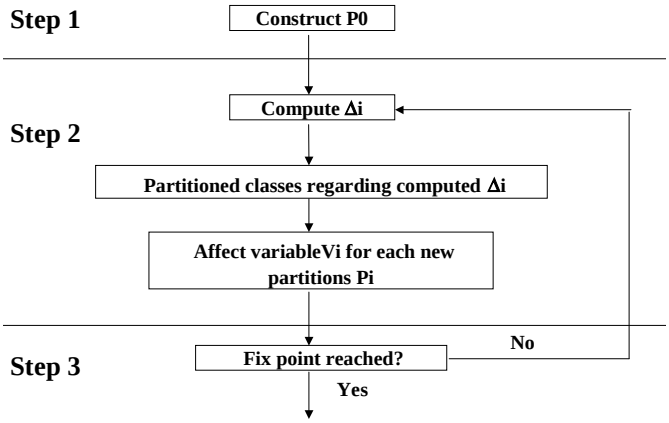


Fig. 16. Our functional matching algorithm

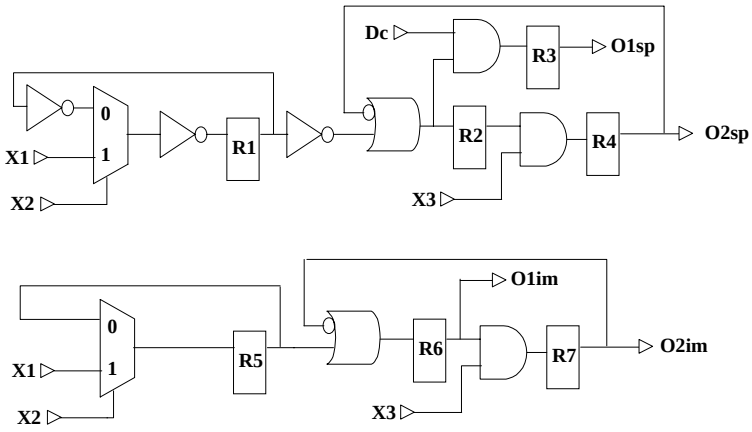


Fig. 17. Example of application of our functional matching algorithm

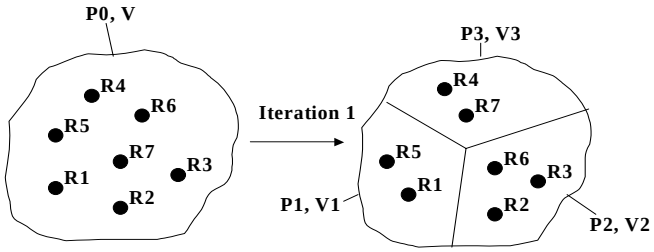


Fig. 18. First iteration of our functional matching in the example of Figure 17

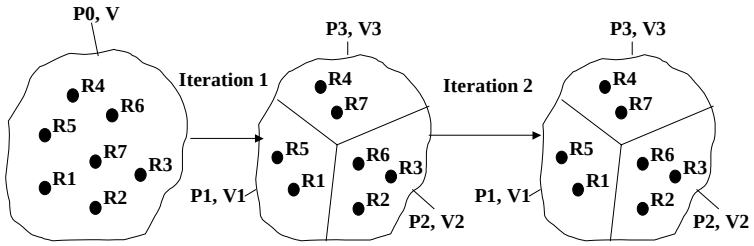


Fig. 19. Second iteration of our functional matching in the example of Figure 17

functions of FFs depending only on the primary inputs do not change. Moreover, when a new class is created, only the input functions of the FFs present in the transitive fan out (TFO) of the FFs in the new class need to be computed, other functions remain the same. Another improvement consists not in starting with only one class, but using random simulation to create more classes. The experimental results show that these improvements greatly reduce the CPU time.

However, due to function representation issue (BDD blow-up), our method cannot in practice match all the FFs in the two designs under verification. Thus, a structural matching is combined with our functional matching technique in order to solve this problem.

5.3 Structural Matching

One major problem of existing structural techniques is that they are not able to handle sequential optimizations. Indeed, in case of optimizations, existing structural matching are often incomplete and lead to false negatives problem. In order to solve this problem for an important set of designs, our structural matching algorithm combines several structural methods to be as complete as possible in case of sequential optimizations. The structural methods used are simulation, name and support matching.

Structural techniques only give the assumption that the input function of two FFs is equivalent or included. In order to avoid false negatives problems, this assumption should be correct. The combination of different structural techniques produces more correct assumptions that existing structural matching technique based only on name or support matching.

Moreover, in order to decrease the use of structural matching as much as possible, we combine our structural matching algorithm with our functional matching algorithm in a fixpoint algorithm as illustrated in Figure 20. Indeed, matching FFs with a structural technique may lead to new functional matching. All FFs matched with our structural technique need then to be proved with classical combinational techniques.

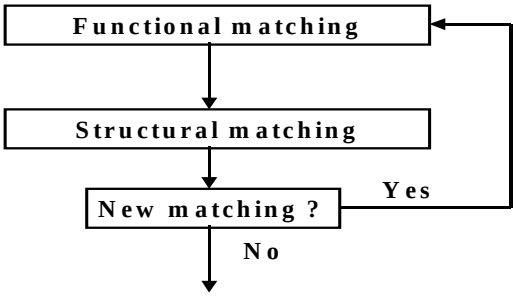


Fig. 20. Combination of our structural and functional matching algorithms

5.4 Redundancy Removal Engine

Synthesis tools can remove redundant FFs from the design under conception. In this case, the redundant FFs of the specification design do not have any correspondent FFs (equivalent or included FFs) in the implementation design. The use of matching algorithms, which only try to find a match between the FFs, is therefore not efficient in order to handle such optimizations. Our matching engine as illustrated in Figure 13 integrates a redundancy removal engine in order to find redundant FFs, removed them from the specification design and propagated the redundancy value. The redundancy removal engine algorithm is illustrated in Figure 21.

The algorithm is based on two fixpoint algorithms. The first is used to handle all constant, "stuck at" FFs, the second one to find all non-observable FFs. Fixpoint algorithms are needed since every time a FF is removed from the design, new FFs can become redundant. Two fixpoints are needed because removing constant, "stuck at" FFs does not result in new non-observable FFs, and removing non-observable FFs does not result in new constant or "stuck at" FFs.

```
Void RemoveRedFFs(Netlist *nl) {
        While (fixpoint1) {
                RemoveConstFFs(nl);
                RemoveStuckAtFFs(nl);
        }

        While(fixpoint2) {
                RemoveNonObsFFs(nl);
        }
}
```

Fig. 21. Our redundancy removal engine algorithm

5.4.1 Removing Constant FFs

To detect constant input FFs, the input function of some candidate FFs is computed using BDD representation. All FFs with input function equals to *bdd_zero* or *bdd_one* are constant input FFs. Constant propagation through the FFs is then done and the FFs are removed. The candidate FFs come from a random simulation. This reduces the number of BDDs to compute.

5.4.2 Removing "stuck at" FFs

The algorithm to detect "stuck at" FFs is based on an induction technique. The algorithm is presented in Figure 22. First the initial state of the FF is considered. If the initial state of the FF is 0 (resp. 1), it is supposed that the input of the FFs is equal to 0 (resp. 1) at time T. Then, the input function of the FF is computed at time $T + 1$, if the input function of the FF is still equal to 0 (resp. 1) at this time, the FF is considered as a stuck at 0 FF. The FF is then removed and the value of the redundancy 0 (resp. 1) is propagated.

```
RemoveStuckAtFFs(Netlist *nl) {
    If (init_state(Ri) = = 0) {
        Δi (T) = 0;
        Compute(Δi (T+1));
        If ((Δi (T+1) = = 0) {
            Remove(Ri);
            Propagate(0);
        }
    }
    If (init_state(Ri) = = 1) {
        Δi (T) = 1;
        Compute(Δi (T+1));
        If ((Δi (T+1) = = 1) {
            Remove(Ri);
            Propagate(1);
        }
    }
}
```

Fig. 22. Algorithm to remove "stuck at" FFs

5.4.3 Removing Non-Observable FFs

To remove non-observable FFs, our algorithm check if the output variable V_i of a candidate FF R_i is present in the function of all FFs and primary output in their transitive fanout. This can be done using AIG ("And Inverter Graph") or BDD

representation. If the variable is not present in the AIG of the FFs and primary output in the transitive fanout of the candidate FF, R_i is a non-observable FF. If the variable is present is the AIG, no conclusion can be made and the BDD representation is needed. This algorithm can be very time consuming because of the BDDs complexity. However the use of simulation techniques on the BDDs shows that the time can be decreased considerably, an average gain of 80 % has been achieved.

5.4.4 Coupling with the Matching Engine

In this section, we illustrated on an example how our redundancy removal engine is coupled with our matching engine on a fixpoint algorithm. The two designs under verification are represented in Figure 23. In the specification design, the synthesis tool has removed the framed part because it has been found redundant. Therefore, only two FFs (R_8 and R_9) remain in the implementation design.

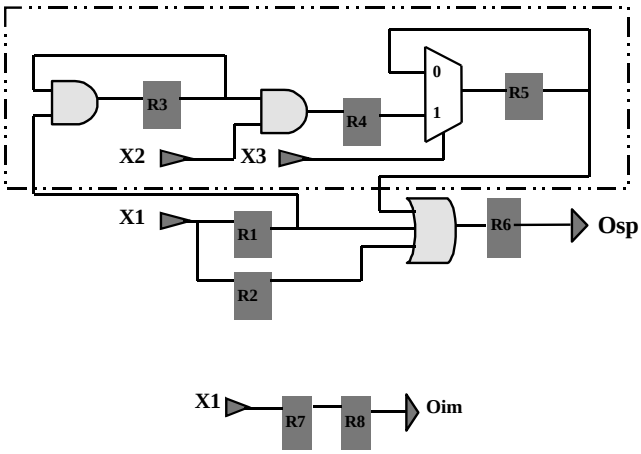


Fig. 23. Example of application of our matching engine

The first step of our algorithm is to run our functional matching algorithm. This results to match R_1, R_2 and R_7 together. All other FFs remain unmatched as illustrated in Figure 24.

The second step of the algorithm is to run our redundancy removal engine. The first iteration of this engine removes R_3 because it is a "Stuck at" 0 FF as illustrated in Figure 25. The second iteration removes R_4 because it is a constant 0 FF as shown in Figure 26. The third iteration removes R_5 because it is a "stuck at" 0 FF as illustrated in Figure 27. At the next iteration the fixpoint is reached (no more redundant FF have been found).

The third step of the algorithm is to run our functional matching again. This results to match R_6 and R_8 together as illustrated in Figure 28.

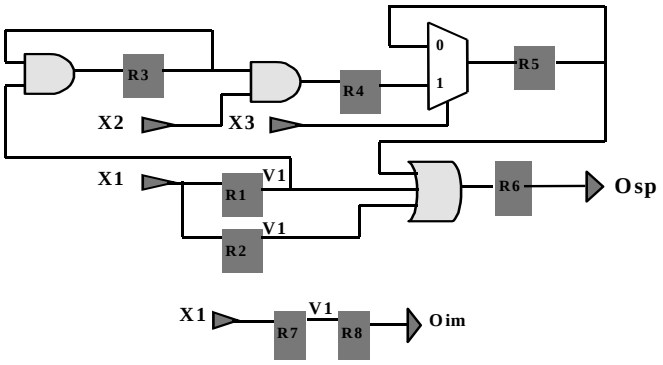


Fig. 24. First step of our matching engine (functional matching match R_1, R_2 and R_7 together)

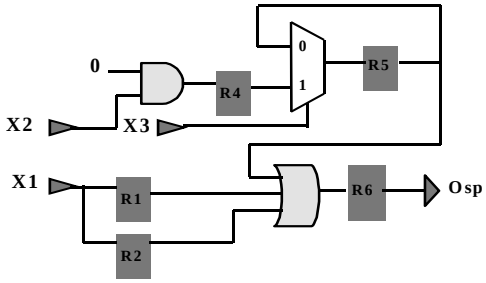


Fig. 25. First iteration of the redundancy removal engine (R_3 is removed)

After the third step, it can be concluded that the outputs of the two designs under verification are equivalent.

6 EXPERIMENTAL RESULTS

This section reports the experimental results of our procedure on circuits synthesized and sequentially optimized using Synplify Pro 7.3 tool (`www.synplicity.com`). We use a corporate BDD package for the functional matching. All tests are performed using a Dell Precision 833 Mhz workstation with a memory limit of 1 Gb. We compare our method with the last VIS version on ISCAS89 circuits (Table 1), and with an industrial sequential equivalence-checking tool on some industrial designs (Table 2). For all tests, we have reported the matching (complete/incomplete), the rate of matched FFs by the functional and non-functional part of the procedure, the CPU time and whether the test is proved. The time limit has been set to 1 hour for the comparison with VIS on the ISACS89 benchmarks. For the comparison with

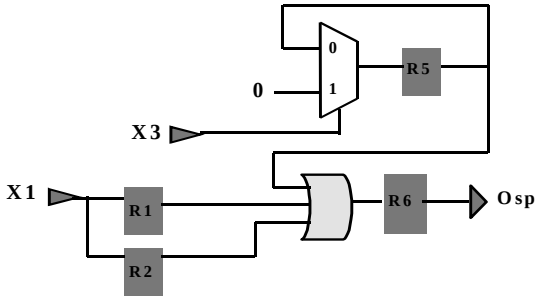


Fig. 26. Second iteration of the redundancy removal engine (R_4 is removed)

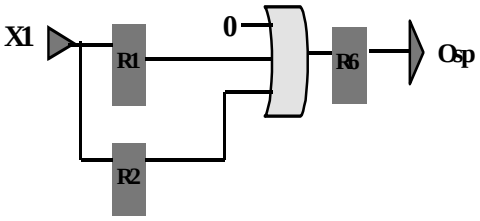


Fig. 27. Third iteration of the redundancy removal engine (R_5 is removed)

the industrial tool, the time limit has been set to 3 days. Note that VIS runs out of time for industrial circuits.

Table 1 shows that our procedure passes quickly 24 tests over 25, VIS passes only 17 tests. The average rate of FFs matched functionally is 91 % and structurally 9 %. The circuit S38584 does not pass with our procedure because we cannot produce a completion for this test due to function representation issues (BDD blow-up).

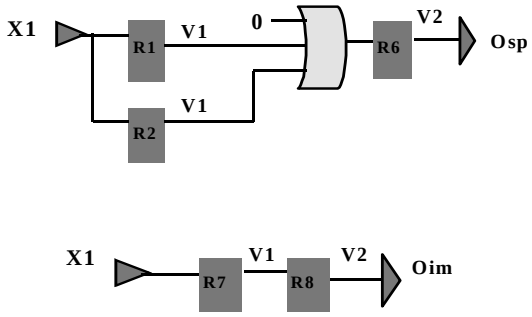


Fig. 28. Third iteration of our matching engine (functional matching match R_6 and R_8 together)

Circuit	# FFs orig./opt	VIS		Our procedure				
		Time (s)	Pass	Time (s)	Pass	Matching	FM (%)	NFM (%)
S208	8/12	0.8	Yes	0.9	Yes	Complete	100	0
S298	14/29	3.2	Yes	0.9	Yes	Complete	100	0
S344	15/27	18	Yes	1.4	Yes	Complete	100	0
S382	21/24	24	Yes	0.9	Yes	Complete	100	0
S386	6/6	10	Yes	0.9	Yes	Complete	100	0
S420	16/20	82	Yes	0.9	Yes	Complete	100	0
S444	21/29	15	Yes	1.4	Yes	Complete	100	0
S510	6/12	2932	Yes	1.5	Yes	Complete	100	0
S526	21/32	76	Yes	1.4	Yes	Complete	100	0
S641	19/17	4	Yes	1.4	Yes	Complete	100	0
S713	19/17	5	Yes	1.4	Yes	Complete	100	0
S832	5/5	236	Yes	1.4	Yes	Complete	100	0
S838	32/74	–	No	4.5	Yes	Complete	88	12
S953	29/62	186	Yes	1.5	Yes	Complete	100	0
S1196	18/18	3.8	Yes	1.5	Yes	Complete	100	0
S1238	18/18	4.3	Yes	1.75	Yes	Complete	92	8
S1423	74/154	–	No	27.3	Yes	Complete	78	22
S1488	6/6	1.1	Yes	1.4	Yes	Complete	100	0
S1494	6/6	0.9	Yes	1.5	Yes	Complete	100	0
S5378	179/253	–	No	5.2	Yes	Complete	78	22
S9234	160/62	–	No	2.3	Yes	Complete	82	18
S13207	648/431	–	No	3.1	Yes	Complete	88	12
S15850	563/304	–	No	11	Yes	Complete	69	31
S35932	1728/1728	–	No	9.1	Yes	Complete	74	26
S38584	1301/1406	–	No	68	No	Incomplete	36	64

Table 1. Experimental results on sequential and logically optimized ISCAS89 circuits

Table 2 shows that the proposed matching engine passes 21 tests over 23 industrial circuits where the industrial tool passes only 14 tests.

The average rate of FFs matched functionally is 69 % and non-functionally 31 %. The non-passed tests using the industrial tool occur when the tests are proved different due to incomplete matching (False negative problem). Our procedure results in a false negative for circuit I17 due to an incomplete matching. Our procedure does not find a FFs replication optimization because of BDD size limit. The circuit passes if the replication information is given using the user matching input of our procedure. Circuit I21 does not pass with our method because the procedure does not finish with in the time limit of 3 days. This circuit is successfully tested with the industrial tool. We believe that the industrial tool has a better and faster SAT/BDD solver than ours. This is confirmed with circuit I20, where both our procedure and the industrial tool pass, but the industrial tool verified the circuit

Circuit	# FFs orig./opt	Industrial tool			Our procedure				
		Time (s)	Pass	Matching (s)	Time	Pass	Matching	FM (%)	NFM (%)
I1	255/277	2.2	Yes	Complete	2.9	Yes	Complete	72	28
I2	255/270	1.9	Yes	Complete	2.2	Yes	Complete	100	0
I3	158/206	5.2	No	Incomplete	2.3	Yes	Complete	18	82
I4	17/18	0.9	Yes	Complete	1.5	Yes	Complete	100	0
I5	1437/1437	345	Yes	Complete	595	Yes	Complete	95	5
I6	101/105	1.5	Yes	Complete	1.7	Yes	Complete	100	0
I7	222/211	36	No	Incomplete	24	Yes	Complete	12	88
I8	523/568	11	No	Incomplete	7.2	Yes	Complete	100	0
I9	354/219	6.3	No	Incomplete	5.4	Yes	Complete	95	5
I10	565/537	4.1	Yes	Complete	4.6	Yes	Complete	100	0
I11	381/384	3.4	Yes	Complete	3.3	Yes	Complete	38	62
I12	72/72	18.2	Yes	Complete	20.6	Yes	Complete	0	100
I13	120/162	36.3	No	Incomplete	32.5	Yes	Complete	34	66
I14	996/1251	78.6	No	Incomplete	56.4	Yes	Complete	100	0
I15	524/611	24.5	No	Incomplete	19.7	Yes	Complete	100	0
I16	48/52	3.4	Yes	Complete	4.2	Yes	Complete	21	79
I17	250/246	345	No	Incomplete	222	No	Incomplete	0	100
I18	419/439	12.5	Yes	Complete	18.3	Yes	Complete	76	24
I19	358/369	14.1	Yes	Complete	21.2	Yes	Complete	82	18
I20	357/360	2378	Yes	Complete	5288	Yes	Complete	42	58
I21	347/249	26378	Yes	Complete	–	No	Complete	57	43
I22	432/444	92.3	No	Incomplete	68.1	Yes	Complete	76	24
I23	236/237	36200	Yes	Complete	172800	Yes	Complete	75	25

Table 2. Experimental results on sequential and logically optimized industrial circuits

twice faster (2 378 s) than our method (5 288 s). Note that both the industrial tool and our matching procedure produce a false negative with an incomplete matching. This confirms the importance of a complete matching to resolve the sequential equivalence-checking problem.

Comparisons done in [9, 14] and [15] show better performances, but we could not optimize the circuits using the same tools and we did not know the exact machine used for the testing. For example, circuit S838 is proved in 55.6 seconds with apparently the same optimizations (same number of FFs after optimizations) in [9] on a HP9000/755, our method proved this circuit in 4.5 s. Circuit S1196 is proved in 159.3 s with apparently the same optimizations in [14] on a Sun machine; our method proved the circuit in 1.5 s. Circuit S35932 is proved in 75 s in [15] on a Sun-sparc5 machine; our method proved it in 9.1 s.

7 CONCLUSION

In this paper, we show why sequential optimizations are a real bottleneck for matching based equivalence checking (Section 3). We therefore introduce an efficient and complete matching engine in Section 4. This engine uses several techniques to find complex optimizations. We present a functional matching technique (Section 5.1) based on a powerful partitioning technique, which is able to find functional equivalent and included FF. A redundancy removal engine is also introduced in order to perform complex pruning as stuck at FFs (Section 5.3). We show why structural

matching should sometimes be applied to improve the matching process. We present our structural matching, which combines several techniques to make the assumption correct in most cases. Finally, we show how this matching is combined with our functional matching (Section 5.2).

The experimental results show a clear advantage to our procedure. The results show that our procedure is more reliable than an industrial tool (less false negatives). These improved results are due to several factors. The first is our incremental approach; every time new matching is found, the entire procedure is run again. The second is the use of several engines (functional matching, structural matching, complex redundancy removal), which is more likely to produce a complete matching in most cases. Another advantage of our method is due to our functional matching, which handles 69 % of the matched FFs for the industrial circuits and 91 % for the ISCAS89. Thus, our procedure finds a greater of complex sequential optimizations and produces a complete and better matching than the industrial tool.

REFERENCES

[1] Akers, S.: Binary Decision Diagram. IEEE Trans. on Computers, Vol. C-27, No. 6, pp. 509–516 (June 1978).

[2] Kuelmann, A.—Krohm, F.: Equivalence Checking Using Cuts and Heaps. In Design Automation Conference (DAC) 1997, pp. 263–268.

[3] Reddy, S. M.—Kunz, W.—Pradhan, D. K.: Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment. In Design Automation Conference (DAC) 1995, pp. 414–419.

[4] Brand, D.: Verification of Large Synthesized Designs. In IEEE International Conference on Computer-Aided Design (ICCAD), 1993, pp. 534–537.

[5] van Eijk, C.: Formal Methods for the Verification of Digital Circuits. Ph.D. Thesis of the Eindhoven University of Technology, Eindhoven, The Netherlands, September 1997.

[6] Burch, J. R. et al.: Symbolic Model Checking for Sequential Circuit Verification. IEEE Trans. On Computer-Aided Design Of Integer Circuits and Systems, Vol. 13, No. 4, pp. 401–424, April 1994.

[7] Rahim, S.—Rouzeyre, B.—Torres, L.—Rampon, J.: Loop Problem in Sequential Equivalence Checking. In SAME 2002, pp. 52–58.

[8] Burch, J. R.—Singhal, V.: Robust Latch Mapping for Combinational Equivalence Checking. In ICCAD, 1998, pp. 563–569.

[9] van Eijk, C.: Sequential Equivalence Checking without State Traversal. In Design Automation and Test in Europe (DATE), February 1998, pp. 618–623.

[10] Rahim, S.—Rouzeyre, B.—Torres L.—Rampon, J.: An Efficient Flip-Flops Matching Engine. In DDECS (IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems), Stará Lesná, 2004.

[11] Rahim, S.—Rouzeyre, B.—Torres L.—Rampon, J.: An Equivalence Checking Flow to Verify Sequential Optmizations. In poster session of Formal Methods

and Models for Codesign (MEMOCODE '2004), June 22–25, 2004, San Diego Hilton Harbor Island, San Diego, California.

[12] Rahim, S.: Equivalence Checking for Sequentially Optimized Designs. Ph.D. Thesis of Montpellier University, LIRMM, Montpellier, France, July 2004.

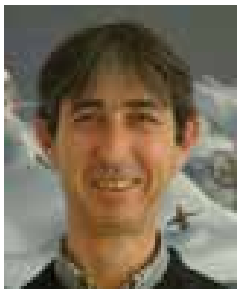
[13] Rahim, S.—Rouzeyre, B.—Torres L.—Rampon, J.: Matching in the Presence of Don't Cares and Redundant Elements for Sequential Equivalence Checking. In HLDVT, 2003, pp. 129–135.

[14] Huang, S-Y.—Cheng, K.-T.—Chen, K-C.—Glaeser, U.: An ATPG-Based Framework for Verifying Sequential Equivalence. In ITC, 1996, pp. 865–874.

[15] Huang, S.-Y.—Cheng, K.-T.: AQUILA: A Local BDD-Based Equivalence Verifier. Informal Equivalence Checking and Design Debugging, pp. 90–109.

Solaiman Rahim received his Master degree in microelectronics and computer sciences in 2001 from the French "grande école" ESEO (École Superieur d'Electronique de l'Ouest) and his Ph.D. degree in 2004 from LIRMM (Laboratory of computer sciences, Microelectronic and Robotics of Montpellier, France), attached to the University of Montpellier. He is currently a Senior Formal Verification Engineer at Synplicity in Sunnyvale, CA, USA. His research interests include design verification, synthesis, and optimizations.

Lionel Torres received his Master degree in 1993 and his PhD degree in 1996 from the University of Montpellier in France. From 1996 to 1997 he worked as IP core methodology engineer for ATMEL company. He is currently Professor in the microelectronic department of the LIRMM (Laboratory of computer sciences, Microelectronic and Robotics of Montpellier, France) at University of Montpellier II (France). He has been active in several microelectronics design conferences as topic chair, steering committee member, program committee member, conference organization team member. His research interests include computer architectures, reconfigurable computing, and design verification.



Bruno Rouzeyre received his M.S. degree in mathematics in 1978, PhD degree in CAD in 1984 and "Habilitation á diriger les recherches" in 1992, all degrees from the University of Montpellier, France. Currently he is Professor at the University of Montpellier where he heads the electronic engineering school. He does his research at LIRMM. His main research interests include synthesis and test of integrated circuits and formal verification.