AUTOMATING THE DIAGNOSIS AND THE RECTIFICATION OF DESIGN ERRORS WITH PRIAM

Jean Christophe Madre
Olivier Coudert
Jean Paul Billion
BULL Corporate Research Center P.C. 58B1
68, Route de Versailles
78430 Louveciennes FRANCE

Abstract
This paper presents the original extensions brought to PRIAM to automate both the diagnoses and the rectification of the design errors detected by this tool. PRIAM is an industrial automated formal verifier used to check the functional correctness for digital circuits of up to 20000 transistors. These extensions implement a new approach to diagnosis based on boolean equation solving. In particular, no enumeration of the faulty patterns is necessary to find out the incorrect gates in the circuit. The diagnosis system can handle any circuit that can be verified by PRIAM.

1. Introduction

PRIAM is an automated formal verifier of digital circuits now used by industrial designers [6]. It checks the functional correctness of circuits with respect to their specifications. The behavioural specifications as well as the circuit descriptions are written in the hardware description language LDS. PRIAM formally proves the equivalence or the implication between LDS programs and thus establishes the correctness of the described circuits.

When PRIAM detects an error during the verification of a circuit, it provides the circuit designer with a description of the error: its kind, the related variable, and the equation of the set of input patterns that make the error observable. However, until now, no help was given to the designer to diagnose this error, that is to find out the reasons of the incorrect behaviour of the circuit. Except for very simple errors, this diagnosis task generally takes several hours, to which must be added several hours for the rectification.
Table 1. Verification times with PRIAM (BULL DPX5000).

<table>
<thead>
<tr>
<th>circuit</th>
<th>#inputs</th>
<th>#outputs</th>
<th>#trans.</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>operabc</td>
<td>160</td>
<td>50</td>
<td>4100</td>
<td>9mn</td>
</tr>
<tr>
<td>addmul</td>
<td>100</td>
<td>45</td>
<td>5400</td>
<td>15mn</td>
</tr>
<tr>
<td>tdat</td>
<td>297</td>
<td>192</td>
<td>8400</td>
<td>12mn</td>
</tr>
<tr>
<td>scd5</td>
<td>663</td>
<td>591</td>
<td>17000</td>
<td>20mn</td>
</tr>
</tbody>
</table>

This paper presents the diagnosis and rectification system we have developed for the debugging of gate level circuit descriptions. This system is based on the equation solving facility of the automated propositional theorem prover of PRIAM. The diagnosis and the rectification methods are pure formal methods. The diagnosis method overcomes the defaults of the methods based on enumeration of the faulty patterns. For instance under the single fault assumption, each gate connected to some incorrect output is analyzed once to determine whether it is responsible for the incorrect value of this output.

The paper is divided in 4 parts. Part 2 presents the verifier PRIAM and its underlying propositional prover. Part 3 describes the diagnosis method. Part 4 explains the rectification system determines whether the possibly incorrect gates of the circuit can be modified in order to make the circuit function well. Part 5 gives the fundamental results on theorem proving and boolean equation solving that underly the diagnosis system.

2. The Formal Verifier PRIAM

PRIAM is a formal verifier of functional correctness of digital circuits. This tool has been integrated in the CAD system used by the circuit designers at BULL. Table 1 gives some verification times for industrial circuits. Note that none of these circuits could have been verified by simulation because of their very large number of inputs.

Within BULL’s design methodology, the behaviour of synchronous circuits is described with the hardware description language LDS [6]. Descriptions are at the cycle level. They are written in a procedural way like VHDL processes [11]. This means that the behavioural specification of a circuit written in LDS describes how the output and the transition functions are computed. All the storage elements of the circuit must be declared and there must be no loop without an included storage element in the circuit.
PRIAM uses symbolic execution of the LDS description of the circuit to compute its output and transition functions. Symbolic execution consists in executing a program $P$ with symbolic instead of logical values assigned to its inputs [4]. Since the output and transition functions of the circuit can be partial functions, PRIAM manipulates partial boolean functions that are represented by contextual values. A contextual value $(C_f, V_f)$ is a couple of boolean expressions where $C_f$ represents the domain of definition of the function $f$ and $V_f$ represents the value of function $f$ on its domain.

All along the symbolic execution of a LDS program $P$, PRIAM has to make proofs in order to establish that the program $P$ is correct with respect to LDS semantics [6]. PRIAM is built on a powerful propositional prover that makes these proofs simple to perform. This prover is based on a new canonical representation of propositional formulae called the typed decision graphs (TDG) [6].

The formal comparison between two LDS programs $P_S$ and $P_R$ is performed through two steps. First PRIAM symbolically executes the programs $P_S$ and $P_R$ to compute the contextual values of each output $o$ of the programs. We note these contextual values $(C_o(P_S), V_o(P_S))$ and $(C_o(P_R), V_o(P_R))$ respectively. Then PRIAM compares these contextual values according to the selected comparison criterion. If $P_R$ must be proved to be implied by $P_S$, which is the most used comparison criterion between LDS programs, the formulae to be proved valid for each output $o$ are the following:

$$C_o(P_S) \Rightarrow C_o(P_R), \text{ and } C_o(P_S) \Rightarrow (V_o(P_S) \Leftrightarrow V_o(P_R))$$

This means that the values computed by $P_S$ and $P_R$ are equal on the domain specified by $P_S$. When PRIAM proves that these formulae do not hold for some output of the programs $P_S$ and $P_R$, then the circuit is declared to functionally incorrect. PRIAM then provides the designer with a description of the error: the associated output, the error kind (incorrect context of incorrect value) and the equation of the set of all the faulty patterns. However until now PRIAM did not provide any help to the designer for finding the causes of this error.

Next section shows that the tedious diagnosis task can be automated. It then shows that under some conditions the diagnosis system can also provide the designer with possible rectifications of the circuit.
3. Automating the Diagnosis of Design Errors

This section shows that diagnosing design errors is a simpler problem than diagnosing faults in real circuits. It then presents the diagnosis method and it gives some experimental results.

Diagnosing design errors is quite different from diagnosing faults in real circuits. Many different kinds of errors can be introduced during the fabrication of a circuit that are difficult to model correctly [7]. On the other hand, when dealing with design errors, only one kind of fault needs to be modeled: a circuit does not function well because some of its gates are incorrect. This does not mean that these gates are incorrectly implemented but rather that they are misused in the logical network. For this reason when several outputs of a circuit are declared incorrect by PRIAM the errors are considered independent and they are analyzed separately.

Different methods have been proposed to diagnose errors in circuits whose structure is known [9, 10]. All are based on enumeration of the faulty patterns, which are the input patterns that make the error observable. In the worst case all the faulty patterns must be enumerated to determine the set of possibly incorrect gates. Moreover none of these methods directly supports automated rectification. The diagnosis method proposed here eliminates this enumeration. Under the single fault assumption, a one-pass process over the gates of the circuit produces the exact set of gates that can be held responsible for the error detected by PRIAM. This set is empty if and only if the single fault assumption does not hold.

3.1 The Diagnosis Method

For the sake of clarity this section presents the diagnosis method on a pure combinational circuit, that is a circuit without any transistors used as switches. We consider a combinational circuit $C$ with $n$ inputs noted $i_1, \ldots, i_n$. This circuit has several outputs and at least one of these outputs $o$ has an incorrect value. This means that the specified boolean function $f_s$ and the boolean function $f_r$ produced by the circuit at this output $o$ are not equivalent: $f_r \neq f_s$.

The first step in the diagnosis consists in computing the set $S$ of gates in the circuit that are directly or indirectly connected to the output $o$. Indeed only these gates can be responsible for the error. Following [7], we call this set $S$ of gates the coverage cone of the output $o$. It contains in general a small part of all the gates of the circuit. It is computed, like in [7], by an algorithm that traverses the circuit $C$ backward from the output $o$ to the inputs $i_1, \ldots, i_n$. 
The second step in the diagnosis is to determine which gates in the coverage cone $S$ are really responsible for the incorrect value of the variable $o$. Under the single fault assumption, this problem is the same than determining whether the behaviour of any gate $G$ in $S$ can be modified in such a way that the output function $f_r$ becomes equivalent to the expected function $f_s$. If we note $[o_1 \cdots o_p]$ the outputs of the gate $G$, the diagnosis problem comes down to finding a $p$-tuple of boolean functions $[f'_o \cdots f'_p]$ that should be produced by the gate $G$ so that $f_r \equiv f_s$.

In order to find the functions $[f'_o \cdots f'_p]$ we create a vector of boolean variables noted $[z_1 \cdots z_p]$ and we propagate them in the circuit $C$ in place of the functions $[f_o \cdots f_p]$ actually produced by the gate $G$. This propagation, which uses PRIAM’s symbolic execution mechanism, produces the function $f'_r(i_1, \ldots, i_n, z_1, \ldots, z_p)$ at the output $o$. Thanks to the free variables $z_1, \ldots,$ and $z_p$, the boolean function $f'_r$ represents all the boolean functions that could be obtained at the output $o$ by modifying the behaviour of the gate $G$. In order for the functions $f_s$ and $f'_r$ to be equivalent, we must find, for each set of values that can be assigned to the inputs of the circuit, the values of the variables $z_1,\ldots,z_p$ such that: $f_s(i_1,\ldots,i_n) = f'_r(i_1,\ldots,i_n, z_1,\ldots,z_p)$. This is expressed by the following theorem.

**Theorem.** The gate $G$ is possibly responsible for the incorrect value at the output $o$ if and only if the following formula is valid:

$$\forall i_1 \cdots i_n, \exists z_1 \cdots z_p, f'_r(i_1, \ldots, i_n, z_1, \ldots, z_p) = f_s(i_1, \ldots, i_n). \quad (1)$$

Section 5 explains how this non-trivial theorem can be automatically proved by the propositional prover of PRIAM. It also shows that the proof procedure can give us the functional values of the variable $z_1,\ldots,$ and $z_p$. These functional values represent all the functions that could be produced by the gate in order for the circuit to be correct. These functional values are used by the rectification system presented in Section 4 to provide the designer with the correct equations of the gate $G$.

### 3.2 Experimental Results

The performance of the diagnosis method mainly depends on the time needed to propagate the vector $[z_1 \cdots z_p]$ to the output $o$. The variables $z_1, \ldots,$ and $z_p$ replace possibly complex boolean functions so their propagation in place of these functions can be expected to require less time. This is clearly shown in Table 2. It guarantees that any circuit that can be verified with PRIAM can be handled by the diagnosis system.

Table 2 gives the diagnosis times for some industrial circuits. It gives the number “#stats” of statements in the LDS program describing the
Table 2. Diagnosis time for 32 bit circuits (BULL DPX5000).

<table>
<thead>
<tr>
<th>circuit</th>
<th>#stats</th>
<th>#cone</th>
<th>CPU time</th>
<th>#faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>add32/s15</td>
<td>269</td>
<td>53</td>
<td>48s</td>
<td>3</td>
</tr>
<tr>
<td>alu32/s14</td>
<td>174</td>
<td>51</td>
<td>192s</td>
<td>4</td>
</tr>
<tr>
<td>alu32/s0</td>
<td>174</td>
<td>89</td>
<td>660s</td>
<td>12</td>
</tr>
</tbody>
</table>

circuit, the number of statements “#cone” in the coverage cone of the incorrect output, the total time “time” needed for the diagnosis and finally the number of possibly incorrect statements (or gates) “#faults”. Note that the circuit add32 is described at the gate level and that the circuit alu32 is built out of standard cells.

The diagnosis procedure given above uses the single fault assumption. Experience shows that experienced designers make very few faults, so this assumption holds in many cases. When it does not hold, the diagnosis system will be unable to find a gate that can be held responsible for the incorrect behaviour of the circuit. Nevertheless the diagnosis procedure can be applied to multiple faults except that a tuple of gates of the set $S$ instead of only one gate must be considered at each step. We are then faced with the usual combinatorial explosion in the search.

4. Rectifying Design Errors

Section 3 has presented a procedure for finding the gates that can be responsible for the incorrect value of some output of a circuit. We now show that this procedure gives us enough information for determining which of these gates can be rectified in order for the output value to become correct.

The key idea here is that the formula 1 given in Section 3.1 can be considered as a theorem to be proved but also as an equation to be solved. Solving this equation with the procedure given in Section 5 provides us with the functional values $[f'_{o_1} \cdots f'_{o_p}]$ of the variables $[z_1 \cdots z_p]$. These functions are the boolean functions that should be produced by the gate $G$ under analysis for the output function $f_r$ to be correct. Note that there can exist more than one solution to the boolean equation 1. The solver then introduces a finite number of parameters in the functions. This means that $[f'_{o_1} \cdots f'_{o_p}]$ are higher-order functions. They represent the set $F$ of all different vectors of output functions of the gate $G$ for which the value of the output $o$ is correct.
The problem we address here is to determine whether it is possible to rectify the gate $G$, without changing the global structure of the circuit. If this is possible, then the cost of rectifying the design error is minimal because the structure of the circuit is not modified. We note $[f_1 \cdots f_m]$ the tuple of boolean functions taken as inputs by the gate $G$. The gate $G$ produces the outputs functions $[f_{o_1} \cdots f_{o_p}]$. This means that $G$ implements $p$ compositions noted $h_1, \ldots, h_p$ such that $h_i(f_1, \ldots, f_m) = f_{o_i}$. To rectify the gate $G$ is to find $p$ compositions $h_1, \ldots, h_p$ that produce correct functions on the gate outputs. This is expressed by the following theorem.

**Theorem.** The gate $G$ can be rectified if and only if the following formula is valid:

$$\exists h_1 \cdots h_p, [h_1(f_1, \ldots, f_m) \cdots h_p(f_1, \ldots, f_m)] \in \mathcal{F}. \quad (2)$$

The resolution procedure of this equation is given in Section 5.2. It returns, if they exist, all the functions $[h_1 \cdots h_p]$ that compose the input functions of the gate $G$ in such a way that the circuit $C$ produces a correct function at the output $o$. When several solutions exist the designer has to choose between them the better one according to some criteria.

The rectification problem is essentially combinatorial. When the gate $G$ has $m$ inputs and $p$ outputs, the resolution procedure given in Section 5.2 must deal with $p2^m$ boolean variables. This combinatorial explosion restricts the rectification method to be applied on circuits whose gated have a small number of inputs and outputs. When this is not the case, for instance when complex standard cells are used in the circuit, the rectification problem looks very much like the synthesis one and the same problems are encountered.

### 4.1 Experimental Results

The CPU time needed for trying to rectify a gate directly depends on its number of inputs and the complexity of its input functions. No rectification is attempted when the number of inputs is larger than 10. Experience shows that for the 32 bit circuits, whose diagnosis times are given in Table 2, these CPU times are less than 5 seconds. None of the $3$ possible incorrect gates of the 32 bit adder $add32$ can be rectified. For the 32 bit ALU $alu32$ the system finds that only one gate can be rectified in order for the output $s14$ to become correct, and also only one gate for the output $s0$. Eventually these gates are the same.
5. The Boolean Equation Solver

We have presented in [6] the propositional theorem prover underlying PRIAM. This prover is based on a new canonical form of the propositional formulae that we have called Shannon’s typed canonical form. Formulae in this form are represented by graphs called Typed Decision Graphs (TDG).

In PRIAM the prover is mainly used as a rewriting system for reducing propositional formulae to their canonical form. In this section we show that it can also be used to prove quantified formulae. We then show that proving a quantified formula valid is similar to solving a boolean equation and we give in Section 5.2 several resolution procedures.

5.1 Validity of a Quantified Expression

We consider here a higher-order logic with a finite domain of interpretation. Since the domain of interpretation is finite, any closed formula in this logic can be rewritten into a closed formula whose variable are propositional variables [5]. For instance the formula \((\forall x_1 x_2, \exists x_3, (x_1 \lor x_3) \Leftrightarrow (x_2 \lor x_3))\) is such a quantified formula. A term is a formula without any quantifier. We note \(f/x=y\) the formula obtained by substituting each occurrence of the variable \(x\) in \(f\) by the term \(y\).

The validity of any closed quantified formula is inductively defined in the following standard way:

- the formula \(1\) is valid and the formula \(0\) is not valid.
- the closed formula \((\forall x f)\) is valid if and only if the formula \(f/x=0\) and the formula \(f/x=1\) are valid.
- the closed formula \((\exists x f)\) is valid if and only at least one of the formulae \(f/x=0\) and \(f/x=1\) is valid.

The proof procedure called \texttt{valid}\ given in Fig. 5.1 is a direct implementation of this inductive definition. The formula to be proved valid is \((Q_1x_1, \ldots, Q_nx_n t)\) where \(Q_1, \ldots, Q_n\) are quantifiers and \(t\) is a term. The function valid takes as input Shannon’s tree of the term \(t\) built with the order \(x_1 < \cdots < x_n\) [6]. The proof procedure traverses this tree and determines, at each step, whether the subterm represented by some vertex in the tree is valid. The function returns \(1\) if the subtree is valid, else it returns \(0\).

When all the quantifiers of the formula \((Q_1x_1, \ldots, Q_nx_n t)\) are identical, the canonicity of the representation makes the proof trivial. For instance, the formula \((\forall x_1 \cdots x_n t)\) is valid if and only if the term \(t\) is a tautology. In the same way, the formula \((\exists x_1 \cdots x_n t)\) is valid if and only
Automating the Diagnosis and the Rectification Of Design Errors

Figure 1. Proof procedure of a quantified closed formula.

```pascal
type vertex = record
  index : integer; low, high : tdg;
end;
type tdg = record
  tag : '-', '+': node : vertex;
end;
var True, False : tdg;
function valid (t : tdg) : tdg;
var v : vertex;
begin
  if (t = True) or (t = False) then return t;
  v = t.node;
  if (∀-quantified v.root) then
    if (t.tag = '+') then
      return And(valid(v.low), valid(v.high));
    else return And(valid(Not(v.low), valid(Not(v.high))));
  else return Or(valid(v.low), valid(v.high));
end;
```

if \( t \) is not identically equal to 0. These remarks are used to optimize the proof procedure given in Fig. 5.1. In a more general way there is a proof procedure [3] that does not require the TDG of the term \( t \) to be built with the order \( x_1 < \cdots < x_n \).

5.2 Solving Equations

Any closed formula \((Q_1 z_1, \ldots, Q_n x_n t)\) can be seen as an equation whose unknown variables are its existentially quantified variables. From the theoretical point of view, solving this boolean equation \( t = 1 \) is the same problem than finding Skolem’s function of the existentially quantified variables [3]. We give here the resolution procedure for an equation that has only one unknown variables. The general resolution procedure can be found in [3].

Consider the equation \( t(x_1, \ldots, x_n, y) = 1 \) with only one unknown variables \( y \). We note tree \( t \) Shannon’s decomposition tree of the term \( t \) with the order \( y < x_1 < \cdots < x_n \). This tree can be written \(((\neg y) \land L) \lor (y \land H))\) where \( L \) and \( H \) are both in canonical form. Then two cases must be considered:
the formula \((L \lor H)\) is not a tautology. Then for some interpretation of the variables \(x_1, \ldots, x_n\), the formulae \(H\) and \(L\) both evaluates to 0, and the term \(t\) also evaluates to 0. It is this not possible to assign a value to \(y\) such that \(t\) evaluates to 1. This means that the equation has no solution.

the formula \((L \lor H)\) is a tautology. Then for any interpretation of the variables \(x_1, \ldots, x_n\), either the formula \(H\) or the formula \(L\) evaluates to 1. In any case it is possible to assign a value to \(y\) such that \(t\) evaluates to 1. When \(L\) and \(H\) both evaluate to 1 (this can happen if the formula \((L \land H)\) is not an antilogy) then any value can be assigned to \(y\). The solution of the equation is: \(y = (\neg L) \lor (H \land v)\) where \(v\) is a new free variable.

5.3 Solving a Functional Equation

This section explains how the rectification system solves the functional equation 2 presented in Section 4. Consider a set of boolean functions \(f_1, \ldots, f_m\), and \(f\) whose variables are \(i_1, \ldots, i_n\). The problem we address here is to find a boolean function \(h\) such that:

\[ h(f_1, \ldots, f_m) \equiv f. \]

(3)

There are \(2^{2m}\) Boolean functions with \(m\) inputs \(v_1, \ldots, v_m\). All these functions can be represented by a single TDG \(F_m\) built out of the variables \(v_1, \ldots, v_m\) and \(N = 2^m\) new boolean variables that we note \(h_1, \ldots, h_N\). This TDG has \(2^{m+1} - 1\) vertices. Any interpretation of the variables \(h_1, \ldots, h_N\) defines one and only one of these boolean functions. Fig. 5.3.1 shows \(F_2\).

Starting from the TDG \(F\) we can compute the TDG \(F_C\) that represents the set of boolean functions that can be obtained by composing the boolean functions \(f_1, \ldots, f_m\). This TDG is obtained by substituting the TDGs of the functions \(f_1, \ldots, f_m\) to the variables \(v_1, \ldots, v_m\) in the TDG \(F\). The variables occurring in \(F_C\) are the inputs variables \(i_1, \ldots, i_n\) and the variables \(h_1, \ldots, h_N\). Finally finding a function \(h\) that is solution of equation 3 is the same than solving a tuple \((h_1, \ldots, h_N)\) that is solution of the equation:

\[ \exists h_1 \cdots h_N, \forall i_1 \cdots i_n, F_C(h_1, \ldots, h_n, i_1, \ldots, i_n) = f(i_1, \ldots, i_n) \]

(4)

When this equation has solutions the resolution procedure returns the TDGs representing the values of \(h_1, \ldots, h_N\). If there is only one solution to the equation then all these TDGs are equal to 1 or to 0. However if there are several solutions then some parameters (at least \(N\)) occur in
Figure 2. Resolution of a functional equation.

these TDGs. These TDGs are then substituted back in the TDG $F_m$ and we obtain the TDG that represents all the boolean functions that are solution of the functional equation 3.

Fig. 5.3 shows the application of this resolution procedure to the case where $f_1 = ((\neg a) \land b) \lor (a \oplus (\neg c))$, $f_2 = (a \lor (\neg c))$ and the function to be obtained is $f = ((\neg a) \land b \land c)$. Fig. 5.3.2 shows the TDG $F_C$ that represents all the functions that can be obtained by composing the function $f_1$ and $f_2$. Fig. 5.3.3 shows the TDG of $f$. In order for the TDG of $F_C$ and $f$ to be isomorphic we must have $h_1 = h_2 = h_4 = 0$ and $h_3 = 1$. Fig. 5.3.4 is the TDG of the function $h = (v1 \land (\neg v2))$ that is the only solution of the functional equation.

6. Conclusion

This paper has presented the procedures we have developed to automate the diagnosis and the rectification of the design errors detected by the formal verifier PRIAM. The original diagnosis method is based on algorithms for proving quantified propositional formulae and for solving boolean equations. These algorithms operate on formulae in Shannon’s typed canonical form that are represented by Typed Decision Graphs. The diagnosis procedure can be applied to any circuit that can be verified by PRIAM. When the diagnosis system has formally proved that some gate can be held responsible for the incorrect behaviour of the circuit, the rectification system is called. It determines whether the circuit can be rectified without changing it’s structure and when this is possible it provides the designer with the rectified equations of the gate.
References


