Chapter 2
Preliminaries

2.1 Circuits and Sensitized Paths

Each combinational circuit is represented by a directed acyclic graph $C = (V, E)$, referred to as the *circuit graph*, where V is the set of circuit nodes and $E \subseteq V \times V$, the set of edges, corresponds to the gate input-output connections in the circuit [LRS89]. For gate-level benchmarks, we consider the nodes to be gates with symmetric functions. Each node in the circuit graph is associated with a symmetric function which represents the corresponding behavior of that gate in the circuit. A symmetric function does not depend on the order of inputs but only on the sum of variables assigned to 0 or to 1, respectively.

The basic gates used in this book for gate-level benchmarks are BUF, NOT, AND, OR, XOR, NAND, NOR and XNOR corresponding to the logic operators. Where relevant a FANOUT-gate may be inserted to differentiate between multiple fanout-branches of a gate. A FANOUT gate implements the identity function. For the RTL benchmarks, in addition to the logic operators, the word-level operators such as arithmetic operations (e.g. ADD, MUL, DIV), control and relational operations (e.g. ITE, LEQ, EQ) and array operations (e.g. READ, WRITE) are used [SKF+09]. In these cases the function associated to a node is not symmetric. Thus, the order of inputs to an operator is relevant to determine the function of the circuit. However, to simplify the presentation we do not reflect this in the following definitions. In the examples and explanations of this book, for the sake of simplicity, we use gates to explain the idea of our approaches.

The successors of a node $g \in V$ are given by a set of nodes $succ_g = \{h | (g, h) \in E\}$ and $|succ_g|$ is the number of successors of g. The predecessors of a node $g \in V$ are given by a set of nodes $pred_g = \{h | (h, g) \in E\}$ and $|pred_g|$ is the number of predecessors of g.

Gates are usually denoted by lower case latin letters in the circuit representation, e.g., gate a in Fig. 2.1. Letter I indicates the set of *Primary Inputs* (PI) and O is the set of *Primary Outputs* (PO). The PIs of a circuit are often denoted by $i_1, i_2, \ldots, i_n$

© Springer International Publishing Switzerland 2015

M. Dehbashi, G. Fey, *Debug Automation from Pre-Silicon to Post-Silicon*,
DOI 10.1007/978–3-319-09309-3_2

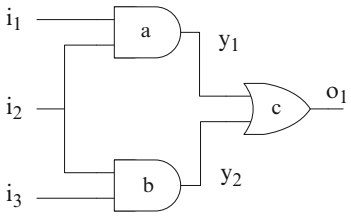


Fig. 2.1 An example circuit

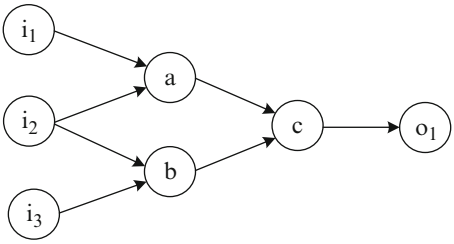


Fig. 2.2 Circuit graph

while the POs are often denoted by $o_1, o_2, \ldots, o_m$. The wires in the circuit are also denoted by lower case latin letters. Additional indices can be used for ordering wires and gates.

An example combinational circuit and its corresponding circuit graph are shown in Figs. 2.1 and 2.2, respectively. The circuit has three primary inputs which are shown by letters i_1, i_2 and i_3. Also there is one primary output o_1. In the circuit graph, each intermediate node represents a gate in the circuit. The source nodes, as the nodes which have only outgoing edges in the circuit graph, represent primary inputs. The primary outputs are represented by sink nodes, as the nodes which have only incoming edges in the circuit graph.

For symmetric gates, an input to a node is said to have a *controlling value (cv)* if it determines the value of the node output regardless of the values on the other inputs to the node. If the value on some input is the complement of the cv, the input is said to have a *non-controlling value (ncv)*. An input with value X does neither have a cv nor a ncv. Figure 2.4 shows the truth table of AND and OR gates. For an AND gate the value 0 is cv and the value 1 is ncv. For an OR gate, the value 1 is cv and the value 0 is ncv. For gates XOR and XNOR, no cv and ncv are defined because the output of gates XOR and XNOR always depends on the values of both inputs of the gate (in the case of two inputs). Therefore, if one of the inputs of gates XOR and XNOR has the value X (denoting an unknown value in three-valued logic), the output value will be X.

A path P from node g_1 to node g_r is a sequence of nodes $(g_1, g_2, \ldots, g_r)$. An edge as a wire in the circuit can have value 0 or 1 in the binary encoding. An edge whose value changes under the presence of some fault(s) is called a *sensitized edge*, and a path of sensitized edges is called a *sensitized path* [VH99]. In the ternary encoding, an edge can have value 0, 1 or X.

In contrast to a combinational circuit, a sequential circuit includes flipflops or state elements (Fig. 2.3). A flipflop is used to store state information in the circuit.

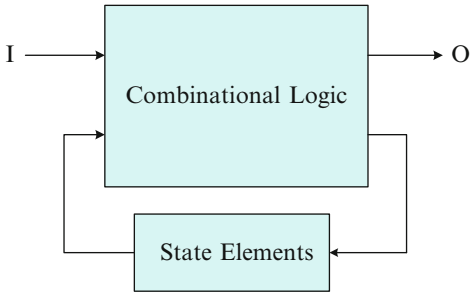


Fig. 2.3 Sequential circuit

In a sequential circuit, output values of the circuit depend on the input values and the current state of the circuit (values of all flipflops). In general, the next state of the circuit is determined by the input values and the current state.

For sequential circuits, we denote a flipflop as $b = FF(a, clk, init)$, where a is the data input of the flipflop, *clk* indicates the clock frequency of the corresponding flipflop, and *init* is the initial state of the flipflop. Wires a and *clk* and initial state $init$ are represented by Boolean variables with the same name in our models. The data output of the flipflop is denoted by b. A flipflop can change its state on either rising clock edge or falling clock edge. When a circuit has a single clock and the initial states of flipflops are 0, a flipflop is denoted as $b = FF(a)$.

2.2 Conjunctive Normal Form

Boolean Satisfiability (SAT) has emerged as a powerful reasoning engine for verification [Gup07] and also related areas, e.g., debugging [FSBD08] as well as diagnosis [SVAV05]. A SAT solver is applied to the logic formula of a circuit which has been converted into *Conjunctive Normal Form* (CNF). A CNF Φ on n Boolean variables $x_1, x_2, \ldots, x_n$ is a conjunction (AND) of m clauses $\omega_1, \omega_2, \ldots, \omega_m$. Each clause is the disjunction (OR) of one or more literals. A literal is a Boolean variable (x) or its complement ($\overline{x}$). The CNF as a Boolean formula is written as follows: $\Phi = f(x_1, x_2, \ldots, x_n)$.

The task of a SAT solver is to find an assignment $a_1, a_2, \ldots, a_n$ for $x_1, x_2, \ldots, x_n$ such that $f(a_1, a_2, \ldots, a_n) = 1$ or to prove that no such assignment exists. The Boolean formula f is called *satisfiable* (SAT) if such an assignment exists. Otherwise, f is called *unsatisfiable* (UNSAT).

For a circuit $C(V, E)$, where circuit nodes and edges are gates and their connections, respectively, CNF Φ_C has to be derived. To derive CNF Φ_C, a Boolean variable x_e is assigned to each connection $e \in E$. Then, for each gate $g \in V$, CNF Φ_g is derived from the characteristic function [Tse68, Lar92]. The characteristic function of a gate denotes whether a respective assignment to the gate inputs

Table 2.1 CNF formulas for simple gates [MS95]

Gate type g	Gate function	Φ_g
AND	$y = AND(w_1, \ldots, w_j)$	$\prod_{i=1}^{j} (w_i + \bar{y}) \cdot \sum_{i=1}^{j} (\bar{w}_i + y)$
NAND	$y = NAND(w_1, \ldots, w_j)$	$\prod_{i=1}^{j} (w_i + y) \cdot \sum_{i=1}^{j} (\bar{w}_i + \bar{y})$
OR	$y = OR(w_1, \ldots, w_j)$	$\prod_{i=1}^{j} (\bar{w}_i + y) \cdot \sum_{i=1}^{j} (w_i + \bar{y})$
NOR	$y = NOR(w_1, \ldots, w_j)$	$\prod_{i=1}^{j} (\bar{w}_i + \bar{y}) \cdot \sum_{i=1}^{j} (w_i + y)$
NOT	$y = NOT(w_1)$	$(y + w_1) \cdot (\bar{y} + \bar{w}_1)$
BUFFER	$y = BUFFER(w_1)$	$(\bar{y} + w_1) \cdot (y + \bar{w}_1)$

and output is consistent or inconsistent with the gate's function. The CNF Φ_C is constructed by the conjunction of the CNF of each gate:

$$\Phi_C = \prod_{g_i \in V} \Phi_{g_i} \tag{2.1}$$

The number of clauses in Φ_C is linearly related to the number of circuit nodes if only gates up to a fixed number of inputs or no XOR/XNOR-gates are considered. Table 2.1 shows the CNF formulas for simple gates. For each simple gate with j inputs, $j+1$ clauses are created. For example, for a two-input AND gate $y = w_1 \cdot w_2$, the CNF formula is given by the following expression:

$$\Phi_{AND} = (w_1 + \overline{y}) \cdot (w_2 + \overline{y}) \cdot (\overline{w_1} + \overline{w_2} + y) \tag{2.2}$$

The CNF formula for the example circuit of Fig. 2.1 is given by the following expression in which the CNF formula is created from Table 2.1 for each gate:

$$\begin{aligned} \Phi_C = {} & (i_1 + \overline{y_1}) \cdot (i_2 + \overline{y_1}) \cdot (\overline{i_1} + \overline{i_2} + y_1) \cdot \\ & (i_2 + \overline{y_2}) \cdot (i_3 + \overline{y_2}) \cdot (\overline{i_2} + \overline{i_3} + y_2) \cdot \\ & (\overline{y_1} + o_1) \cdot (\overline{y_2} + o_1) \cdot (y_1 + y_2 + \overline{o_1}) \end{aligned} \tag{2.3}$$

In *Bounded Model Checking* (BMC) the behavior of a sequential circuit is modeled in CNF. In BMC, a sequential circuit is unrolled for some time steps (clock cycles). In this case, the flipflops are removed and the input wire of a flipflop from clock cycle i is connected to the appropriate gates in clock cycle $i + 1$. In each unrolled copy of the circuit, only the combinational logic remains. Then the whole CNF is created by the conjunction of the CNFs of all unrolled copies.

2.3 Three-Valued Logic in Boolean Satisfiability

In three-valued logic, each variable can have the value 0, 1, or don't care X. This logic has been used in the field of formal hardware verification for creating strong counterexamples [RS04, GK05] and faster verification engines [Vel05, SVD08]. Also, this logic has been used for generating high quality counterexamples [SFB+09].

An encoding in *Conjunctive Normal Form* (CNF) is needed for the three-valued logic defined over {0, 1, X} to apply a standard SAT solver. Accordingly, the modeling of gates and components in the CNF formula has to be adjusted. Here, three-valued logic is encoded by using two variables for each signal similar to [Vel05]. The three-valued constants 0, 1, and X are defined by pairs $(0, 0)$, $(0, 1)$ and $(1, -)$, respectively. The first variable of a pair determines whether the signal has a binary value or value X. If the first variable is 0, the signal has binary value. In this case, the second variable shows a binary value as 0 or 1. If the first variable is 1, the signal has value X. In this case, the second variable is not used. Figure 2.4 shows the truth table of AND and OR gates regarding three-valued logic. When one input has a ncv, value X can be propagated to the gate output.

2.4 Model-Based Diagnosis

MBD is an approach to diagnose errors from first principles [Rei87, FSW99, MS07]. In MBD, a system model is provided in terms of components and their interconnections [Rei87, dKK03]. The component models describe how each component behaves. The behavior of the system is modeled as a logic formula. Then a domain-independent reasoning engine calculates the diagnosis from the model and system observations. Independently, SAT-based reasoning engines have been shown as a robust and efficient approach to diagnose faults and localize the bugs, called SAT-based debugging [SVAV05]. The underlying principle of MBD and SAT-based debugging is very similar. In the following section, SAT-based debugging is explained.

AND	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

OR	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

Fig. 2.4 Truth table of AND and OR gates with three-valued logic

2.4.1 SAT-Based Debugging

Debugging is a procedure in a design process that is started when the implementation of the design has failed verification. The output of the verification engine is typically returned as a set of counterexamples which proves the existence of a bug in the implementation. The set of counterexamples is denoted by CEs. Each counterexample is denoted by $CE_i \in CEs$. A counterexample $CE_i \in CEs$ includes input stimuli causing erroneous behavior, and the expected correct output response, i.e., $CE_i = (I_i, O_i)$. An input stimulus (input test vector) is a single assignment of the PIs $I = (i_1, i_2, \ldots, i_n)$ and is denoted by $(v_1, v_2, \ldots, v_n)$ where $v_i \in \{0, 1, X\}$. When sequential behavior is considered a superscript or subscript may be used to denote time points.

A circuit is generally divided into components. The components specify the granularity of the debugging result. Typical components are gates or modules, but also the hierarchical and structural information can be taken into account [ASV+05,FD05,FSBD08,SVAV05]. We consider gates as components in examples of this book for the sake of simplicity.

An approach for SAT-based debugging was presented in [SVAV05] that searches for all possible fault candidates in the implementation. Each fault candidate is a set of components of the implementation which may be modified to fix the erroneous behavior of the counterexamples. Firstly, the circuit is enhanced with correction blocks by adding a multiplexer at the output of each component. The original output function f_c of component c is replaced by f'_c as shown in Fig. 2.5a. The select line sel_c of the added multiplexer controls f'_c such that if sel_c is activated $f'_c = r_c$ where r_c is an unconstrained variable and a value for correcting the erroneous behavior may be injected, otherwise $f'_c = f_c$. The select line is also called *abnormal predicate*.

Given an implementation of a circuit and a set of counterexamples, one copy of the circuit is created for each counterexample. Then, the inputs and outputs are constrained to the input stimuli and to the correct output response of the corresponding counterexample as shown in Fig. 2.5b. When abnormal predicates are inactive, the created problem instance yields a contradiction as the circuit produces erroneous output. In Fig. 2.5b,c for the sake of simplicity, only the select lines of the correction block are shown. In the figures, the block "+" is a binary adder with m one-bit inputs [SVAV05].

When debugging is started, the SAT solver searches for satisfying assignments by activating some of the abnormal predicates. The number k of active abnormal predicates is controlled by a fault cardinality constraint as shown in Fig. 2.5b. The debugging procedure increases the number k from 1 until a satisfiable instance is found. Here, the number k shows the minimal number of changes leading to a satisfiable instance.

Debugging for sequential circuits is done by unrolling the circuit for some time steps equal to the length of the counterexample [FAVS+04] as shown in Fig. 2.5c. The length of the counterexample is the number of clock cycles in which the input

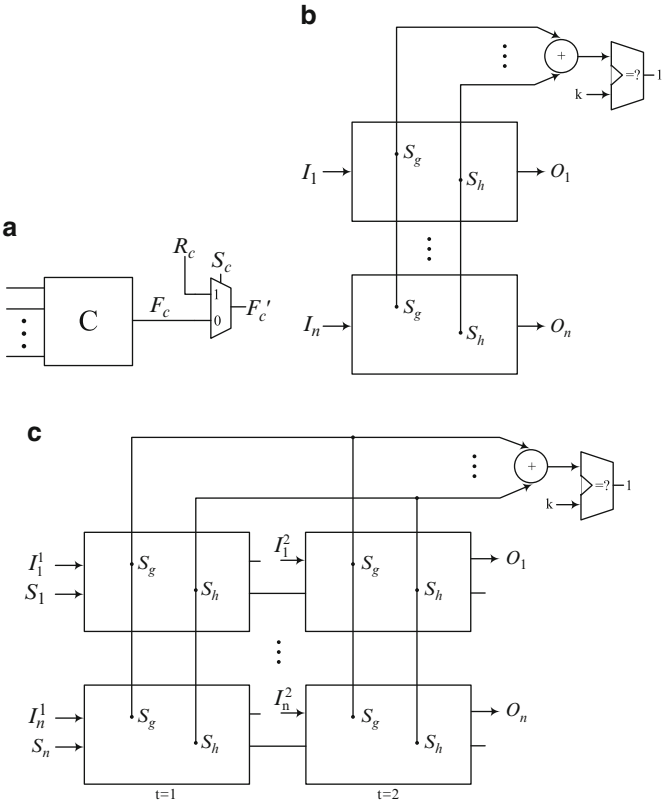


Fig. 2.5 SAT-based debugging. (**a**) Correction block. (**b**) Combinational debugging. (**c**) Sequential debugging

stimuli create the erroneous output value. In the example of Fig. 2.5c, the length of counterexamples is two time steps. The correction block is added as in the combinational case and usually the same abnormal predicate is used for the same gate in all time steps and for all counterexamples. The set of the states of the circuit (flipflops) is denoted by S. An index is used to show the time step of the states.

The case in which one component in the circuit is faulty is called *single fault*. In the case of *multiple faults*, more components than one are faulty in the circuit. Conceptually, for each counterexample $CE_i, i = 1, 2, \ldots, n$, where n is the number of counterexamples, there is a set of fault candidates $\mathscr{F}_{CE_i}$. In the case of single faults, these sets are intersected to return the final set of fault candidates $\mathscr{F}$ regarding all counterexamples:

$$\mathscr{F} = \mathscr{F}_{CE_1} \cap \mathscr{F}_{CE_2} \cap \ldots \cap \mathscr{F}_{CE_n} = \bigcap_{i=1}^{n} \mathscr{F}_{CE_i} \tag{2.4}$$

The set $\mathscr{F}$ contains a set of components of the circuit. In the examples of this book, the set $\mathscr{F}$ is a set of gates. Each fault candidate is denoted by $FC_i \in \mathscr{F}$. Parameter n has a direct impact on the memory and the run time of debugging engines. A debug algorithm which can minimize the number of fault candidates with a minimum number of counterexamples in a reasonable time improves the performance of the debugging engines.

Diagnosis accuracy is a function of the fault candidates $\mathscr{F}$. In general, the quantity and quality of fault candidates determine the diagnosis accuracy. For design bugs, we use the number of fault candidates as an estimate for diagnosis accuracy. In this case, a smaller number of fault candidates indicates a higher diagnosis accuracy. For delay faults, not only the number of fault candidates, but also the number of paths constituted by fault candidates and also the length of the shortest path from a real fault location to the fault candidates are used to indicate the diagnosis accuracy. A smaller number of paths with a smaller number of gates on them with a shorter distance from a real fault location indicates a higher diagnosis accuracy.

For multiple faults, each fault candidate $FC_i \in \mathscr{F}$, $i = 1, 2, \ldots, |\mathscr{F}|$, is a tuple of k components, $FC_i = \{FC_{i,1}, FC_{i,2}, \ldots, FC_{i,k}\}$. $FC_{i,j}$ is the jth component of the fault candidate FC_i. The set $\mathscr{F}$ is constituted in a way that each component of a fault candidate FC_i is part of at least one sensitized path of one counterexample. Each counterexample implies one or more sensitized paths. The gates along the sensitized paths are the potential places on which the fault may be located. When there are multiple counterexamples and in each counterexample all faults are sensitized, the sensitized paths of counterexamples have common gates which define their intersection. Because the sensitized paths are derived from the faulty components. In this case, the gates on the intersected sensitized paths are potential components of fault candidates which propagate the error through different paths. All components of a fault candidate can propagate the error through all sensitized paths determined by the counterexample. If one fault is sensitized and another fault is not sensitized in a counterexample, the sensitized paths of counterexamples for multiple faults may not have an intersection.

Fault model-free SAT-based debugging does not make any assumption on the fault types present in the circuit for the both cases of single fault and multiple faults. The advantage is that faults with a nondeterministic (unmodeled) behavior can also be captured [Ait97, LV05]. As shown in Fig. 2.5a, there is no restriction on the correction variable r_c. When select line sel_c is activated, the SAT solver can assign any value to the correction variable r_c to justify the erroneous behavior of the outputs corresponding to the counterexample. Since there is no restriction on the correction variables, the fault model-free debugging is performed [Ait97]. The correction blocks used for model-free debugging can change the function of each component in the circuit. In this case, local correction regarding one component is performed and there is no assumption about the function of a top module which contains the component having an activated correction block.

In the case of multiple faults, fault masking may occur in which one fault (dominant fault) masks the behavior of another fault (masked fault). In this case, the behavior of the masked fault is not propagated to an output and only dominant fault

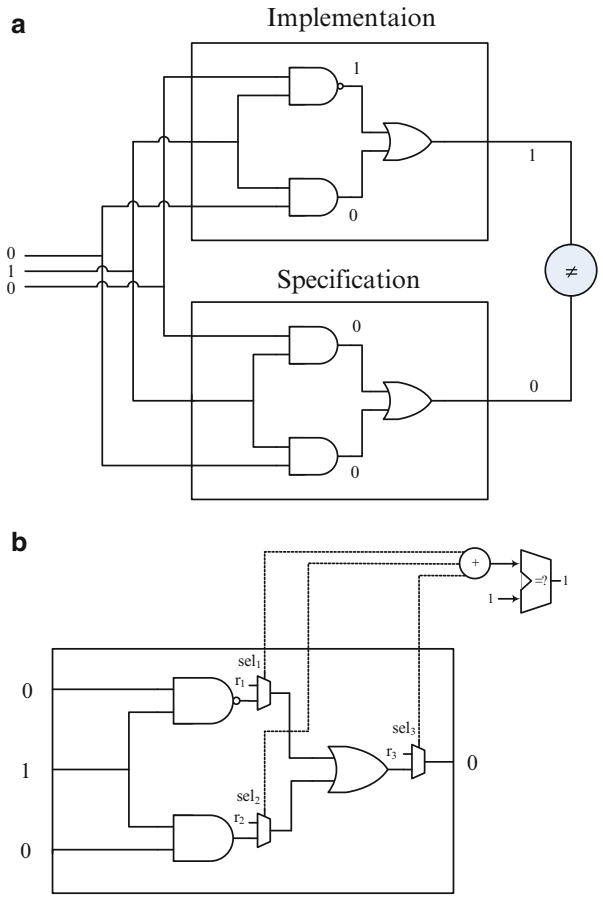


Fig. 2.6 Verification and debugging example. (**a**) Verification. (**b**) Debugging instance

is found as fault candidate. The masked fault is missed. This is a known problem but not addressed in this book. The quality of counterexamples may influence the fault masking. If a counterexample can activate different sensitized paths for different faults such that the behavior of each fault can be propagated to outputs through at least one sensitized path, the fault masking does not occur.

Example 2.1. Figure 2.6 shows the verification and the debugging processes for an implemented circuit. In the verification process, the implemented circuit (implementation) is compared to the specification. Here, the specification is given as a formal specification. If there is any inconsistency between the specification and the implementation, this inconsistency is considered as a counterexample. For instance in Fig. 2.6a, when the input vector is $(i_1, i_2, i_3) = (0, 1, 0)$, the outputs of the implementation and the specification are different. In this case, the input test vector and the correct output value obtained from the specification constitute a counterexample, i.e., $CE = (I, O) = (0, 1, 0, 0)$.

The counterexample *CE* is given for debugging. Debugging finds the fault candidates according to the given counterexample. To debug the implementation, first the implementation is enhanced by correction blocks to create the debugging instance (Fig. 2.6b). One correction block is inserted at the output of each gate. The select lines of the correction blocks are controlled by the fault cardinality constraint.

The debug instance is translated into CNF. A SAT solver is utilized to find and to enumerate all possible solutions (fault candidates). The SAT solver finds the following solutions: $(sel_1 = 1, r_1 = 0)$, $(sel_3 = 1, r_3 = 0)$. In this case, the NAND and OR gates are considered as fault candidates, as their corresponding multiplexers are the solutions found by the SAT solver. The solution $(sel_1 = 1, r_1 = 0)$ means if the select line sel_1 is activated and the correction value $r_1 = 0$ is inserted in the circuit, the correct output value is created. Accordingly, the solution $(sel_3 = 1, r_3 = 0)$ means that activating the select line sel_3 and injecting the correction value $r_3 = 0$ through the input of the multiplexer leads to the correct output value.

Figure 2.7 shows the algorithm of SAT-based debugging as pseudocode. The inputs of the algorithm are an implemented design and a set of counterexamples. In line 2, the design is copied as many times as the number of counterexamples. Also, the inputs and the output of each copy is constrained according to the inputs and the output values of one counterexample. Then the correction blocks are inserted in the model (line 3). The set of fault candidates $\mathscr{F}$ is initially empty (line 5) and variable k is initialized to 1 (line 6). The select lines of multiplexers are constrained according to k in line 9. If the SAT solver can find a solution (line 11), the solutions

```
1  algorithm SAT_Based_Debugging(In : Design, CEs, Out : F)
2  Copy_Constrain(Design, CEs)
3  SEL = Insert_Correction_Block()
4  // sel_i ∈ SEL, i = 1,2,...,m
5  F = ∅
6  k = 1
7  do
8  {
9     Add_Constraint((Σ_{i=1}^{m} sel_i) = k)
10
11    if Solve() == SAT then
12    {
13       F = Extract_All_Solutions()
14       break
15    }
16    else
17    {
18       Remove_Constraint((Σ_{i=1}^{m} sel_i) = k)
19       k = k + 1
20    }
21 } while k ≤ |SEL|
22 end function
```

Fig. 2.7 SAT-based debugging

are returned as fault candidates (line 13) and the algorithm terminates. Otherwise, the previous constraint is removed (line 18) and variable k increases (line 19). The algorithm repeats until variable k reaches the total number of select lines (line 21). In the worst case, the loop in the algorithm is executed $|SEL|$ times, where $|SEL|$ is the number of components in the circuit.

2.5 Transaction-Based Debug

In this section, the basic information about transaction-based debugging is explained. Transaction based communication-centric debugging was introduced to debug complex and large SoCs [GVVSB07]. Transaction-based debugging has more coarse-grain diagnosis accuracy in comparison to debugging at the gate level because in transaction-based debugging only transactions and events are considered. However, the approach is especially suitable for large SoCs. The output of transaction-based debugging can be utilized when debugging at RTL and gate level in order to find the root cause of the error. In an SoC, there are some IP cores which interact through a network. The transactions are observed using monitors [GVVSB07] and a debug unit controls the execution of the SoC. Also, the transactions are monitored online and are stored in on-chip memories such as a trace buffer. After running the SoC, the transaction-based assertions are used to find certain patterns in the extracted transactions [GF09]. *Transaction Debug Pattern Specification Language* (TDPSL) is used to define the transaction-based assertions. As the name suggests TDPSL is derived from *Property Specification Language* (PSL) [IEE05]. In the following, the concept of transaction and the TDPSL are explained.

2.5.1 Transaction

This section explains the transaction elements from [Ope09] and [GF09]. In TLM, abstract operations which describe the communication in a system are called transactions. Each transaction consists of a request from a master and a response from a slave. More precisely, a transaction in TLM has four basic elements:

- *Start of Request (SoRq)* corresponds to putting the request in the channel (communication medium) by the master.
- *End of Request (EoRq)* is getting the request from the channel by the slave.
- *Start of Response (SoRp)* corresponds to putting the response in the channel by the slave.
- *End of Response (EoRp)* is getting the response from the channel by the master.

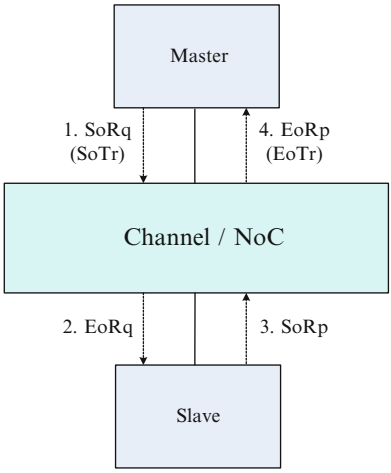


Fig. 2.8 Elements of a transaction

Figure 2.8 shows the basic elements of a transaction which are transferred via a channel or an NoC. Also there are two additional elements which are called: *Request Error* (ErrRq) and *Response Error* (ErrRp). These elements handle error conditions and correspond to any kind of error that causes a request or a response to fail.

2.5.2 Transaction Debug Pattern Specification Language

TPDSL has three layers as shown in Fig. 2.9: the Boolean layer, the temporal layer, and the verification layer [GF09]. The Boolean layer contains *trans_exp* which represents the basic elements of transactions. The *trans_exp* format is as follows:

trans_type (*master*, *slave*, *type*, *address*, *tag*)

Field *trans_type* can be any transaction element mentioned in Sect. 2.5.1 as well as the *Start of Transaction* (SoTr) or the *End of Transaction* (EoTr) which are the same as SoRq and EoRp, respectively. The elements SoTr and EoTr model transactions more coarsely than the elements SoRq, EoRq, SoRp and EoRp. Fields *master* and *slave* specify the master and slave IDs which are an abstraction of the higher address bits. Field *type* can be read Rd or write Wr. Field *address* indicates the slave address symbolically as SAME, SEQ, and OTHER. Field *tag* indicates the transaction number and is only used for buses that allow non-blocking requests and out-of-order responses [GF09]. In this book, we show a transaction without considering the field *tag*.

The motivation to use symbols for the address field is to abstract and to compress the address bits. In this case, only the compact address information is stored or sent via network for debugging. The symbols can be defined with respect to the application and the granularity of debugging. The following symbols are used: SAME specifies that in the current transaction, the slave address is the same as

```
/********** Boolean Layer **********/
trans_exp ::=
    trans_type "(" master "," slave "," type "," address "," tag ")";
trans_type ::=
    "SoRq" | "EoRq" | "SoRp" | "EoRp" | "ErrRq" | "ErrRp" |
    "SoTr" | "EoTr" | "ErrTr" ;
master ::= var | id | "-" ;
slave ::= var | id | "-" ;
tag ::= var | id | "-" ;
type ::= "Rd" | "Wr" | "-" ;
address ::= "SAME" | "SEQ" | "OTHER" | "-" ;
var ::= IDENTIFIER ;
id ::= NUMBER ;

/********** Temporal Layer **********/
fl_exp ::= ("always" | "never" | "eventually") temporal_exp ;
temporal_exp ::= seq_exp | temporal_exp t_op temporal_exp ;
t_op ::= "->" | "&&" | "||" ;
seq_exp ::= sere_exp | "{" sere_exp "}" | seq_exp repeat_exp;
repeat_exp ::= repeat_op [ count_exp ] ;
repeat_op ::= "*" | "+" | "=" ;
count_exp ::= NUMBER [ ":" (NUMBER | "inf") ] ;
sere_exp::= trans_exp | sere_exp sere_op sere_exp ;
sere_op ::=";" | ":" | "|" | "&" ;

/********** Verification Layer **********/
property_exp ::= "assert" fl_exp [ filter_exp ] ;
filter_exp ::= "filter" "(" masters "," slaves "," types ")" ;
masters ::= "*" | "-" | id_list ;
slaves ::= "*" | "-" | id_list ;
types ::= "*" | type ;
id_list ::= id | id_list "&" id ;
```

Fig. 2.9 The syntax of TDPSL [GF09]

the previous transaction address for this slave. SEQ specifies that in the current transaction, the slave address has one word difference with the previous transaction address for this slave. OTHER specifies that in the current transaction, the slave address is neither SAME nor SEQ.

Example 2.2. Transaction $EoTr$ ($m1$, $s2$, Rd, $-$) represents the end of a read transaction from master $m1$ to slave $s2$ with any address. The symbol "–" indicates that we leave the address field as don't care. As explained before, the tag field is not shown.

The properties in terms of transaction sequences are defined at the temporal layer. Different operators are available at this layer such as *concatenation operator* (;), *fusion operator* (:), *or operator* (|), *and operator* (&) and *repetition operators*

(*, +, =) [GF09]. The *always* operator means that the temporal expression has to hold at any time. The *never* operator means that the temporal expression has to never hold. The *eventually* operator means that the temporal expression has to hold at the current time or some future time [GF09].

Last layer in Fig. 2.9 is verification layer. In the verification layer, the *assert* statement is defined. Also, a *filter* can be defined which specifies a filter over the execution path for the evaluation of the assertion statement. The filter expression *filter_exp* in the verification layer of Fig. 2.9 can be used to filter some of the unwanted transactions. Filters can be defined over masters, slaves and transaction types. Filters can be defined by explicitly naming the masters, slaves or transaction types. The symbol "−" means no filter and "∗" means that only consider related masters, slaves or transaction types [GF09]. In the following example, the related masters are $m2$ and $m1$. The related slave is $s1$.

Example 2.3. A simple assertion in TDPSL is as follows:

$assert\ never\{$
$\quad EoTr\,(m2,\ s1,\ Wr,\ -);\ SoTr\,(m1,\ s1,\ Rd,\ -)$
$\}filter(*, *, *)$

This assertion specifies that start of a read transaction from master $m1$ to slave $s1$ must never be directly after the end of a write transaction from master $m2$ to slave $s1$. Sign $*$ in the filter means only the related transaction types, masters, and slaves have to be considered.

For SoCs, debugging can be performed at the granularity of transaction-level and message-level [GVVSB07]. For transaction-level debugging, we need to observe each transaction including SoTr and EoTr. As Fig. 2.8 shows, SoTr and EoTr can be observed at master interconnect. Monitors on master interconnects can be utilized to observe transactions (Fig. 2.10b). Message-level debugging has a finer granularity. A message is a request or a response. Typical requests are read and write commands. Read data and write acknowledgment are typical responses [GVVSB07]. To perform message-level debugging, we need to observe each message including

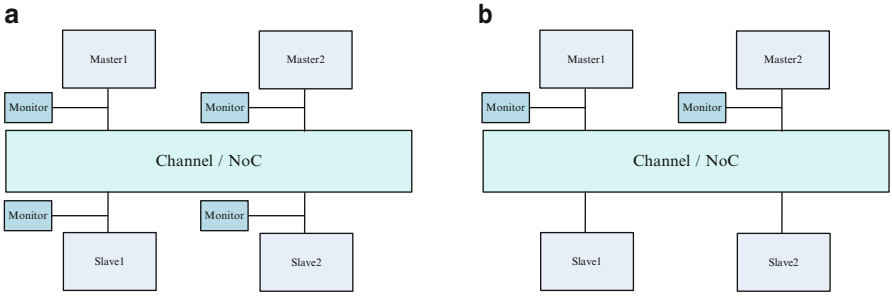


Fig. 2.10 Monitoring. (**a**) Message-level monitoring. (**b**) Transaction-level monitoring

SoRq and EoRq as well as SoRp and EoRp. As Fig. 2.8 shows, both master and slave interconnects have to be observed for message-level debugging. Monitors on both master and slave interconnects can be utilized to observe the messages (Fig. 2.10a).

2.6 Overview of Benchmarks

In this book, we propose automated debugging approaches for a hardware system at different abstraction levels, i.e., transaction-level, RTL and gate-level. To show the efficiency of our approaches, we use suitable benchmarks at each abstraction level. Figure 2.11 shows the structure of our benchmarks for debugging at different levels. As Fig. 2.11 shows, at transaction-level we use an NoC-based SoC. In this case, we utilize the Nirgam environment which is a cycle accurate NoC simulator implemented in SystemC language [Jai07]. In Fig. 2.11, the system consists of *Routers* (R) and *Intellectual Property* cores (IP). Each IP has two main elements: *Processing Element* (PE) and *Communication Element* (CE).

The elements of an IP are considered for debugging at RTL. In this case, we consider the OpenRISC processor [LB14], ITC-99 [Dav99] and LGSynth-93 [McE93] benchmark suites. Using ITC-99 and LGSynth-93 benchmarks also help us to compare our approaches with previous debugging approaches which will be mentioned in later chapters.

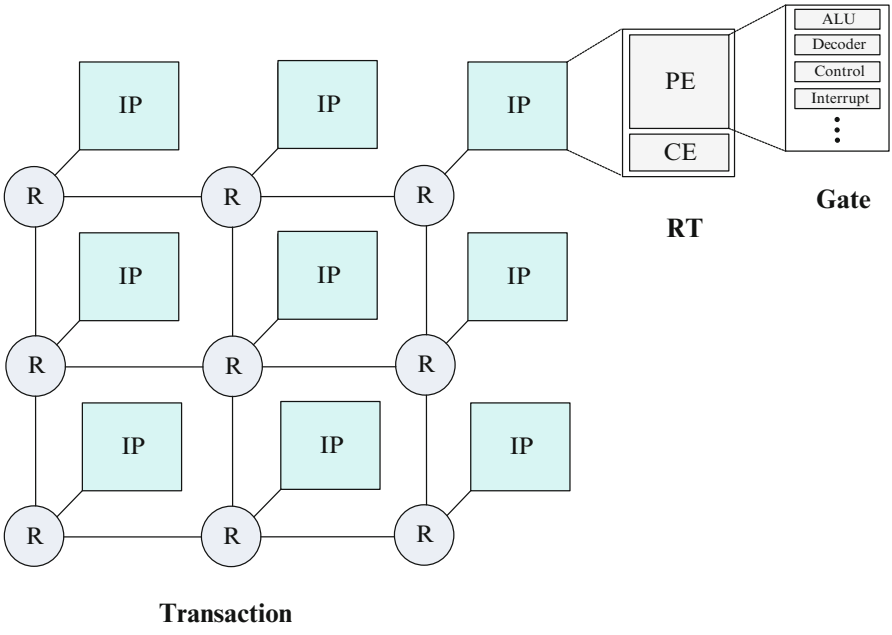


Fig. 2.11 Structure of debugging benchmarks at different levels: Transaction, RT, Gate

Table 2.2 Benchmarks

Debug of transactions	Debug of design bugs	Debug of electrical faults
NoC-Based SoC (Nirgam)	OpenRISC ITC-99 LGSynth-93	ISCAS-85 ISCAS-89

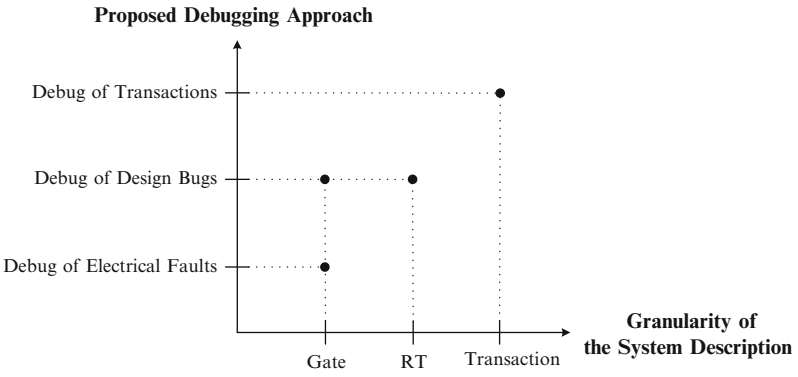


Fig. 2.12 The proposed debugging approaches regarding the granularity of the system description

A PE can have different modules such as ALU, decoder, control unit, interrupt controller, etc., which are represented by the ISCAS benchmarks [BPH85, Brg85, BBK89]. The ISCAS benchmarks are known benchmarks in the research area for test of delay faults. Also, we use the ISCAS benchmarks to evaluate our approaches to debug delay faults.

Table 2.2 summarizes the benchmarks used for our debugging approaches. The first row indicates the proposed debugging approach, i.e., debug of transactions, debug of design bugs and debug of electrical faults (delay faults). The second row indicates the used benchmark suite.

Figure 2.12 shows the proposed debugging approaches regarding the granularity of the system descriptions used in this book. The system description at gate-level is the most fine-grain granularity considered in this book. Many details of a hardware system are available at this level. We even include some technological parameters like timing behavior of gates in respective models. The most coarse-grain granularity of the system description in this book is at transaction-level.

For debug of electrical faults, we use the gate-level description of a system as certain information such as delay and frequency of the circuit is only available at gate-level. The gate-level benchmarks are obtained by synthesizing the circuits using Synopsys Design Compiler with Nangate 45nm Open Cell Library [Nan11]. For debug of design bugs, we use both RTL and gate-level descriptions of a system. Because a design bug may escape the verification at RTL and may slip into gate-level. The transaction-level description of a system is used for validating the methodology for transaction-based debugging.

Part I
Debug of Design Bugs



http://www.springer.com/978-3-319-09308-6

Debug Automation from Pre-Silicon to Post-Silicon

Dehbashi, M.; Fey, G.

2015, XIV, 171 p. 93 illus., 55 illus. in color., Hardcover
ISBN: 978-3-319-09308-6