基於第一原理所發展之數位電路診斷系統
DSDC: A Diagnosis System for Digital Circuits Based on First Principles

韓定中 **Benjamin Han**　　楊欣泰 **Hsin-Tai Yang**　　李錫智 **Shie-Jue Lee**

國立中山大學電機系
Department of Electrical Engineering
University of Sun Yat-Sen, Kaohsiung, Taiwan, R.O.C.
{ben|styang|leesj}@water.ee.nsysu.edu.tw

摘要

在本論文中，我們提出一個完整的數位電路診斷系統。我們的系統主要是根據 Reiter 所提出的第一原理診斷理論[6]，再輔以 Hou 的測量理論[4]。此外，為了要決定量測的最佳順序，我們提出一套利用基因演算法來決定最佳量測的方法。

關鍵字： 電路診斷、第一原理、量測、基因演算法

Abstract

In this paper we propose a complete diagnostic system for digital circuits (DSDC). Our work is based on Reiter's theory of diagnosis from first principles [6], incorporated with Hou's theory of measurements [4] to discriminate among all possible diagnoses in a fault diagnosis task. Meanwhile, in order to determine the best order in which measurements are to be taken, a measurement selection strategy using the genetic algorithm (MSSGA) is proposed.

Keywords: circuit diagnosis, first principles, measurement, genetic algorithm

1 Introduction

Although almost all theories of diagnosis intend to be domain-independent, digital circuits are the most frequent and suitable test-bed for these theories. One of the reasons is that the model of digital circuits can be easily built in a compact form, and their behaviors have been well studied and formalized. The other reason is that due to the advances of VLSI technology in recent years, the dramatically increasing complexity of digital circuits together with their short life cycle makes any diagnostic approaches based only on intuitions impractical. It is then necessary and important to build automatic diagnostic systems for such diagnostic tasks.

In this paper we propose a complete *d*iagnostic *s*ystem for *d*igital *c*ircuits (DSDC) based on the work of Reiter [6] and Hou [4]. DSDC has the following characteristics:

1. DSDC is complete in the sense that all possible diagnoses under the observations of a malfunctioning circuit obtained so far can be found.
2. DSDC uses a simple language called DSDC Language (DSDCL) for describing digital circuits. Hierarchical diagnosis is possible by using circuit specifications at different granularity.
3. DSDC derives minimal conflict sets (MCSs) by the derivation method using a CS-tree with mark set, which is an improvement to Hou's method.
4. DSDC uses an incremental propositional calculus prover (PC prover) for logic inferences. To be more specific, a PC prover implemented by trie data structure (PCPT) with chronological-backtracking capability [7] is incorporated in DSDC as the underlying inference engine.
5. DSDC selects the best next measurement via the genetic algorithm [3].

2 Overview of DSDC

DSDC receives a circuit specification in DSDC Language (DSDCL), evaluates different possible measurement orderings via the genetic algorithm and prompts for measurement results from user input. The interactive diagnostic process is going on until only one possible diagnosis is left. Upon this point the user may replace the faulty components found within the system, and use a fault-detection method to verify the soundness of the system [5]. Fig 2.1 shows the block diagram of

DSDC.

DSDC Preprocessor translates the circuit specification in DSDCL into a set of clauses in conjunctive normal form (CNF) and other parameters for the use of Inference Engine and Measurement Request (MR) Generator. A circuit specification contains component definitions, component declarations, component connections, circuit observation and other related parameters for DSDC.

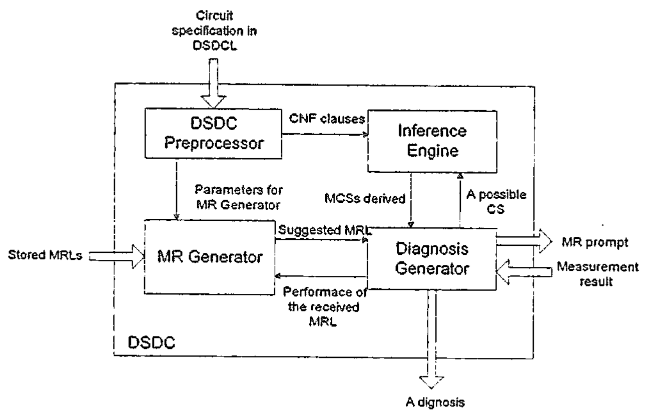

Fig. 2.1. The block diagram of DSDC

Inference Engine receives the CNF clauses from DSDC Preprocessor and responds the minimal conflict set (MCS) derivation requests from Diagnosis Generator with the derived MCSs. It uses an efficient incremental Propositional Calculus (PC) prover for deriving MCSs from a possible conflict set (CS) using CS-tree with mark set. To improve the efficiency of the PC prover, we choose trie data structure for representing CNF clauses internally, and various reasoning rules are performed based on the work of Zhang and Stickel [8].

Diagnosis Generator is responsible for generating all possible diagnoses, viz., all Minimal Hitting Sets (MHSs) from the collection of MCSs received from Inference Engine. To discriminate among competing diagnoses, Diagnosis Generator issues a request for a Measurement Request List (MRL) from MR Generator and prompts for the measurement result from user input in the order specified in the MRL received. Such interactive diagnostic process is going on until only one possible diagnosis is left.

MR Generator generates a promising MRL according to past experiences of diagnosing the circuit. A promising MRL should minimize the effort for deriving the unique cause of the disorders of the circuit. Specifically speaking, the MR Generator of DSDC exploits Measurement Selection Strategy using the Genetic Algorithm (MSSGA) for evolving and determining promising MRLs. The population of MRLs required by MR Generator is generated randomly or obtained from the previous runs on the same circuit. Various parameters for GA are given in the circuit specification in DSDCL.

3 DSDC Preprocessor and DSDC Language (DSDCL)

In DSDC, the model of a digital circuit is built by using DSDCL. DSDC Preprocessor translates the circuit specification in DSDCL into a set of CNF clauses and other related parameters for other modules in DSDC. This section presents the details of DSDCL, and a complete language formulation in Backus-Naur form is given.

The terminals of DSDCL are listed in Table 3.1. The meaning and the precedence of each operator used in DSDCL is listed in Table 3.2.

keyword:	one of `define` `declare` `connect` `observe` `assign` `in/inx` `out/outx`
token:	Any string other than any one of the keywords
real:	A real number
integer:	An integer
output-predicate:	`out`[*token*] `outx`[*token*]
input-predicate:	`in`[*token*] `inx`[*token*]
truth-value:	either `0` or `1`
unary-operator:	~
binary-operator:	one of () ~ * + @ = >

Table 3.1. All terminals of DSDCL

DSDC Preprocessor translates an expression using *Huntington's postulates* [2] into its equivalent CNF clauses. In particular, the equality "$a = b$" is translated into $(\neg a \vee b) \wedge (\neg b \vee a)$. Table 3.3 shows the basic structure of a circuit description.

Operator	Meaning	Precedence
()	Parentheses	0
~	Not	1
*	And	2
+	Or	3
@	Xor	3
=	Equality	4
>	Implication	5

Table 3.2. The meaning and the precedence of each operator used in DSDCL

circuit-specification:	*define-block declare-block connect-block observe-block assign-block*

Table 3.3. The basic structure of a circuit specification in DSDCL

A *define-block* is defined as shown in Table 3.4.

define-block:	`define` { *define-sub-blocks* }
define-sub-blocks:	*define-sub-block*
	define-sub-block define-sub-blocks
define-sub-block:	*token* { *define-expressions* }
define-expressions:	*define-expression*
	define-expression define-expressions
define-expression:	*output-predicate* = *define-primary-expression*;
define-primary-expression:	*input-predicate*
	unary-operator define-primary-expression
	define-primary-expression binary-operator define-primary-expression
	(*define-primary-expression*)

Table 3.4. The formulation of *define-block* in DSDCL

The formulation of *declare-block* is shown in Table 3.5.

declare-block:	`declare` { *declare-sub-blocks* }
declare-sub-blocks:	*declare-sub-block*
	declare-sub-block declare-sub-blocks
declare-sub-block:	*token* { *tokens* }
tokens:	*token*;
	token; *tokens*

Table 3.5. The formulation of *declare-block* in DSDCL

Table3.6 shows the formulations of the other blocks in DSDCL.

4 Inference Engine

In DSDC, Inference Engine is responsible for deriving all MCSs from a possible CS via a sound and complete PC prover. We enhance the efficiency of Inference Engine in DSDC by using an incremental Davis-Putnam PC prover [1] implemented by trie data structure [8].

connect-block:	`connect` { *connect-expressions* }
observe-block:	`observe` { *observe-expressions* }
assign-block:	`assign` { *assign-sub-blocks* }
connect-expressions:	*connect-expression*
	connect-expression connect-expressions
observe-expressions:	*observe-expression*
	observe-expression observe-expressions
assign-sub-blocks:	*assign-expression*
	assign-sub-block
	assign-expression assign-sub-blocks
	assign-sub-block assign-sub-blocks
connect-expression:	*output-predicate* = *input-predicate*;
	input-predicate = *output-predicate*;
observe-expression:	*output-predicate* = *truth-value*;
	input-predicate = *truth-value*;
assign-sub-block:	*token* { *assign-sub-blocks* }
assign-expression:	*token* = *real*;
	token = *integer*;

Table 3.6. The formulations of *connect-block*, *observe-block* and *assign-block* in DSDCL

4.1 A PC Prover Implemented by Trie Data Structure (PCPT)

PCPT represents ground CNF clauses in trie data structure, and efficiently performs unit propagation on the representation. The nondestructive approach used in PCPT makes it very suitable for repeated satisfiability computation.

A trie structure of a set of ground CNF clauses is then defined as follows.

Definition 4.1 A trie structure of a set of ground CNF clauses is a 3-ary *tree* defined below.

(1) Each node of the structure is either empty (●), or a clause end-mark ( ), or labeled by a 4-tuple <*var*, *pos*, *neg*, *rest*>, where *var* is a variable index, *pos* is its positive child node, *neg* is its negative child node and *rest* is its brother node.
(2) An empty node or a node labeled by a clause end-mark has no child and brother nodes.
(3) The left edge of a node labeled by <*var*, *pos*, *neg*, *rest*> is interpreted as the positive literal of the variable with index *var*. The right edge of the node is interpreted as the negative literal of the variable with index *var*. Brother edges, viz. the edges between brother nodes, represent nothing.

(4) A clause is obtained from the collection of left/right/brother edges of the path starting from the root node to some clause end-mark node.

A trie structure of a set of ground CNF clauses is said to be *ordered* if for each node labeled by *<var, pos, neg, rest>* in the tree, *var* is smaller than any variable index appearing in *pos*, *neg* and *rest*. Fig. 4.1 shows the ordered trie structure of the clause $(a \vee b) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee c)$.

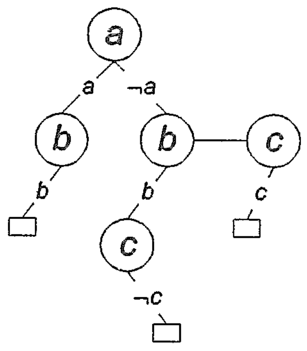

Fig. 4.1. The ordered trie structure of the clause $(a \vee b) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee c)$

In PCPT, the *unit resolution* technique of Davis-Putnam procedure is implemented on an ordered trie structure by first building a *variable table.*

Definition 4.2 A *variable table* is an array of variable records. Each variable has its own variable record, which is a 4-tuple *<truth, head-list, positive-tail-list, negative-tail-list>*, where *truth* denotes the truth value (TRUE, FALSE or UNASSIGNED) of the corresponding variable, *head-list* is a list containing all occurrences (viz. the corresponding nodes in the trie structure) of the corresponding variable as the first variable in the clauses, and *positive-tail-list* (resp. *negative-tail-list*) is a list consisting of all occurrences of the corresponding variable in positive (resp. negative) form as the last variable in the clauses. Note that *head-list* has at most one element.

The unit resolution can then be achieved by updating the variable table. The following procedure is used for updating the variable table in PCPT.

Procedure 4.3 Assume that we assign TRUE/FALSE to a variable v, whose variable record is *<truth, head-list, positive-tail-list, negative-tail-list>*. Update the variable table as follows.

(1) Assign TRUE/FALSE to *truth*.
(2) Update *head-list* if it is not empty:
Let the only element in *head-list* points to node n in the trie structure, and remove the element. Assume the case is *truth* = TRUE (resp. FALSE). If the right (resp. left) child of n in the trie structure is a clause end-mark, then a null clause is found. If the right (resp. left) child is empty, stop updating *head-list*. Otherwise add the right (resp. left) child of n and the brothers of the right (resp. left) child of n in the trie structure into their corresponding *head-list* if the truth values of the corresponding variables are UNASSIGNED. For each of those added nodes, assume its corresponding variable is v'. If *positive-tail-list/negative-tail-list* of v' contains the same node, then either v' or $\neg v'$ is a unit clause. For each of the right (resp. left) child of n and the brothers of the right (resp. left) child of n in the trie structure such that the truth value of the corresponding variable v'' is not UNASSIGNED, set $v = v''$ and repeat this step.
(3) Update *positive-tail-list* (resp. *negative-tail-list*) if it is not empty:
For the case *truth* = TRUE (resp. FALSE), remove all elements of *positive-tail-list* (resp. *negative-tail-list*). If no parent node exists for the nodes pointed by the elements of *negative-tail-list* (resp. *positive-tail-list*), then a null clause is found. Otherwise for each of the nodes pointed by the elements of *negative-tail-list* (resp. *positive-tail-list*), add the parent node of the node in *positive-tail-list/negative-tail-list* of the corresponding variable of the parent node if the truth value of the corresponding variable of the parent node is UNASSIGNED. For each of those added nodes, assume its corresponding variable is v'. If *head-list* of v' contains the same node, then either v' or $\neg v'$ is a unit clause. Moreover, if the tail list with the opposite polarity to the tail list which the new node is added contains the same node, then a null clause is found. For each of the nodes pointed by the elements of *negative-tail-list* (resp. *positive-tail-list*) such that the truth value of the corresponding variable v'' of its parent node is not UNASSIGNED, set $v = v''$ and repeat this step.

The unit clauses found in the unit resolution process are collected in a unit clause list in PCPT. If a null clause is found in the unit resolution process, PCPT will rewind the assignments to the most recent splitting point.

The splitting rule of Davis-Putnam method is handled in PCPT as follows. If no unit clause is available, an unassigned variable is chosen such that the total number of elements in its *head-list, positive-tail-list* and *negative-tail-list* is the largest one compared to all other unassigned variables. The intuition is that selecting such a variable might lead us to update more variable records than choosing other unassigned variables. Each assignment step is then recorded down in a stack so that future rewinding is possible.

4.2 Incremental Proving Using PCPT

In DSDC, we exploit simple idea of chronological backtracking [7] to improve the efficiency of deriving MCSs from a CS. At the root node in a CS-tree with mark set, for each unit assumption except for the last one, we insert it into PCPT's unit clause list and initiate PCPT to run the assignment process to the point where no unit clause is available and a splitting has to be made. An *s-point* mark is then placed at the top of the rewinding stack. For the last unit assumption, we do the same thing except that we run the assignment process to the end, and we do not place any s-point mark. When testing the descendant set, we simply rewind PCPT to the last s-point marked.

To make such incremental proving possible, the order of inserting unit assumptions is important. In DSDC, we insert the unit assumptions in a mark set first, then we push the other unit assumptions into the unit clause list of PCPT.

5 Diagnosis Generator

Diagnosis Generator in DSDC is responsible for generating all possible diagnoses under the observations obtained so far. In [6], Reiter has shown that an MHS for the collection of all MCSs for a system is a diagnosis for it. Moreover, he has proposed an MHS derivation method using a pruned HS-tree to computes all MHSs for a collection of MCSs. To refine old diagnoses, Hou [4] suggested a method by deriving new MCS from old diagnoses. However, his method is incomplete and would miss some MCSs. Therefore, we propose an improvement to Hou's method, which is called CS-tree with mark set.

Definition 5.1 A *CS-tree with mark set* T_M rooted in a CS C is defined as follows:

(1) Its root is labeled by $[C, \varnothing]$;
(2) Each node n of T_M is labeled by $[S_n, S_{M,n}]$, where $S_n \subseteq C$ is the *label set* of node n, and $S_{M,n} \subseteq S_n$ is the *mark set* of node n. If $S_{M,n} = S_n$ or $|S_n| = 1$, then node n has no descendants. Otherwise for each $c \in S_n - S_{M,n}$, node n_c is a descendant of node n such that node $n_{c'}$ is the *immediate* left brother of node n_c and node n_c is labeled by $[S_{M,n} \cup (S_n - S_{M,n} - \{c\}),\ S_{M,n_{c'}} \cup \{c\}]$.
(3) For the leftmost node n in every subtree of T_M, $S_{M,n} = S_{M,n_p}$, where n_p is the parent node of node n.

To derive all MCSs from C, we generate a pruned CS-tree with mark set $T_M{}'$ rooted in C by the following procedure.

Procedure 5.2 Generate a pruned CS-tree with mark set $T_M{}'$ rooted in a CS C by the following rules:

(1) Generate $T_M{}'$ depth-first, i.e., generate the descendants of a node before generating its brothers.
(2) Pruning rule:
    (a) If S_n of node n is not a CS, then we close node n.
    (b) If $S_{M,n}$ of node n is a superset of some MCS already used as a label set of some node in $T_M{}'$, then we close node n and do not generate any right brothers of node n.

Note that not every label set requires a PC prover call for testing if it is a CS because a superset of a known CS is a CS. Also note that for a node n labeled by a CS C, if n does not have any descendant or all of its descendant nodes are labeled by non-CSs, C is not necessarily minimal - a check must be made to see if it is minimal against all other CSs of this kind.

Example 5.3 Let $C = \{c_1, c_2, c_3, c_4\}$ and suppose that all the MCSs we can derive from C are $\{c_1\}$, $\{c_2, c_3\}$ and $\{c_2, c_4\}$. The following figure shows the derivation using our method:

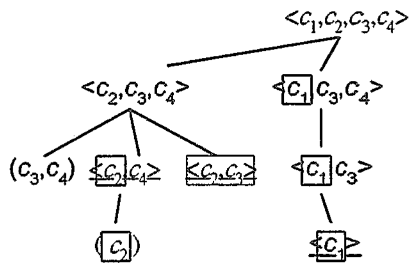

Fig. 5.1. Deriving the MCSs from C, using our method

In some case, Hou's method would lose $\{c_2, c_4\}$. Besides, compared to Hou's approach, out method needs fewer PC prover calls and nodes generated.

After deriving all possible diagnoses, Diagnosis Generator prompts for the measurement result from user input according to the MRL it receives from MR Generator. It discriminates among competing diagnoses based on the measurement result. When only one possible diagnosis is left, Diagnosis Generator reports the number of prover calls of each measurement request to MR Generator as the indication of the MRL's performance. This data forms the basis on which MR Generator will gradually find effective MRLs for future operations.

6 Measurement Request Generator

For a nontrivial circuit with the initial observation only, we always have more than one possible diagnosis. To discriminate among these competing diagnoses, new information must be observed by probing some terminals of the components within the system. The effects brought by such measurements on the possible diagnoses have been formalized in the work of Reiter [6] and Hou [4]. However, they did not address the problem as how to select the best next measurement. In DSDC, MR Generator is responsible for pointing out the best next measurement by MSSGA.

A good MRL should shorten the diagnostic process while lower down the cost for obtaining the only one possible diagnosis. In particular, two important quantities are used as indications of the quality of an MRL: the first one is the number of measurements needed for a diagnostic system to settle down to the only one possible diagnosis, and the second one is the total number of PC prover calls needed during the course of reasoning. These two numbers, n_M and n_P, respectively, are then combined into one quantity *PI* (performance index) in the following simple form:

$$PI = n_M \times n_P$$

Our goal here is to minimize the average *PI* values of the population of MRLs using MSSGA.

MSSGA consists of the following steps:

(1) Initialization: Randomly initialize a population of MRLs.

(2) Initial evaluation:
Compute the *PI* value of each MRL within the population by performing diagnosis using the MRL. Then calculate the fitness value of the MRL based on its *PI* value.

(3) Test if one of the stopping criteria (time, fitness, etc.) holds. If yes, stop the procedure. For example, one would like to stop the procedure when the average fitness value goes down under some threshold.

(4) Genetic operator selection:
Choose a genetic operator using the roulette wheel selection method. Assume that the operator selected requires *n* MRLs as operands.

(5) MRL selection:
Select *n* MRLs from the population using the roulette wheel selection method.

(6) Using the genetic operator:
Generate descendant chromosomes by using the genetic operator selected in (4) on the MRLs selected in (5).

(7) Evaluation:
Select one of the new MRLs generated in (6). Compute the *PI* value of the MRL by performing diagnosis using the MRL. Then calculate the fitness value of the MRL based on its *PI* value.

(8) Updating the population:
If the new MRL evaluated in (7) already exists in the population, replace the MRL in the population with the newly generated one. Otherwise replace the worst MRL, i.e. the MRL with largest *PI* value in the population, with the newly generated one.

(9) Updating the fitness values of the genetic operators:
Update the fitness values of all related genetic operators according to the performance of the new MRL.

(10) Repeat (7) to (9) until all of the descendant MRLs generated in (6) are evaluated and inserted into the population.

(11) Repeat (3) to (10).

7 Conclusion

In this paper, we proposed and implemented a complete *d*iagnostic *s*ystem for *d*igital *c*ircuits (DSDC). Our work is based on Reiter's theory of diagnosis from first principles [6], incorporated with Hou's theory of measurements [4] to discriminate among all possible diagnoses in a fault diagnosis task. In addition, our system is capable of improving its performance *online* by incorporating MSSGA.

References

1. M. Davis and H. Putnam, A Computing Procedure for Quantification Theory, *Journal of ACM* 7 (3) (1960), 201 - 215.
2. J. P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley, 1993.
3. J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
4. A. Hou, A Theory of Measurement in Diagnosis from First Principles, *Artificial Intelligence* 65 (1994) 281 - 328.
5. P. K. Lala, *Fault Tolerant and Fault Testable Hardware Design*, Prentice-Hall, 1985.
6. R. Reiter, A Theory of Diagnosis from First Principles, *Artificial Intelligence* 32 (1987) 57 - 95.
7. S. Russell and P. Norvig, "Logical Reasoning Systems," in *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 1995.
8. H. Zhang and M. Stickel, *Implementing the Davis-Putnam Algorithm by Tries*, typescript, University of Iowa, 1994.