UNIUNEA EUROPEANĂ

GUVERNUL ROMÂNIEI

Instrumente Structurale 2007-2013

Platformă de e-learning și curriculă e-content pentru învățământul superior tehnic

Testarea Sistemelor

11. GAT pentru blocaje simple în circuite combinaţionale

GENERAREA AUTOMATĂ A TESTELOR PENTRU DEFECTELE BLOCAJE SIMPLE DIN CIRCUITELE COMBINAŢIONALE

În cele ce urmează vor fi considerate metodele de generare a testelor pentru defecte blocaje simple. Procesul de generare al testelor depinde în primul rând de tipul de experiment de testare pentru care sunt generaţi vectorii de test.
Toate consideraţiile ce urmează sunt dedicate unei testări off-line, la nivelul pinilor de intrare-ieşire, folosind pentru testare vectori de test stocaţi. Decizia detecţiei unui defect are loc prin comparaţia completă a rezultatelor ieşirii circuitului testat în raport cu cel etalon. Se presupune că atât la pinii de intrare ai circuitului testat cât şi la bornele celui etalon se aplică aceleaşi valori, atunci când se face testarea

Elemente fundamentale

Generarea testelor este o problemă complexă cu multe aspecte de interacţiune. Cele mai importante sunt:

- costul generării testelor;
- calitatea testelor generate;
- costul aplicării testelor.

Costul generării testelor depinde de complexitatea metodei de generare. Generarea testelor aleatoare (GTA) este un proces simplu care implică doar o generare aleatoare a vectorilor de test. Totuşi, pentru atingerea unei bune calităţi a testului - calitate apreciată prin acoperirea defectelor - este nevoie de un set mai mare de vectori de test generaţi aleator.
Chiar dacă generarea testelor poate fi mai simplă simplă, determinarea calităţii testului, prin simularea defectelor, spre exemplu, poate fi un proces costisitor.
Mai mult, un test mai bogat (constituit dintr-o secvenţă mai numeroasă de vectori de test poate costa mai mult deoarece creşte timpul de testare şi volumul de memorie al dispozitivului care implementează testarea.
Generarea aleatoare a testelor, în general, nu ia în consideraţie funcţia sau structura circuitului ce urmează a fi testat. Spre deosebire de aceasta, generarea deterministică a testelor produce teste pornind de la un model al circuitului. Comparativ cu generarea aleatoare, generarea deterministică este mult mai costisitoare, dar produce teste mai scurte şi de calitate mai bună, în general. Generarea deterministică a testelor poate fi realizată manual (pentru circuite cu complexitate mică) sau automat. În continuare se vor aborda metodele de generare deterministică automata a testelor.

Generarea deterministică a testelor poate fi orientată spre defect sau poate fi independentă de defect. În primul caz testele sunt generate pentru defecte specificate din universul de defecte (universul de defecte definit printr-un model explicit al defectelor). Generarea deterministică a testelor independentă de defect nu ţine seama de specificitatea unui defect ţintă anume, individual.
Costul generării deterministice a testelor depinde de complexitatea circuitului.
Metodele de reducere a complexităţii circuitelor vizând modalităţi mai simple de testare sunt grupate sub denumirea generică de metode de proiectare pentru testabilitate.

Figura 1 prezintă schema generală a unui sistem de generarea deterministică a testelor. Testele sunt generate în baza unui model al circuitului şi pentru un model al defectului precizat.

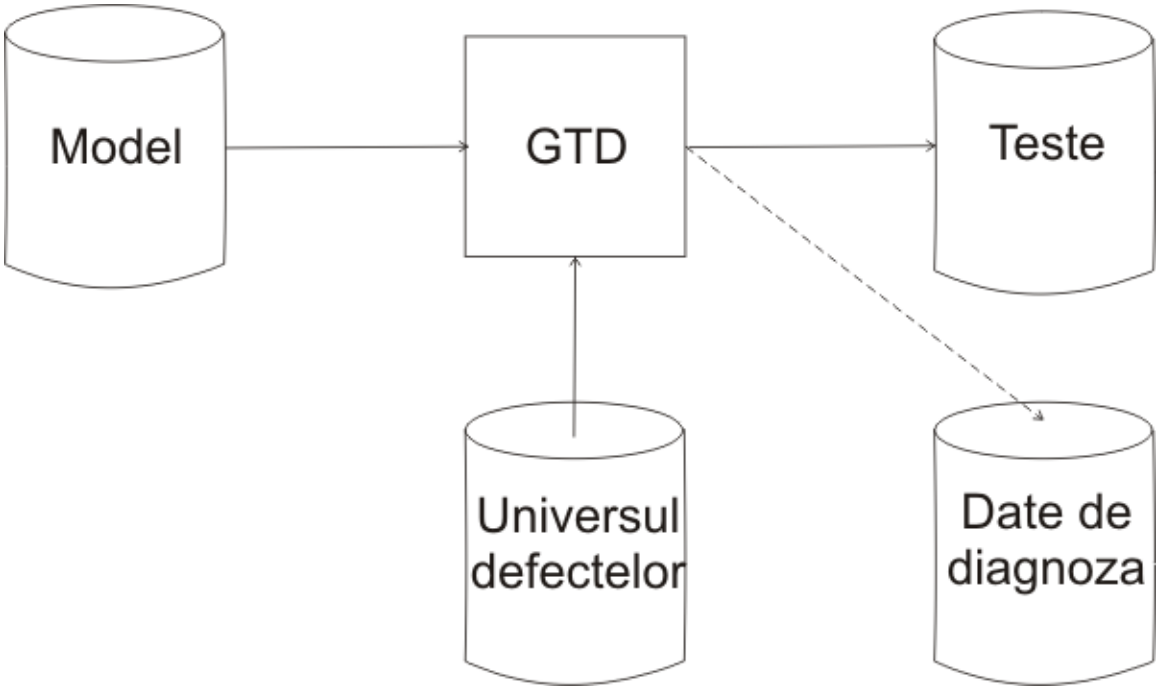


Figura 1. Generarea deterministică a testelor.

Testele determinate includ atât stimulii ce vor fi aplicaţi circuitului testat cât şi răspunsul aşteptat al circuitului corect funcţional, etalonul. Anumite generatoare deterministice de testelor produc şi date de diagnoză ce pot fi folosite pentru localizarea defectelor.

Generarea testelor pentru circuitele combinaţionale

În această secţiune se vor considera circuite combinaţionale compuse din porţi elementare ŞI, ŞI-NU, SAU, SAU-NU şi NU. Pentru ilustrarea principalelor categorii de concepte se vor aborda, pentru început, circuitele fără ramificaţii.

Circuitele combinaţionale fără ramificaţii

Cei doi paşi fundamentali în generarea unui test pentru defectul blocaj al liniei *w b-l-v* (unde *v* are valoarea 0 sau 1) sunt:

- activarea (excitarea) defectului,
- propagarea erorii rezultate la una dintre liniile primare de ieşire (LPE)

Activarea defectului înseamnă poziţionarea valorilor liniilor primare de intrare (LPI) astfel încât să se cauzeze (în circuitul liber de defecte) aducerea liniei *w* la valoarea *v'* (valoare opusă valorii blocajului).
Aceasta constituie o instanţiere a problemei *justificării unei linii*, problemă ce tratează găsirea unei atribuiri a valorilor liniilor primare de intrare astfel încât să rezulte valoarea

dorită pentru o anumită linie din circuit. Pentru a urmări propagarea erorii trebuie să considerăm valori atât în circuitul liber de efecte N cât şi în circuitul defect N_f definit de defectul ţintă.
Valorile logice compozite care reprezintă erorile - 1/0 şi 0/1 - sunt tradiţional notate prin simbolurile D şi respectiv D'. În scrierea 1/0 se înţelege valoarea aceleiaşi linii dar în circuitul etalon (1) şi respectiv în circuitul defect (0)
Celelalte două valori compozite - 0/0 şi 1/1 - sunt notate 0 şi respectiv 1.
Orice operaţie logică dintre două valori compozite poate fi realizată prin procesarea separată a celor două valori.
Astfel:

$$D' + 0 = 0/1 + 0/0 = (0 + 0) / (1+ 0) = 0/1 = D'.$$

Acestor patru valori binare compozite se adăugă o a cincea valoare (x) care reprezintă o valoare compozită nespecificată, cu alte cuvinte oricare valoare din mulţimea {0, 1, D, D'}. În practică operaţiile logice cu valori compozite sunt definite prin tabele (figura 2).

ŞI	0	1	D	D'	x
0	0	0	0	0	0
1	0	1	D	D'	x
D	0	D	D	0	x
D'	0	D'	0	D'	x
x	0	x	x	x	x

(a) Operatorul ŞI extins.

SAU	0	1	D	D'	x
0	0	1	D	D'	x
1	1	1	1	1	x
D	D	1	D	1	x
D'	D'	1	1	D'	x
x	x	1	x	x	x

(b) Operatorul SAU extins.

Figura 2. Operatorii ŞI şi SAU extinşi peste mulţimea de valori {0, 1, D, *X*}.

În figura 3 este descris un algoritm de generare a unui test pentru *w: b-l-v*. Sunt iniţializate în valoarea *x* toate liniile din circuit şi apoi procedează în cei doi paşi de bază, reprezentaţi de rutinele Justify şi Propagate.

```
begin
    toate liniile din circuit au valoarea x
    Justify (w, v')
    if v = 0 then Propagate (w, D)
    else Propagate (w, D')
end
```

Figura 3 generarea testului pentru defectul *l*: *b-l-v* într-un circuit fără ramificaţii.

Justificarea unei linii (Figura 4) este un proces recursiv în care valoarea ieşirii fiecărei porţi este justificată de valorile intrărilor porţii şi astfel recursiv până se ajunge la o linie primară de intrare. Se consideră o poartă ŞI-NU cu k intrări. Există o singură cale de justificare a unei valori 0 a liniei de ieşire; în schimb se pot alege 2^k-1 combinaţii de intrare ce pot justifica o valoare 1 a ieşirii.
Cea mai simplă cale ar fi atribuirea valorii 0 unei singure linii de intrare (aleasă arbitrar) lăsând celelalte intrări nespecificate. Aceasta corespunde alegerii unuia din cele k cuburi primitive în care această poartă ia valoarea 1.

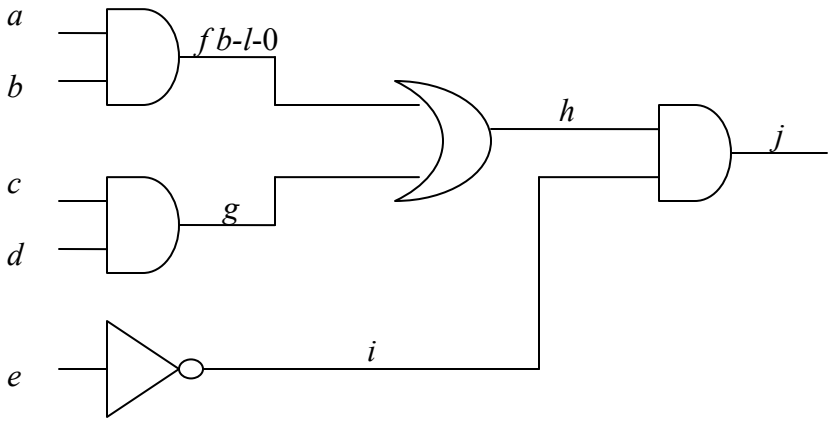
```
Justify (w, val)
begin
        poziţionează linia w la valoarea val;
        /* l este ieşirea unei porţi */
        c = valoarea de control a porţii w;
        i = inversiunea porţii w;
        inval = val ⊕ i;
        if  (inval = c')
                then for every intrare j a porţii w
                        Justify (j, inval);
else
    begin
                alege o intrare (j) a porţii w;
                Justify  (j, inval);
    end
end
```

Figura 4. Justificarea valorii unei linii dintr-un circuit combinaţional fără ramificaţii.

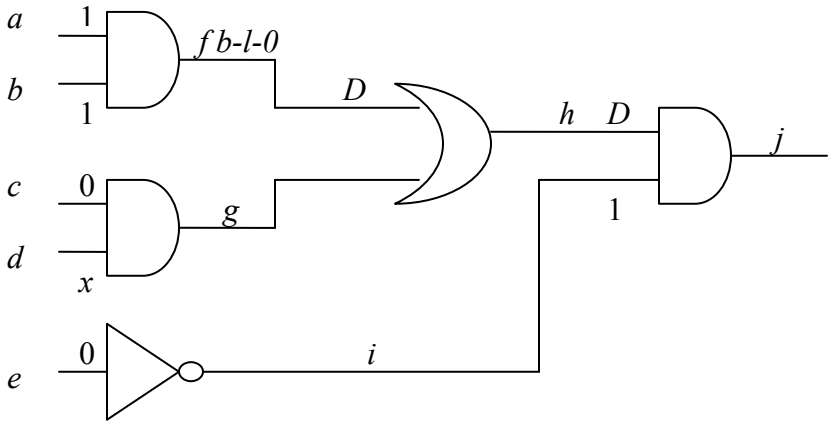
Propagarea unei erori către o linie primară de ieşire a circuitului necesită *senzitivizarea* unei căi unice de la *w* la o inie primară de ieşire. Fiecare poartă de pe această cale are exact o singură intrare *senzitivă* la defect. Evident trebuiesc poziţionate toate celelalte intrări la valori *non-control* pentru poarta respectivă.
Exemplul 1*:* Generarea unui test pentru defectul *f b-l*-0 în circuitul din figura 6 (a). Problemele iniţiale sunt: Justify (f, 1) şi Propagate (f, D). Justify (f, 1) este rezolvată.

```
    Propagate (l, err)
/* err is D or D' */
begin
        w = err;
        if w este LPE then return
        k = forcount (w);
        c =valoarea_de_control (k);
        i = inversiune (k);
        for every intrare j a porţii k alta decât l
                Justify (j, c');
                Propagate (k, err ⊕ i);
end
```

Figura 5. Propagarea erorii într-un circuit fără ramificaţii.



a)



b)

Figura 6

Prin $a = b = 1$ Propagate (*f, D*) necesită Justify (*g*, 0) şi Propagate (*h, D*). Se rezolvă Justify (*g*, 0) prin alegerea unei intrări a porţii *g* – poate fi *c* - şi se poziţionează în 0. Propagate (*h, D*) conduce la Justify (*i*, 1) ceea ce conduce la $e = 0$. Acum eroarea ajunge la LPE *j*.

Figura 6.*b* arată valorile rezultante. Testul generat este 110*x*0 (*x* poate fi arbitrar 0 sau 1). Este important de observat că într-un circuit liber de ramificaţii fiecare problemă de justificare a unei linii poate fi rezolvată independent de toate celelalte, deoarece mulţimile de LPI ce sunt eventual poziţionate (setate) pentru ca prin respectivele atribuiri să satisfacă justificarea unor linii interioare sunt mutual disjuncte unele faţă de celelalte.

Circuitele combinaţionale cu ramificaţii

Se consideră în continuare cazul general al circuitelor cu ramificaţii în comparaţie cu cazul particular anterior examinat. În principiu trebuie atinse aceleaşi scopuri de bază: activarea defectului şi propagarea erorii.

Din nou activarea defectului se traduce într-o problemă de justificare a unei linii. O primă diferenţă cauzată de ramificaţii este aceea că acum pot fi mai multe căi de propagare a unei erori către o LPE. Dar o dată aleasă o cale reducem din nou problema propagării erorii la un set de probleme de tipul justificare-linie.

Dificultatea fundamentală cauzată de ramificaţiile neconvergente este aceea că, în general, probleme de tipul justificare a liniilor ce rezultă nu mai sunt independente.

Exemplul 2: Să considerăm iredundant din figura 7 şi defectul G_1: *b-l*-1. Pentru a activa defectul trebuie să justificăm $G_1 = 0$. Acum există alegerea propagării erorii printr-o cale ce trece prin G_5 sau prin G_6. Să presupunem că s-a decis să se aleagă prima variantă. Atunci este de justificat $G_2 = 1$. Setul rezultant de probleme - Justify (G_1, 0) şi Justify (G_2, 1) - nu pot fi soluţionate simultan. Aceasta demonstrează că propagarea erorii prin G_5 a fost greşită. Nu ne rămâne decât să încercăm decizia cealaltă: propagarea erorii prin G_6. Aceasta necesită $G_4 = 1$, care este eventual soluţionat de $c = 1$ şi $c = 0$. Testul rezultant este 111*x*0 .

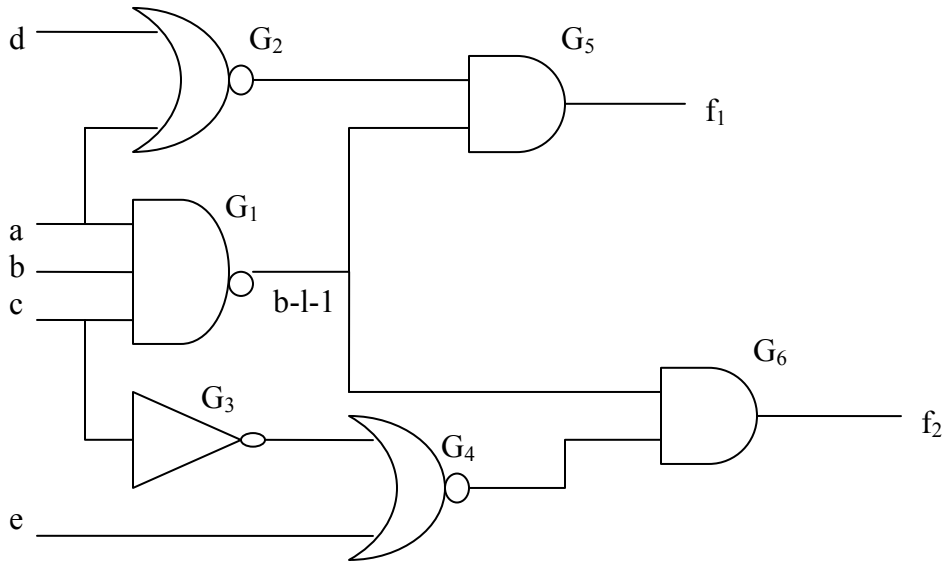


Figura 7.

Acest exemplu arată că este necesar să se exploreze alternative pentru propagarea erorii. Similar, se pot încerca variante diferite pentru justificarea liniilor.

Exemplul 3: Se consideră defectul *h*: *b-l*-1 din circuitul desenat în figura 8. Pentru a activa acest defect h=0. Există o unică cale de propagare a erorii, şi anume prin *p* şi *s*. Pentru aceasta e nevoie de: $e = f = 1$ şi $q = r = 1$. Valoarea $q = 1$ poate fi justificată de *l* =

1 sau de $k = 1$. Întâi se încearcă $l = 1$. Aceasta conduce la $c = d = 1$. Totuşi aceste două atribuiri împreună cu cea anterior specificată $e = 1$, ar implica $r = 0$, ceea ce conduce la o inconsistenţă. În consecinţă justificarea $q = 1$ prin $l = 1$ a fost incorectă. Deci e de ales alternativa $k = 1$ care implică $a = b = 1$. Acum rămâne numai problema de justificare a liniei $r = 1$. Fie $m = 1$, fie $n = 1$ conduce la o soluţie consistentă.

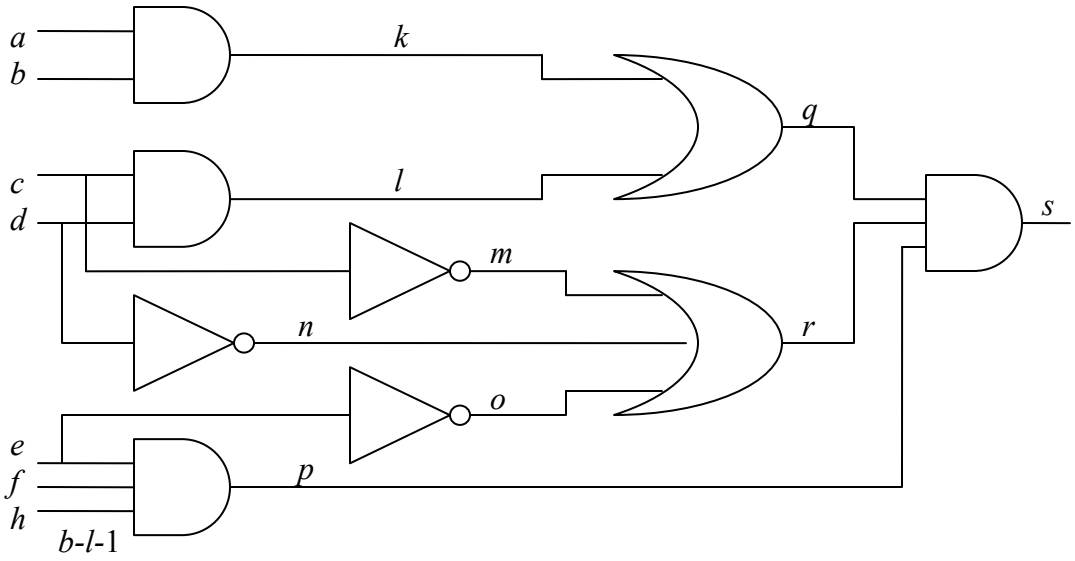


Figura 8.

Backtracking

S-a putut remarca faptul că în căutarea unei soluţii de găsire a unui test apare un proces de decizie. Ori de câte ori există anumite alternative de justificare a unei linii sau de propagare a unei erori, se alege o alternativă şi aceasta e încercată. Dar procedând astfel se poate alege o decizie care să conducă la o inconsistenţă (se foloseşte de asemenea termenul de contradicţie sau conflict). În consecinţă în procesul de căutare după un test trebuie folosită o strategie de *backtracking*, care să permită o explorare sistematică a spaţiului complet al tuturor soluţiilor şi să se poată recupera anumite decizii incorecte. Recuperarea implică restaurarea stării de calcul din starea dinaintea deciziei incorecte.

În mod uzual atribuirile realizate ca urmare a unei decizii determină unic (implică) alte valori. Procesul de calcul al acestor valori şi verificarea consistenţei acestora cu valori anterior determinate se numeşte *implicaţie*.

Figura 9 arată modul în care au progresat valorile calculate pentru exemplul 3, făcând distincţie între valorile rezultate din decizii şi cele generate prin implicaţie. Implicaţiile iniţiale urmează din soluţia unică a activării defectului şi din problemele de propagare a erorii.

În majoritatea algoritmilor de *backtracking* trebuie să se înregistreze toate valorile atribuite ca rezultat al unei decizii pentru a avea posibilitatea ştergerii acestora de îndată ce decizia a condus la o inconsistenţă. În exemplul 3 toate valorile ce rezultă din decizia $l = 1$ sunt şterse, atunci când are loc *backtracking*-ul.

Decizii	Implicaţii	Observaţii
	h = D' *e* = 1 *f* = 1 *p = D'* *r* = 1 *q* = 1 *o* = 0 s = *D'*	*Implicaţii iniţiale*
l = 1	*c* = 1 *d* = 1 *m* = 0 *n* = 0 *r* = 0	*Pentru a justifica q* = 1 *Contradicţie!*
k = 1	*a* = 1 *b* = 1	*Pentru a justifica q* =1
n =1	*c* = 0 *l* =0	*Pentru a justifica r* =1

Figura 9. Deciziile şi implicaţiile pentru exemplul 3.

Figura 10 schiţează o schemă de algoritm GTD cu *backtracking*. Problema iniţială de rezolvat în generarea unui test pentru defectul *l b-l-v* constă din justificarea valorii *v'* pe linia *l* şi propagarea erorii din linia *l* la o LPE. Ideea de bază este că neputând rezolva problema direct, se transformă recursiv problema în subprobleme şi se încearcă soluţionarea întâi a acestor sub-probleme. Soluţionarea unei probleme poate avea drept rezultat SUCCES sau INSUCCES.

```
Solve(.)
begin
        if Implică-şi-verifică()=INSUCCES then return  INSUCCES;
        if (eroare la LPE & toate liniile sunt justificate)
                then return SUCCES;
        if (nici o eroare nu poate fi propagată la o LPE)
                then return INSUCCES;
        alege o problemă nerezolvată;
        repeat
                begin
                        alege o cale neîncercată de soluţionare a
                         problemei;
                        if Solve()=SUCCES than return SUCCES;
                end;
        until toate căile de rezolvare ale problemei au fost încercate;
        return INSUCCES;
end;
```

Figura 6.10 Schiţa generală a unui algoritm de GTD cu backtracking.

Întâi algoritmul tratează toate problemele ce au soluţii unice şi pot fi soluţionate, în consecinţă, prin implicaţie. Este cazul procedurii Implică-şi-verifică, care face de asemenea o verificare a consistenţei. Procedura Solve() raportează INSUCCES

dacă verificarea consistenţi eşuează; în schimb, se raportează SUCCES atunci când se atinge scopul dorit: anume atunci când s-a propagat o eroare la o LPE şi când toate problemele de justificare a liniilor au fost soluţionate. Chiar şi într-o stare consistentă algoritmul poate determina că nici o eroare nu poate fi propagată mai departe către o LPE şi atunci nu are nici un rost să se continue căutarea raportându-se INSUCCES.

Dacă Solve nu poate determina imediat SUCCES sau INSUCCES atunci alege o problemă curent nerezolvată. Aceasta poate fi ori o justificare de linie ori o problemă de propagare a unei erori. În acest punct există anumite căi alternative de soluţionare a problemei alese. Algoritmul selecţionează una dintre căi şi încearcă să rezolve problema la următorul nivel de recursivitate.
Procesul continuă până când se găseşte o soluţie sau toate alegerile posibile au faliat. Dacă activarea iniţială a lui Solve eşuează atunci algoritmul încetează să genereze un test pentru defectul specificat.

Alegerea unei probleme nerezolvate pentru soluţionare şi selecţia unei căi neîncercate de soluţionare pot fi - în principiu - arbitrare. Adică, dacă defectul este detectabil vom genera un test pentru respectivul defect independent de ordinea în care problemele şi soluţiile sunt abordate. Totuşi, procesul de alegere poate afecta în mare măsură eficienţa algoritmului. Există câteva GTD ai căror algoritmi au structura similară cu Solve.