UNIVERSITATEA TEHNICĂ "GH. ASACHI"
IAŞI
FACULTATEA DE AUTOMATICĂ ŞI
CALCULATOARE

Aplicații ale Inteligenței Artificiale în sinteza structurilor numerice complexe

Teză de doctorat

Doctorand: Ms. Ing. Codrin Pruteanu
Conducător ştiinţific: Prof. Dr. Ing. Dan Gâlea

- 2006 -

Cuprins

Capitolul 1 – Introducere
1.1 Microelectronică ………... 6
1.2 Taxonomia circuitelor ………... 7
1.3 Metode de proiectare a circuitelor ……… 7
1.4 Sinteza circuitelor ………. 8
1.5 Proiectarea asistată pe calculator a circuitelor ……… 9

Capitolul 2 – Metode de proiectare a automatelor finite
2.1 Etapele proiectării automatelor finite ……….. 11
2.2 Alocarea stărilor prentru automatele finite ………. 12
2.2.1 Metode de abordare existente pentru alocarea starilor ………. 12
2.2.2 Probleme care apar la realizarea unui sistem de asignare a stărilor ………. 13
2.3 Minimizarea numarului de stări ale automatelor finite ………... 14
2.3.1 Necesitatea minimizării stărilor în sistemele de proiectare CAD ……… 14
2.3.2 Abordări curente ale minimizării stărilor ………. 14
2.4 Alte abordări ale optimizării automatelor finite ……….. 15
2.4.1 Asignarea concurentă a starilor pentru reducerea suprafeței ………... 15
2.4.2 Metode de descompunere şi partiționare a automatelor finite ………. 16

Capitolul 3 – Introducere în teoria laticilor
3.1 Operații algebrice ……… 17
3.2 Latice ………... 18
3.3 Semilatice ……… 19
3.4 Latice ca seturi parțial ordonate ……….. 19
3.5 Diagrame de latice ………... 20
3.6 Sublatice ……….. 20
3.7 Limitele unei latici ………. 21
3.8 Elementele prime şi ireductibile ale unei latici ………... 21
3.9 Latici complete. Sublatici complete ……… 22
3.10 Spații topologice. Seturi parțial ordonate ……… 22
3.11 Latici distributive ……… 23
3.12 Latici modulare ………... 24
3.13 Latici de echivalența ………... 25

Capitolul 4 – Aplicații ale teoriei automatelor
4.1 Specificații inițiale despre automatele finite ………... 26
4.2 Descompunerea automatului de către utilizator ……….. 26
4.3 Descrierea la nivel înalt ………... 27
4.4 Captura grafică a schemei bloc la nivel înalt ……….. 27
4.5 Taxonomia automatelor finite ………. 28
4.6 Reprezentările automatelor ……… 33
4.7 Homomorfismul automatelor ……….. 34
4.8 Echivalența automatelor ……….. 36
4.9 Automate Moore ……… 36
4.10 Reprezentarea automatelor incomplet specificate ………... 37
4.11 Operații cu automate ………... 38
4.12 Operații cu automate utilizînd metode implicite ………. 41
4.12.1 Gruparea în paralel utilizînd metode implicite ……… 41
4.12.2 Gruparea în serie utilizînd metode implicite ……… 41
4.12.3 Descompunerea în serie şi paralel a automatelor finite ………... 42
4.12.4 Descompunerea prin factorizare a automatelor finite ……….. 44
4.12.5 Optimizarea structurilor logice secvențiale complexe cu pachetul ***fsmtool*** ……… 45

Capitolul 5 – Metode de reprezentare a circuitelor complexe
5.1 Metode de reprezentare a mulțimilor de obiecte ………. 46
5.2 Reprezentarea cubică pozițională a mulțimilor de obiecte ………. 47
5.3 Reprezentări implicite ale automatelor finite ……….. 48
5.4 Implementarea operatorilor impliciți ……….. 51

5.5 Tabela de calcul funcțională …… 52
5.6 Implementare cu diagrame de decizie binară …… 52
5.6.1 Ordinea variabilelor …… 53
5.6.2 Reducerea dimensiunilor …… 54
5.6.3 Pachete de programe ce implementează diagrame de decizie binară …… 54
5.6.4 Variante ale ROBDD …… 55
5.7 Metode de reprezentare utilizînd librăria GMP …… 56

Capitolul 6 – Modalități de sinteză a circuitelor digitale complexe
6.1 Minimizarea rețelelor logice …… 57
6.1.1 Necesitatea programelor de minimizare logică …… 57
6.1.2 Minimizarea circuitelor FPGA şi a circuitelor bivalente …… 57
6.1.3 Minimizarea circuitelor logice multivalente …… 58
6.2 Clasificarea procesului de sinteză …… 60
6.3 Optimizarea circuitelor …… 61
6.4 Reprezentarea funcțiilor logice …… 64
6.5 Sinteza funcțiilor logice …… 66
6.5.1 Optimizarea logică independentă de tehnologie …… 66
6.5.2 Optimizarea logică dependentă de tehnologie …… 72
6.6 Optimizarea logică secvențială …… 72
6.7 Minimizarea stărilor …… 74
6.7.1 Minimizarea stărilor la automate finite complet specificate …… 74
6.7.2 Minimizarea stărilor la automate finite incomplet specificate …… 76
6.8 Codificarea stărilor …… 78
6.8.1 Codificarea stărilor pentru circuite bivalente …… 78
6.8.2 Codificarea stărilor pentru circuite multivalente …… 80
6.9 Hazardul circuitelor …… 82

Capitolul 7 – Metode de estimare a efortului logic în proiectarea circuitelor
7.1 Estimarea timpului de răspuns pentru o poartă logică …… 85
7.2 Rețele logice cu mai multe stagii …… 87
7.3 Alegerea lungimii unei căi …… 88
7.4 Optimizarea funcției de fanin a unui circuit logic …… 90
7.4 Optimizarea funcției de fanout a unui circuit logic …… 93

Capitolul 8 – Metode de optimizare globală în proiectarea circuitelor complexe
8.1 Introducere …… 96
8.2 Formularea problemei …… 97
8.3 Implementarea spațiului de soluții …… 98
8.4 Rezultatele implementarii …… 101
8.5 Concluzii şi implementări viitoare …… 105

Capitolul 9 – Strategii evolutive
9.1 Noțiuni introductive …… 106
9.2 Paradigma evolutivă …… 107
9.3 Comparație intre SE, PE si AG …… 109
9.3.1 Rezumat al caracteristicilor …… 110
9.4 Evoluția …… 111
9.5 Codificarea şi transmiterea informației. Cromozomi …… 111
9.6 Operatorii genetici …… 112
9.7 Reprezentarea soluțiilor …… 114
9.8 Blocuri de construcție şi scheme …… 115
9.9 Considerații asupra operatorilor genetici …… 116
9.10 Teorema schemei …… 117
9.11 Paralelismul implicit …… 118
9.12 Metode de căutare a informațiilor …… 118
9.13 Permutări …… 121
9.13.1 Maparea unui set de întregi în permutări …… 121
9.13.2 Inversul unei permutări …… 122

9.13.3 Funcția de mapare 122
9.13.4 Metode de permutare 123
9.13.4.1 Aplicarea metodei de crossover propusă de Davis 123
9.13.4.2 Metoda de crossover mapat parțial (PMX) 124
9.13.4.3 Metoda de crossover ordonat 125
9.13.4.4 Metoda de crossover pozițional 125
9.13.4.5 Metoda de crossover uniform 125
9.13.4.6 Metoda recombinării muchiilor 126
9.13.4.7 Metoda conservativă maximală (MPX) 127
9.14 Selecția 127
9.14.1 Selecția parametrilor 128
9.14.2 Intensitatea selecției 128
9.15 Evaluarea funcției de fitness 129
9.16 Exemplu de implementare a unui algoritm evolutiv 130
9.17 Circuite hardware evolutive 130
9.17.1 Abordarea orientată pe inginerie 132
9.17.2 Abordarea orientată pe embriologie 132

Capitolul 10 – Aplicații ale algoritmilor evolutivi în proiectarea circuitelor digitale complexe
10.1 Preliminarii 133
10.2 Metode de reprezentare a soluțiilor 135
10.3 Seturi de funcții de test 139
10.4 Reprezentări grafice ale funcțiilor de test 142
10.5 Abordări precedente 143
10.6 Metode de implementare 144
10.7 Concluzii şi implementări ulterioare 154

Capitolul 11 – Concluzii finale. Contribuții originale. Obiective de viitor
11.1 Concluzii finale 155
11.2 Contribuții originale 158
11.3 Obiective de viitor 159

Referințe 160

Lista figurilor

Figura 1 – Diagrame de latice 20
Figura 2 – Laticea unui subgrup 24
Figura 3 – Reprezentarea unui automat prin graf de tranziție 33
Figura 4 – Gruparea a două automate 38
Figura 5 – Gruparea în cascadă 39
Figura 6 – Gruparea în serie 40
Figura 7 – Gruparea în paralel 40
Figura 8 – Reprezentările ROBDD ale funcțiilor caracteristice a 3 funcții 47
Figura 9 – Reprezentarea implicită a automatului a.kiss2 49
Figura 10 – Reprezentarea implicită multivalentă a automatului a.kiss2 50
Figura 11 – Nivele de abstractizare şi aspectele corespunzătoare 61
Figura 12 – Modele de circuite, sinteză şi optimizare 63
Figura 13 – Rețeaua booleană 66
Figura 14 – Reprezentarea prin STT a automatului 69
Figura 15 – Reprezentarea prin STG a automatului 69
Figura 16 – Implementarea fizică la nivel RTL a automatului 71
Figura 17 – Diagrama bloc a implementării unui automat finit 72
Figura 18 – Modele de circuite secvențiale 73
Figura 19 – Diagrama de stări. Tabelul de stări 74

Figura 20 – Diagrama şi tabelul de tranziție minimală ……………………………………………….. 75
Figura 21 – Diagrama de stări. Diagrama de stări minimală …………………………………………… 77
Figura 22 – Soluționarea hazardului static utilizînd diagrame Karnaugh …………………………... 83
Figura 23 – Calculul efortului logic pentru un buffer ………………………………………………… 89
Figura 24 – Graful de fluență pentru automatul finit dk27.kiss2 ………………………………………. 90
Figura 25 – Graful de fluență pentru automatul finit dk27.kiss2 ………………………………………. 93
Figura 26 – Circuit secvențial. Maşina prototip ……………………………………………………… 98
Figura 27 – Topologia descompunerii generale … …………………………………………………… 98
Figura 28 – Schema bloc la nivel Top a maşinii prototip …………………………………………... 100
Figura 29 – STG pentru maşina prototip …………………………………………………………... 100
Figura 30 – Schema la nivel RTL pentru maşina descompusă ……………………………………….. 100
Figura 31 – STG pentru prima masină ………………………………………………………………... 101
Figura 32 – STG pentru a doua masină ……………………………………………………………….. 101
Figura 33 – Circuitul secvențial. Maşina prototip. Topologia descompunerii generale ……………. 135
Figura 34-39 – Reprezentarea grafică a funcțiilor de test …………………………………………... 142
Figura 40 – Descrierea functională pentru AG conventional …………………………………………. 145
Figura 41 – Graful de fluență pentru automatul prototip …………………………………………… 148
Figura 42 – Schema top pentru automatul prototip …………………………………………………. 148
Figura 43 – Schema top pentru automatul descompus ……………………………………………… 148
Figura 44 – Graful de fluență pentru aubautomatul 1 ………………………………………………... 148
Figura 45 – Graful de fluență pentru aubautomatul 2 ………………………………………………... 148

Lista tabelelor

Tabelul 1 – Descrierea automatului ………………………………………………………………. 34
Tabelul 2 – Funcțiile automatului A ………………………………………………………………… 35
Tabelul 3 – Funcțiile automatului B ………………………………………………………………… 35
Tabelul 4 – Descrierea automatului incomplet specificat …………………………………………… 37
Tabelul 5 – Reprezentarea prin STT a automatului …………………………………………………... 70
Tabelul 6 – Reprezentarea stărilor interne şi a ieşirilor ……………………………………………… 70
Tabelul 7 – Reprezentarea prin STT a valorilor stărilor următoare ………………………………… 70
Tabelul 8 – Reprezentarea prin STT a valorii ieşirilor ………………………………………………. 71
Tabelul 9 – Efortul logic pentru intrări porți CMOS statice ………………………………………… 86
Tabelul 10 – Estimarea întîrzierii parazitice pentru diferite tipuri de porți logice ………………….. 86
Tabelul 11 – Optimizarea funcției de fanin pentru automatele finite …………………………………... 92
Tabelul 12 – Optimizarea funcției de fanout pentru automatele finite ………………………………. 95
Tabelul 13 – Raportul pentru exemplul shfitreg.verilog ……………………………………………... 102
Tabelul 14 – Raportul pentru exemplul shfitreg-top.verilog ………………………………………... 102
Tabelul 15 – Rezultatele experimentale obținute pt setul de test în procesul de optimizare ……….. 104
Tabelul 16 – Tabelul muchiilor ……………………………………………………………………… 126
Tabelul 17 – Funcțiile de test De Jong ……………………………………………………………… 140
Tabelul 18 – Partiționarea automată a subautomatelor de stări ..…………………………………… 146
Tabelul 19 – Raport pentru Spartan-XL ……………………………………………………………….. 152
Tabelul 20 – Raport pentru Xilinx-XC4000XL ………………………………………………………. 152
Tabelul 21 – Raport pentru Virtex-II ………………………………………………………………... 153
Tabelul 22 – Raport pentru Altera Flex-10k ………………………………………………………….. 153

Anexe

Anexa A – Exemple în format Verilog a circuitelor obținute ……………………………………….. 166

Capitolul 1

Introducere

1.1 Microelectronică

Dezvoltarea continuă a industriei microelectronice a permis evoluția tehnologiilor de proiectare a sistemelor hardware şi software pe parcursul ultimilor ani. Creşterea continuă a nivelului de integrare a dispozitivelor electronice de la un singur strat la nivel multistrat a condus la producerea unor sisteme din ce în ce mai complexe. Dacă în perioada de început a proiectării circuitelor digitale densitatea de integrare era de ordinul miilor de tranzistoare (LSI), în prezent densitatea de integrare a ajuns la nivelul milioanelor de tranzistoare (VLSI), şi se prognozează că în viitor să crească exponențial la ordinul zecilor şi sutelor de milioane de tranzistoare (ULSI).

Legea lui Moore prezice că densitatea de integrare a circuitelor se dublează o dată la 18 luni. Acest fapt conduce la creşterea complexitătii circuitelor integrate şi scăderea la jumătate a perioadelor de tact. Din acest motiv pot apărea dificultăți în satisfacerea condițiilor de timing impuse în proiectarea circuitelor complexe.

Proiectarea circuitelor digitale complexe necesită investiții mari de capital datorită procesului de fabricare a dispozitivelor din ce în ce mai performante necesare pentru creşterea densității de integrare. Cu toate acestea, tendința de creştere a densitătii de integrare a circuitelor este pozitivă din punct de vedere economic deoarece conduce la reducerea numărului de componente din sistem, ceea ce presupune reducerea implicită a costurilor de împachetare şi interconectare precum şi o crestere a fiabilității generale a sistemului şi o scădere a efectelor parazitice din cadrul structurii interne a componentelor.

În prezent există un număr ridicat de sisteme electronice care necesită componente integrate dedicate care sunt specializate să execute o funcție sau un set specific de funcții. Aceste componente se numesc "Application Specific Integrated Circuits" , sau **ASIC**-uri, şi ocupă o porțiune ridicată pe piața de componente electronice.

Tehnologiile CAD au un rol foarte important în reducerea timpului de proiectare şi de optimizare a calității circuitului. Odată cu creşterea nivelului de integrare proiectarea unui circuit fără erori devine o sarcina deosebit de dificilă pentru o echipa de ingineri proiectanți. Tehnologiile CAD sunt o codificare a fluxului de proiectare top-down a circuitelor complexe deoarece permit abordări fezabile în situații specifice.

Programele de sinteză curente țin cu greu pasul cu complexitatea în creştere a circuitelor moderne: procesoare dedicate, acceleratoare grafice, tehnologiile multi-strat şi multi-core iar proiectarea circuitelor digitale devine din ce în ce mai dependentă de numărul de operații dintre sinteza logică şi sinteza fizică, denumită şi „timing-closure", care este o zonă de cercetare din cadrul companiilor EDA (Cadence, Synopsis, IBM, Cypress, TSMC, etc) precum şi a centrelor de cercetare (AT&T, Bell Labs, INRIA) şi a universitătilor mari de profil (MIT, Stanford, U.C. Berkeley, Eindhoven TU, etc).

Descompunerea funcțională[124] a fost utilizată cu succes în analiza şi optimizarea circuitelor discrete binare şi multivalente, precum şi a sistemelor simbolice[68] în domeniul ştiințelor ingineresti moderne.[82] L. Józwiak este unul din cei care au formulat teoria şi metodologia descompunerii generale care înlocuieşte un circuit complex cu o rețea de subcircuite mai mici interconectate, care au acelaşi comportament şi sunt echivalente din punct de vedere functional cu circuitul original, considerat prototip.[61]

1.2 Taxonomia circuitelor

Circuitele microelectronice exploatează proprietățile materialelor semiconductoare. Cele mai utilizate familii de circuite pentru substraturile de silicon sunt “Complementary Metal Oxide Semiconductor”, sau CMOS, cu versiunea bipolară denumită BiCMOS, utilizată încă de la începuturile proiectării circuitelor digitale, precum şi recenta apariție a noilor circuite electronice bazate pe tehnologia “Silicon-On-Insulator”, sau SOI. Pentru circuitele bazate pe tehnologia CMOS perioada de încărcare şi descărcare a joncțiunii substratului de capacitantă este foarte mare ceea ce permite atingerea unor performanţe de funcționare pîna la frecvența de maxim 1 GHz. Pentru circuite implementate în tehnologie SOI, suprafața substratului de capacitantă este eliminată astfel încît tranzistorul este capabil să funcționeze mai rapid întrucit procesul de încarcare a fost eliminat. Se preconizează că în următorii ani circuitele se vor putea proiecta pîna în frecvența de 5 Ghz pe baza tehnologiei SOI, care prezintă un consum redus de curent, timp de raspuns mai bun şi densitate mai mare de integrare.

O alta formă de clasificare a circuitelor este în circuite digitale şi analogice. În varianta analogică informațiile sunt codificate după valori cu parametri electrici continui, precum tensiune şi curent. În circuitele digitale informația este cuantificată. Întrucit majoritatea circuitelor sunt binare, cuantumul informației are o valoare de 0 sau 1, Fals sau Adevarat. Circuitele digitale sunt superioare celor analogice în multe situații, de aceea în ultima perioada sunt utilizate pe scară largă în domeniile de procesare a semnalelor şi telecomunicații, depăşindu-le ca grad de utilizare pe cele analogice.

Dupa modul de operare circuitele digitale pot fi clasificate în circuite **sincrone** şi circuite **asincrone**. Cele sincrone sunt caracterizate prin prezența unui semnal global de procesare denumit clock, pe cînd cele asincrone nu necesită prezența unui semnal de ceas global. Pe parcursul evoluției proiectarii circuitelor au propuse diferite metode de abordare. Pentru circuite cu densitate mare de integrare sunt mai avantajos de utilizat operații sincrone deoarece sunt conceptual mai simple, prezintă un nivel ridicat de cunoaştere funcțională şi există multe metode de verificare. Pe de altă parte avantajul utilizării circuitelor asincrone este dispariția semnalului de ceas global, ceea ce permite pe de o parte obținerea unor performanțe mai ridicate de funcționare, dar pe de altă parte abordarea lor functională este mai dificilă, iar corectitudinea răspunsurilor obținute este mai greu de apreciat.

1.3 Metode de proiectare a circuitelor

Viabilitatea proiectării circuitelor depinde de anumiți factori precum timpul de proiectare, preț şi performanță. Pentru proiectarea circuitelor sunt utilizate diverse metode de proiectare. Acestea sunt de obicei clasificate în proiectare de tip “custom” sau “semicustom”. În primul caz proiectarea funcțională şi fizică se realizează manual, necesitînd un efort suplimentar de optimizare din partea unei echipe de proiectare, detaliat pentru fiecare porțiune din circuit. În acest caz, efortul de proiectare şi costurile sunt mari si de obicei ele sunt compensate prin obținerea unor circuite de o calitate foarte bună. Proiectarea “custom” a fost utilizată în special în perioada de început a proiectării circuitelor digitale, în prezent complexitatea crescută a circuitelor limitînd acest tip de proiectare doar la anumite porțiuni specifice de proiecte şi circuite, precum unitățile de execuție şi virgulă mobilă la unele procesoare. Proiectarea “semicustom” este bazată pe conceptul de restricționare a părților principale ale unui circuit la un număr limitat şi reducerea posibilităților de abordare detaliată

a tuturor părților circuitului. Reducerea numărului posibil de modalități de implementare uşurează dezvoltarea de programe CAD de proiectare şi optimizare, precum şi reducerea timpului de proiectare per ansamblu.[59] Pierderea de performanță este de obicei redusă, deoarece abordarea detaliată poate fi extrem de dificilă pentru proiecte mari, iar tehnologiile de optimizare automată pentru metoda “semicustom” poate explora o gamă mai largă de implementări decît îşi poate permite o echipă de proiectanți, de aceea în prezent numărul de proiecte “semicustom” îl depăşeşte pe cel de tip “custom”, existînd microprocesoare de înaltă performanță proiectate în acest mod. [60]

Proiectarea “semicustom” poate fi împărțită în 2 clase principale: proiectare bazată pe celule, sau “cell base design”, şi proiectare bazată pe arii programabile, sau “array-based design”. Proiectarea bazată pe celule se bazează pe celule de librărie care pot fi proiectate o dată şi apoi utilizate, sau pe utilizarea generatoarelor de celule care sintetizează suprafețe de macrocelule din specificațiile lor funcționale. În proiectarea bazată pe celule, deşi procesul nu este complet simplificat, totuşi este optimizat datorită utilizării blocurilor funcționale deja proiectate.

Proiectarea bazată pe celule include proiectarea cu celule standard. În acest caz celulele fundamentale sunt stocate într-o librarie. Celulele sunt proiectate o singura dată, dar sunt necesare update-uri pe masură ce progresul tehnologic în tehnologia semiconductorilor permite geometrii mai reduse. Menținerea librariei este un proces dificil deoarece fiecare celulă necesită a fi parametrizată în termeni de suprafață şi timp de răspuns la anumite temperaturi şi tensiuni de alimentare. Utilizatorul unei librarii de celule standard trebuie mai întîi să îşi adapteze proiectul la primitivele din librarie disponibile într-un pas denumit **mapare tehnologică**. Apoi celulele sunt plasate şi rutate, toate aceste etape fiind automatizate. O extensie este metoda celulelor standard ierarhice, unde celule mai mari pot fi derivate prin combinarea celor mai mici.

Proiectarea bazată pe macrocelule constă în combinarea blocurilor care pot fi sintetizate de programe denumite **generatoare de module**. Primele generatoare au fost adresate sintezei automate a ariilor de memorii şi suprafețelor logice programabile, sau PLA-uri. În prezent există generatoare sofisticate care sunt capabile să sintetizeze o aranjare a circuitelor cu o densitate şi performanță egală sau superioară celor care pot fi realizate de un proiectant uman. Utilizatorul generatoarelor de macrocelule trebuie să prezinte doar descrierea funcțională, după care macrocelulele sunt plasate şi rutate.

1.4 Sinteza circuitelor

În crearea unui circuit integrat există 4 etape principale: *proiectare*, *fabricare*, *testare* şi *împachetare*. Sinteza constă în rafinarea modelului circuitului, dintr-un model abstract în unul detaliat, care prezintă toate detaliile necesare procesului de fabricare. Sinteza circuitului este cel de al doilea proces creativ. Primul se desfasoară în mintea proiectantului atunci cînd conceptualizeaza circuitul şi schițează primul model. Scopul principal al sintezei circuitelor este de a genera un model detaliat al circuitului, cum ar fi o aranjare geometrică, care sa poată fi folosită pentru fabricarea circuitului integrat. Optimizarea circuitului este de multe ori combinată cu sinteza. Rolul optimizării este de a imbunătăți calitatea generală a circuitului. Un prim aspect ar fi **performanța** circuitului. Performanța ține atît de timpul de procesare al unei informații, precum şi de cantitatea de informație care poate fi procesată într-o perioadă de timp dată. Un alt aspect ar fi **aria** circuitului, un circuit mai mic însemnind o densistate mai mare de integrare pe wafer, deci implicit un cost de producere mai scăzut şi un numar mai mic

de rebuturi. Tehnologiile de sinteză îmbunătăţesc calitatea circuitului, reducînd atît perioada de proiectare cît şi efortul uman.

După taxonomia etapelor de sinteză avem:

- Sinteza la **nivel arhitectural:**

  - constă în generarea unei vederi structurale a unui model la nivel arhitectural. Aceasta corespunde determinării unei asignări a funcţiilor circuitului către operatori, denumiţi resurse, precum şi interconectarea dintre ei şi timpul lor de execuţie. Acest tip de sinteză se mai numeşte şi sinteză de nivel înalt deoarece determină structura la nivel de blocuri a circuitului. Poate specifica modelul unei unităţi de control.

- Sinteza la **nivel logic:**

  - constă în generarea unei vederi structurale a unui model la nivel logic. Acest tip de sinteză manipulează specificaţiile logice în scopul creării de modele logice ca o interconectare a primitivelor logice, generînd structura la nivel de poartă a circuitului. Procesul de transformare a unui model logic într-o interconectare de instanţe de celule de librării este de obicei denumit mapare tehnologică. Poate specifica modelul unui automat finit determinist descris la nivel schematic sau în format de limbaj HDL.

- Sinteza la **nivel geometric:**

  - constă în crearea unei vederi fizice la nivel geometric. Acest tip de sinteză permite specificarea tuturor direcţiilor geometrice definind suprafaţa fizică a circuitului integrat, precum şi poziţionarea acestuia, de aceea mai este denumită şi sinteza la nivel fizic. Poate specifica modelul unei harţi de memorie.

1.5 Proiectarea asistată de calculator a circuitelor

Un proces de proiectare al unui circuit integrat porneşte cu descrierea funcţională a circuitului într-un limbaj de descriere hardware de nivel înalt precum *Verilog* sau *VHDL* şi un set de specificaţii. Scopul procesului de proiectare este de a minimiza funcţia de cost tinînd cont de constrîngeri legate de întîrziere, arie, puterea disipată, etc. Descrierea funcţională este mai întîi trasmisă unui program de sinteză logică pentru a genera un circuit logic optimizat care îndeplineşte constrîngerile conform unui model de cost. Circuitul logic este apoi plasat şi rutat într-un plan bidimensional de un program de plasare şi altul de rutare care minimizează funcţia de cost îndeplinind în acelaşi timp constrîngerile în funcţie de modelul lor de cost. Cu toate că un proiect profesional poate fi realizat manual fără a utiliza programele *CAD* dedicate, în prezent nici un proiect nu mai poate fi realizat fără ajutorul lor. În ultimii ani, s-au putut remarca 2 tendinţe care cuprind metodologia de proiectare şi programele *CAD* prezente. Prima tendinţă este presiunea din ce în ce mai mare a considerentelor de timp de proiectare. Aceasta a impus o presiune enormă asupra proiectantilor de a apela din ce în ce mai mult la programele de proiectare *CAD*. În plus, odată cu descresterea dimensiunilor tehnologice, devin din ce în ce mai importante unele efecte secundare precum întîrzierea crescută a traseelor dintre circuite datorită unei creşteri a rezistenţei sîrmelor. A doua tendinţă este creşterea dimensiunilor de integrare într-un singur chip, în special odată cu explozia cererilor de chip-uri de reţea care să suporte infrastructura Internet. Creşterea integrarii înseamnă de asemenea că proiectanţii trebuie să apeleze din ce în ce mai mult la programele *CAD*, pentru a putea manipula proiecte de complexitate mereu în creştere. Cu toate că aceasta înseamnă că va fi o nevoie crescută pentru instrumente *CAD*, de asemenea va însemna că unele presupuneri şi abstractizări care au fost deja făcute, vor trebui revăzute.[39] De asemenea

trebuie studiat şi numărul în creştere de operaţii dintre sinteza logică şi cea fizică, ceea ce în termeni de specialitate se numeste *"timing closure"*.

Tehnologiile actuale utilizate pentru proiectarea circuitelor digitale includ, printre altele, arii de porti logice , celule de tehnologie standard, circuite executate la comandă, precum şi dispozitive logice programabile *PLD*, arii de porţi logice programabile *FPGA* şi circuite dedicate specifice unor anumite tipuri de aplicaţii *ASIC*. Pe piaţă sunt disponibile o multitudine de pachete software de automatizare a proiectării pentru aceste tehnologii provenite de la diversele companii de profil. Ele permit funcţii precum : captura schemei bloc, minimizare logică, asezarea geometrica a celulelor, plasarea şi rutarea lor, verificarea regulilor de proiectare, simulare, şi altele. Programele de acest tip au fost mai întîi dezvoltate de către marile companii de proiectare hardware precum şi de universitati şi au aparut apoi pe piaţa sub formă de produse comerciale utilizate pentru proiectarea asistată pe calculator *CAD*.[45][46]

În timp ce proiectarea la nivel sistem s-a realizat iniţial începînd de la nivel de descriere sub forma de netlist a circuitelor şi a ecuaţiilor logice ale funcţiilor booleene, în format text sau sub forma de captură de schema a circuitului bloc, noua generaţie de programe utilizate pentru proiectarea hardware porneşte de la descrieri de nivel inalt şi generează datele de ieşire în acest format sau doar in formate intermediare. Descrierile de nivel inalt disponibile de la firmele de profil precum *INTEL* sau *IBM* includ limbaje la nivel de transfer între regiştri, automate finite, expresii booleene şi limbaje structurate de descriere hardware precum *Verilog/VHDL*. În universităţi precum *Stanford, Carnegie-Mellon, UCLA, Technical University-Darmstadt, Oxford University,* se experimentează diverse versiuni de limbaje, expresii regulate, Retele Petri, automate nedeterministe şi paralele, ecuatii recursive, precum şi limbaje de uz general *C/C++*, *Lisp*, *Prolog* sau *Ada,* utilizate pentru automatizarea la nivel înalt a proiectării. [84]

Sunt încă incerte direcţiile de viitor pe care le va urma acest domeniu în privinţa limbajelor de nivel înalt şi a metodelor de proiectare folosite. Unele direcţii sunt oricum deja vizibile din prezentările care au loc în fiecare an la simpozioanele de profil precum *„Design Automation Conference” (DAC), „International Conference on Computer Aided Design” (ICCAD),* sau *„Design Automation and Test in Europe” (DATE),* precum şi din prezentările noilor apariţii software şi din declaraţiile companiilor despre planurile lor de viitor. Acestea includ: integrarea programelor de proiectare, analiză şi verificare în sisteme complete, oferind proiectantului opţiunea de a selecta diverse moduri de proiectare, furnizînd acestuia instrumentele pentru a-şi putea construi propriul său mediu de proiectare *CAD;* interfeţe usor de utilizat bazate pe ferestre şi grafice; incorporarea minimizării funcţiilor booleene şi a implementării tehnologice pentru diverse metode de proiectare , nu doar *FPGA*; utilizarea descrierilor de automate finite pentru proiectarea celulelor executate la comandă.

Capitolul 2

Metode de proiectare a automatelor finite

2.1 Etapele proiectării automatelor finite

Atunci cînd automatul este deja disponibil sub o formă tabelară de 4 tuple, respectiv 3 tuple în cazul maşinilor Moore, se pot face două tipuri de abordări ale sistemului, una simplă, rapidă şi una amplă, complexă.

În prima variantă se preia descrierea automatului într-un limbaj de descriere hardware simplificat şi se generează ecuațiile logice la ieşire. Simbolurile stărilor sunt înlocuite cu codurile stărilor iar tabela de fluență este rescrisă în format de tabelă de adevăr pentru bistabilele de tip D. Atribuirea de coduri stărilor se face conform specificațiilor lor, care este de fapt aleatoare, sau utilizatorul poate să-si declare propriile sale coduri. În sistemele mai avansate, sunt executate o serie de operații de optimizare şi analiză asupra tabelei de fluență pentru conversia în tabela de adevăr. Operațiile de optimizare includ următoarele operații:

- minimizarea numărului de stări interne
- descompunerea stărilor pentru o alocare optimală a acestora
- minimizarea numărului de stări de intrare şi de ieşire
- alocarea stărilor interne
- alocarea stărilor de intrare, ieşire
- conversia din maşina Mealy în maşina Moore sau viceversa

Analiza include o simulare şi o analiză de testabilitate a maşinii, precum şi o analiză a hazardului în funcțiile de ieşire. În unele sisteme tabela stărilor este imbunătățită prin adăugarea de linii şi coloane în plus, în vederea creşterii testabilității acestora. Deşi au fost dezvoltate teoretic mai multe tipuri de analiză a automatelor finite, puține dintre ele au fost implementate în mediul universitar şi în practică. În final, tabela de fluență analizată şi optimizată este convertită în tabela de adevar. Aceasta poate fi creată pentru diverse tipuri de bistabile şi are intrările primare şi ieşirile bistabilelor ca intrări, iar ieşirile primare şi intrările în bistabile ca ieşiri. În majoritatea sistemelor mai recente sunt considerate doar bistabile de tip D, dar există sisteme în care utilizatorul are de ales între mai multe tipuri de bistabile realizate într-un şir. Alegerea unui bistabil potrivit poate micşora esential aria implementării logice pentru unele automate, cum ar fi numărătoarele. În alte sisteme fiecare bistabil poate fi diferit față de celelalte pentru a minimiza şi mai mult suprafața pe siliciu a implementării logice şi a circuitelor bistabile. În literatura de specialitate au fost publicate articole despre noi tipuri de bistabile pentru implementări *VLSI*, care teoretic vor putea reduce şi mai mult suprafața implementării funcțiilor logice, mărind uşor suprafața ocupată de bistabile. În etapa următoare, tabela de adevar este minimizată şi realizată ca o rețea logică. În majoritatea sistemelor curente este obținută o implementare logică bi-nivel, dar în sistemele realizate mai recent există o amplă abordare a proiectării multi-nivel pentru diferite familii tehnologice. Cercetarea în acest domeniu va avea o influență esențială în optimizarea automatelelor finite. Existența unor programe de optimizare rapidă multi-nivel a circuitelor va avea acelaşi efect pe care l-au avut programele bi-nivel atunci cînd au apărut în minimizarea PLA-urilor. [100]

2.2 Alocarea stărilor pentru automatele finite

2.2.1 Metode de abordare existente pentru alocarea stărilor

Una dintre probleleme alocării stărilor este cea a atribuirii de coduri stărilor interne ale automatului cu scopul de a minimiza unele funcții de cost legate de realizarea circuitului (cum ar fi o arie FPGA sau un număr de porți logice). Majoritatea lucrărilor de cercetare abordează doar problema asignării stărilor interne, presupunînd că utilizatorul specifică codurile pentru stările de intrare şi de ieşire. Există alte abordări care se bazează pe principii similare care atribuie numere binare stărilor de intrare şi de ieşire, care sunt specificate inițial după nume. Aceasta are o implicație directă în minimizarea unităților de control microprogramate sau în proiectarea unităților de control cu FPGA-uri. Problema este clasică în Teoria Comutării Circuitelor, dar pină de recent au existat puține instrumente software. Bazele abordării problemei asignării stărilor şi a teoriei automatelelor au fost formulate la începutul anilor '60. Au existat şi unele implementări software la unele din metode, dar rezultatele au fost nesatisfăcătoare. În schimb, recent, problema atribuirii stărilor devine abordabilă datorită numeroaselor încercări de a realiza compilatore pentru implementarea pe siliciu la nivel logic a ariilor de circuite *VLSI*.

Există mai multe tipuri de abordări de principiu a problemei asignării stărilor:

Teoria partiționării dezvoltată de catre Hartmanis, Stearns, Kohavi

Teoria se bazeaza pe conceptul de a partiționa stările automatului în blocuri. Partițiile se găsesc în tabele de fluență şi sunt folosite pentru a găsi seturi de partiții care produc codări unice optimale. Această teorie este foarte elegantă din punct de vedere matematic, şi dă o pătrundere în natura proprietăților structurale ale maşinii, şi de asemenea este foarte generală: poate fi folosită la descompunerea automatelelor, minimizarea stărilor, implementări cu regiştri de shiftare, etc. Rezultatele publicate ale programelor care utilizează această teorie, precum şi implementările practice demonstrează că acest tip de abordare devine dificil de aplicat pentru maşini cu mai mult de 10 stări, 10 intrări şi 10 ieşiri.

Abordarea evaluării pe coloană, dezvoltată de către Dolotta şi Mc Cluskey

Coloanele tabelei de fluență primesc un punctaj în funcție de diverse criterii legate de calitatea asignării stărilor. Scorurile sunt folosite pentru a găsi partițiile potrivite şi apoi atribuirea. Această abordare poate produce rezultate foarte bune, pentru toate tipurile de bistabile; rezultatele sunt chiar mai bune decît cele din primul tip de abordare, dar metoda este de asemenea dificil de aplicat pentru maşini cu mai mult de 12 stări.

Evaluarea enumerativă Story

Sunt evaluate toate partițiile posibile drept candidați posibili pentru asignare, calculînd complexitățile realizărilor funcțiilor booleene corespunzătoare, şi presupunînd ca toate celelalte partiții au fost selectate optimal pentru ele. Este selectat pentru atribuire subsetul partițiilor cu cele mai bune scoruri. Această abordare produce rezultate mai bune decît a 2-a abordare, dar este totuşi lentă.

Abordarea Branch and Bound dezvoltată de Perkovski, Lee şi Zasovska

Are 2 variante. Prima permite realizarea de maşini cu 8 intrări, 8 stări interne şi 8 ieşiri şi produce rezultate optime pentru diverse tehnologii şi bistabile. În această variantă trebuiesc evaluate aproape toate partițiile posibile. Acesta este softul care furnizează cele mai optime soluții din cele publicate, dar are neajunsul că este foarte lent. Metoda de aproximare care se bazează pe această metodă nu evaluează toate partițiile, doar selectează euristic cele mai bune

partiții si permite realizarea de maşini cu 12 stari, care pot fi extinse pîna la 18-20 de stări. Amîndouă variantele realizează asignarea stărilor combinată cu minimizarea lor.

Abordările asingnării quadruple dezvoltate de Armstrong, De Micheli şi Perkovski
Toate aceste abordări se bazează pe transformarea unor grafuri create din tabela de fluență a automatului în grafuri de tip hipercub. Această abordare permite realizarea de maşini cu 100 de stări, dar soluțiile sunt departe de cele optime obținute în varianta 6. Au existat unele îmbunătățiri teoretice, dar nu au fost aplicate. *De Micheli* a implementat 2 algoritmi pentru asignare, la *Berkeley University*. În primul dintre ele, bazat pe metoda asignării quadruple, asignarea stărilor este redusă la problema îmbunătățirii grafului.[28] Cel de al doilea se bazează pe alte principii, şi nu a obținut rezultate satisfacatoare. Rezultatele obținute, unde crearea grafului şi a algoritmului îmbunatatit se bazau pe principii noi, par să fie satisfăcătoare: au fost asignate maşini cu 136 de stări. Dar nu sunt disponibile comparații detaliate cu alte tipuri de abordări. [29]

Algoritmul Moroz
Este un algoritm constructiv îmbunătățit foarte rapid şi eficient care a fost folosit pe larg în proiectarea automatizată a sistemelor din întreprinderile din estul Europei. Autorul a implementat acest algoritm şi a observat că poate găsi asignări pentru maşini cu peste 100 de stări, dar calitatea soluțiilor pentru automatele mici este departe de optim. Acesta este probabil cel mai rapid algoritm disponibil. Viteza rezultă din faptul că nu rezolvă asignările quadruple ci doar dezvoltă problema îmbunătățirii grafului care este creat direct din graful de fluență.

Abordarea De Michelli, Brayton şi Sangivanni-Vincenteli
Abordarea se bazează pe minimizarea funcțiilor booleene multi-valoare cu scopul de a găsi grupuri de stări şi asignări constructive îmbunătățind apoi aceste grupuri prin exprimarea lor în funcțiile de fețele hipercubului. Strategia este foarte inovativă şi rezultatele obținute sunt foarte bune. Este probabil din acest punct de vedere cel mai bun produs disponibil prezent din punctul de vedere al raportului timp/calitate. A fost aplicat la masini cu pina la 100 de stări, dar cel mai mare rezultat publicat a fost pentru 27 de stări.[28] Grupurile de stări sunt găsite cu ajutorul minimizării funcțiilor booleene multi-valoare cu programul *Espresso*, aplicat automatului înainte de asignarea stărilor. Această metodă de găsire a grupurilor de stări sau blocuri, combinată cu minimizarea booleană rapidă a fost sursa de succes a programului. Programul *KISS,* împreună cu formatul de descriere a automatului *.kiss*, respectiv versiunea sa îmbunătățită *.kiss2*, este în prezent folosit pe larg în universități şi de multe companii care produc softuri comerciale. Un program care extinde această abordare a fost produs de firma *Intel.*

2.2.2 Probleme care apar la realizarea unui sistem de asignare a stărilor

În general se doreşte un program care să permită o abordare rapidă, eficientă, şi să fie aplicabilă în proiectarea *VLSI* în cît mai multe situații posibile, ceea ce este practic imposibil. Utilizatorul trebuie să implementeze una din metodele prezentate, specifică pentru clasa lui de automate şi necesități de viteză şi performanță, sau să implementeze mai mulți algoritmi şi să îi aplice interactiv pentru fiecare situație particulară în parte.

Este necesară o analiză detaliată a metodelor de reducere, precum şi evaluarea complexității algoritmilor optimali pentru problemele matematice, cum ar fi imbunătățirea hipercubului, îmbunătățirea fețelor şi a asignării quadruple. De aceea pare a fi de viitor problema utilizării de calculatoare şi arhitecturi paralele precum şi acceleratoare hardware dedicate în acest scop.

2.3. Minimizarea numărului de stări ale automatelor finite

2.3.1 Necesitatea minimizării stărilor în sistemele de proiectare *CAD*

Minimizarea numărului de stări ale automatelelor finite constă în unificarea grupurilor de stări compatibile ale unei maşini într-o singură stare în noua maşina , cu scopul de a obține o reprezentare echivalentă cu un numar redus, şi posibil minimal, de stări. Se presupune că un automat cu mai puține stări permite o modelare superioară. Stările care prezintă aceeaşi comportare la aceleaşi intrări se numesc stări compatibile. Problema se complică atunci cînd, în cazul general, grupuri de stări compatibile nu sunt disjuncte iar compatibilitatea unor grupuri poate duce la incompatibilitatea altor grupui de stări. Deşi există un anumit număr de lucrări publicate în acest domeniu, se pare că nu există nici un sistem în industria americană care să implementeze această abordare. Contrar acestei situații, în industria europeană există numeroase sisteme de acest tip în aplicațiile industriale. În Statele Unite managerii responsabili nu au prevăzut necesitatea unor astfel de sisteme. Unul din motive ar fi din cauza faptului că proiectarea de nivel înalt, arhitecturală, este de cele mai multe ori separată de minimizarea logică iar proiectarea la nivel de aşezare geometrică a tranzistoarelor este realizată de grupuri separate de ingineri. Atunci cînd cei care se ocupă de minimizarea logică primesc specificațiile circuitului de la cei care se ocupă cu proiectarea arhitecturii sistemului, ei nu mai observă atît de multe posibilități de îmbunătățire, cum ar fi cele care rezultă din introducerea stărilor de indiferență care ajută la minimizarea automatului. Maşinile sunt de dimensiuni mici, deoarece descompunerea inițială este realizată intuitiv de către proiectantul arhitecturii. Un alt motiv important ar fi cel legat de faptul că niciodată nu a existat o piață de mari dimensiuni pentru sisteme de control industriale realizate ca automate finite, deoarece ele au fost lăsate în urmă de aparitia microcontrolerelor şi a microprocesoarelor. În Europa de est, aceste tipuri de circuite secvențiale, care de obicei sunt de mari dimensiuni, au fost realizate ca automate finite cu PLA-uri, deci a existat o necesitate de software pentru minimizarea stărilor şi asignarea lor. Necesitatea pentru minimizarea stărilor este mai evidentă în prezent cînd noile instrumente de nivel înalt pentru descrierile automatelelor finite precum *Expresii Regulate*, *Retele Petri*, scheme de programe paralele, şi altele, sunt incorporate în sistemele *CAD* iar conversiile în tabele de fluență vor fi executate automat.[54]

De exemplu, minimizarea unui automat complet definit face parte din procedurile de: conversie a expresiilor regulate în tabele de fluență, conversie în automat paralel, automat nedeterminist sau *Retea Petri* a tabelei de fluență, verificarea echivalenței a 2 automate şi multe altele. O multitudine de noi algoritmi produc tabele de fluență care sunt incomplete, au multe stări de indiferență, şi de aceea pot fi minimizate substantial.

2.3.2 Abordări curente ale minimizarii stărilor

Programele de minimizare a stărilor pot fi impărțite în 2 categorii: cele pentru automate complet definite şi cele pentru automate incomplet definite. Prima problemă este relativ uşoară. Relația de compatibilitate a stărilor automatului în această situație este una de echivalență şi prin urmare au fost dezvoltați algoritmi eficienți de ordinul O(n*log(n)). Problema minimizării automatelor incomplet definite este mult mai dificilă, iar soluția la un izomorfism nu este unică precum în cazul automatelelor complet specificate. Există relativ puțină documentație în acest domeniu, iar toate publicațiile abordează încercarea de a găsi spațiul minim de soluții şi să utilizeze o variantă a algoritmului de închidere/acoperire pentru

grupuri de stări compatibile. Ideea este de a selecta un astfel de subset de stări compatibile, care este complet şi închis, ceea ce înseamnă că toate numerele stărilor inițiale sunt incluse în unele grupuri, şi că fiecare grup implicat de un grup selectat este de asemenea selectat. Au existat versiuni îmbunătățite ale algoritmilor clasici, dar nu au fost implementate software. Este necesar un efort substantal pentru scrierea de software dedicat şi multă cercetare la baza proiectării algoritmilor pentru minimizarea stărilor pentru automate cu sute de stări, intrări şi ieşiri, care apar în circuitele curente, precum module de unităti de control şi microprocesoare.

2.4 Alte abordări ale optimizării automatelor finite

Deoarece abordarea clasică de proiectare a automatelor finite implică rezolvarea unor probleme dificile precum minimizarea stărilor şi asignarea lor, sau poate produce rezultate modeste, au existat multe încercări de a implementa metode de proiectare diferite. Ele încearcă minimizarea suprafeței sau a numărului de circuite şi include urmatoarele metode:

- proiectarea de automate finite de uz general bazate pe regiştri de shiftare sau alte maşini elementare în loc de bistabile
- proiectarea de bistabile speciale cu intrări multiplexate
- proiectarea de arii regulate de maşini configurabile elementare
- descompunerea automatelelor în structuri de automate
- conversia tipului de masina, Moore în Mealy, şi viceversa, pentru a selecta una cu performante mai bune
- minimizarea concurentă a stărilor şi asignarea lor
- structuri speciale de automate, precum diverse tipuri de unităti de control microprogramate cu multe mutiplexoare şi memorii ROM, sau realizări partiționate ale logicii de control
- conversia directă a descrierii de nivel înalt cum ar fi cea a automatelor paralele în simboluri sau asezare geometrica

Aceste tipuri de abordări sunt utilizate selectiv pentru diferitele tipuri de automate. Unele dintre ele pot fi utilizate pentru toate automatele, unele doar pentru cele de mari dimensiuni, altele doar pentru cele de mici dimensiuni. [112]

2.4.1 Asignarea concurentă a stărilor pentru reducerea suprafeței

Abordarile curente ale sintezei structurale a automatelelor nu sunt minimale. Metodele curente sunt de a minimiza inițial numărul stărilor interne ale automatului şi apoi de a realiza o asignare a stărilor. Scopul acestor metode este în general de a micşora suprafața semiconductorilor pentru implementarea pe o anumită tehnologie selectată. Metodele aplică aproximari de cost foarte abstracte pentru a obține acest lucru. Este mult mai indicat să se minimizeze unele funcții de cost generalizate dependente de tehnologia dorită. Minimizarea numărului de stări interne rezultă din următoarea presupunere : "cu cît sunt mai multe stări interne, cu atît este mai complicată realizarea, deoarece sunt necesare mai multe elemente de memorie, există mai multe funcții de excitație, şi prin urmare realizarea este mai complicată". Exemplele practice demonstrează faptul că tabela de fluență cu numărul minim de stări interne nu este neapărat punctul de start pentu a obține realizarea circuitului de o complexitate

minimală per ansamblu, calculată ca o sumă a numărului de bistabile şi a numărului de porţi combinaţionale. Multe dintre tehnologiile curente nu necesită pentru început minimizarea numărului de elemente de memorie, aşa cum este presupus în metodele clasice. Exemplele practice demonstrează faptul că nu trebuie să căutăm un automat cu un numar minim de stări interne sau cu funcţiile de excitaţie care să depindă de numărul minim de variabile. Pot fi găsite usor exemple de automate care au realizări minimale, deşi au un număr de bistabile mai mare decît miniminul, şi asta din cauză că ele prezintă funcţii de excitaţie mai simple. De asemenea, încercarea de a găsi o realizare a setului de funcţii de excitaţie cu numărul minimal de variabile argument este de cele mai multe ori nefolositor, deoarece astfel de realizări pot avea mai multe porţi sau conexiuni decît alte realizări ale acestor funcţii. Mai mult, aceste presupuneri nu ţin cont de realizarea funcţiilor de excitaţie pentru implementări cu bistabile, altele decît cele de tip D. Una dintre soluţiile de abordare a acestei situaţii ar fi înlocuirea celor 2 etape de minimizare a numărului de stări interne şi de atribuire a stărilor, cu o singură etapă de minimizare în acelaşi timp cu asignarea stărilor. În acest caz, funcţia de cost, în raport cu funcţiile de excitaţie alese într-o anumită technologie este optimală.[108]

2.4.2 Metode de descompunere şi partiţionare a automatelelor finite

Alt grup de metode se referă la descompunerea unui automat într-un grup de subautomate. Aici există o multitudine de abordări. Metodele clasice bazate pe teoria partiţionării, caută în general anumite partiţii care permit descrierea automatului ca o compunere în paralel sau cascadă a altor automate. Aceste proceduri pot fi realizate recursiv. Din păcate, aceste tipuri de descompuneri sunt foarte laborioas de găsit, iar situaţiile de aplicaţie practică apar foarte rar în automatele industriale. Unele variante utile ale teoriei clasice discută despre descompunerea unor blocuri importante cum ar fi registrele de shiftare şi numărătoarele.

Noile metode de descompunere încearcă folosirea metodelor de partiţionare a grafurilor [23], pentru partiţionarea grafului de fluenţă. Aceste metode sunt utilizate în sistemele practice ca o necesitate, dar comportamentul lor pare nesatisfăcător. Un alt grup de metode presupun o anumită structură de realizare a automatului, de exemplu o unitate microprogramată, în care unele ieşiri din automat sunt legate de intrarea de adresă a unui multiplexor şi controlează selecţia semnalelor de intrare ale acestui automat, ieşirea multiplexorului fiind una din intrările automatului. Sunt realizate PLA-uri separate pentru codificarea semnalelor de intrare şi de ieşire.[120] PLA-ul principal care realizează funcţia de excitaţie şi cea de ieşire este partiţionat în mai multe PLA-uri. Aceste abordări combinate, pot produce diverse structuri descompuse cu costuri de implementare foarte diferite. Altă abordare posibilă este de a crea metode, care sunt aplicabile în particular pentru unele variante selectate de realizare a automatului.

Capitolul 3

Introducere în teoria laticilor

Descompunerea automatelor finite presupune gǎsirea unor partiții pe mulțimea stǎrilor automatului care satisfac condiția de ortogonalitate. Acest lucru poate fi realizat tinînd seama de diferitele proprietǎți pe care le are aceastǎ mulțime. În acest capitol se vor studia proprietǎțile algebrice legate de existența partițiilor avînd proprietatea de substituție. Mulțimea acestor partiții are o structurǎ de latice a cǎrei cunoaştere teoreticǎ este utilǎ, atît pentru realizarea descompunerilor în serie şi paralel dar şi pentru cazul descompunerii generalizate care le cuprinde şi le inglobeazǎ, şi constituie suportul teoretic fundamental care va sta la baza tuturor operațiilor realizate pe seturi finite de pe parcursul acestei lucrǎri.[35]

3.1 Operații algebrice

În aceastǎ secțiune vom începe cu descrierea setului de teorii prezentate în acest capitol. Faptul cǎ un element x este inclus într-un set M, şi un element y nu este inclus în M, va fi notat cu $x \in M$ respectiv $y \notin M$. Dacǎ pe un set P este definitǎ o relație de ordine, π, atunci π se va numi **set parțial ordonat** în funcție de π. Dacǎ cel puțin una din relațiile $x \leq y(\pi)$ şi $y \leq x(\pi)$ este valabilǎ pentru orice pereche de elemente $(x,y) \in P$, atunci P se va numi **lanț** sau **set complet ordonat** în funcție de π. [101]

Fie c_0 un element oarecare dintr-un set parțial ordonat P. Putem forma un sub-lanț din P în felul urmǎtor: fie c_0 cel mai {mare, mic} element din sub-lant; altfel, fie $c_k (k \geq 1)$ un element din P astfel încît una din relațiile $\{c_k < c_{k-1}, c_k > c_{k-1}\}$este adevǎratǎ. Dacǎ fiecare din lanțurile astfel formate, începînd cu orice c_0 este finit, atunci P satisface condiția de minim(maxim).[3]

Teoremǎ: Dacǎ un set P parțial ordonat satisface condiția de minim (respectiv de maxim), atunci pentru orice $x \in P$, existǎ cel puțin un element minim(respectiv maxim) $m \in P$, astfel încît $x \geq m$ ($x \leq m$). [35]

Definiție: O mapare de o singurǎ valoare $f : x_1 , \ldots , x_n \rightarrow f(x_1 , \ldots , x_n)$ (1) dintr-un set nevid A, este denumitǎ o operație de variabilǎ n, definitǎ în A. Este posibilǎ definirea unui numǎr nelimitat de operații într-un set A. Dacǎ A este infinit, atunci chiar şi o infinitate de operații distincte pot fi definite în A iar numǎrul de variabile poate fi diferit la fiecare operație.

Definiție: Douǎ algebre $A = A(\{f_\gamma\}_{\gamma\in\Gamma})$ şi $B = B(\{g_\delta\}_{\delta\in\Delta})$ sunt *algebre similare* dacǎ existǎ o mapare unu la unu σ din setul Γ aparținînd setului Δ astfel încît pentru fiecare $\gamma\in\Gamma$, operațiile f_γ şi $g_{\sigma(\gamma)}$ au acelaşi numǎr de variabile. În mod evident, similaritatea algebrelor este o relație de echivalențǎ. O mapare de o singurǎ valoare φ este un **homomorfism** al lui A în B dacǎ mapeazǎ pe A în B, şi dacǎ pentru orice index γ ($\gamma\in\Gamma$) şi oricare sistem de elemente $x_1 , \ldots , x_{n(\gamma)} \in A$ avem $\varphi (f_\gamma (x_1 , \ldots , x_{n(\gamma)})) = f_\gamma (\varphi (x_1), \ldots \varphi (x_{n(\gamma)}))$. (2)

Definiție: Dacă o mapare φ de această natură este nu numai de o singură valoare dar şi unu la unu, atunci este denumită **izomorfism**. În cazul special în care $B = A$ avem de a face cu un **endomorfism** în locul unui homomorfism şi cu un **automorfism** în locul unui izomorfism. Dacă algebra A are un homomorfism φ în algebra similară B, atunci B este denumit **imaginea izomorfică** a lui A după homomorfismul φ.

3.2 Latice

Un set L este denumit *latice* dacă sunt definite în L două operații binare, **reuniune** şi **intersecție**, care atribuie fiecărei perechi de elemente $(a,b) \in L$, în mod unic un element $a \cap b$ (intersecția lui a cu b) precum şi un element $a \cup b$ (reuniunea lui a cu b) astfel încît sunt îndeplinite următoarele axiome de latici L_1 - L_6:

L_1 Pentru orice elemente $a,b,c \in$ L, avem $(a \cap b) \cap c = a \cap (b \cap c)$

L_2 Pentru orice elemente $a,b,c \in$ L, avem $(a \cup b) \cup c = a \cup (b \cup c)$

L_3 Pentru orice elemente $a,b \in$ L, avem $a \cap b = b \cap a$

L_4 Pentru orice elemente $a,b \in$ L, avem $a \cup b = b \cup a$

L_5 Pentru orice elemente $a,b \in$ L, avem $a \cap (a \cup b) = a$

L_6 Pentru orice elemente $a,b \in$ L, avem $a \cup (a \cap b) = a$

Cu alte cuvinte, într-o latice ambele operații sunt comutative şi asociative; mai mult, ele au proprietățile exprimate de L_5 şi respectiv L_6, care sunt referite ca **identitate de absorbție a operațiilor de intersecție şi reuniune**. O consecință imediată a identităților de absorbție este:

Teoremă: Fiecare latice are următoarele proprietăți: (Dedekind)

L_7 Operațiunea de intersecție este idempotentă, $a \cap a = a$, pentru $\forall a \in$ L.

L_8 Operațiunea de reuniune este idempotentă, $a \cup a = a$, pentru $\forall a \in$ L.

Corolar: Pentru oricare 2 elemente a,b aparținînd unei latici, atunci $a \cap b = a \cup b$ dacă şi numai dacă $a = b$.

Teoremă: Fiecare latice are următoarea proprietate:

L_9 Pentru orice elemente $a,b \in$ L, $a \cap b = b$ dacă şi numai dacă $b \cup a = a$ (Bergmann).

$L_{9'}$ Pentru orice elemente $a,b \in$ L, $a \cap b = b$ implică $b \cup a = a$

$L_{9''}$ Pentru orice elemente $a,b \in$ L, $b \cup a = a$ implică $a \cap b = b$

În această formă, $L_{9'}$ şi $L_{9''}$ nu sunt duale, întrucît ele diferă nu doar prin simbolurile operaționale $\cap$ şi $\cup$, dar sunt interschimbate de asemenea şi literele a şi b. Întrucit a şi b sunt

elemente arbitrare din *L*, este permis a se interschimba valoarea lui *a* cu cea a lui *b*. Astfel obținem afirmația:

L_{9*}. Pentru o pereche de elemente dată $(a,b) \in L$, $a \cup b = b$ implică $b \cap a = a$.

3.3 Semilatice

Pentru fiecare algebră de 2 operații există o întrebare importantă şi interesantă despre ce algebră va rezulta dacă una din operații ar fi eliminată. În cazul unei latice L, fie $L^{\cup}$ şi $L^{\cap}$, care denotă aceste algebre unare, cu aceleaşi elemente ca cele din L, dar numai operațiunea de intersecție din L este considerată în $L^{\cap}$ şi numai operațiunea de reuniune din L respectiv în $L^{\cup}$.

Considerînd L_1 , L_3 , L_7 , precum şi L_2 , L_4 , L_8 , operația definită pe ambele structuri $L^{\cap}$ şi $L^{\cup}$ este asociativă, comutativă şi idempotentă.

În general, un set *H* este denumit o **semilatice** dacă este definită în el o operație care atribuie fiecărei perechi de elemente *a,b* un element $a \circ b$, astfel încît sunt satisfăcute următoarele *axiome de semilatice*:

H_1: Operația ° este asociativă, ceea ce rezultă că $(a \circ b) \circ c = a \circ (b \circ c)$ pentru orice triplet de elemente $a, b, c \in H$.

H_2: Operația ° este comutativă, ceea ce rezultă că $a \circ b = b \circ a$ pentru orice pereche de elemente din *H*.

H_3: Operația ° este idempotentă, ceea ce rezultă că $a \circ a = a$ pentru orice element $a \in H$.

În consecință, dacă L este latice, atunci ambele $L^{\cap}$ şi $L^{\cup}$ sunt semilatice. Prima este denumită laticea-intersecție, iar a doua laticea-reuniune, din L. Între aceste două semilatice se stabileşte o conexiune foarte strinsă prin L_5 , L_6 şi L_9 .

3.4 Latice ca seturi parțial ordonate

În orice latice poate fi definită în mod natural o relație de ordine, cu ajutorul operațiilor cu latici.

Teoremă: Într-o latice L, avem prescrierile:

1. $a \leq b \Leftrightarrow a \cap b = a \ (a,b \in L)$, defineşte o relație de ordine.
2. $a \leq b \Leftrightarrow a \cup b = b \ (a,b \in L)$, echivalentă cu (1)

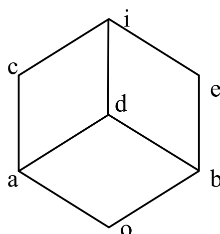
Prin comutativitatea operațiilor cu latici şi datorită proprietății L_9 , prescrierea (2) este echivalentă cu (1), prin urmare, în cazul în care prescrierea (2) mai convenabilă, se poate folosi în locul lui (1).

Teoremă: În funcție de relația de ordine (1), fiecare subset finit $\{a_1, ..., a_n\}$ al unei latici L are un infimum şi supremum, denumite:

$$(3)\ inf_L \{a_1, ..., a_n\} = \bigcap_{j=1}^{n} a_j \quad , \quad sup_L \{a_1, ..., a_n\} = \bigcup_{j=1}^{n} a_j$$

3.5 Diagrame de latice

Orice latice finită, fiind un set finit parțial ordonat, poate fi reprezentată printr-o diagramă denumită **diagramă de latice**. Diferite seturi finite parțial ordonate pot fi reprezentate de aceeaşi diagramă dacă şi numai dacă sunt izomorfe. Prin urmare, orice latice finită este determinată în mod unic prin diagrama sa pîna la izomorfism. Laticile finite sunt în mod normal dat descrise prin intermediul diagramelor lor. Lucrul cu o diagramă este la fel de simplu ca şi lucrul cu tabelele, cu avantajul că diagrama permite în plus definirea ambelor operații în acelaşi timp. De la diagrama unei latice, intersecția şi reuniunea a 2 elemente, *a* şi *b*, poate fi gasită prin faptul că $a \cap b = \inf \{a,b\}$ şi $a \cup b = \sup \{a,b\}$. Prin urmare, dacă $a \le b$, avem $a \cap b = a$, $a \cup b = b$; dacă $a \ge b$, atunci avem $a \cap b = b$, $a \cup b = a$. În fine, dacă $a \| b$, atunci $a \cap b$ este acel element situat cel mai sus (dintre *a* şi *b*) din care este posibil de a trasa segmente linie care conectează cercuri ale diagramei către *a* şi *b*. Spre exemplu, diagrama din Fig. 1 reprezintă o latice în care operațiile sunt definite după cum urmează: pentru fiecare element *x* al laticei, avem: $0 \cap x = 0$, $0 \cup x = x$, $i \cap x = x$, $i \cup x = i$, iar tabelele şi diagramele arată astfel:



∩	**a**	**b**	**c**	**d**	**e**
a	a	o	a	a	o
b	o	b	o	b	b
c	a	o	c	a	o
d	a	b	a	d	b
e	o	b	o	b	e

∪	**a**	**b**	**c**	**d**	**e**
a	a	d	c	d	i
b	d	b	i	d	e
c	c	i	c	i	i
d	d	d	i	d	i
e	i	e	i	i	e

Figura 1. Diagrame de latice

3.6 Sublatice

Ținînd cont de definiția unei „subalgebre” a unei algebre, putem defini un subset nevid $R \in L$ astfel încît:

(1) $a, b \in R \Rightarrow a \cap b , a \cup b \in R$.

este o **sublatice** a laticii *L*. În mod evident, axiomele $L_1 - L_6$ sunt satisfacute de către orice triplet $a, b, c \in R$. După cum este de asteptat, fiecare sublatice a laticii *L* formează o latice în raport cu operațiile definite în *L*. De asemenea *L* este o sublatice a ei. Sublaticile laticii *L*, altele decît *L* însăşi, sunt denumite **sublatice proprii** ale lui *L*. Sublaticile proprii care constau din mai mult de un element sunt denumite **sublatice netriviale**.

Definiție: Un subset *I* al unei latice *L* este denumit un **ideal** al lui *L* dacă *I* satisface următoarele două condiții:

I$_1$: $a,b \in I \Rightarrow a \cup b \in I$;
I$_2$: pentru $\forall x \in L, a \in I \Rightarrow a \cap b \in I.$

Prin această definiție, fiecare ideal al unei latice L este o sublatice a laticii L.

3.7 Limitele unei latici

Dacă o latice L are un element {o, i} astfel încît orice element $x \in L$ satisface inegalitatea { $o \leq x$, $i \geq x$} , atunci {o, i} este denumit cel mai {mic, mare} element din L. Aceste elemente se mai numesc de altfel şi elementele limită ale lui L. Prin definiția ordonării laticilor, cel mai mic element o şi cel mai mare element i al unei latice L satisface identitățile:

(1) $o \cap x = o$, $o \cup x = x \ (x \in L)$

(2) $i \cup x = i$, $i \cap x = x \ (x \in L)$

Altfel spus, cel mai {*mic, mare*} element {*o, i*} al laticii L este elementul {*zero, unitate*} al laticii intersecție $L^{\cap}$ şi elementul {*unitate, zero*} al laticii reuniune $L^{\cup}$.

Teoremă: Fiecare latice are cel mult un element minim şi unul maxim; aceste elemente sunt în acelaşi timp elementul cel mai mic respectiv cel mai mare al laticii.

Corolar: O latice care satisface condiția de {*minim, maxim*} are un element {*minim, maxim*}. În mod particular fiecare latice de lungime finită este marginită.

3.8 Elementele prime şi ireductibile ale unei latici

Fiecare element a al unei latice poate fi reprezentat sub forma $a = x \cap y$. (Spre exemplu x = a si y ≥ a). Totusi o descompunere de acest fel a lui a nu aduce nici o informație în plus, este de interes doar reprezentarea de forma $a = x \cap y$ cu $x,y > a$. Un element $a \in L$ este reductibil la intersecție dacă există în L elementele a_1 şi a_2 astfel încît avem:

(1) $a = a_1 \cap a_2 \ (a_1 , a_2 > a)$

Dacă a nu poate fi descompus în forma (1) atunci se poate spune că este ireductibil prin intersecție. În mod dual, $\forall a \in L$ este reductibil respectiv ireductibil la reuniune dacă poate fi reprezentat în forma:

(2) $a = a_1 \cup a_2 \ (a_1 , a_2 < a)$

În mod evident, cel mai mic element şi fiecare atom al unei latice marginite este ireductibil la reuniune, în timp ce cel mai mare element şi fiecare atom dual a unei latice marginite este ireductibil la intersecție.

Teoremă: Într-o latice care satisface condiția de maxim, fiecare din elementele sale poate fi reprezentat ca o intersecție a unui numar finit de elemente ireductibile la intersecție.

Teoremă: Într-o latice complementară fiecare element prim {*intersecție, reuniune*} cu excepția celui {*minim, maxim*} este un {*atom, atom dual*} al laticii.

3.9 Latici complete. Sublatici complete

Definiție: Dacă pentru orice subset nevid R aparținînd unei latice L există {intersecția $\cap R$, reuniunea $\cup R$}, atunci L este **latice completă** în raport cu {$\cap$, $\cup$}. Dacă L este completă în raport cu ambele operații, atunci laticea este completă. Conceptul de latice completă a fost introdus într-o formă diferită de către Birkhoff [3]. Aparent, din definiție, fiecare latice finită este completă.

Dacă L este o latice completă atunci, în particular, trebuie să existe ambele afirmații: $\cap L = inf_L\ L$ şi $\cup L = sup_L\ L$. Dar prin definiția minimului $\cap L \leq x$ pentru orice element $x \in L$; atunci $\cap L$ este cel mai mic element din L. În mod similar, $\cap L$ este cel mai mare element din L. Prin urmare, fiecare latice completă este marginită. Mai mult, tot prin definiție, dualul unei latici complete este de asemei o latice completă. În consecință dualul fiecarei propoziții despre latici complete este de asemenea adevărat. Aceast fapt poate fi exprimat ca un principiu al dualitătii laticilor complete.

Definiție: Un subset R aparținînd unei latici complete L va fi denumit **sublatice completă** la {intersecție, reuniune} din L, dacă există {$inf_R H$, $sup_L H$} pentru orice subset H din R care coincide cu {$inf_L H$, $sup_L H$}.

Este evident că orice interval al unei latici complete este o sublatice completă. Într-adevăr, dacă $H \subseteq [a,b] \subseteq L$ şi L este o latice completă, atunci a este limita inferioară respectiv b este cea superioară a lui H. Prin urmare, prin definiția la infimum şi supremum obținem relația: $a \leq inf_L H \leq sup_L H \leq b$. Cu alte cuvinte ambele $inf_L H$ şi $sup_L H$ aparțin lui $[a,b]$ şi în consecință $inf_L H$ este cea mai de jos margine inferioară iar $sup_L H$ este de asemenea ultima margine superioară a lui $H \in$ [a,b].

3.10 Spații topologice. Seturi parțial ordonate

Un spațiu topologic poate fi descris prin definirea tuturor seturilor închise din spațiul respectiv. De asemenea o operațiune de închidere poate fi definită ca o operațiune de stabilire a seturilor închise de subseturi de spații care au fost închise în raport cu operațiunea de închidere. Una din topologiile naturale ale seturilor parțial ordonate este **topologia internă** în care elementele sub-bazei sunt doar aceste subseturi. Întrucît fiecare subset de un singur element al setului parțial ordonat este un interval, prin introducerea unei topologii interval pe un set parțial ordonat, este obținut un spațiu T_I. Dacă dorim să introducem o topologie pe un set T, trebuie considerat un subset $\{S_\gamma\}_{\gamma\in\Gamma}$ al laticii subset $\wp(T)$ şi format cea mai mică sublatice $Z \in \wp(T)$ care include fiecare S_γ, la fel ca şi un O şi T. Z este sublaticea completă la intersecție, generată de $\{ S_\gamma\} \cup \{O,T\}$ în $\wp(T)$. Elementele lui vor fi considerate ca cele mai mici subseturi ale lui T. Elementele lui Z astfel definite pot fi exprimate prin intermediul lui S_γ.

Definiție: Un set parțial ordonat P este denumit un **set direct** {*sus* , *jos*} dacă fiecare subset de două elemente din P are cel puțin o limită {joasă , înaltă} în P.Un set parțial ordonat orientat în ambele direcții în acelaşi timp, se numeşte set orientat.

3.11 Latici distributive

Clasa laticilor descrise anterior cuprinde o categorie care satisface următoarele condiții în plus la axiomele precedente ale laticilor.

L_{10} Pentru orice triplet de elemente (a, b, c) ale unei latice, $a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$;

L_{11} Pentru orice triplet de elemente (a, b, c) ale unei latice, $a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$;

O latice pentru care sunt valabile L_{10} şi L_{11} este denumită **latice distributivă**. L_{10} este denumită **identitate distributivă la intersecție** iar L_{11} **identitate distributivă la reuniune**.

Laticile distributive complementare sunt denumite Algebre Booleene, după numele autorului care le-a elaborat. [4]

Inegalitățile $a \cap (b \cup c) \geq (a \cap b) \cup (a \cap c)$ şi $a \cup (b \cap c) \leq (a \cup b) \cap (a \cup c)$ sunt valabile pentru orice triplet de elemente (a, b, c) ale unei latice, similar cu identitățile distributive. Prima este denumită **inegalitate distributivă** la intersecție iar cea de a 2-a este denumită **inegalitate distributivă** la reuniune. Prin aceste două inegalități, pentru a demonstra distributivitatea unei latice este suficient a se demonstra că pentru orice triplet (a, b, c) al unei latice avem:

(1) $a \cap (b \cup c) \leq (a \cap b) \cup (a \cap c)$
(2) $a \cup (b \cap c) \geq (a \cup b) \cap (a \cup c)$

Teoremă: Atunci cînd o latice satisface L_{10}, ea satisface de asemenea şi L_{11} şi viceversa. (Schrőder, p.286).

Corolar 1: O latice în care sunt satisfacute L_{10} şi L_{11} este distributivă.

Corolar 2: O latice în care una din inegalitățile (1) şi (2) este satisfacută fără restricții este distributivă.

Demonstrație: Dacă L_{10} este valabilă pentru o latice, atunci pentru orice triplet a,b,c aparținînd laticii avem:

$$\begin{aligned}(a \cup b) \cap (a \cup c) &= ((a \cup b) \cap a) \cup ((a \cup b) \cap c) = \\ &= a \cup ((a \cap c) \cup (b \cap c)) = \\ &= (a \cup (a \cap c)) \cup (b \cap c) = \\ &= a \cup (b \cap c)\end{aligned}$$

prin urmare L_{11} este valabilă. Datorită considerentelor duale, L_{10} poate fi dedusă din L_{11}. Teorema este astfel demonstrată ; corolarele sunt triviale.

Teoremă: Dualul, fiecare sublatice şi fiecare imagine homomorfică a unei latice distributive este de asemenea o latice distributivă.

3.12 Latici modulare

Se poate demonstra prin simple calcule că în orice latice ecuația în L_{10} este adevarată pentru orice triplet de elemente (a , b, c) ale laticii care satisface relatia $a \leq c$. Există de asemenea latici nedistributive importante în care orice triplet de elemente a, b, c ($a \leq c$) satisface de asemenea ecuația L_{11}. Întrucît în cazul în care $a \leq c$ partea dreaptă a ecuației din L_{11} se reduce la $a \cup (b \cap c)$, aceste latici pot fi caracterizate ca satisfăcînd următoarea condiție:

L_{12} Pentru orice triplet de elemente (a , b, c) ale laticii care satisface relația $a \leq c$, se menține identitatea $a \cup (b \cap c) = (a \cup b) \cap c$.

Identitatea descrisă în L_{12} este denumită **identitate modulară**, iar laticile care au proprietatea L_{12} sunt denumite **latici modulare**. Deci este evident că orice latice distribuită este modulară.

Este util de remarcat că pentru orice triplet de elemente a, b, c ($a \leq c$) al oricărei latici, există inegalitatea modulară $a \cup (b \cap c) \leq (a \cup b) \cap c$. Atunci dacă pentru orice triplet a, b, c ($a \leq c$) al oricărei latici, există şi inegalitatea reversă, atunci laticea este modulară.

Teoremă: O latice L este modulară dacă şi numai dacă fiecare triplet $(a , b, c) \in L$ satisface ecuația:

(1) $a \cup (b \cap (a \cup c)) = (a \cup b) \cap (a \cup c)$. (Jordan).

Laticea tuturor subgrupurilor unei latici necomutative în general nu este modulară. Spre exemplu, laticea subgrup a unui grup alternant de gradul patru α_4 (Fig.2), în cazul în care permutările sunt scrise ca produse de cicluri, este după cum urmează:

a = {(1) , (123) , (132)},
b = {(1) , (124) , (142)},
c = {(1) , (12) , (34)},
d = {(1) , (13) , (24)},
e = {(1) , (14) , (23)},
f = {(1) , (134) , (143)},
g = {(1) , (234) , (243)},
h = {(1) , (12) (34) , (13) (24) , (14) (23)},
o = {(1)}.

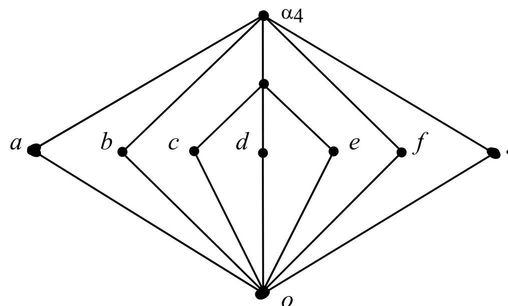


Figura 2. Laticea unui Subgrup

În acest caz avem $c \leq h$, dar niciodată $c \cup (a \cap h) = c \cup o = c$ şi $(c \cup a) \cap h = \alpha_4 \cap h = h$.

Relația dintre structura unui grup şi laticea subgrupului său a fost studiată de numeroşi matematicieni. În mod evident dacă doua grupuri sunt izomorfe, atunci laticile subgrupurilor lor sunt de asemenea izomorfe. Pe de altă parte, grupuri care au laticile subgrupurilor izomorfe, nu sunt în mod necesar izomorfe. Oricum, pot apărea cazuri particulare de clase de grupuri, cum ar fi cele care pot fi descompuse în produse libere (Sadovski), care sunt **grupuri izomorfe unic determinate** de către laticile subgrupurilor lor.

3.13 Latici de echivalență

Fie o latice M şi $\xi(M)$ setul de relații de echivalență care pot fi definite în setul M. Se poate defini o partiție a setului M, o **descompunere** a lui M în subseturi disjuncte mutual, a cărui reprezentare este de forma:

$$M = \bigcup_{\gamma} M_\gamma \qquad (M_{\gamma'} \cap M_{\gamma''} = O \text{ , daca } \gamma' \neq \gamma'')$$

Subseturile M_γ sunt denumite **clase ale partitiei** considerate. În continuare setul tuturor partițiilor va fi notat cu $\wp(M)$. Este de asemenea ştiut faptul că pornind de la orice relație de echivalență $\phi \in M$, se poate forma o partiție $c(\phi) \in M$ după cum urmează: elementele x şi $y \in M$ trebuie să aparțină aceleiasi clase din $c(\phi)$, dacă şi numai dacă $x \equiv y(\phi)$. În mod contrar, fiecare partiție a dintr-un set M dă nastere la o echivalență $\Theta(a) \in M$. Pentru unele elemente x şi $y \in M$, fie $x \equiv y(\Theta(a))$, valabilă daca şi numai dacă x şi y sunt în aceeaşi clasă din a. Mai mult, avînd formată partiția $c(\phi)$, scoasă din relația de echivalență ϕ după cum a fost descris mai sus, şi relația de echivalentă $\Theta(c(\phi)) = \phi$ din $c(\phi)$ dupa cum tocmai a fost definită, se ajunge înapoi la ϕ. Acesta poate fi exprimat simbolic prin $\Theta(c(\phi)) = \phi$. În mod similar, avem $c(\Theta(a)) = a$. Prin urmare, corespondența:

$$(1) \qquad \phi \rightarrow c(\phi) \qquad (\phi \in \xi (M);\ c(\phi) \in \wp(M))$$

este o mapare unu la unu a lui $\xi (M)$ în $\wp(M)$. În consecință, se poate spune că $c(\phi)$ este o **partiție aparținînd relației de echivalență ϕ**. În mod evident, orice relație de echivalență şi partiția aparținînd ei se determină mutual una pe cealaltă. Clasele lui $c(\phi)$ sunt denumite în mod frecvent **clase ale relației de echivalență ϕ**, sau, pe scurt, *clase-ϕ*.

Determinarea relațiilor de echivalență ale unui set finit M dat este cel mai rapid îndeplinită cu ajutorul partițiilor corespunzătoare. Partițiile unui set finit M de acest tip pot fi descrise prin subsumarea elementelor lui M care aparțin unei clase subunitare şi aceleiaşi perechi de paranteze. În această notație, toate parțiile setului de patru elemente $M = \{1,2,3,4\}$ pot fi enumerate după cum urmează:

(1) (2) (3) (4);	(1,2) (3) (4);	(1,3) (2) (4);
(1,4) (2) (3);	(1) (2,3) (4);	(1) (2,4) (3);
(1) (2) (3,4);	(1,2) (3,4);	(1,3) (2,4);
(1,4) (2,3);	(1,2,3) (4);	(1,2,4) (3);
(1,3,4) (2);	(1) (2,3,4);	(1,2,3,4).

După cum se poate observa, setul $\wp(M)$ este finit dacă şi numai dacă M este finit.

Capitolul 4

Aplicații ale teoriei automatelor

4.1 Specificații inițiale despre automatele finite

Automatele finite, *FSM*-urile, sunt reprezentate prin grafuri sau tabele de fluență. Uneori tabelele sunt specificate ca seturi de tranziții: (stare prezentă, stare de intrare, stare următoare, stare de ieşire), unde starea prezentă şi starea următoare sunt stări interne ale automatului, iar stările de intrare respectiv ieşire sunt valori ale combinațiilor semnalelor de intrare şi de ieşire. Stările interne sunt reprezentate prin numere sau cifre; în cazul maşinilor cu stări codificate, ele sunt şiruri de 0 şi 1. Numărul de elemente din şir corespunde numărului de bistabile din automat. Tabelele de fluență sunt foarte utile pentru multe tipuri de automate. Cu toate acestea ele au două dezavantaje: nu pot fi utilizate pentru automate complexe şi sunt greu de modificat, adăugarea unor noi semnale de intrare necesitînd rescrierea întregului tabel. Un alt tip de reprezentare este cea folosind diagramele temporale. Ele reprezintă legătura dintre intrări şi iesiri prin figurarea acestora de-a lungul axei timpului. Aceste reprezentări nu sunt atît de utile în cazul în care numărul posibil de secvențe de intrare este foarte mare sau dacă se doreşte descrierea în întregime a comportamentului unui sistem secvențial. Aceste descrieri sunt utile atunci cînd este dată doar o descriere parțială, în cazul în care se doreşte scoaterea în evidență a anumitor restricții temporale ce apar în funcționarea sistemului. Reprezentările interne trebuie să fie mai putin sugestive din punct de vedere al utilizatorului, în schimb trebuie să fie compacte şi usor de manipulat de către diverşii algoritmi. În acest scop sistemele numerice secvențiale sunt reprezentate în mod implicit sau explicit. În mod explicit pentru un automat finit se reprezintă în memorie următoarele:

- tabela de fluență sub forma unei matrici de stări de intrări şi ieşiri
- graful de fluență sub forma informatiilor referitoare la noduri şi arcele constituente
- mulțimea tranzitiilor sub forma unor cvadruple de tip (intrare, stare prezentă, următoare, şi ieşire)

Prin metoda implicită, aceste sisteme sunt reprezentate prin intermediul funcțiilor caracteristice corespunzătoare funcțiilor de tranziție şi de ieşire sau prin intermediul funcției caracteristice a relației de tranziție-ieşire. Acestea prezintă avantajul că pot fi reprezentate şi manipulate în mod eficient cu ajutorul diagramelor de decizie binară.

În proiectarea sistemelor sunt utilizate următoarele tipuri de abordări:

4.2 Descompunerea automatului de către utilizator

Această situație este în principal cauzată de incapacitatea sistemelor şi a tehnologiei folosite de a putea manipula automate de dimesiuni mari. În acest caz utilizatorul este responsabil de a specifica automatul nu doar ca o singură entitate, ci ca o colecție de automate mai mici care comunică reciproc, fiecare dintre ele avînd un număr redus de stări interne, intrări şi ieşiri. Deşi există puțina documentație despre cum se descompun automatele, cercetarea care se face în acest domeniu de către firmele de profil şi universități poate oferi ca rezultat versiuni de programe îmbunătățite, pornind de la teoria clasică de descompunere a automatelor bazată pe partiționarea acestora, teorie dezvoltată pentru prima dată de către Harrison şi Hartmani/Stearns, dar folosind metoda partiționării grafurilor care se aplică grafurilor de fluență ale automatelor.

4.3 Descrierea la nivel înalt

Este în prezent folosită cel mai mult în majoritatea sistemelor academice şi în unele sisteme ale marilor companii. Limbajele de descriere a comportării la nivel de registru reprezintă mediul cel mai folosit acum, conversia din acest format în tabele de fluenţă fiind cunoscută. Proiectarea la acest nivel facilitează integrarea sistemelor *CAD* în acest cadru cu instrumente precum: simulatoare la nivel de registru, care simulează comportamentul circuitului; programe de optimizare a grafului de fluenţă, care optimizează viteza şi costul căii critice precum şi părti ale unitătii de control a circuitelor; programe de alocare a căii datelor, care ajută la găsirea structurilor/planurilor optimale pentru alocarea datelor, precum şi alte utilitare de proiectare a părţilor unităţii de control precum şi a fluxului de date. Există în prezent şi un interes pentru alte forme de descriere la nivel înalt, cum ar fi *Expresiile Regulate* şi *Retele Petri*. *Expresiile Regulate* sunt expresii care descriu o clasă de de limbaje regulate. Ele compun expresii de forma *E1* şi *E2* cu ajutorul operaţiilor *'sumare de expresii': E1* sau *E2*, *'concatenare': E1* plus *E2*, şi '*iteratie': E1* repetat de un număr arbitrar de ori. Sistemele de proiectare a automatelor pornind de la expresii regulate au fost concepute în unele universităţi printre care: *Stanford, Carnegie-Mellon* şi *Columbia Universiy.* Sistemul Stanford utilizează un limbaj special, care este o combinaţie de stări maşină şi expresii regulate. Prima versiune utiliza anumite reguli pentru a mapa geometric diferite expresii; următoarea versiune, îmbunătăţită , utilizează o abordare mai clasica în care din expresie este mai întîi extras automatul finit care este apoi codificat şi realizat cu ajutorul funcţiilor booleene. Un alt tip de aplicaţie este convertirea unui vector de limbaje regulate într-un automat Mealy sau Moore, cu un program scris într-un limbaj de inteligenţă artificială precum Prolog.

4.4 Captura grafică a schemei bloc la nivel înalt

Descrierea automatului prin intermediul unei interfeţe grafice pare a fi o soluţie promiţătoare pentru sistemele viitoare şi poate fi folosită în conjuncţie cu toate celelalte metode specificate anterior. Sunt utilizate sistemele cu ferestre şi meniuri pentru captura grafică a schemei logice sau bloc şi diagrame de sistem în timp real pentru proiectarea software, cum ar fi cel de la firma *Intel* în care se foloseşte captura schematică pentru a capta grafurile de fluenţă ale automatelor. De asemenea, o metodă similară poate fi aplicată şi pentru captura diagramelor de fluenţă, *Retele Petri*, *Expresii Regulate*, precum şi automate nedeterministe. Interfeţele grafice realizate pentru aceste tipuri de date trebuie să prezinte toate compatibilităţile standard, precum: mărire, liste de meniuri, ferestre separate pentru grafice şi text care pot fi selectate de la un nivel înalt de descriere prin imagini şi butoane. Ele trebuiesc să permită introducerea datelor orientate obiect organizate ierarhic. Există multă documentaţie şi algoritmi despre conversia descrierilor de nivel înalt a automatelor în tabele de fluenţă, sau în descrieri la nivel scăzut precum reţele de blocuri logice. Unele metode de conversie directă a grafurilor de fluenţă, *Expresii Rregulate* sau *Retele Petri*, chiar şi pînă la nivel de descriere geometrică a schematicului au fost deja create. Unelele dintre conversii sunt probleme NP-complete, pentru care nu este cunoscut un algoritm secvenţial determinist cu timp de execuţie polinomial, ceea ce le face greu de aplicat pentru automate complexe, prin urmare descompunerea este din nou o necesitate. De asemenea este necesară mai multă activitate de cercetare, care să folosească mai mult metodele algoritmice şi euristice descoperite de recent în domeniile de *CAD* pentru *VLSI* precum şi *Inteligenţa Artificială* şi *Programarea Algoritmică*. Apariţia unor procesoare mai performante produse de către firmele de profil precum Intel Itanium sau noul Clawhammer de la AMD, va imbunătăţi considerabil performanţele întrucît majoritatea algoritmilor de conversie sunt uşor de modificat în vederea obţinerii paralelismului în execuţie.

4.5 Taxonomia automatelor finite

Funcția caracteristică este utilizată în domeniul automatelor finite pentru a reprezenta seturi şi relații. În continuare se vor defini cîteva clase utile de automate finite precum şi posibilitățile de interconectare dintre acestea.[55][62][119]

Definiție: Pentru un set finit de simboluri *S* (un alfabel finit), se defineşte un limbaj ξ ca un set de şiruri de simboluri din *S*.

Definiție: Pentru un subset $S \subseteq U$, unde *U* este un domeniu finit, fie X_S funcția caracteristică a lui *S*. Funcția $X_S : U \rightarrow B$ este definită astfel:

pentru fiecare element $x \in U$, avem $X_S = \begin{cases} 0, daca\ x \notin S, \\ 1, daca\ x \in S. \end{cases}$

Definiție: Pentru o relație $R \subseteq XxY$, unde *X* şi *Y* sunt nişte domenii finite, fie X_R funcția caracteristică a lui *R*. Funcția $X_R : X\ x\ Y \rightarrow B$ este definită astfel:

Pentru fiecare pereche $(x,y) \in X\ x\ Y$, avem $X_R(x,y) = \begin{cases} 0, \text{daca } x\ si\ y \text{ nu sunt in relatia R}, \\ 1, \text{daca } x\ si\ y \text{ sunt in relatia R}. \end{cases}$

Definiția poate fi extinsă pentru orice relație *n*-ară.

Definiție: Un automat finit nedeterminist este definit ca o 5-tuplă *M = (S, I, O, T, R)*, unde:

- mulțimea *S* reprezintă spațiul finit de stare,
- mulțimea *I* reprezintă spațiul finit de intrare,
- mulțimea *O* reprezintă spațiul finit de ieşire,
- mulțimea *R* reprezintă starea de reset,
- mulțimea *T* reprezintă relația de tranziție definită ca o funcție caracteristică $T : I\ x\ S\ x\ S\ x\ O \rightarrow B$.

Pentru o mărime de intrare *i*, automatul finit nedeterminist aflat în starea prezentă *p*, poate tranzita într-o nouă stare *n* şi un output *o*, dacă şi numai dacă *T(i,p,n,o)* = 1 (adică *(i,p,n,o)* este o tranziție). Există una sau mai multe tranziții pentru fiecare combinație de stări prezente *p* şi intrări *i*.

Un automat finit poate fi specificat printr-un tablel de tranziție, care este o listă tabelară de tranziții în *T*. Un automat finit defineşte o structură de tranziții care poate fi de asemenea descrisă printr-un graf de tranziții (fluență). Automatul tranzitează dintr-o stare *p* cu intrarea *i* într-o stare *n* cu ieşirea *o*, printr-un arc etichetat cu $p \xrightarrow{i/o} n$

Definiție: Pentru un automat finit *M = (S, I, O, T, R)*, graful stărilor de tranziție al lui *M* este un graf orientat, notat cu *G(M) = (V,E)*, unde fiecare stare $s \in S$ corespunde unui nod în *V* etichetat cu *s*, şi fiecare tranziție $(i,n,p,o) \in T$ corespunde unui arc orientat în *E* de la nodul *p* la nodul *n*, iar arcul este etichetat de către perechea de intrare/ieşire *i/o*.

Definiție: O relație *T* de tranziție de stare este completă dacă: $\forall\ i,p\ \exists n,o\ [T(i,p,n,o)] = 1$.

- $\forall$ este folosit pentru a nota o cuantificare universală
- $\exists$ este folosit pentru a nota o cuantificare existentială

Ecuația definită este o abreviere la următoarea expresie:

$\forall i \in I\ \forall p \in S,\ \exists n \in S\ \exists o \in O$ astfel incit *T(i,p,n,o)* = 1.

Reprezentarea lui T permite abordarea unor tranziții nedeterministe în funcție de stările următoare şi ieşiri, şi de asemenea permite corelarea între stările următoare şi ieşiri. Forme mai specializate de automate finite pot fi obținute din rescricționarea formei tranzițiilor permise în T.[64]

Definiție: Un automat finit pseudo nedeterminist este definit ca o 6-tuplă $M = (S, I, O, \delta, L, R)$, unde:

- mulțimea S reprezintă spațiul finit de stare,
- mulțimea I reprezintă spațiul finit de intrare,
- mulțimea O reprezintă spațiul finit de ieşire,
- funcția δ reprezintă funcția stărilor următoare, definită ca $\delta : I \times S \times O \rightarrow S$, unde fiecare combinație de intrări, stări prezente şi ieşiri este mapată către o stare următoare unică,
- mulțimea L reprezintă relația de ieşire definită ca o funcție caracteristică $L : I \times S \times O \rightarrow B$, unde fiecare combinație de intrări şi stări prezente este legată de una sau mai multe ieşiri,
- mulțimea $R \subseteq S$ reprezintă starea de reset.

Cu alte cuvinte, pentru un automat finit pseudo nedeterminist $T(i,p,n,o) = 1$ dacă şi numai dacă $L(i,s,o) = 1$ şi $n = \delta(i,s,o)$. Întrucît starea următoare n este unică pentru o combinație dată de intrare, stare prezentă şi ieşire, aceasta poate fi dată de o funcție de stare următoare $n = \delta(i,p,o)$. Ieşirea este reprezentată de relația L, întrucît mărimea de ieşire este în general nedeterministică.

Definiție: Un automat finit pseudo nedeterminist de pas k este definit ca o 6-tupla $M = (S, I, O, \delta, L, R)$, unde:

- mulțimea S reprezintă spațiul finit de stare,
- mulțimea I reprezintă spațiul finit de intrare,
- mulțimea O reprezintă spațiul finit de ieşire,
- mulțimea $R \subseteq S$ reprezintă starea de reset,
- funcția stărilor următoare este definită ca $\delta : I \times \ldots \times S \times O \times \ldots \times O \rightarrow S$, unde fiecare combinație a stării prezente, k intrări şi k iesiri produc o stare următoare unică,
- mulțimea L reprezintă relația de ieşire definită ca o funcție caracteristică $L : I \times S \times O \rightarrow B$, unde fiecare combinație de intrări şi stări prezente este legată de una sau mai multe ieşiri.

Definiție: Un automat finit nedeterminist de tip Mealy este definit ca o 6-tuplă $M = (S, I, O, D, L, R)$, unde:

- mulțimea S reprezintă spațiul finit de stare,
- mulțimea I reprezintă spațiul finit de intrare,
- mulțimea O reprezintă spațiul finit de ieşire,
- mulțimea $R \subseteq S$ reprezintă starea de reset,
- mulțimea D reprezintă relația stărilor următoare, definită ca o funcție caracteristică $D : I \times S \times S \rightarrow B$, unde fiecare combinație dintre intrări şi starea curentă este legată de un set nevid de stări următoare,
- mulțimea L reprezintă relația de ieşire definită ca o funcție caracteristică $L : I \times S \times O \rightarrow B$, unde fiecare combinație dintre intrări şi starea prezentă este legată de un set nevid de ieşiri.

În un automat finit de tip Mealy, stările următoare şi ieşirile nu sunt corelate în relația de tranziție a stărilor.

În un automat finit nedeterministic de tip Moore există o relație a stărilor următoare de forma $D : I \times S \times S \rightarrow B$ şi o relație de ieşire de forma $L : S \times O \rightarrow B$ astfel încît pentru toate $(i,n,p,o) \in I \times S \times S \times O$, $T(i,n,p,o) = 1$ dacă şi numai dacă $D(i,p,n) = 1$ şi $L(p,o) = 1$.

Definiție: Un automat finit nedeterminist de tip Moore este definit ca o 6-tuplă $M = (S, I, O, D, L, R)$, unde:

- mulțimea S reprezintă spațiul finit de stare,
- mulțimea I reprezintă spațiul finit de intrare,
- mulțimea O reprezintă spațiul finit de ieşire,
- mulțimea $R \subseteq S$ reprezintă starea de reset,
- mulțimea D reprezintă relația stărilor următoare, definită ca o funcție caracteristică $D : I \times S \times S \rightarrow B$, unde fiecare combinație dintre intrări şi starea curentă este legată de un set nevid de stări următoare,
- mulțimea L reprezintă relația de ieşire definită ca o funcție caracteristică $L : S \times O \rightarrow B$, unde fiecare stare prezentă este legată de un set nevid de ieşiri.

Observatii: Automatele de tip Moore sunt un caz special al maşinilor Mealy, în care funcția de ieşire depinde doar de starea prezentă, şi nu de intrări. Automatele de tip Mealy pot fi convertite în automate echivalente de tip Moore. Cu toate acestea există situații de automate de tip Mealy care nu au automate echivalente de tip Moore.[58] Un exemplu în acest sens ar putea fi un automat de tip Mealy cu o stare şi cu intrarea conectată direct la ieşire.

Un automat finit nedeterminist este un automat finit incomplet specificat dacă pentru fiecare pereche $(i,p) \in I \times S$ astfel încît $T(i,p,n,o) = 1$, automatul poate tranzita într-o stare următoare n unică sau în orice stare următoare, şi automatul poate produce o ieşire unică o sau orice ieşire.

Definiție: Un automat finit incomplet specificat este definit ca o 6-tuplă $M = (S, I, O, D, L, R)$, unde:

- mulțimea S reprezintă spațiul finit de stare,
- mulțimea I reprezintă spațiul finit de intrare,
- mulțimea O reprezintă spațiul finit de ieşire,
- mulțimea $R \subseteq S$ reprezintă starea de reset,
- mulțimea D reprezintă relația stărilor următoare, definită ca o funcție caracteristică $D : I \times S \times S \rightarrow B$, unde fiecare combinație dintre intrări şi starea curentă este legată de o singură stare următoare sau toate stările,
- mulțimea L reprezintă relația de ieşire definită ca o funcție caracteristică $L : I \times S \times O \rightarrow B$, unde fiecare combinație dintre intrări şi starea prezentă este legată de o singură ieşire sau de toate ieşirile.

Un automat finit determinist, sau complet specificat, este un automat finit nedeterminist unde pentru fiecare pereche $(i,p) \in I \times S$ există o stare următoare n unică şi o ieşire o unică astfel încît $T(i,p,n,o) = 1$, adică spunem că există o tranziție unică de la (i,p). În plus, mulțimea R conține o stare de reset unică.

Definiție: Un automat finit determinist sau complet specificat este definit ca o 6-tuplă $M = (S, I, O, \delta, \lambda, r)$, unde:

- mulțimea S reprezintă spațiul finit de stare,
- mulțimea I reprezintă spațiul finit de intrare,
- mulțimea O reprezintă spațiul finit de iesire,
- mulțimea $r \in S$ reprezintă starea de reset, unică,
- mulțimea δ reprezintă funcția stărilor următoare definite ca $\delta : I \times S \rightarrow S,$ unde $n \in S$ este starea următoare a stării prezente $p \in S$ pentru intrarea $i \in I$ dacă şi numai dacă $n = \delta(i,p)$,
- mulțimea λ reprezintă funcția de ieşire şi este definită ca $\lambda : I \times S \rightarrow O,$ unde $o \in O$ este ieşirea stării prezente $p \in S$ pentru intrarea $i \in I$ dacă şi numai dacă $o = \lambda(i,p)$.

Observație: În cazul unui automat finit incomplet specificat precum şi în cazul unui automat finit determinist, fiecare dintre ele prezintă ieşirile stării următoare necorelate deoarece starea următoare şi informația de ieşire se pot reprezenta separat. În cazul unui automat finit pseudo determinist, starea următoare şi ieşirea sunt corelate întrucît starea următoare este corelată cu ieşirea prin $n = \delta(i,p,o)$.

Un automat Moore este un automat finit nedeterminist Moore în care pentru fiecare pereche $(i,p) \in I \times S$ există o stare următoare unică n şi pentru fiecare $p \in S$ există o ieşire unică o astfel încît $T(i,p,n,o) = 1$. În plus, mulțimea R conține o stare de reset unică.

Definiție: Un automat finit determinist de tip Moore este definit ca o 6-tuplă $M = (S, I, O, \delta, \lambda, r),$ unde:
- mulțimea S reprezintă spațiul finit de stare,
- mulțimea I reprezintă spațiul finit de intrare,
- mulțimea O reprezintă spațiul finit de ieşire,
- mulțimea $r \in S$ reprezintă starea de reset,
- mulțimea δ reprezintă funcția stărilor următoare definite ca $\delta : I \times S \rightarrow S,$ unde $n \in S$ este starea următoare a stării prezente $p \in S$ pentru intrarea $i \in I$ dacă şi numai dacă $n = \delta(i,p)$,
- mulțimea λ reprezintă funcția de ieşire şi este definită ca $\lambda : S \rightarrow O,$ unde $o \in O$ este ieşirea stării prezente $p \in S$ dacă şi numai dacă $o = \lambda(p)$.

Definiție: Pentru un automat finit nedeterminist, o stare s este deterministă la ieşire dacă pentru orice intrare i, există o ieşire unică o astfel încît starea s are la ieşire o pentru intrarea i. Un automat finit nedeterminist este determinist la ieşire dacă fiecare stare accesibilă este deterministă la ieşire.

Definitie: Un automat finit nedeterminist este fals nedeterminist dacă pentru fiecare secvență de intrare, există o secvență de ieşire unică. Dacă un automat finit nedeterminist este fals nedeterminist, atunci toate stările sale accesibile sunt deterministe la ieşire.

Definiție: Un automat finit determinist sau simplu, automat finit, este definit ca o 5-tuplă $A = (S, s, \delta, r, F),$ unde:
- mulțimea S reprezintă spațiul finit de stare,
- mulțimea s reprezintă un alfabet finit,
- mulțimea $r \in S$ reprezintă starea de reset,
- mulțimea $F \subseteq S$ reprezintă setul stărilor finale,

- mulțimea δ reprezintă funcția stărilor următoare definite ca $\delta : S \, x \, S \rightarrow S$, unde $n \in S$ este starea următoare a stării prezente $p \in S$ pentru simbolul $i \in S$ dacă şi numai dacă $n = \delta(i,p)$,

Spunem că un şir x este acceptat de către un automat finit A dacă $\delta(x,r)$ este o stare din F. Limbajul acceptat de către A, notat cu $\alpha(A)$, este setul de şiruri $\{ x \mid \delta(x,r) \in F\}$.

Spunem că un şir x este acceptat de către un automat finit nedeterminist D dacă există o secvență de tranziții corespunzătoare lui x astfel încît $D(x,r)$ conține o stare în F. Limbajul acceptat de A, notat cu $\alpha(A)$, este setul de şiruri $\{ x \mid D(x,r) \cap F \neq \emptyset \}$.

Teoremă: Fie α un limbaj acceptat de un automat finit nedeterminist. Atunci există un automat finit determinist care acceptă limbajul α.

Definiție: Un *Automat* se numeşte *finit* dacă sunt finite toate cele trei mulțimi ale sale **Z**, **X**, **Y**.

Pot fi definite şi alte tipuri de finitudine ale automatelor, cum ar fi de exemplu automatele (**X**,**Y**)-*finite*, în care sunt finite mulțimea semnalelor de intrare şi mulțimea semnalelor de ieşire, iar mulțimea stărilor poate fi arbitrară. Definiția introdusă mai sus este mai generală decît cea introdusă de Mealy prin faptul că mulțimile **Z**, **X**, **Y** pot fi luate arbitrare, şi nu în mod obligatoriu mulțimi finite.

Fie $\mathbf{X} = \{x_1 , x_2 , \dots , x_n\}$ şi $\mathbf{Y} = \{y_1 , y_2 , \dots , y_m\}$ mulțimea semnalelor de intrare şi respectiv a semnalelor de ieşire ale automatului $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$. Se convine notarea elementelor mulțimii **X** litere de intrare, iar elementele multimii **Y** litere de ieşire. De asemenea, multimea **X** se numeşte alfabetul de intrare, iar mulțimea **Y** alfabetul de ieşire. Elementele mulțimii $\mathbf{Z} = \{z_1 , z_2 , \dots , z_l\}$ sunt literele de stare, iar **Z** este alfabetul stărilor.

Automatului $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ i se ataşeaza trei semigrupuri libere $\Im(\mathbf{X})$, $\Im(\mathbf{Y})$ şi $\Im(\mathbf{Z})$ generate de mulțimile **X**, **Y** si **Z** ale semnalelor de intrare, semnalelor de ieşire şi stărilor sale. Elementele semigrupului liber $\Im(\mathbf{X})$, numit semigrup de intrare, vor fi şiruri finite de stări din **Z**, denumite cuvinte de stare ale automatului.

Cu aceste definiții se face o extindere naturală a domeniului de definiție al funcției de tranziție $\delta(\mathbf{z},\mathbf{x})$ şi al funcției de ieşire $\lambda = \lambda(\mathbf{z},\mathbf{x})$ ale automatului, înlocuind mulțimea $\mathbf{Z} \, x \, \mathbf{X}$ prin mulțimea $\Im(\mathbf{X})$. Fie $p = x_1x_2 \dots x_k$ un cuvînt de intrare arbitrar cuprins în $\Im(\mathbf{X})$ şi o stare arbitrară $z \in \mathbf{Z}$. Se defineşte $\delta(z,p) = z_1z_2 \dots z_k$, unde $z_1 = \delta(z , x_1)$, $z_2 = \delta(z_1 , x_2)$, … , $z_k = \delta(z_{k-1} , x_k)$. De asemenea se defineşte $\lambda(z,p) = y_1y_2 \dots y_k$, unde $y_1 = \lambda(z , x_1)$, $y_2 = \lambda(z_1 , x_2)$, … , $y_k = \lambda (zk\text{-}_1 , x_k)$

Definiție: Automatul $B(\mathbf{Z}_1, \mathbf{X}_1, \mathbf{Y}_1, \delta_1, \lambda_1)$ se numeşte *subautomat* al automatului $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$, dacă $\mathbf{Z_1} \subseteq \mathbf{Z}$, $\mathbf{X_1} \subseteq \mathbf{X}$, $\mathbf{Y_1} \subseteq \mathbf{Y}$ iar funcțiile δ_1 şi λ_1 coincid respectiv cu funcțiile δ şi λ pe submulțimea $\mathbf{Z}_1 \, x \, \mathbf{X}_1$ a lui $\mathbf{Z} \, x \, \mathbf{X}$. O categorie interesantă de subautomate este constituită de subautomatele pentru care $\mathbf{Z_1} \subset \mathbf{Z}$, $\mathbf{X_1} = \mathbf{X}$ şi $\mathbf{Y_1} = \mathbf{Y}$.

4.6 Reprezentările automatelor

Automatele au trei reprezentări utilizate mai des în literatura de specialitate. Prima este reprezentarea electrică, prin porți logice, cea de a doua este reprezentarea logică, prin grafuri, care va fi detaliată în continuare în acest subcapitol, iar cea de a treia este metoda tabelară. Se convine asocierea unui graf unui automat după urmatoarele reguli:

a) mulțimii stărilor automatului $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ i se asociaza nodurile grafului $\wp$

b) dacă starile z_1 şi z_2 sunt legate prin relația $z_2 = \delta(z_1 , x)$, atunci nodul z_1 este legat cu nodul z_2 printr-un arc orientat de la z_1 spre z_2. Arcul (z_1 , z_2) astfel obținut este marcat cu x.

Este posibil să existe mai multe semnale de intrare $x_1, x_2, \ldots, x_p$ pentru care $z_2 = \delta(z_1, x_i)$, $i \in \{1,2,\ldots,p\}$. În loc de a se trasa p arce de la z_1 la z_2 se va duce un singur arc care va fi marcat cu fiecare din literele $x_1, x_2, \ldots, x_p$. Dacă $y = \lambda(z_1, x)$ atunci arcul (z_1 , z_2) va fi marcat şi cu litera y şi în general dacă $x_1, x_2, \ldots, x_p$ verifică relația $y_i = \lambda(z_1, x_i)$, $i \in \{1,2,\ldots,p\}$, atunci arcul (z_1, z_2) va fi marcat cu fiecare din literele $y_1, y_2, \ldots , y_p$.

Exemplu: Fie automatul cu două semnale de intrare x_1 şi x_2, două semnale de ieşire y_1 şi y_2 şi patru stări z_1, z_2, z_3 şi z_4, pentru care funcțiile de tranziție şi de ieşire sunt date de relațiile:

$\delta(z_1 , x_1) = z_2$ $\quad$ $\delta(z_1 , x_2) = z_3$ $\quad$ $\lambda(z_1 , x_1) = y_1$ $\quad$ $\lambda(z_1 , x_2) = y_2$

$\delta(z_2 , x_1) = z_1$ $\quad$ $\delta(z_2 , x_2) = z_3$ $\quad$ $\lambda(z_2 , x_1) = y_1$ $\quad$ $\lambda(z_2 , x_2) = y_1$

$\delta(z_3 , x_1) = z_4$ $\quad$ $\delta(z_3 , x_2) = z_1$ $\quad$ $\lambda(z_3 , x_1) = y_2$ $\quad$ $\lambda(z_3 , x_2) = y_2$

$\delta(z_4 , x_1) = z_2$ $\quad$ $\delta(z_4 , x_2) = z_4$ $\quad$ $\lambda(z_4 , x_1) = y_2$ $\quad$ $\lambda(z_4 , x_2) = y_1$

se reprezintă dupa metoda descrisă pe graful din Fig 3.

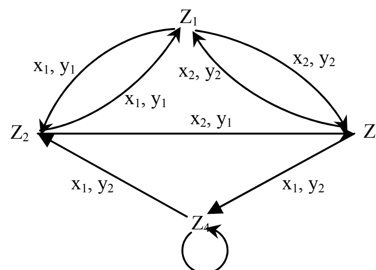


Figura 3. Reprezentarea unui automat prin graf de tranziție

A treia şi cea mai des utilizată metodă de reprezentare a automatelor este metoda tabelară. Se convine asocierea unui automat $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ a unui tabel $\mathbf{T}(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ construit astfel:

a) liniile tabelului corespund stărilor automatului;

b) coloanele tabelului corespund semnalelor de intrare;

c) dacă z_i este una din stările automatului şi x_j unul dintre semnalele de intrare ale acestuia, atunci în tabelul $\mathbf{T}(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$, la intersecția liniei i cu coloana j se vor scrie starea $\delta(z_i , x_j)$ şi semnalul de ieşire dat de $\lambda(z_i , x_j)$.

Automatul din exemplul anterior se reprezintă prin metoda tabelară descrisă în Tabelul 1, detaliat mai jos:

Stare \ Intrare	x_1	x_2
z_1	z_2 y_1	z_3 y_2
z_2	z_1 y_1	z_3 y_1
z_3	z_4 y_2	z_1 y_2
z_4	z_2 y_2	z_4 y_1

Tabelul 1. Descrierea automatului

4.7 Homomorfismul automatelor

Se presupune că funcțiile $\delta(z, x)$ si $\lambda(z, x)$ sunt definite pentru mulțimea tuturor perechilor $(z, x) \in \mathbf{Z} x \mathbf{X}$.

Definiție: **1**. Automatele pentru care funcțiile δ şi λ sunt definite pentru orice pereche (z, x) $(z, x) \in \mathbf{Z} x \mathbf{X}$ se numesc *automate complet definite* sau *automate complete*. Dimpotrivă, automatele pentru care există cel puțin o pereche $(z, x) \in \mathbf{Z} x \mathbf{X}$ pentru care funcția δ sau funcția λ nu este definită se numesc *automate incomplet definite* sau *automate incomplete*.

Definiție: **2**. Un **homomorfism** ψ al automatului $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ în automatul $A(\mathbf{Z_1}, \mathbf{X_1}, \mathbf{Y_1}, \delta_1, \lambda_1)$ este dat de mulțimea de trei aplicații ψ_1: $\mathbf{Z} \rightarrow \mathbf{Z}_1$, ψ_2: $\mathbf{X} \rightarrow \mathbf{X}_1$ şi ψ_3: $\mathbf{Y} \rightarrow \mathbf{Y}_1$ care satisfac pentru orice pereche $(z, x) \in \mathbf{Z} x \mathbf{X}$, relațiile:

$$\psi_1(\delta(z, x)) = \delta_1(\psi_1(z), \psi_2(x)),$$
$$\psi_3(\lambda(z, x)) = \lambda_1(\psi_1(z), \psi_2(x)).$$

Un homomorfism biunivoc a două automate $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ şi $B(\mathbf{Z}_1, \mathbf{X}_1, \mathbf{Y}_1, \delta_1, \lambda_1)$ va fi un **izomorfism** al acestor automate. Pot fi întîlnite diferite tipuri de homomorfisme în funcție de cele trei aplicații care intervin la definiția 2. Dacă corespondența celor două automate este o funcție injectivă atunci avem un **monomorfism**, dacă este surjectivă atunci este un **epimorfism**. Dacă domeniul este egal cu codomeniul atunci este un **endomorfism**. Dacă are proprietatea de izomorfism şi endomorfism atunci este un **automorfism**. Dacă are proprietatea de monomorfism şi epimorfism atunci este un **bimorfism**.

De exemplu, pentru automatele la care coincid mulțimea semnalelor de intrare $\mathbf{X} = \mathbf{X}_1$ şi mulțimea semnalelor de ieşire $\mathbf{Y} = \mathbf{Y}_1$, ca aplicații ψ_2 şi ψ_3 pot fi luate aplicațiile identice, astfel că relațiile de omomorfism se reduc la următoarele relații:

$$\psi_1(\delta(z, x)) = \delta_1(\psi_1(z), x),$$
$$\lambda(z, x) = \lambda_1(\psi_1(z), x), \quad (z, x) \in \mathbf{Z} x \mathbf{X}.$$

În acest caz homomorfismul automatelor $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ şi $B(\mathbf{Z}_1, \mathbf{X}_1, \mathbf{Y}_1, \delta_1, \lambda_1)$ se reduce la o singură aplicație ψ_1: $\mathbf{Z} \rightarrow \mathbf{Z}_1$. Un astfel de homomorfism este un homomorfism după stare. La fel se pot defini homomorfismele după intrări, ieşiri sau combinații ale celor trei mulțimi care intervin în definiția automatelor.

Exemplu: Fie automatele din tabelele 2 şi 3 de mai jos. Se observă că aplicaţia ψ care realizează corespondenţa de mai jos a stărilor celor două automate:

$\psi(z_1) = t_1$,	$\psi(z_4) = t_4$,	$\psi(z_7) = t_7$,	$\psi(z_{10}) = t_8$,
$\psi(z_2) = t_2$,	$\psi(z_5) = t_5$,	$\psi(z_8) = t_3$,	$\psi(z_{11}) = t_6$,
$\psi(z_3) = t_3$,	$\psi(z_6) = t_6$,	$\psi(z_9) = t_7$,	$\psi(z_{12}) = t_1$,

este un homomorfism după stare al automatului din tabelul 2 în automatul din tabelul 3, întrucît verifică relaţiile de homomorfism. Funcţiile δ_1 şi λ_1 ale automatului din tabelul 3 sunt date de relaţiile:

$$\delta_1(t , x) = \psi(\delta(z , x)) \ , \qquad \lambda(t , x) = \lambda(z , x).$$

unde $t = \psi(z)$, δ şi λ fiind funcţiile corespunzătoare ale automatului din Tabelul 3.

	x_1	x_2	x_3	x_4
z_1	Z_1 , y_1	z_2 , y_3	z_6 , y_4	z_3 , y_2
z_2	Z_4 , y_1	z_2 , y_3	z_5 , y_4	z_3 , y_2
z_3	Z_1 , y_1	z_7 , y_3	z_6 , y_4	z_3 , y_2
z_4	Z_4 , y_1	z_7 , y_3	z_5 , y_4	z_8 , y_2
z_5	Z_1 , y_1	z_9 , y_3	z_5 , y_4	z_3 , y_2
z_6	z_{12} , y_1	z_2 , y_3	z_6 , y_4	z_{10} , y_2
z_7	Z_1 , y_1	z_7 , y_3	z_{11} , y_4	z_3 , y_2
z_8	z_{12} , y_1	z_9 , y_3	z_{11} , y_4	z_8 , y_2
z_9	z_{12} , y_1	z_9 , y_3	z_6 , y_4	z_8 , y_2
z_{10}	Z_1 , y_1	z_7 , y_3	z_6 , y_4	z_{10} , y_2
z_{11}	Z_1 , y_1	z_2 , y_3	Z_{11} , y_4	z_{10} , y_2
z_{12}	z_{12} , y_1	z_2 , y_3	z_{11} , y_4	z_8 , y_2

Tabelul 2. Funcţiile automatului *A*

	x_1	x_2	x_3	x_4
t_1	t_1 , y_1	t_2 , y_3	t_6 , y_4	t_3 , y_2
t_2	t_4 , y_1	t_2 , y_3	t_5 , y_4	t_3 , y_2
t_3	t_1 , y_1	t_7 , y_3	t_6 , y_4	t_3 , y_2
t_4	t_4 , y_1	t_7 , y_3	t_5 , y_4	t_3 , y_2
t_5	t_1 , y_1	t_7 , y_3	t_5 , y_4	t_3 , y_2
t_6	t_1 , y_1	t_2 , y_3	t_6 , y_4	t_8 , y_3
t_7	t_1 , y_1	t_7 , y_3	t_6 , y_4	t_3 , y_2
t_8	t_1 , y_1	t_7 , y_3	t_6 , y_4	t_8 , y_3

Tabelul 3. Funcţiile automatului *B*

4.8 Echivalența automatelor

Definiție: Se numeşte *aplicație* φ_z, indusă de starea $z \in \mathbf{Z}$ a automatului $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ o aplicație a semigrupului său de intrare în semigrupul de ieşire, care pune în corespondență unui cuvint arbitrar $p = x_1 x_2 \ldots x_k$ de intrare cuvîntul de ieşire: $\varphi_z(p) = \lambda(z , p)$.

Se numeşte *familie de aplicații* induse de automatul A mulțimea tuturor aplicațiilor φ_z diferite două cîte două, $z \in \mathbf{Z}$.

Definiție: Două sau mai multe stări ale unui automat sau ale mai multor automate se numesc *echivalente* dacă induc aceleaşi *aplicații*.

Definiție: Două automate $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ şi $B(\mathbf{Z}_1, \mathbf{X}_1, \mathbf{Y}_1, \delta_1, \lambda_1)$ care au aceleaşi alfabete de intrare şi de ieşire se numesc *echivalente* dacă induc aceleaşi *familii de aplicații*.

Teoremă: Dacă există un homomorfism ψ al automatului $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ pe automatul $B(\mathbf{Z}_1, \mathbf{X}_1, \mathbf{Y}_1, \delta_1, \lambda_1)$, atunci automatele A şi B sunt echivalente. Pentru orice stare $z \in \mathbf{Z}$, stările $z \in \mathbf{Z}$, $\psi(z) \in \mathbf{Z}_1$ sunt echivalente.

Exemplu: Se verifică uşor că automatele din tabele 4 şi 5 sunt echivalente deoarece $\lambda(z , x) = \lambda_1(\psi(z) , x) = \lambda_1(t , x)$ pentru $\surd x \in \mathbf{X}$, $z \in \mathbf{Z}$, de unde rezultă că orice cuvînt de intrare $p = u_1 u_2 \ldots u_k$ va genera în cele două automate acelaşi cuvînt de ieşire:

$q = v_1 v_2 \ldots v_k \lambda = (s_1 , u_1)\ \lambda(s_2 , u_2) \ldots \lambda(s_k , u_k) = \lambda_1(\psi(s_1), u_1)\ \lambda_1(\psi(s_2), u_2) \ldots \lambda_1(\psi(s_k), u_k)$,
$s = s_1 s_2 s_3 \ldots s_k$, unde $s_2 = \delta(s_1 , u_1)$, $s_3 = \delta(s_2 , u_2)$, ... , $s_k = \delta(s_{k-1} , u_{k-1})$, s_1 fiind dat odată cu $p = u_1 u_2 \ldots u_k$. În particular, pentru: $p = x_1 x_1 x_2 x_3 x_4 x_1$, $s_1 = z_6$ se obțin cuvintele de ieşire şi de stare următoare: $q = y_1 y_1 y_3 y_4 y_2 y_1$, $s = z_6 z_{12} z_{12} z_2 z_5 z_3$.

Aplicînd homomorfismul ψ, se obține $t = \psi(s) = \psi(z_6)\ \psi(z_{12})\ \psi(z_{12})\ \psi(z_2)\ \psi(z_5)\ \psi(z_3)$ sau $t = t_6 t_1 t_1 t_2 t_5 t_3$, la care corespunde cuvîntul de ieşire $\lambda(t , p) = \lambda(t_6 , x_1)\ \lambda(t_1 , x_1)\ \lambda(t_1 , x_2)\ \lambda(t_2 , x_3)\ \lambda(t_5 , x_4)\ \lambda(t_3 , x_1) = y_1 y_1 y_3 y_4 y_2 y_1$, adică tot q.

4.9 Automate Moore

Definiție: Un automat $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ se numeşte *Automat Moore* dacă funcția de ieşire $\lambda(z , x)$ poate fi reprezentată sub forma $\lambda(z , x) = \mu(\delta(z , x))$, $(z , x) \in \mathbf{Z} \, x \, \mathbf{X}$, unde $\delta(z , x)$ este funcția de tranziție a automatului, iar $\mu(z)$ este o funcție care aplică multimea $\mathbf{Z}$ a stărilor automatului A în mulțimea semnalelor de ieşire. Funcția $\mu(z)$ se numeşte *funcția de marcaj* iar valoarea $\mu(z)$ a funcției în starea z se numeşte *marcajul acestei stări*.

Deşi constituie un caz particular de automate Mealy, automatele Moore au caracter destul de general şi se bucură de proprietatea cu totul remacabilă dată de teorema următoare:

Teoremă: Pentru orice automat Mealy $A(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \delta, \lambda)$ există un automat Moore B, echivalent cu A, iar în cazul cînd A este finit, automatul B poate fi ales şi el finit cu numărul de stări egal cu $(m+1)n$, unde m este numărul semnalelor de intrare, iar n este numărul stărilor în automatul A.

Exemplu: Automatele din tabelele 4 şi 5 sunt automate de tip Moore. Într-adevar, se verifică imediat că pentru automatul din tabelul 4 există funcţia μ(z) definită cu relaţiile:

$\mu(z_1) = y_1$, $\mu(z_2) = y_3$, $\mu(z_3) = y_2$, $\mu(z_4) = y_1$, $\mu(z_5) = y_4$, $\mu(z_6) = y_4$,
$\mu(z_7) = y_3$, $\mu(z_{10}) = y_3$, $\mu(z_{11}) = y_4$, $\mu(z_{12}) = y_1$, $\mu(z_8) = y_2$, $\mu(z_9) = y_3$.

Pentru automatul din tabelul 5 există funcţia $\mu(t)$ definită de relaţiile:
$\mu(t_1) = y_1$, $\mu(t_2) = y_3$, $\mu(t_3) = y_2$, $\mu(t_4) = y_1$, $\mu(t_5) = y_4$, $\mu(t_6) = y_4$,
$\mu(t_7) = y_3$, $\mu(t_8) = y_3$

4.10 Reprezentarea automatelor incomplet specificate

Prin definiţie automatele pentru care există cel puţin o pereche $(z, x) \in \mathbf{Z}\ x\ \mathbf{X}$ pentru care funcţia $\lambda = \lambda(z, x)$ nu este definită se numeşte automat incomplet definit sau simplu automat incomplet. Pentru studiul automatelor incomplete nu se mai folosesc noţiunile de homomorfism şi echivalentă introduse în studiul automatelor complete. În locul noţiunii de echivalenţă se utilizează în cazul automatelor incomplete noţiunea de compatibilitate, care este mai slabă decît prima, din cauza lipsei axiomei tranzitivităţii. O consecinţă imediată ar fi ca împărţirea în clase de compatibilitate a mulţimii stărilor automatului nu va conduce la partiţii ale mulţimii **Z**, ci la submulţimi oarecare, fiind posibil ca două astfel de submulţimi să conţină aceeaşi stare. După cum se poate vedea, acest lucru nu constituie un inconvenient în simplificarea automatelor şi nici în stabilirea echivalenţei automatelor incomplete.

Se consideră un automat arbitrar cu n stări specificat în tabelul 4, unde pentru prescurtarea notaţiilor s-a notat $\delta_{ij} = \delta(z_i, x_j)$ şi $\lambda_{ij} = \lambda(z_i, x_j)$ pentru $\forall i \in \{1,2, ... n\}$ si $\forall j \in \{1,2, ... r\}$.

	X^1	...	X^j	...	x^r
z_1	$\delta_{11}, \lambda_{11}$	...	$\Delta_{1j}, \lambda_{1j}$	...	$\delta_{1r}, \lambda_{1r}$
...	...	...	...	...	...
z_i	$\delta_{i1}, \lambda_{i1}$	...	$\delta_{ij}, \lambda_{ij}$	...	$\delta_{ir}, \lambda_{ir}$
...	...	...	...	...	...
z_k	$\delta_{k1}, \lambda_{k1}$	...	$\delta_{kj}, \lambda_{kj}$	...	$\delta_{kr}, \lambda_{kr}$
...	...	...	...	...	...
z_n	$\delta_{n1}, \lambda_{n1}$	...	$\delta_{nj}, \lambda_{nj}$	...	$\delta_{rn}, \lambda_{rn}$

Tabelul 4. Descrierea automatului incomplet specificat

Fie $\mathbf{Z}_{ij} \subset \mathbf{Z}\ x\ \mathbf{Z}$ mulţimea de perechi de stări din **Z** definită astfel:

$\mathbf{Z}_{ij} = \{(\delta_{ik}, \delta_{jk}), k \in \mathbf{K}_{ij}$, unde $\mathbf{K}_{ij} \subset \{1,2, ... , r\}$, este mulţimea de indici pentru care sunt definite atît starea δ_{ik} cît şi starea δ_{jk}, cu i şi j fiind daţi.

Definiţie: Submulţimea $\mathbf{Z}_{ij}$ este denumită *submulţimea de perechi de stări asociate* perechii (z_i, z_j) sau , simplu, *multimea asociată perechii* (z_i, z_j).

Definiţie: Două stări z_i şi z_j sunt *compatibile* dacă submulţimea asociată $\mathbf{Z}_{ij}$ îndeplineşte oricare dintre condiţiile:

a) este mulțimea vidă;
b) conține perechea (z_i , z_j) şi/sau perechi (z_k , z_k), k$\in$\{1, 2, ... , n\}, n fiind nr. de stări;
c) conține perechi de stări compatibile, iar ieşirile îndeplinesc oricare dintre condițiile:
i. $\lambda_{ik} = \lambda_{jk}$, k$\in$\{1, 2, ... , r\};
ii. oricare dintre ieşiri este nedefinită;
iii. ambele ieşiri sunt nedefinite.

Două stări z_i şi z_j pentru care mulțimile $\mathbf{Z}_{ij}$ asociate îndeplinesc punctele a) şi b) referitoare la perechile de stări şi punctele i. , ii. , iii. Referitoare la ieşiri se vor numi *stări direct compatibile*.

Definiție: Două stări z_i şi z_j sunt *incompatibile* dacă verifică oarecare dintre condițiile:

a) $\exists$ k$\in$\{1, 2, ... , r\} astfel încît $\lambda_{ik} = \lambda_{jk}$, cu λ_{ik} , λ_{jk} ambele definite;
b) $\exists$ 2 stări z_r şi z_s incompatibile, $(z_r , z_s) \in \mathbf{Z}_{ij}$.

Stările care îndeplinesc condiția a) se numesc *direct compatibile*.

Definiție: Două stări z_i şi z_j sunt *nedeterminate* dacă ieşirile îndeplinesc condițiile de compatibilitate dar există cel puțin o pereche $(z_r , z_s)\in Z_{ij}$, $(z_r , z_j) \neq (z_i , z_j)$, $r \neq s$.

Stările z_i şi z_j nedeterminate pot fi compatibile sau incompatibile, dar nu se poate decide acest lucru apriori.

Lema: Relația de compatibilitate este *reflexivă* şi *simetrică*.

4.11 Operații cu automate

Definiție: Fie I_0 şi O_0 două mulțimi nevide, $A_j=(I_j, O_j, S_j, \tau_j, \omega_j)$, j=1,2 două automate deterministe şi aplicațiile $a_1:I_0xO_2 \rightarrow I_1$, $a_2:I_0xO_1 \rightarrow I_2$ şi $a_3:O_1xO_2 \rightarrow O_0$. Compunerea automatelelor A_1 şi A_2 prin aplicațiile a_1, a_2 şi a_3 este automatul $A_{2x(a1,a2,a3)}A_1=(I_0, S_2xS_1, O_0, \delta_{(a1,a2,a3)}, \lambda_{(a1,a2,a3)})$, unde:

$$\delta_{(a1,a2,a3)}((s_2,s_1),i_0)=(\delta_2(s_2,a_2(i_0,o_1)),\ \delta_1(s_1,a_1(i_0,o_2)))$$
$$\lambda_{(a1,a2,a3)}((s_2,s_1),i_0)=a_3(\lambda_2(s_2,a_2(i_0,o_1)),\ \lambda_1(s_1,a_1(i_0,o_2)))$$

Legarea celor 2 automate este realizată schematic în Fig 4:

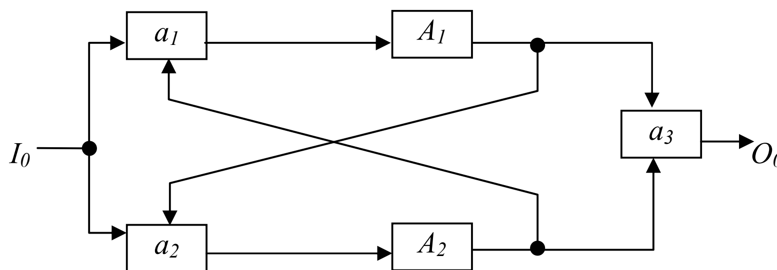


Figura 4. *Gruparea a două automate prin aplicațiile a_i (i=1,2,3)*

În cazul cel mai general perechile de mulţimi (I_0, I_1), (I_0, I_2), (I_0, O_1), (I_0, O_2), (S_1, O_1), (S_2, O_2), (O_0, O_1), (O_0, O_2) pot să nu fie disjuncte. În urma acestei compuneri se poate considera că cele două automate schimbă informaţii între ele transmiţînd unul altuia informaţii atît despre starea în care se află cît şi despre ieşirile pe care le emit. Anumite informaţii pot fi trimise direct la ieşirea automatului rezultant, inclusiv o serie de semnale de intrare. Există următoarele situaţii:

1. $S_i \cap O_i \neq \phi$ corespunde situaţiei în care o serie de semnale de ieşire ale automatului A_i corespund unor variabile de stare ale acestuia. În acest fel este disponibilă explicit informaţia privind starea automatului
2. $S_i \cap O_i = \phi$. nu este disponibilă explicit informaţia privind starea automatului A_i. Starea automatului este cunoscută implicit prin valorile semnalelor de ieşire modulo o relaţie de echivalenţă
3. $I_0 \cap O_0 \neq \phi$. înseamnă că o serie de semnale de intrare sunt disponibile şi la ieşire

Particularizînd cele 3 aplicaţii a_1, a_2 şi a_3 putem obţine cîteva din operaţiile întîlnite în literatură. Pentru definirea operaţiilor de grupare în cascadă precum şi operaţiile particulare de grupare în serie şi paralel trebuiesc următoarele definiţii:

***Definiţie**:* Fie I_0 o mulţime nevidă, $A_j=(I_j, O_j, S_j, \delta_j, \lambda_j)$, j=1,2 două automate deterministe şi aplicaţiile $a_1:I_0 \rightarrow I_1$, $a_2:I_0 x O_1 \rightarrow I_2$. *Cascadarea automatelelor* A_1 *si* A_2 prin aplicaţiile a_1 şi a_2 este automatul $A_{2x(a1,a2)}A_1=(I_0, S_2xS_1, O_2xO_1, \delta_{(a1,a2)}, \lambda_{(a1,a2)})$, unde:

$$\delta_{(a1,a2)}((s_2,s_1),i_0)=(\delta_2(s_2,a_2(i_0,\lambda_1(s_1,a_1(i_0)))),\ \delta_1(s_1,a_1(i_0)))$$
$$\lambda_{(a1,a2)}((s_2,s_1),i_0)=(\delta_2(s_2,a_2(i_0,\lambda_1(s_1,a_1(i_0)))),\ \lambda_1(s_1,a_1(i_0)))$$

Legarea în cascadă a celor 2 automate este redată în Fig 5:

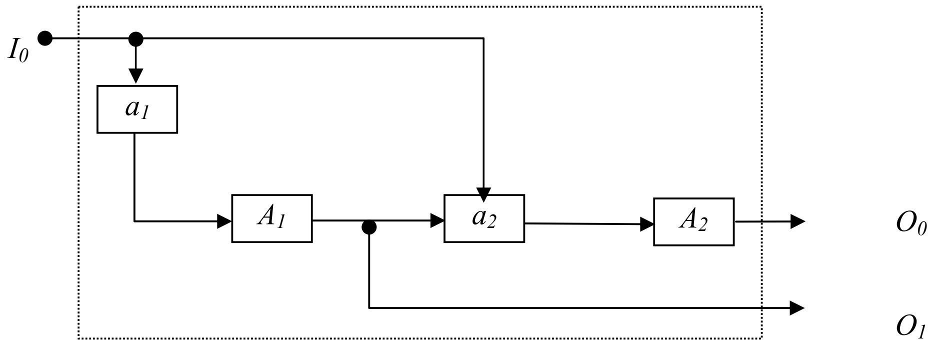


Figura 5. *Gruparea în cascadă*

***Definiţie**:* Gruparea în serie în sens Hartmanis-Stearns a 2 automate se obţine ca o particularizare a grupării în cascadă pentru cazul în care $I_1 = I_0$ şi $a_2(i_0,o_1)= o_1$, adică $O_1=I_2$. Automatul obţinut se notează cu $A_1 \rightarrow A_2=(I_1, S_2xS_1, O_2, \delta_S, \lambda_S)$, unde:

$$\delta_S((s_2,s_1),i_0)=(\delta_2(s_2,\lambda_1(s_1,i_1)),\ \delta_1(s_1,i_1))$$
$$\lambda_S((s_2,s_1),i_0)=(\delta_2(s_2,a_2(i_0,\delta_1(s_1,a_1(i_0)))),\ \lambda_1(s_1,a_1(i_0)))$$

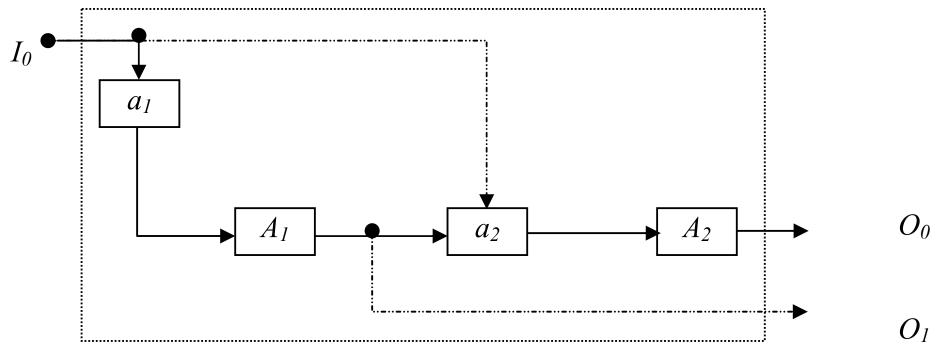


Figura 6 *Gruparea în Serie*

***Definiție**:* Gruparea în paralel în sens Hartmanis-Stearns a 2 automate se obține ca o particularizare a grupării în cascadă pentru cazul în care $I_1 = I_0$ şi $I_2 = I_0$. Automatul obținut se notează cu $A_1||A_2=(I_1, S_2xS_1, O_2xO_1, \delta_P, \lambda_P)$, unde:

$$\delta_P((s_2,s_1),i_0)=(\delta_2(s_2,i_0),\ \delta_1(s_1,i_0))$$
$$\lambda_P((s_2,s_1),i_0)=(\lambda_2(s_2,i_0),\ \lambda_1(s_1,i_0))$$

Gruparea în paralel a celor 2 automate este redată în Fig 7:

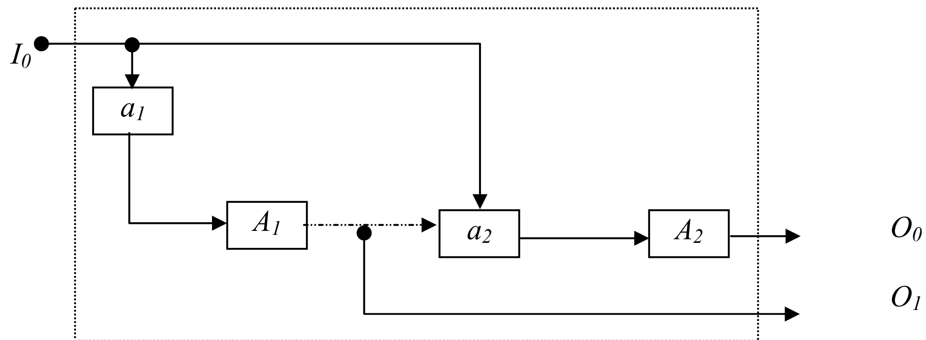


***Figura** 7 Gruparea în Paralel*

4.12 Operații cu automate utilizînd metode implicite

4.12.1 Gruparea în paralel utilizînd metode implicite

Presupunem că automatele sunt date prin relațiile de tranziție-ieşire după cum urmează:

$T_1(i, s^1, S^1, o^1) = 1$
$T_2(i, s^2, S^2, o^2) = 1$

Fiind date stările inițiale ale automatelelor, r^1 *si* r^2, relația de tranziție a automatului ce reprezintă gruparea în paralel a celor 2 automate $T_{||}$ poate fi obținută după următorul algoritm:

$T_{||}^0 = \phi$
$\pi^0 = (r^1, r^2)$
execută {
$T_{||}^{k+1}(is^1s^2S^1S^2o^1o^2) = T_{||}^k(is^1s^2S^1S^2o^1o^2) + T_1(is^1S^1o^1)\ T_2(is^2S^2o^2)\ \pi^k(s^1s^2)$
$\pi^{k+1}(S^1S^2) = \exists\ is^1s^2o^1o^2T_{||}^k(is^1s^2S^1S^2o^1o^2)$
} cît timp $T_{||}^{k+1} \neq T_{||}^k$
$T_{||}^k(is^{||}S^{||}o^{||}) = \exists\ s^1s^2S^1S^2[T_{||}^k(is^1s^2S^1S^2o^1o^2)\ assign(s^1s^2, s^{||})\ assign(S^1S^2, S^{||})]$
unde $assign(s^1s^2, s^{||})$ este relația prin care se asignează fiecărei perechi distincte de stări (s_1s_2) în mod unic o stare unică $s^{||}$ în automatul $A^{||}$ iar $o^{||}=o^1o^2$.

4.12.2 Gruparea în serie utilizînd metode implicite

Presupunem că automatele sunt date prin relațiile de tranziție-ieşire după cum urmează:

$T_1(i^1, s^1, S^1, o^1) = 1$
$T_2(i^2, s^2, S^2, o^2) = 1$

şi care satisfac condiția $I_2 = O_1$. Din acestea se pot obține relațiile de tranziție şi cele de ieşire după cum urmează:

$\delta_1(i^1,s^1,S^1) = \exists\ o^1\ T_1(i^1,s^1,S^1,o^1)$
$\delta_2(i^2,s^2,S^2) = \exists\ o^2\ T_2(i^2,s^2,S^2,o^2)$
$\lambda_1(i^1,s^1,o^1) = \exists\ S^1\ T_1(i^1,s^1,S^1,o^1)$
$\lambda_2(i^2,s^2,o^2) = \exists\ S^2\ T_2(i^2,s^2,S^2,o^2)$

Fiind date stările inițiale ale automatelelor, r^1 şi r^2, relația de tranziție a automatului ce reprezintă gruparea în serie T_S a celor 2 automate poate fi obținută după următorul algoritm:

$T_S^0 = \phi$
$\pi^0 = (r^1, r^2)$
execută {
$\omega(i^1s^1s^2o^2) = \exists\ o^1[\lambda_2(o^1,s^2,o^2)\ \lambda_1(i^1,s^1,o^1)]$

$$\sigma(i^1s^1s^2S^1S^2) = \exists\, o^1[\delta_2(o^1,s^2,S^2)\ \delta_1(i^1,s^1,S^1)]$$

$$T_S^{k+1}(i^1s^1s^2S^1S^2o^2) = T_S^{k}(i^1s^1s^2S^1S^2o^2) + \sigma(i^1s^1s^2S^1S^2)\ \omega(i^1s^1s^2o^2)\ \pi^k(s^1s^2)$$

$$\pi^{k+1}(S^1S^2) = \exists\, i^1s^1s^2o^1o^2 T_S^{k}(i^1s^1s^2S^1S^2o^1o^2)$$

} cît timp $T_S^{k+1} \neq T_S^{k}$

$$T_S^{k}(is^SS^So^S) = \exists\, s^1s^2S^1S^2[T_S^{k}(i^1s^1s^2S^1S^2o^2)\ \mathrm{assign}(s^1s^2, s^S)\ \mathrm{assign}(S^1S^2, S^S)]$$

unde assign(s^1s^2, s^S) este relația prin care se asigneaza fiecărei perechi distincte de stări (s_1s_2) în mod unic o stare unică s^S în automatul A^S.

4.12.3 Descompunerea în serie şi în paralel a automatelor finite

Definiție: O *partiție* π definită pe spațiul stărilor *S* este o colecție de mulțimi de stări disjuncte din *S*. Mulțimile disjuncte se denumesc *grupuri de blocuri* aparținînd partiției π. Partiția π este *partiție închisă* dacă şi numai dacă pentru oricare 2 stări s_i şi s_j care aparțin aceluiaşi bloc al partiției π, atunci pentru orice intrare $i \in I$ stările următoare $\sigma(s_i)$ şi $\sigma(s_j)$ aparțin aceluiaşi bloc din π. O partiție este generală dacă nu este închisă.

Definiție: Produsul a 2 partiții, π_p , conține cîte un bloc în π_p pentru fiecare pereche de blocuri din cele 2 partiții, unde elementele blocului sunt intersecția elementelor din perechea de blocuri. Dacă intersecția este nulă atunci blocul nu este prezent în π_p. ***Partiția zero***, $\pi(0)$, este partiția în care fiecare bloc conține o singură stare. Se poate demonstra că un automat ***M*** poate fi descompus într-un set de *n* subautomate care realizează aceeaşi funcție dacă şi numai dacă există un set de partiții nebanale şi ortogonale, astfel încît $\pi_1\pi_2 \ldots \pi_n = \pi(0)$.

Exemplu: Pentru un *S* avînd 4 stări, $S=\{s_1, s_2, s_3, s_4\}$, şi 2 partiții $\pi_1 = \{(s_1, s_2), (s_3,s_4)\}$ si $\pi_2 = \{(s_1, s_3), (s_2,s_4)\}$ din *S*, atunci $(s_1, s_2), (s_3, s_4)$ sunt blocuri din partiția π_1 iar produsul lor este partiția $\pi_p = \pi_1\pi_2 = \{s_1, s_2, s_3, s_4\}$

Definiție: O ***relație de congruență*** pe mulțimea stărilor unui automat este o relație de echivalență $\sigma \in Eq(S)$ avînd proprietatea:

$$\forall (s_1,s_2) \in \sigma \Rightarrow (f(s_1,i), f(s_2,i) \in \sigma,\ \forall i \in I.$$

Partiția indusă de relația de congruență pe mulțimea stărilor unui automat este denumită ***partiție închisă*** sau partiție avînd proprietatea de distribuție.

Teoremă: Produsul $\pi_1\pi_2$ şi suma $\pi_1+\pi_2$ a două partiții avînd proprietatea de substituție pe mulțimea stărilor unui automat este tot o partiție avînd proprietatea de substituție.

Din această teoremă rezultă că pentru fiecare pereche de partiții închise există cel mai mare minorant comun $\pi_1\pi_2$, şi cel mai mic majorant comun $\pi_1+\pi_2$. Rezultă de asemenea că mulțimea partițiilor închise pe mulțimea stărilor unui automat este închisă față de operațiile de adunare şi înmulțire deci are o structură de latice. Teorema următoare indică o metodă prin care pentru un automat finit se pot determina toate partițiile avînd proprietatea de substituție.

Teoremă: Fie $\theta_{s1,s2}$ cea mai mica congruență pe A avînd proprietatea $(s_1s_2) \in \theta_{s1,s2}$ şi θ o congruență. Atunci:

$$\theta = \forall\{\ \theta_{s',s''} | (s',s'') \in \theta\}$$

Pe baza acestei relații se dă un procedeu de determinare a partițiilor avînd proprietatea de substituție care constă în determinarea congruențelor $\theta_{s',s''}$ pentru toate perechile (s',s'') $\in$SxS. Procedeul este descris în continuare.

Se notează cu $I=\{i_1,i_2,...,i_m\}$ mulțimea combinațiilor de intrare ale automatului considerat. Fie blocul determinat de perechea de stări (s',s'') şi anume B_0={s',s''}. Pentru fiecare intrare i_j se calculează $S^{ij}_{01}= f(B_0,i_j),\ j=1,..,m$. Se reunesc într-un singur bloc toate perechile de mulțimi oarecare $S^{ij'}_{01}$, $S^{ij''}_{01}$ care nu sunt disjuncte şi se obțin blocurile $B_1^1,...,B_1^{k1}$. Se calculează mulțimile $S_{h2}^{ij}=f(B_1^h,i_j),\ h=1,...,k,\ si\ j=1,...,m$ care sunt reunite din nou în blocuri noi în cazul în care nu sunt disjuncte, $B_2^1,...,B_2^{k2}$. Procesul continuă pînă cînd nu se mai obține nici un bloc nou. Oprirea acestui algoritm este garantată avînd în vedere că numărul de stări ale automatului este finit. Se notează cu $B_1,...,B_\alpha$ blocurile obținute în finalul paşilor descrişi anterior şi se notează $B_{\alpha+1} = S \setminus B_j,\ j=1,...,\alpha$. Dacă $B_{j1} \cap B_{j2} \neq \phi$, se vor reuni blocurile B_{j1} şi B_{j2}. Blocurile $B'_1,...,B'_\beta$ obținute din blocurile $B_1,...,B_\alpha B_{\alpha+1}$ prin eventualele reuniuni constituie mulțimea $S/\theta_{s',s''}$.

O alta metodă de determinare a partițiilor avînd proprietatea de substituție este determinarea inițial a celor mai mici partiții care conțin stările s' şi s'' pentru toate perechile de stări (s',s'') numite şi partiții de bază. Restul de partiții se determină efectuînd cu partițiile de bază toate sumele posibile. Practic, după determinarea partițiilor de bază se calculează suma tuturor perechilor de partiții de nivel 2 pentru a se obține partițiile de nivel 3, şi asa mai departe pîna cînd nu se mai obțin partiții noi.

***Teoremă**:* Pentru un automat $A=(I,\ O,\ S,\ \delta,\ \lambda)$ putem obține o descompunere în cascadă formată din 2 automate $A_i=(I_i,\ O_i,\ S_i,\ \delta_i,\ \lambda_i)$, $|S_j|<|S|$ dacă şi numai dacă există o congruență nebanală pentru A.

Cele 2 automate componente sunt $M_1=(I,S/\sigma,S/\sigma xI,\delta_1,\lambda_1)$, numit şi automatul predecesor şi $M_2=(I,S/\sigma xI,S/\tau,O,\delta_2,\lambda_2)$, numit şi automatul succesor, unde:

$$\delta_1([s]_\sigma,i) = [f(s,i)]_\sigma$$
$$\lambda_1([s]_\sigma,i) = ([s]_\sigma,i)$$
$$\delta_2([s_1]_\sigma,([s]_\sigma,i)) = [s_2]_\tau$$
$$\delta_2 = \delta([s_1]_\sigma \cap [s]_\sigma,\mathrm{i})$$
$$\lambda_2([s_1]_\sigma,([s]_\sigma,i)) = \lambda([s_1]_\sigma \cap [s]_\sigma,\mathrm{i})$$

unde τ este o partiție astfel încît $\sigma x \tau = I_S$. Partiția τ nu trebuie să aibă proprietatea de substituție. Funcționarea celor două componente ale legării în serie poate fi explicată după cum urmează: calculul stării următoare este impărțit între cele 2 automate. Primul automat determină clasa S/σ din care face parte starea următoare, automatului succesor revenindu-i sarcina de a determina din acestă clasa exact care este această stare.

***Teoremă**:* Pentru un automat $A=(I,\ O,\ S,\ \delta,\ \lambda)$ putem obține o descompunere în paralel formată din 2 automate $A_i=(I_i,\ O_i,\ S_i,\ \delta_i,\ \lambda_i)$, $|S_j|<|S|$ dacă şi numai dacă există două congruențe netriviale σ_1 şi σ_2 pe mulțimea stărilor automatului A astfel încît produsul lor să fie egal cu I_S.

Pentru cele 2 automate $M_j=(I,S/\sigma_j,S/\sigma_j xI,\sigma_j,\lambda_j),\ j=1,2$ sunt definite relațiile:

$$\delta_j([s]_{\sigma j},i) = [f(s,i)_{\sigma j}$$
$$\lambda_j([s]_{\sigma j},i) = ([s]_{\sigma j},i)$$

4.12.4 Descompunerea prin factorizare a automatelelor finite

Este o formă mult mai puternică de descompunere în care ambele componente ale automatului descompus interacționează între ele. Aceasta presupune identificarea subrutinelor sau a factorilor din automatul inițial şi extragerea şi reprezentarea lor ca un automat separat. Descompunerea în cascadă are o acoperire limitată în proiectarea automatelor finite moderne, cum ar fi specificațiile pentru controlere centralizate în microprocesoare, care nu au o bună descompunere în cascadă. Fiind dată o descriere a tabelei de fluență, factorii, care identifică seturile de stări şi de arce de tranziție, sunt extraşi şi reprezentați ca un subautomat factorizat în care instanțele lor din automatul original sunt înlocuite prin apeluri către subautomatul factorizat. Factorizarea exactă presupune reducerea la maxim a numărului de stări precum şi a arcelor de tranziție din automatul inițial. Problema asignării optimale a stărilor implică găsirea atît a unei codificări binare a stărilor interne dintr-un automat finit cît şi producerea unei implementări logice care să ocupe o arie cît mai mică după codificare şi optimizarea logică. Există o relație strinsă între descompunerea şi asignarea stărilor unei maşini secvențiale.[95] Au fost propuse numeroare tehnici, folosind implementări de tip binivel sau sumă de produse[73], şi de recent, implementări logice multinivel. În timp ce tehnologiile care folosesc implementarea binivel încearcă să minimizeze numărul de termeni produs într-o eventuală implementare, cele multinivel încearcă să minimizeze variabilele din implementarea finală. *Hartmanis şi Stearns* au formulat teoria perechilor de partiții şi au descris descompunerea în cascadă şi algoritmii de asignare a stărilor bazați pe teoria lor. *Karp* a introdus noțiunea de perechi de partiții critice şi *Kohavi* a abordat problema partițiilor suprapuse şi a împărțirii stărilor pentru descompunerea în cascadă şi asignarea stărilor pentru maşinile secvențiale. *DeMicheli* a propus utilizarea minimizării multinivel pentru a găsi o codificare optimală a intrărilor şi a stărilor prezente obținînd rezultate îmbunătățite decît cele precedente. O insuficiență a acestei strategii este faptul că ignoră complet spațiul stării următoare a automatului finit, ceea ce rezultă uneori în asignări de stări neoptimale. Dacă un automat finit, factorizat înainte de a fi codificat şi optimizat,produce un rezultat mai bun decît codificarea şi optimizarea fără descompunere, atunci însemnă că algoritmul utilizat a furnizat un rezultat neoptimal. O decompunere optimală a automatului poate fi definită ca una care reprezintă o codificare care minimizează numărul de termeni produs din maşina originală. Premiza este de a obține o decompunere generală optimală multiplă a automatului, în sensul de a obține mai mult de 2 subautomate.[34]

Pentru un automat care poate fi factorizat nebanal, rezultatele experimentale demonstrează că descompunerea urmată de asignarea stărilor produce rezultate superioare pentru maşini secvențiale mari. Factorizarea inițială rezultă în explorarea efectivă a relației dintre spațiul intrărilor şi cel al ieşirilor în subautomatele descompuse pentru ambele tipuri de implemetări: binivel şi multinivel. Factorii exacți pot să nu existe într-un anume automat, sau pot fi prea mici pentru a putea produce o descompunere eficientă. Factorii inexacți, dar buni, pot produce realizări economice descompuse şi pot fi de asemenea folosiți pentru a ghida algoritmii euristici de asignare a stărilor. [31]

4.12.5 Optimizarea structurilor logice secvențiale complexe cu pachetul *fsmtool*

Dezvoltarea fără precedent din ultimii ani a sistemelor de proiectare asistată a circuitelor digitale a fost determinată de cerința de a realiza sisteme de complexitate din ce în ce mai mare într-un timp din ce în ce mai scurt. Avansul tehnologic a permis utilizarea unor pachete de programe care permit abordarea unor circuite de complexitate din ce în ce mai ridicată, care îndeplinesc funcții complexe. În acelaşi timp ele asigură că acestea se încadrează în anumite limite impuse dimensiunii, vitezei de lucru, suprafeței ocupate, puterii disipate, etc. Deşi la început programele care permiteau proiectarea circuitelor *VLSI* erau folosite doar în cadrul marilor firme de profil, în ultimii ani ele sunt disponibile şi în universități, la costuri reduse, putind fi folosite atît pentru proiectare cît şi pentru scopuri didactice [16].

O serie de programe sunt disponibile în rețeaua universitară;[33] la acestea se adaugă pachetul de programe ***fsmtool*** *(Finite State Machines tool)* proiectat de către autor, şi care urmează a fi folosit în continuare. Este prezentat de asemenea şi modul în care acest program poate fi folosit pentru optimizarea automatelelor finite. Pachetul ***fsmtool*** conține funcțiile şi procedurile prezentate anterior care permit realizarea diverselor operații asupra automatelelor finite. Programul *Stamina* este un program care permite minimizarea numărului de stări ale unui automat folosind anumite tehnici de minimizare. Acest program realizează minmizarea exactă a stărilor automatului utilizînd metodele de acoperire bivalentă. El ofera de asemenea solutii euristice de minimizare a stărilor automatului bazate pe calculul unor anumite limite superioare şi pe baza izomorfismului dintre stări.[86] Unul din programele mai performante în domeniul codificărilor automatelelor finite este programul *Nova*,[114] care realizează asignarea optimală a codurilor pentru stările unui automat prin minimizarea numărului de termeni produs care implementeaza automatul finit codificat pentru un anumit numar de biți utilizați la codificare. Codificarea se realizează în 2 etape. Prima etapă determină constrîngerile la care vor fi supuse codurile prin realizarea unei minimizări simbolice.[68] Cea de a doua etapă, de codificare, are ca rezultat obținerea codurilor stărilor automatului care satisfac toate sau majoritatea constrîngerilor determinate în prima etapă. Utilizatorul are posibilitatea de a selecta algoritmul care să fie aplicat şi de asemenea, numărul de biți utilizați pentru codificarea stărilor.[36] Toate programele de mai sus, primesc la intrare automatul specificat în formatul *Kiss*. Acest format este utilizat în sistemele de proiectare asistată pentru descrierea automatelelor finite şi este în corespondență directă cu graful de fluență. Pachetul ***fsmtool*** a fost realizat ca o interfață care permite utilizarea programelor prezentate astfel încît să se poată realiza operații precum: citirea şi prelucrarea grafului de tranziție a unui automat, salvarea lui în format *kiss* sau verilog, minimizarea stărilor folosind programul *Stamina*, codificarea stărilor automatului folosind programul *Nova*,[115] optimizarea automatelor prin descompunere folosind programul ***fsmtool***, în serie, paralel, sau generalizat, vizualizarea descrierilor implicite cu ajutorul *ROBDD* folosind programul „*dotty*” (*Bell Labs.*)[65], conversia automată a descrierii unui automat finit din format *kiss* în format *Verilog* recunoscut de componenta ***kiss2vl*** a pachetului de programe ***fsmtool***. De asemenea componenta ***genfsm*** a pachetului fsmtool este utilizată pentru generarea de automate finite deterministe aleatoare cu un anumit număr de intrări, stări interne şi ieşiri, specificate ca parametri de intrare la execuția programului. Automatele obținute pot constitui un bogat set de exemple usor de utilizat ca un set de test pentru a afla funcționalitatea programului ***fsmtool*** în descompunerea şi optimizarea setului de automate finite generate cu ***genfsm***.

Capitolul 5

Metode de reprezentare a circuitelelor digitale complexe

5.1 Metode de reprezentare a mulțimilor de obiecte

Proiectarea structurilor logice secvențiale presupune reprezentarea şi explorarea unui spațiu mare al soluțiilor. Aceasta implică reprezentări şi manipulări ale unor mulțimi de obiecte ale căror dimensiuni cresc de cele mai multe ori exponențial odată cu creşterea complexității sistemului proiectat. Găsirea unor reprezentări optime din punct de vedere al dimensiunilor ocupate şi din punct de vedere al simplității şi vitezei algoritmilor de manipulare a acestor reprezentări, constituie o temă de mare interes în domeniu.

***Definiție**:* Spunem că o reprezentare este *explicită* atunci cînd obiectele sunt reprezentate intern în memorie ca o listă de obiecte a cărei dimensiune este proportională cu numărul de obiecte considerat. Spunem că obiectele sunt *manipulate explicit* atunci cînd ele sunt prelucrate unul cîte unul.

***Definiție**:* Spunem că o reprezentare este *implicită* atunci cînd obiectele sunt reprezentate folosindu-se anumite proprietăți comune ale acestora, ceea ce face ca dimensiunile reprezentării să nu mai fie proporționale cu numărul de obiecte. Spunem că aceste obiecte sunt *manipulate implicit* atunci cînd într-un singur pas sunt manipulate mai multe obiecte odată.

Ideea constă în reprezentarea şi manipularea mulțimilor de mari dimensiuni prin intermediul funcției caracteristice a acestora. La rîndul lor funcțiile caracteristice pot fi reprezentate eficient folosind arborii de decizie binară a căror dimensiune nu depinde de numărul de elemente ale mulțimii. Astfel, în cazurile de interes practic, operațiile asupra mulțimilor sunt transpuse în operații asupra arborilor de decizie binară, în realitate operații booleene asupra funcțiilor booleene. Folosind tehnicile implicite, calcule care erau imposibile folosind reprezentări explicite pot fi abordate eficient şi într-o perioadă de timp rezonabilă. Există totuşi situații în care nici aceste reprezentări implicite nu pot fi construite, dimensiunile lor depăşind dimensiunea reprezentării. Acest lucru se datorează fie structurii intrinseci a funcției, fie faptului că nu se găseşte o ordine convenabilă a variabilelor. [85]

5.2 Reprezentarea cubică pozițională a mulțimilor de obiecte

Este utilizată pentru reprezentarea unei mulțimi de obiecte. Pentru o mulțime cu n obiecte există 2^n submulțimi de obiecte. Oricare submulțime de obiecte poate fi reprezentată folosind o variabilă booleană n-dimensională $x=(x_1x_2...x_n)$ fiecare variabilă x_i corespunzînd obiectului s_i. Prezența unui obiect s_k în mulțimea considerată este indicată prin faptul că variabila x_k are valoarea 1. Cazul $x_k = 0$ corespunde situației în care obiectul s_k nu aparține mulțimii considerate. [74]

Exemplu 1*:* Fie n=5 obiectul s_3 poate fi reprezentat prin vectorul boolean *(00100)*. Submulțimea $M_1=\{s_1, s_3, s_5\}$ poate fi reprezentată prin vectorul boolean *(10101)*. Mulțimea $M_2=\{\{s_1, s_3, s_5\}, \{s_3, s_4\}, \{s_1, s_2, s_4, s_5\}\}$. Această mulțime de submulțimi poate fi reprezentată prin următoarea mulțime de vectori booleeni *{{10101}, {00110}, {11011}}*.

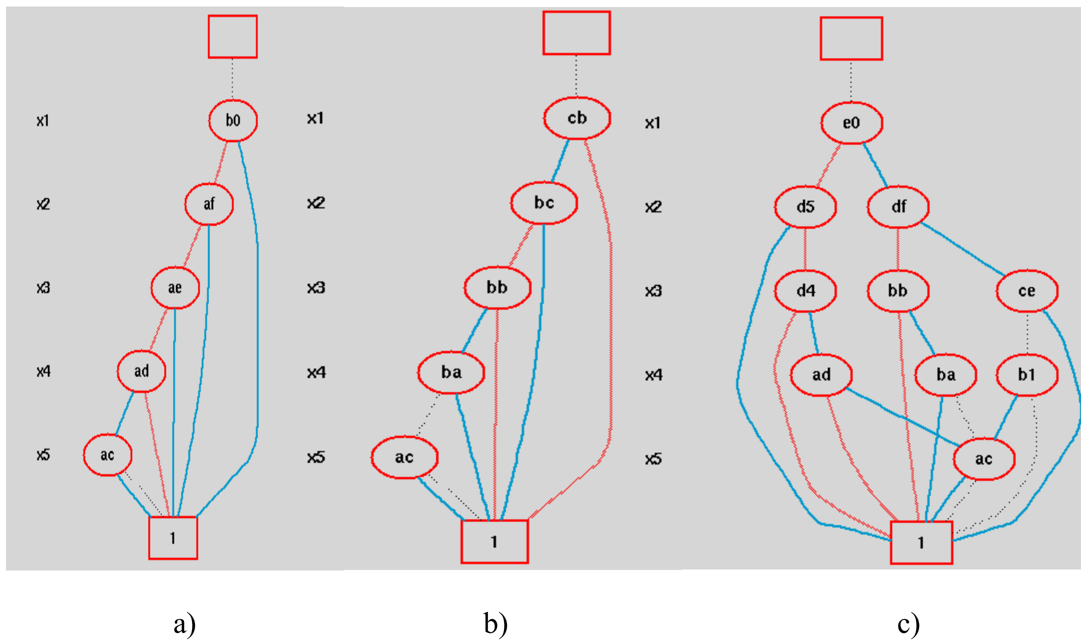
Această mulțime de vectori booleeni poate fi reprezentată cu ajutorul funcției caracteristice a acestei mulțimi. Se definește $f_s:B^n \rightarrow B$, unde $f_s(x)=1$ dacă și numai dacă mulțimea de obiecte reprezentată în notație cubică pozițională prin x este în mulțimea S.

Exemplu 2*:* Pentru mulțimile din exemplul următor funcțiile caracteristice sunt:

a) $f_{s3}(x)=f_{s3}(x_1,...x_5)=\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4x_5$

b) $f_{M1}(x)=f_{M1}(x_1,...x_5)=\bar{x}_1x_2\bar{x}_3x_4x_5$

c) $f_{M2}(x)=f_{M2}(x_1,...x_5)=\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4x_5 + \bar{x}_1x_2\bar{x}_3x_4x_5 + x_1\bar{x}_2x_3x_4x_5$



a) b) c)

Figura 8. *Reprezentările ROBDD ale funcțiilor caracteristice ale celor 3 funcții*

Funcția caracteristică astfel definită poate fi reprezentată cu ajutorul arborilor de decizie binară. Fiecare minterm al acestei reprezentări va corespunde unei mulțimi de obiecte. În cazul în care mulțimea S este mulțimea stărilor unui automat, se poate folosi și reprezentarea cubică pozițională pentru a reprezenta stări, mulțimi de stări și mulțimi de mulțimi de stări.

5.3 Reprezentări implicite ale automatelelor finite

Se ştie că automatele finite pot fi reprezentate printr-o formă binivel unde tranziţiile sunt precizate explicit una cîte una. În cazul automatelelor finite de mari dimensiuni numărul acestor tranziţii şi dimensiunile termenilor produs care reprezintă aceste funcţii pot fi foarte mari. De asemenea, în cadrul algoritmilor de sinteză a automatelelor finite obiectele folosite: mulţimi de stări, mulţimi de mulţimi de stări, relaţii de echivalenţă, relaţii de compatibilitate, toate acestea pot deveni foarte mari şi imposibil de reprezentat în memorie şi de manipulat în timp optim dacă acestea sunt reprezentate explicit. Pentru a depăşi aceste inconveniente, se caută ca aceste mulţimi să se reprezinte implicit şi manipularea să se facă folosind tehnici care să utilizeze cît mai eficient proprietăţile acestor reprezentări.[47] Reprezentarea implicită a unui automat finit se poate obţine plecînd de la următoarele definiţii:

***Definiţie**:* Se numeşte *automat incomplet definit* o cvintuplă $A=(I,\ O,\ S,\ \tau,\ \omega)$, unde I este mulţimea semnalelor de intrare, O este mulţimea semnalelor de ieşire iar S este mulţimea stărilor. τ este *relaţia de tranziţie* definită ca funcţia caracteristică $\tau{:}IxSxS\rightarrow B$ unde fiecărei combinaţii intrare-ieşire i se pune în corespondenţă o stare sau toate stările. ω este *relaţia de ieşire* definită ca funcţia caracteristică $\omega{:}IxSxO\rightarrow B$ unde fiecărei combinaţii intrare-ieşire i se pune în corespondenţa o ieşire sau toate ieşirile.

***Definiţie**:* Se numeşte *automat finit* o cvintuplă $A=(I,\ O,\ S,\ \tau,\ \omega)$, unde I este mulţimea semnalelor de intrare, O este mulţimea semnalelor de ieşire iar S este mulţimea stărilor. τ este *relaţia de tranziţie* definită ca funcţia caracteristică $\tau{:}IxSxS\rightarrow B$ unde $\tau(i,p,n)=I$ dacă si numai dacă $n = \delta(p,i)$, iar ω este *relaţia de ieşire* definită ca funcţia caracteristică $\omega{:}IxSxO\rightarrow B$ unde $\omega(i,p,o)=I$ dacă şi numai dacă $o = \lambda(p,i)$.

În reprezentarea acestor relaţii, i şi o sunt de obicei vectori binari iar p şi n sunt reprezentate folosind notaţia cubică poziţională. Cele 2 relaţii pot fi grupate într-o singură relaţie de tranziţie-ieşire conform definiţiei următoare:

***Definiţie**:* Se numeşte *automat finit* o cvintuplă $A=(I,\ O,\ S,\ \tau,\ \omega)$, unde I este mulţimea semnalelor de intrare, O este mulţimea semnalelor de ieşire iar S este mulţimea stărilor. Relaţia π este *relaţia de tranziţie-ieşire* definită ca funcţia caracteristică $\pi{:}IxSxSxO\rightarrow B$ unde:

$\pi(i,p,n,o)=I$ dacă şi numai dacă n este starea următoare definită pentru starea prezentă p şi intrarea i, iar o este ieşirea definită pentru starea prezentă p şi intrarea i, deci dacă şi numai dacă $n = \delta(p,i)$ şi $o = \lambda(p,i)$.

***Exemplu 3**:* Considerăm automatul definit prin tabela următoare şi a cărui reprezentare în format *KISS* este următoarea:

St	***0***	***1***
St1	St2/1	St1/0
St2	St2/1	St1/0

Fisierul de descriere al automatului a.kiss2:

.i 1
.o 1
.s 2
.p 4
.r s1
0 st1 st2 1
1 st1 st1 0
0 st2 st2 1
1 st2 st1 0
.e

Automatul din acest exemplu are 2 stări, prin urmare reprezentarea în notația cubică pozițională a acestora va necesita 2 variabile s_1, s_2 pentru starea prezentă şi 2 variabile S_1, S_2 pentru starea următoare.

Variabilele de intrare şi de ieşire sunt binare de aceea ele nu vor fi codificate. În acest fel se reduce numărul de variabile utilizate pentru reprezentarea relațiilor ceea ce reduce probabilitatea ca numărul de noduri ale diagramei de decizie binară corespondente să devină prea mare şi astfel mai greu de manipulat. Astfel stările vor fi codificate după cum urmează:

$st_1 \rightarrow s_1\bar{s}_2$ $st_2 \rightarrow \bar{s}_1 s_2$ (pentru stările prezente)
$st_1 \rightarrow S_1\bar{S}_2$ $st_2 \rightarrow \bar{S}_1 S_2$ (pentru stările urmatoare)

Utilizînd codificarea de mai sus, pentru relațiile π, τ si ω vor rezulta următoarele expresii:

$$\pi(i,s_1,s_2,S_1,S_2,o) = \bar{i}s_1\bar{s}_2\bar{S}_1S_2o + i\bar{s}_1s_2\bar{S}_1\bar{S}_2o + \bar{i}\bar{s}_1s_2\bar{S}_1S_2o + \bar{i}s_1s_2\bar{S}_1\bar{S}_2o$$
$$\tau(i,s_1,s_2,S_1,S_2) = \bar{i}s_1\bar{s}_2\bar{S}_1S_2 + i\bar{s}_1s_2\bar{S}_1S_2 + \bar{i}s_1\bar{s}_2S_1S_2 + \bar{i}s_1s_2\bar{S}_1S_2$$
$$\omega(i,s_1,s_2,o) = \bar{i}\bar{s}_1\bar{s}_2o + is_1\bar{s}_2o + \bar{i}s_1s_2o + i\bar{s}_1\bar{s}_2o$$

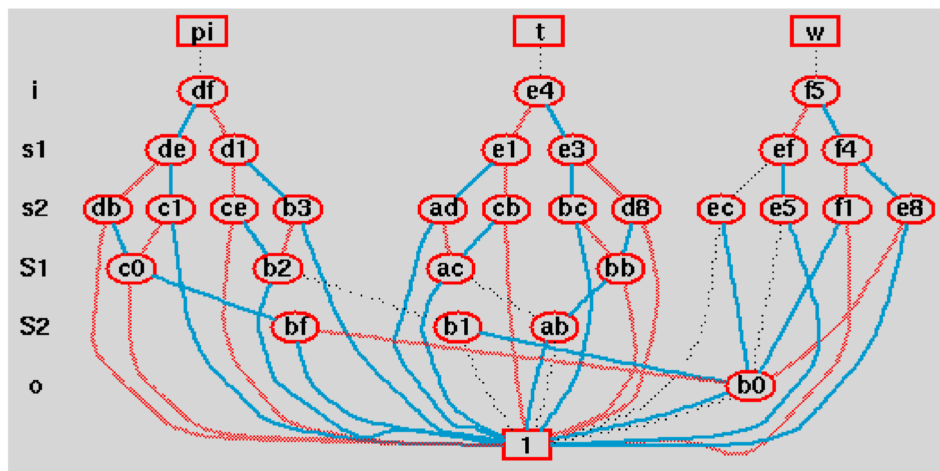


Figura 9. *Reprezentarea implicită a automatului a.kiss2*

Există soluții în care şi intrările şi ieşirile sunt reprezentate folosind notația cubică pozițională. Una dintre aceste situații este cea în care intrările sau ieşirile sunt simbolice. O altă situație este cazul în care intrările şi ieşirile sunt binare dar se doreşte o reprezentare uniformă a variabilelor care intră în aceste relații, mai exact, ca variabile multivalente, pentru aceasta utilizîndu-se reprezentarea folosind notația cubică pozițională. În acest caz automatul poate fi reprezentat tinînd cont că stările, intrările şi ieşirile vor fi codificate după cum urmează:

$st_1 \rightarrow s_1\overline{s_2}$ $st_2 \rightarrow \overline{s_1}s_2$ (pentru stările prezente)
$st_1 \rightarrow S_1\overline{S_2}$ $st_2 \rightarrow \overline{S_1}S_2$ (pentru stările următoare)
$i_0 \rightarrow i_1\overline{i_2}$ $i_1 \rightarrow \overline{i_1}i_2$ (pentru intrări)
$o_0 \rightarrow o_1o_2$ $o_1 \rightarrow o_1o_2$ (pentru ieşiri)

Va rezulta următoarea reprezentare a celor 3 relații:

$$\pi(i,s_1,s_2,S_1,S_2,o) = i_1i_2s_1s_2\overline{S_1}\,\overline{S_2}\,\overline{o_1}\,\overline{o_2} + \overline{i_1}i_2\overline{s_1}s_2\overline{S_1}\,\overline{S_2}o_1o_2 + \overline{i_1}\,\overline{i_2}\,\overline{s_1}s_2S_1\overline{S_2}o_1o_2 + \overline{i_1}\,\overline{i_2}s_1\overline{s_2}S_1\overline{S_2}o_1o_2$$
$$\tau(i,s_1,s_2,S_1,S_2) = i_1\overline{i_2}s_1\overline{s_2}\,\overline{S_1}S_2 + \overline{i_1}i_2\overline{s_1}s_2S_1\overline{S_2} + \overline{i_1}\,\overline{i_2}s_1\overline{s_2}S_1S_2 + \overline{i_1}i_2s_1s_2\overline{S_1}S_2$$
$$\varpi(i,s_1,s_2,o) = i_1\overline{i_2}s_1\overline{s_2}\,\overline{o_1}o_2 + \overline{i_1}i_2\overline{s_1}\,\overline{s_2}o_1o_2 + \overline{i_1}\,\overline{i_2}s_1\overline{s_2}o_1o_2 + \overline{i_1}\,\overline{i_2}s_1s_2\overline{o_1}o_2$$

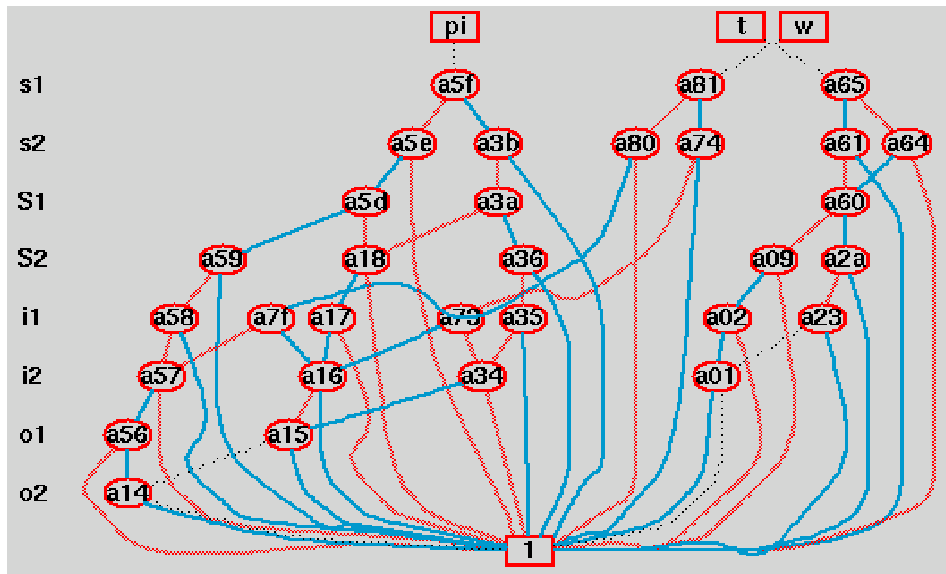


Figura 10. *Reprezentarea implicită a automatului a.kiss2 utilizînd reprezentarea multivalentă pentru variabilele de intrare şi de ieşire*

După cum se observa relațiile sunt definite sub formă de sume de produse. Atunci cînd se construieşte reprezentarea implicită folosind arborii de decizie binară pornind de la o expresie de tip sumă de produse, la fiecare nivel se calculează mai întîi subarborii pentru operanzi dupa care se aplică operatorul corespunzator '+' sau '.'. În funcție de ordinea în care se aplică operatorii, numărul de noduri suplimentare, numite şi noduri nefolosite, care nu vor face parte din arborele binar final corespunzător rezultatului operației, va fi mai mare sau mai mic şi de asemenea şi timpul de construire al arborelui de decizie binar va creşte şi va descreşte odată cu el.

Pentru a obține un număr de noduri nefolosite cît mai mic şi pentru a reduce astfel timpul de construcție al arborelui de decizie şi al relației considerate, arborii binari pentru termenii produs se construiesc sortînd subarborii în ordinea inversă a variabilei lor corespunzătoare rădăcinii. Astfel construcția este una de jos în sus dacă ne referim la arborele binar sau de la dreapta la stînga dacă ne referim la termenul produs construit. Rezultatele obținute pentru diversele metode de construire a relațiilor de tranziție-ieşire sunt concludente în acest sens şi arată că metoda construirii arborelui binar corespunzător, pornind în ordinea inversă celei pe care o au variabilele în scrierea normala a relației, este cea mai avantajoasă şi mai rapidă.

5.4 Implementarea operatorilor impliciți

Operatorii impliciți au fost implementați utilizînd pachetul *CUDD*.[99] Operatorii impliciți au fost concepuți pentru a putea fi utilizați la implementarea unor algoritmi de manipulare a stărilor sau a mulțimilor de stări ale automatelelor finite. În cadrul acestor algoritmi, mulțimile de stări sunt reprezentate prin intermediul mai multor variabile multidimensionale. În plus, în cadrul acestor algoritmi, apare de multe ori necesitatea utilizării unor variabile auxiliare care să conțină rezultatele unor calcule intermediare. Cu cît numărul de astfel de variabile este mai mare, cu atît creşte probabilitatea ca o operație ce manipulează diagrame de decizie binară ce depinde de aceste variabile să aibă ca rezultat o diagramă rezultat care să sufere o explozie exponentială a numărului de noduri. Pentru a micşora această probabilitate, s-a căutat implementarea operatorilor astfel încît ei să foloseasca un număr fix de variabile. Din acest punct de vedere, operatorii implementați pot fi împărțiți în două categorii:

- independenți de setul de variable fixat
- dependenți de setul de variabile fixat

Operatorii din prima categorie prezintă un grad de generalitate mai ridicat ceea ce îi face mai apți de a fi grupați într-o bibliotecă de operatori care să poată fi folosiți şi în cadrul altor domenii decît cel al automatelelor finite.

Operatorii din cea de a doua categorie sunt tributari setului de variabile fixat, ca urmare gradul de generalitate este mai mic şi ei pot fi utilizati numai pentru algoritmi care manipulează stări şi mulțimi de stări ale automatelelor finite. Cei care doresc să utilizeze aceşti operatori trebuie să țină seama de variabilele de care depind aceşti operatori şi rezultatele lor de aceea trebuie să citească cu atenție documentația care descrie operatorii respectivi.

5.5 Tabela de calcul funcțională

Pentru manipularea eficientă a diagramelor de decizie binară se utilizează tabelele în care sunt memorate rezultatele intermediare. Aceste tabele sunt numite *tabele de cache* deoarece ele sunt manipulate şi utilizate ca şi memoriile cache care au o mărime variabilă dar limitată. În aceste tabele fiecare intrare costă din mai multe cîmpuri: operatorul, operanzi şi rezultat. Operatorul poate fi oricare dintre operatorii predefiniți ai limbajului folosit sau poate fi orice funcție definită de utilizator. La fel, tipul operatorilor poate fi de oricare tip predefinit sau poate fi un tip definit de utilizator. Pentru ca memoria cache să poată fi utilizată pentru operanzi şi pentru rezultate de orice tip se preferă ca în locul operanzilor să fie memorați pointeri către operanzi, iar în locul rezultatului, să fie memorat un pointer către rezultat. În acest fel se obține şi o reducere a memoriei alocate cîmpurilor intrărilor în memoria cache. În pachetele de programe concepute pentru manipularea diagramelor de decizie binară, majoritatea operatorilor sunt binari sau ternari. De aceea, tabelele cache au fost realizate pentru memorarea rezultatelor calculate pentru operatori avînd maximum 3 operanzi. Pentru cazul în care operatorii au mai multi operanzi şi în cazul în care ei produc mai multe rezultate, ramine la latitudinea programatorului să aleagă dintre mai multe soluții posibile. Una dintre soluții este de a construi o memorie cache definită de utilizator folosind o bibliotecă de rutine adecvată, fie să combine mai multe rezultate sau mai mulți operanzi într-unul singur folosind structurile specifice. S-a observat de mai multe ori că în cazul unor algoritmi, prin introducerea unor structuri de date suplimentare, s-a reuşit să se mărească viteza de execuție a lor. Avînd în vedere că folosirea structurilor de date suplimentare implică mărirea memoriei folosite, rezultă ca întotdeauna soluția optimă va trebui să realizeze un compromis între viteza de execuție şi necesarul de memorie. Pentru a reduce timpul de calcul al unor algoritmi care folosesc tehnicile implicite de manipulare a funcțiilor booleene se utilizează ***tabela de calcul functională***. Ca şi în cazul tabelei de cache normale, această structură este o tabelă utilizată pentru a memora rezultate parțiale ce sunt susceptibile să apară din nou în etape ulterioare ale calculelor. Diferența față de tabela de calcul normală constă în faptul că valorile calculate sunt memorate în tabela de calcul funcțională ca o structură care include implicit mai mult de un rezultat al aceluiasi operator. În funcție de operanzii cu care este apelat operatorul, dimensiunea rezultatului memorat se poate modifica.

5.6 Implementare cu diagrame de decizie binară

Diagramele de decizie binară au căpătat în ultimii ani o largă utilizare în toate domeniile proiectării structurilor numerice dar şi în alte domenii conexe. Principala caracteristică care le face atît de utilizate este capacitatea acestora de a reprezenta în mod foarte eficient funcțiile booleene. Avînd în vedere că o gama întreaga de problemele privind proiectarea structurilor numerice pot fi exprimate şi rezolvate utilizînd funcțiile booleene, utilizarea diagramelor de decizie binară apare ca iminentă.[37]

Principala problemă care apare la utilizarea diagramelor de decizie binară este limitarea dimensiunilor diagramelor. Se caută să se obțina diagrame care să aiba un numar redus de noduri astfel încît memoria ocupată şi viteza de manipulare să fie cît mai reduse. Există diverse abordări pentru rezolvarea acestei probleme din care cele mai importante sunt: ordinea variabilelor, extinderea acestor tipuri de reprezentări şi utilizarea diverselor metode de memorare a acestor structuri. Avînd în vedere că dimensiunile diagramelor de decizie binară depind de ordinea variabilelor de intrare, o rezolvare a acestei probleme presupune găsirea unei ordini a variabilelor astfel încît, diagrama utilizată pentru reprezentarea unei mulțimi de funcții să aibă un număr de noduri cît mai mic.[44]

O altă problemă importantă privind utilizarea diagramelor de decizie binară este legată de viteza de manipulare a acestora. Rezolvarea acestei probleme presupune găsirea unor algoritmi de manipulare care să aiba viteze cît mai mari permițind astfel prelucrarea într-un timp scurt, a unui numar mare de diagrame sau a unor diagrame cu dimensiuni mari. [80]

Cele 2 probleme sunt antagoniste în sensul că se poate obține pe de o parte reducerea dimensiunilor diagramelor, rezultînd o creştere a complexității algoritmilor şi implicit a vitezei de execuție, iar pe de altă parte se poate obține o creştere a vitezei algoritmilor prin utilizarea unor structuri suplimentare care au ca efect creşterea memoriei utilizate. De obicei se încearcă obținerea unui compromis între cele 2 astfel încît să se obțină utilizarea rezonabilă a memoriei şi o viteză suficient de mare pentru manipularea reprezentărilor.

Datorită caracteristicii pe care le posedă, diagramele de decizie binară au fost utilizate în cele mai diverse domenii, pecum: alocarea resurselor (mapare), analiza temporală, compararea funcțiilor booleene, demonstrarea automată a teoremelor, descompunerea funcțiilor booleene, implementarea tehnicilor implicite de manipulare a funcțiilor booleene, manipularea automatelelor finite, manipularea polinoamelor, optimizarea logică a circuitelor combinaționale, simulare simbolică, sinteza multi-nivel, sinteza şi alocarea resurselor pentru circuitelele *FPGA*, testarea circuitelor numerice şi generarea vectorilor de test, testarea echivalenței automatelelor finite, testarea echivalenței funcțiilor booleene, verificarea circuitelor structurilor numerice combinaționale. [81][111]

5.6.1 Ordinea variabilelor

Una din condițiile pentru ca diagramele de decizie binară ordonate *OBDD,* să reprezinte o mulțime de funcții booleene în mod unic este ca, pentru toate funcțiile, *OBDD* să fie construită utilizînd aceeaşi ordine a variabilelor. Alegerea acestei ordini a variabilelor poate avea un impact esențial asupra dimensiunilor diagramei binare care rezultă. Problema are 3 aspecte: primul vizează găsirea unei ordini eficiente la momentul construcției reprezentării *OBDD* pe baza unor informații legate de structura funcției booleene reprezentate; al 2-lea are în vedere că după construirea *OBDD* folosind o ordine inițială, mai mult sau mai puțin optimă, să se obțina o nouă ordine a variabilelor care să fie optima sau mai bună decît cea inițială; al 3-lea aspect ține cont de faptul că, în procesul de manipulare a funcțiilor booleene folosind *OBDD,* sunt create noi diagrame sau sunt eliminate diagramele vechi ceea ce face ca la un moment dat o anumită ordine a variabilelor să nu mai fie optimă pentru noua mulțime de funcții reprezentate. De aceea ordonarea statică a variabilelor, care presupune utilizarea unei singure ordini a variabilelor pentru întreaga durată a manipulărilor, a fost înlocuită cu ordonarea dinamică a variabilelor, care presupune reactualizarea ordonării la diverse momente de timp pe parcursul manipulării funcțiilor conform unor criterii diverse. O serie întreagă de studii s-au ocupat de găsirea unor algoritmi eficienți care să determine cea mai bună ordine a variabilelor vizînd unul sau o combinație a aspectelor prezentate. Dintre metodele de ordonare cele mai importante găsite în literatură găsim:

- aleatorie, perechi de stări sunt alese în mod aleator, poziția în ordinea variabilelor fiind schimbată între ele
- aleatorie cu pivot, similară cu cea precedentă, doar că prima variabilă este aleasă să fie deasupra variabilei cu numărul maxim de noduri iar cea de a 2-a va fi dedesubtul acelei variabile
- prin analiza minuțioasă, cernere, în care fiecare din variabile este deplasată în sus şi în jos în ordinea variabilelor astfel încît ea sa ia toate pozițiile posibile şi se determină

poziția cea mai bună, aceasta devenind noua poziție a variabilei
- cu analiza simetriei, la fel cu metoda anterioară, cu modificarea că variabilele care devin adiacente în timpul analizei sunt testate dacă sunt simetrice; în caz afirmativ ele sunt grupate şi analiza continuă deplasînd întregul grup de variabile
- prin analiza grupată, care este similară cu cea precedentă, cu diferența că gruparea nu mai este restrictionată la proprietatea de echivalență
- folosind permutarea fereastră, în care ordonarea este studiată numai pentru un anumit grup de variabilele
- folosind principiul călirii simulate (*simulated annealing*) se bazează pe utilizarea algoritmilor ce folosesc conceptul de călire simulată pentru determinarea unei ordini optime a variabilelor
- folosind algoritmi genetici, se bazează pe utilizarea algoritmilor genetici pentru determinarea unei ordini a variabilelor[38]
- folosind programarea dinamică se bazează pe algoritmii programării dinamice pentru determinarea unei ordini optime a variabilelor
- folosind informații comune, se bazează pe folosirea informațiilor ce se pot obține prin faptul că anumite structuri sunt utilizate în comun cu reprezentarea funcțiilor booleene
- folosind analiza nucleului expresiilor booleene, se bazează pe informațiile obținute prin analiza expresiilor booleene corespunzătoare funcțiilor multi-nivel
- folosind echivalența functională, se bazează pe împarțirea funcțiilor booleene în clase de echivalență prin gruparea funcțiilor avînd aceeaşi structură

5.6.2 Reducerea dimensiunilor

Pentru a construi mulțimea stărilor ce pot fi atinse pentru un automat finit se poate ajunge la manipularea *OBDD* pentru funcții incomplet definite. Reducerea dimensiunilor *ROBDD* utilizate pentru a reprezenta această mulțime de stări a condus la ideea de a găsi acele *ROBDD* care să acopere funcția incomplet definită şi să aibă dimensiunea minimă. Aceeaşi problemă apare şi atunci cînd sunt manipulate funcțiile incomplet definite şi pentru care se caută să fie reprezentate prin *ROBDD.* Alte aplicații şi algoritmi care să rezolve aceste probleme sunt minimizarea funcțiilor cu mai multe ieşiri incomplet specificate în care reducerea se face utilizînd o partiție a mulțimii variabilelor de care depinde o funcție incomplet definită în grupări simetrice, sau minimizarea *BDD* prin permutarea tabelelor de adevăr.[102]

5.6.3 Pachete de programe ce implementează diagrame de decizie binară

Datorită amplorii pe care a luat-o utilizarea *BDD*-urilor, o serie de colective de cercetare au dezvoltat pachete software care să implementeze aceste structuri, precum şi algoritmii corespunzători de manipulare a lor. Cele mai cunoscute sunt: *BDD*, dezvoltat la *Universitatea Berkeley California, CMU-BDD,* dezvoltat la *Carnegie-Mellon University*, *CAL , CUDD*, autor Fabio Somenzi, *Universitatea Colorado, etc.*

Toate aceste pachete au următoarele caracteristici comune:
- construcția *BDD*-ului se face prin traversare în adîncime (depth-first)
- tabela unicatelor este implementată folosind tabele de dispersie în care coliziunile sunt rezolvate prin înlanțuirea elementelor în liste de coliziuni
- tabela de calcul (*computed table*) este implementată utilizînd tehnicile memoriei cache

- sunt utilizate arcele negate implementate prin folosirea unui bit suplimentar care indică dacă o funcție este inversată sau nu
- eliberarea memoriei nefolosite se bazează pe contorizarea referințelor către noduri, această operație este activată atunci cînd nodurile nefolosite depăşesc o anumită valoare limită
- este utilizată reordonarea dinamică a variabilelor

Datorită avantajelor pe care le oferă utilizarea diagramelor de decizie binară în manipularea funcțiilor booleene, aceste reprezentări au fost implementate în diverse sisteme de proiectare asistată, fiind utilizate în diverse etape ale proiectării. Dintre acestea sunt: *SIS (System for Synthesis of Sequential Circuits), VIS (Verification Interacting with Synthesis),* de la *Berkeley University,* pachetul de programe *ALLIANCE,* de la *Universitatea Marie Curie, Paris,* precum şi *MVSIS*, versiunea modificată de *SIS*, pentru
implementaera cu *Diagrame de Decizie Multinivel.*

5.6.4 Variante ale ROBDD

Diagramele de decizie binară clasice s-au dovedit a fi deosebit de eficiente pentru reprezentarea anumitor clase de funcții booleene. S-a observat însă că acestea işi pierd eficiența în cazul altor clase de funcții. De aceea s-a căutat extinderea acestor tipuri de reprezentări astfel încît să fie depaşite aceste neajunsuri. S-au propus o serie de structuri alternative care au la bază diagramele de decizie binară dar la care, anumite proprietăți ale diagramelor de decizie binară au fost eliminate sau au fost înlocuite cu altele noi sau au fost adăugate proprietăți noi. Aceste modificări au avut ca efect extinderea mulțimii obiectelor ce pot fi reprezentate, modificarea dimensiunilor reprezentărilor şi modificarea complexității algoritmilor. Cele mai importante extinderi ale acestor reprezentări sunt:

- diagramele de decizie binară bazate pe grafuri (*graph driven BDD's*) sunt caracterizate prin faptul că ordinea în care sunt evaluate variabilele este dată de un graf asociat numit şi graf oracol
- diagrame de decizie binară indexate *(indexed BDD's*) sunt utilizate pentru a reprezenta în mod compact funcțiile pentru care *OBDD* au dimensiuni foarte mari; la aceste diagrame o variabilă apare de 2 ori pe un drum în *BDD*
- diagramele de decizie binară extinse (*extended BDD's*), implică un anumit grad de nedeterminare
- diagramele de decizie binară generale (*general BDD's*)
- diagramele de decizie binară libere (*free BDD's*), pe fiecare drum în *BDD*, fiecare variabilă este testată cel mult o singură dată
- diagramele de decizie binară partiționate (*PBDD's* - *partitioned based DD's*)
- diagramele de decizie binară bazate de transformarea produselor *CTBDD* (*Cube Transformation BDD's*) se bazează pe faptul că în loc să reprezinte funcția studiată, ea este mai întîi pusă în corespondență cu o altă funcție printr-o transformare injectivă a domeniului, funcția rezultată fiind cea care este reprezentată
- diagramele de decizie algebrice *ADD's* (*algebraic DD's*) sunt asemanatoare *BDD-urilor,* doar că nodurile terminale pot lua şi alte valori decît 0 şi 1
- diagrame de decizie binară cu valori pentru arce *EVBDD's* (*edge valued BDD's*) constituie o reprezentare canonică şi compactă a funcțiilor ce implică valori booleene şi numere întregi
- diagrame de moment binare *BMD* (*binary moment diagrams*) sunt utilizate pentru a

reprezenta funcţii numerice, în special pe cele care sunt intîlnite la verificarea circuitelor aritmetice
- diagrame de decizie multiplă *MDD's* (*multiple DD's*) utilizate pentru reprezentarea şi manipularea funcţiilor multivalente
- diagrame de decizie binară partajate cu atribute pentru arce (*shared BDD's with attributed edges*) se bazează pe reprezentarea diagramelor de decizie binară izomorfe în mod unic în memorie, iar arcele pot fi marcate cu diverse atribute cum ar fi inversoare pentru intrări şi ieşiri, şi valori de decalare a variabilelor
- diagrame de decizie binară ordonate funcţionale *OFDD's* (*ordered functional DD's*) sunt diagrame de decizie binară bazate pe dezvoltarea Reed-Muller (faţă de *OBDD* clasice care sunt diagrame de decizie binară bazate pe dezvoltarea Shannon)

5.7 Metode de reprezentare utilizînd librăria GMP

Librăria GMP sau "big-numbers library" este o librărie distribuită în mod gratuit, parte a setului de proiecte sub licenţă GNU. Această librărie facilitează efectuarea operaţiilor aritmetice cu precizie arbitrară, operaţiile pe seturi de numere întregi, numere raţionale, precum şi operaţii cu numere în virgulă mobilă. Nu există o limită practică a preciziei utilizate decît cea implicată de disponibilul de memorie de pe maşina pe care este folosită librăria GMP. Librăria prezintă un set bogat de funcţii matematice, precum şi funcţii care au o interfaţă clasică. Principalele aplicaţii ale librăriei GMP sunt cele legate de criptografie şi cercetare, aplicaţii de securitate pe Internet, sisteme algebrice, cercetare algebrică computaţională, etc. Viteza de execuţie obţinută prin utilizarea librăriei GMP pe parcursul rezolvării problemelor matematice complexe se datorează utilizării de cuvinte complete ca tip de bază aritmetic, prin utilizarea de algoritmi rapizi, care au înglobat cod de asamblare foarte optimizat pentru cele mai utilizate bucle de execuţie, cu adaptări specifice pentru diverse tipuri de procesoare, precum şi prin o atenţie deosebită acordată optimizării algoritmilor care produc o creştere asimptotică a vitezei de execuţie.

In această lucrare autorul a studiat şi utilizat funcţii de bază din librăria GMP pentru reprezentarea soluţiilor obţinute în urma operaţiilor efectuate asupra setului de automate finite, considerate ca mărimi de intrare. Reprezentarea datelor în memorie utilizînd setul de funcţii specifice din librăria GMP a permis autorului nu doar rezolvarea problemelor legate de complexitatea operaţiilor care trebuiesc efectuate asupra automatelor de o complexitate foarte mare, dar şi o reorientare a efortului de cercetare către direcţia dorită, aceea de optimizare a procesului de sinteză a circuitelor complexe, şi nu de a optimizare a reprezentării datelor în memoria internă a calculatorului.

De asemenea, prin optimalitatea funcţiilor utilizate în pachetul ***fsmtool***, performanţele programului au crescut simţitor, acest lucru putînd fi cu uşurinţă remarcat în cazul descompunerii automatelor finite cu peste 1.000 de stări, ceea ce a permis obţinerea de rezultate foarte bune într-o perioadă de execuţie rezonabilă pentru situaţii care în mod normal ar fi durat de citeva sute sau mii de ori mai mult în raport de timp per cantitate de rezultate obţinute.

Capitolul 6

Modalități de sinteză a circuitelor digitale complexe

6.1 Minimizarea retelelor logice

6.1.1 Necesitatea programelor de minimizare logică

Porțiunea dintr-un circuit integrat care implementează un set de funcții booleene de obicei deține un procentaj ridicat din suprafața acestuia. Din păcate, nu există înca o teorie generală care să funizeze margini mai reduse pentru aria totală a implementărilor logice în sistemele integrate atît proiectanților cît şi programelor de automatizare a proiectării. De aceea, sarcina principală a programelor de optimizare trebuie să fie cea a alegerii circuitului cu cea mai convenabilă aşezare geometrică, minimizînd suprafața dar în acelaşi timp îndeplinind şi multe alte condiții de timing, putere disipată, şi utilizarea celulelor standard. Sunt utilizate numeroase abordări pentru diferitele tehnologii şi tipuri de proiectare (celule standard, *RAM*, *PLA*, *FPGA*, poziționare geometrică *Weinberger*, arii de porți, etc). Există numeroase surse de non-optimalitate în specificațiile inițiale ale circuitelor logice. Dacă logica este creată din descrierea automatului, ea nu a fost minimizată de proiectant deloc. Dacă este specificată ca o descriere sursă, atunci ea poartă toate non-optimalitățile felului în care proiectantul şi-a conceput ideea lui. Descrierea logicii în sistemele de proiectare integrată automatizată este rezultatul compilării hardware a unui sistem de nivel înalt sau al unui limbaj la nivel de transfer între regiştri. Această descriere de obicei nu este optimală şi trebuie deci apoi optimizată cu transformări bazate pe algebra booleană mai întîi independente şi apoi dependente de tehnologie. Prin urmare minimizarea logică este tot timpul o necesitate, în care se aplică diferite metode, în funcție de stadiul proiectării, scopul propus, şi tehnologia de destinație. [8] În principiu sunt integrate 2 stagii în sistemele de optimizare logică: optimizare independentă de tehnologie, urmată de maparea din tehnologia generică în tehnologia dorită, şi apoi optimizarea dependentă de tehnologie. Metodele de proiectare a circuitelor logice includ în principiu 2 categorii: proiectare bi-nivel, şi proiectare multi-nivel. Circuitele bi-nivel sunt realizate de obicei cu circuite *PLA* şi *FPGA*. [5]

6.1.2 Minimizarea circuitelor FPGA şi a circuitelelor bivalente

Circuitelele FPGA sunt utilizate în proiectare pentru a implementa partea logică a automatelelor sau blocuri de funcții combinaționale. Dacă nu sunt minimizate, ele pot deveni de dimensiuni extrem de mari, creste mult puterea disipată, şi devin lente. Minimizarea logică constă în minimizarea numărului de termeni şi uneori a numărului de variabile din termeni, pentru scăderea puterii consumate, siguranța şi împachetare. Algoritmii booleeni de minimizare pentru circuite *PLA* şi *FPGA* sunt optim sau sau aproximare. Primul poate fi folosit pentru maxim 14 variabile în cazul general, şi pînă la 20 de variabile în majoritatea aplicațiilor industriale. Cel mai folosit program de aproximare, *Espresso*, a introdus un numar de idei noi teoretice şi de implementare în univerități şi industrie. Optimizarea topologică include partiționare, şi împachetare (*Hachtel*) pentru a reduce suprafața neutilizată din interiorul circuitului PLA. Prin împachetare se incearcă să se gasească o permutare de linii

şi/sau coloane de planuri *AND* sau *OR* din cadrul circuitului, astfel încît cît mai multe din linii sau coloane să fie combinate (împachetate) într-o singură linie sau coloană. Au fost implementate numeroase arhitecturi, algoritmi de împachetare şi metode de partiţionare topologică la *Berkeley University*, *IBM* sau la alte firme şi universităţi. Problemele parţiale importante legate de majoritatea operaţiilor de sinteză bi-nivel sunt cele de complementare a funcţiilor booleene, şi de recunoaştere şi minimizare a funcţiilor unare (sume de produse a variabilelor ne-negate). [9] Deoarece operaţiile pe şiruri de cuburi sunt relativ lente şi sunt utilizate în mod repetat în proiectare, s-a propus realizarea de acceleratoare hardware dedicate pentru ele, cum ar fi aceleratoare specifice pentru verificarea tautologiilor logice, sau acceleratoare pentru rezolvarea problemelor combinaţionale, bazate pe reducerea problemelor la probleme combinaţionale generice precum colorarea grafului sau satisfiabilitate şi implementarea lor în arhitecturi sistolice paralele.

6.1.3 Minimizarea circuitelor logice multivalente

Există un mare numar de stiluri de proiectare alternative posibile pentru logica multi-nivel [10]:

- logica domino-ului
- celule statice *CMOS*
- celule standard
- cascadare de FPGA-uri
- asezări geometrice Weinberger
- arii logice programabile multinivel
- asezări geometrice speciale regulate , pentru funcţii simetrice

În prezent programele de optimizare se realizează ţinînd cont de aceste abordări. Structura celor mai multe sisteme de sinteză este:

- programele de interfaţă convertesc descrierea limbajelor de proiectare de nivel înalt de la intrare din format funcţional, structural sau comportamental, în reprezentări interne a reţelelor logice
- se realizează optimizarea logică independentă de tehnologie, în care sunt reduse numărul de porţi, numărul de ieşiri, şi numărul de tranzistoare
- se realizează apoi optimizarea dependentă de tehnologie în care sunt realizate transformările logice aferente
- în final se generează optimizarea topologică cum ar fi împachetarea celulor, împachetarea sîrmelor în reprezentarea geometrică Weinberger , etc, pentru a genera reprezentarea geometrică finală

Există 3 tipuri de abordări ale proiectării multi-nivel:

- optimizarea globală (Silicon Compiler)
- optimizare locală (Logic Synthesis System -LSS)
- combinaţii ale primelor 2 abordări

Optimizarea globală include descompunerea funcţiilor booleene. Algoritmii utilizaţi includ:

1. **extragerea** - subexpresiile comune sunt recunoscute şi înlocuite cu o singură variabilă
2. **reducerea** - variabilele intermediare care nu sunt optimale sunt eliminate
3. **simplificarea** - o expresie logică booleană bi-nivel este minimizată utilizînd algoritmi de minimizare logică
4. **descompunerea** - o funcţie complexă, prea mare pentru a putea fi implementată într-o tehnologie standard, este factorizată şi descompusă.

Rezultatul primului stagiu de compilare al programului de descriere comportamentală de nivel înalt este o reţea logică non-optimală. La acest nivel sunt posibile unele transformări

de optimizare a rețelei, care îşi mențin valoarea de adevăr pentru implementarea rețelei logice în orice tehnologie. Aceste transformări constau în general în eliminarea unor părti din rețea care nu influențează comportamentul, adică nu există nici un traseu de la ele către o ieşire a circuitului, şi în reducerea numărului de intrări, coform regulilor algebrei booleene. Pot fi găsite parți ale rețelei care sunt neconectate la ieşiri , care sunt efectul indirect al unor transformări executate în etape anterioare ale implementării structurale. Faptul că apar aceste părți de rețea poate fi ori eroarea proiectantului, ori mai degrabă rezultatul unei specificații inițiale de nivel înalt. Este posibilă reducerea numărului de intrări ale porților, precum şi numărul de porți logice în unele situații. Aceasta rezultă din valori constante de 0 sau 1 logic la intrările porților. Este de obicei efectul aplicării, la unele etapte anterioare ale proiectării, a blocurilor standard precum registre, sumatoare, multiplexoare sau comparatoare, care au valori fixe la unele date de intrare. Spre exemplu operația unui sumator de a aduna o variabilă, cum ar fi o magistrală pe 4 biți, şi o constantă, o secvență binară pe 4 biți, oferă posibilitatea de simplicare a sumatorului, care în rețeaua rezultată nu işi va mai menține universalitatea de funcționare. Transformările aplicate pentru a minimiza aceste rețele se bazează pe reguli foarte simple a căror succesiune permite uneori reducerea subtanțială a rețelei. Următoarele transformări optimizează rețeaua prin aplicarea unei abordări bazate pe reguli, care este de asemenea independentă de tehnologie. Un astfel de sistem a fost proiectat de către *IBM*.

Alt tip de abordare, care de obicei porneşte de la un set de ecuații booleene, se bazează pe factorizare. Regulile de bază ale factorizării, precum ab+ac = a(b+c), pot fi de obicei aplicate în multe locuri ale setului de expresii, iar aplicarea unei factorizări o perturbă pe cealaltă. Ideea este de a selecta factorizarea optimală pentru seturi mari de ecuații logice. Factorizarea este puternic legată de descompunere. Funcția ar trebui să fie prezentată ca o compunere a unor funcții, care posibil functionează pe seturi disjuncte de variabile de intrare. Un astfel de algoritm de tip *Branch and Bound* de proiectare optimală de rețele multi-nivel cu porti *NAND* a fost implementat de către Davidson. O abordare modernă a descompunerii factorizării a fost introdusă de prof. Brayton, *Berkeley Univ.*, iar softul respectiv este implementat în prezent la *IBM*. În cazul acestui tip de descompunere, ambele componente primesc informații despre starea în care se află cealaltă componentă, prin urmare, ea se încadrează în categoria descompunerii generale a unui automat. S-a arătat de asemenea că această metoda poate fi folosită şi ca ajutor pentru ghidarea etapei de codificare a stărilor unui automat. Pentru realizarea descompunerii prin factorizare se caută găsirea unor subgrafuri izomorfe de preferința avînd dimensiunile cît mai mari. Această metodă are şi ea limitele ei date de faptul că funcția de cost se bazează pe reducerea numărului de stări sau arce ale grafului de fluență. În cazul în care există mai mulți factori, va fi necesară o funcție de cost care să măsoare cu o cît mai mare acuratețe calitatea descompunerii obținute după etapa de factorizare. Rezultatele experimentale au demonstat că în cazul în care există factori exacți de dimensiuni mari, ansamblul obținut în urma descompunerii va avea o complexitate redusă. Alte structuri speciale ale implementării funcțiilor logice sunt: circuite pe 3 nivele cu *NAND*-uri, arbori, cascade, şi arii de porți logice programabile.

Teoria implicanților primi generalizați introdusă de *Devadas*, a fost aplicată în domeniul descompunerii automatelelor, astfel încit prin utilizarea lor se permite obținerea unui algoritm exact pentru realizarea descompunerii unui automat care să fie realizat dintr-un număr oarecare de componente şi avînd o structură de interconectare oarecare. Algorimul ia în considerare o funcție de cost care corespunde unei sume ponderate a numărului de termeni produs din componentele descompunerii atunci cînd acestea sunt codificate şi minimizate pentru cazul bi-nivel. Se arată astfel că această funcție de cost este mai precisă în a prezice necesarul de elemente în cazul descrierii la nivel logic decît celelalte metode anterioare utilizate pentru descompunere [21].

6.2 Clasificarea procesului de sinteză

Procesul de sinteză a circuitelor complexe este o problemă de tip NP-complete. Clasificarea modelelor de sinteza se face prin o taxonomie a proceselor de sinteză a circuitelor. Sinteza poate fi vazută ca un set de transformări care se desfaşoară între două etape. Se poate realiza o descriere a proceselor de sinteză la diferite nivele de modelare a circuitelor, dupa cum urmează:

- Sinteza la nivel arhitectural

Constă în generarea unui aspect structural al unui model la nivel arhitectural. Acest lucru presupune determinarea şi atribuirea funcțiilor circuitului unor operatori, denumiți **resurse**, precum şi interconectarea lor şi calcularea timpului lor de execuție. Mai este denumită şi sinteză de nivel înalt sau sinteză structurală, deoarece determină structura macroscopică, la nivel de bloc, a circuitului. Pentru a evita ambiguitățile este denumită sinteză arhitecturală. Un model comportamental la nivel arhitectural poate fi exprimat ca un set de operații şi dependențe. Sinteza arhitecturală presupune identificarea resurselor hardware care pot implementa operațiile şi maparea lor pe resurse. Cu alte cuvinte, sinteza defineşte un model structural al unei **căi de date**, ca o interconectare de resurse, şi un model la nivel logic al unei **unități de control** care trimite semnalele de control către calea de date.

- Sinteza la nivel logic

Este procesul de generare al unui model structural al nivelului logic. Sinteza logică este procesul de manipulare a specificațiilor logice pentru a crea modele logice ca o interconexiune între primitivele logice. Prin urmare sinteza logică determină structura microscopică, la nivel de poartă, a circuitului. Procesul de transformare a modelului logic în o interconexiune de instanțe a celulelor de librărie este de obicei denumit mapare tehnologică. Un model la nivel logic al unui circuit poate fi dat de o diagrama a tranzițiilor de stări ale unei maşini cu stări finite, de o schemă a unui circuit sau de un model HDL echivalent. Poate fi specificat de către un proiectant sau sintetizat dintr-un model la nivel arhitectural. Procedeele de sinteză logica pot fi de diferite tipuri în funcție de natura circuitului, secvențiale sau combinationale, şi în funcție de reprezentare, diagrame de stări sau scheme bloc. Există multe configurări posibile ale unui circuit. Optimizarea joacă un rol major, în completare cu sinteza, în determinarea modelelor microscopice care trebuie implementate. Realizarea finală a procesului de sinteză logică este o reprezentare structurală completă, cum ar fi o listă de circuite la nivel de porți logice.[71]

- Sinteza la nivel geometric

Constă în crearea unui model fizic la nivel geometric. Atrage dupa sine specificarea tuturor structurilor geometrice definind în acelaşi timp planul fizic al circuitului precum şi poziționarea lor. Este de obicei denumită proiectare la nivel fizic a planului circuitului. Nivelele planului sunt în corespondență cu maştile utilizate pentru proiectarea circuitelor. Prin urmare, proiectarea la nivel geometric este etapa finală a proiectării circuitelor. Proiectarea la nivel fizic depinde în mare măsura de metoda de proiectare utilizată. Pntru proiectarea de tip „custom”, proiectarea la nivel fizic este realizată cu ajutorul editoarelor de suprafețe de circuit. Acest lucru înseamnă că proiectantul renunță la utilizarea uneltelor de sinteză automată în favoarea unei căutari manuale a unei poziții geometrice optime a circuitului, denumită optimizare la mînă. În cazul circuitelor prefabricate, proiectarea la nivel fizic este

realizată virtual, întrucît circuitele sunt deja fabricate înainte. În loc, personalizarea circuitului este realizată prin o mapare de memorie. Principalele sarcini ale proiectării la nivel fizic sunt **plasarea** şi **legarea traseelor**, denumite de asemenea şi **rutare**. **Generarea celulelor** este esenţială în cazul particular al proiectării macro-celulelor, unde celulele sunt sintetizate şi nu extrase dintr-o librărie.[125]

6.3 Optimizarea Circuitelor

Optimizarea circuitelor este de cele mai multe ori realizată concomitent cu sinteza. Optimizarea este motivată nu numai de necesitatea de a spori calitatea circuitelor, dar şi de faptul că procesul de sinteză fără optimizare ar produce circuite necompetitive şi prin urmare funcţionarea lor ar fi destul de ineficientă. Optimizarea se poate face atît pentru reducerea suprafeţei ocupate cît şi pentru cresterea performanţei şi reducerea timpului de răspuns al circuitelor. Optimizarea suprafeţei circuitelor este un domeniu vast care este măsurat prin suma suprafeţelor componentelor circuitelor şi prin urmare poate fi calculat dintr-o abordare structurală a circuitului, o dată ce suprafaţa şi componentele sunt cunoscute. Calcularea ariei poate fi realizată ierarhic. Un model de nivel de abstractizare este descris în Fig 11.

Aspect Comportamental	Aspect Structural	Aspecte / Nivele
… PC = PC + 1; FETCH (PC); DECODE (inst); …	MULT ADD CONTROL RAM	Nivel Arhitectural
$State_0$ $State_3$ $State_1$ $State_2$	*a* *b* *c* *d* *x* *y*	Nivel Logic

Figura 11. Nivele de abstractizare şi aspectele corespunzătoare

În mod uzual, componentele fundamentale ale circuitelor digitale sunt portile logice şi regiştrii, ale căror suprafeţe sunt cunoscute dinainte. Suprafaţa ocupată de conexiuni are un rol important şi nu trebuie nicidecum neglijată. Poate fi derivată dintr-o abordare fizică completă, sau estimată dintr-o abordare structurală utilizînd modele predictive sau statistice. Performanţa circuitului ocupă un rol important, de aceea calcularea ei necesită o analiză a structurii şi de cele mai multe ori a comportamentului circuitului.

În caz particular, performanţa unui circuit logic combinaţional este masurată prin **întirzierea propagării** intrărilor şi ieşirilor. În multe situaţii este realizată o prezumţie de simplificare: toate intrările sunt disponibile în acelaşi timp şi performanţa se referă la întîrzierea de pe parcursul **căii critice** a circuitului.

Performanța unui circuit sincron secvențial poate fi măsurată prin perioada sa de **ciclu secvențial**, cum ar fi perioada celui mai rapid tact de ceas care poate fi aplicat circuitului, cu notificarea că întîrzierea de pe componenta combinațională a circuitului secvențial este mai mică decît perioada de secvențiere.

Atunci cînd este considerat un model la nivel arhitectural al unui circuit ca o secvență de operații, cu o implementare secvențială sincrona, o metodă de măsurare a performanței este **întîrzierea circuitului**, în caz particular timpul de execuție al operațiilor. Întîrzierea poate fi măsurată în termeni de frecvență de ceas sau tacturi. Prin urmare produsul dintre frecvența de ceas şi întîrzieri determină timpul de execuție per ansamblu. De multe ori, frecvența de ceas şi întîrzierile sunt optimizate independent, pentru a simplifica problema de optimizare precum şi pentru a satisface alte constrîngeri de proiectare, cum ar fi interfațarea cu alte circuite.

Circuitele sincrone pot implementa o secvență de operații printr-o metoda „pipeline", în care circuitul realizează operații concurente pe seturi de date diferite. O masură de performanță în plus este apoi rata în care datele sunt produse şi consumate, denumită **capacitatea de prelucrare** a circuitului. Într-un circuit „non-pipeline" capacitatea de prelucrare este mai mică sau egală cu inversul produsului dintre frecvența de ceas şi întîrziere. Metoda „pipeline" permite unui circuit să-si mărească capacitatea de prelucrare peste această limită. Rata maximă de „pipeline" apare atunci cînd rata de prelucrare este inversul frecvenței de ceas, atunci cînd data este produsă şi consumată la fiecare ciclu de ceas.

În acord cu aceste definiții şi modele, optimizarea performanței constă în minimizarea întîrzierilor pentru circuitele combinaționale, a frecvenței de ceas şi a întîrzierilor pentru circuitele secvențiale şi maximizarea capacității de prelucrare a circuitelor de tip „pipeline".

Posibilele implementări structurale diferite ale unui circuit definesc **spațiul de proiectare** al circuitului. Spațiul de proiectare este un set finit de puncte de proiectare, unde există funcții de evaluare pentru suprafață şi performanță asociate cu fiecare punct de proiectare. Există de asemenea numeroase funcții de evaluare precum: suprafața, întîrzierea, frecvența de ceas şi capacitatea de prelucrare. Spațiul multidimensional definit de către aceste obiective se numeşte **spațiu de evaluare a proiectului**.

Optimizarea circuitului corespunde căutarii celei mai bune soluții, cum ar fi o configurare a circuitului care optimizează toate obiectivele. Întrucît problema de optimizare implică **criterii multiple**, trebuie acordată o atenție specială definirie punctelor optimale. Un punct din spațiul de proiectare este denumit **punct Pareto** dacă nu există nici un punct în spațiul de proiectare cu cel puțin un obiectiv inferior, toate celelalte puncte fiind inferioare sau egale lui. Un punct Pareto corespunde unui optim global într-un spațiu de evaluare monodimensional. Pot exista mai multe puncte Pareto, care corespund implementării proiectelor care nu sunt dominate de altele, deci merită a fi luate în considerație. Imaginea punctelor Pareto în spațiul de evaluare a proiectului este setul de puncte de schimb optimale. Interpolarea lor produce o curbă sau suprafață de schimb. Optimizarea circuitului implică funcții obiectiv multiple. Problema de optimizare este dificil de rezolvat datorită naturii discontinue a funcțiilor obiectiv şi a naturii discrete a spațiului de proiectare. În general, punctele Pareto sunt solutii pentru problemele de optimizare cu constrîngeri. Spre exemplu, se impun constrîngeri de întîrziere pentru minimizarea suprafeței, respectiv constrîngeri de suprafață pentru minimizarea timpului de răspuns. Din nefericire, datorită dificultăților apărute la problemele de optimizare, pot fi calculate doar aproximări ale punctelor Pareto.

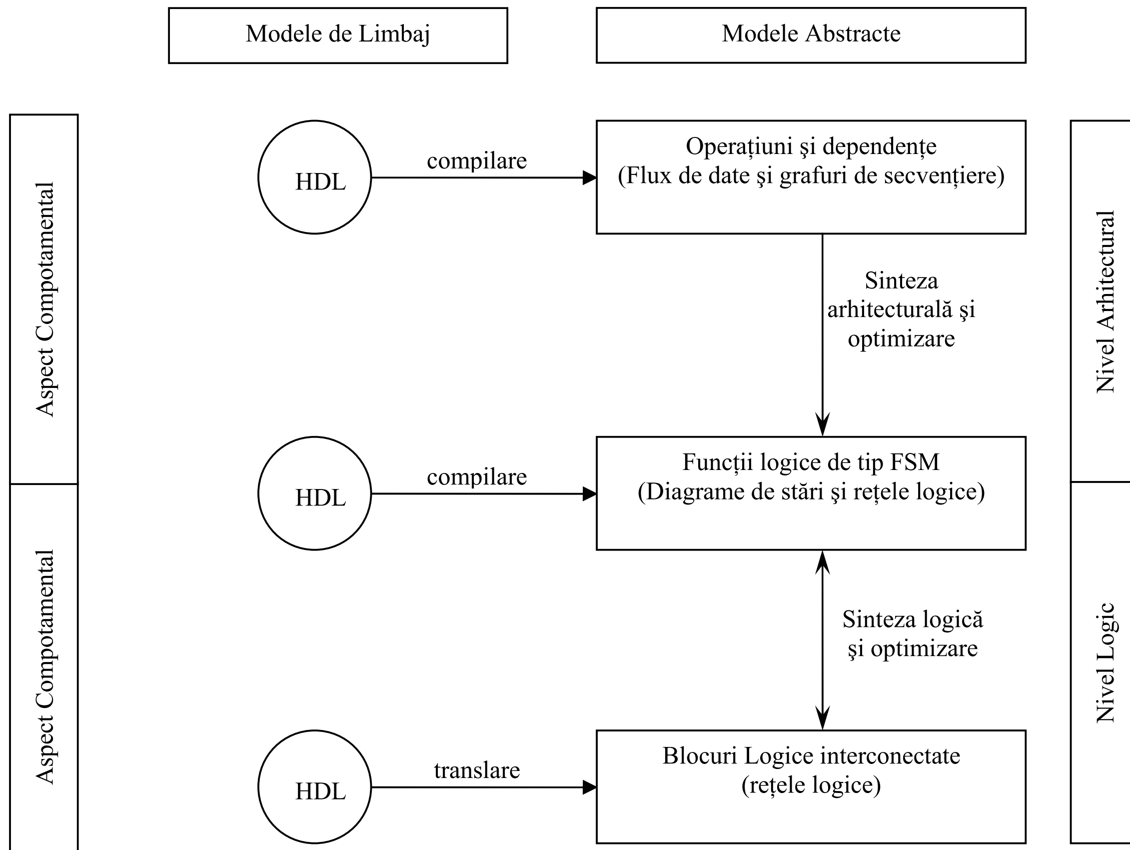


Figura 12. Modele de circuite, sinteză şi optimizare

In Fig. 12 sunt reprezentate schematic procesele de sinteză şi de optimizare şi relațiile lor cu modelele. Se presupune că specificațiile circuitelor sunt realizate cu modele HDL. Algoritmii de sinteză şi de optimizare sunt descrişi ca fiind bazați pe modele abstracte, dar care sunt destul de puternice ca să poată prelua informațiile esențiale din modelele HDL şi să poată separa sinteza de specificațiile caracteristice limbajelor de descriere hardware. Exemple de modele abstracte care pot fi derivate din modelele HDL prin compilare sunt: secvențierea şi grafurile de structuri de date reprezentînd operații şi dependențe, diagrame de tranziții de stare reprezentînd comportament de automate finite şi retele logice reprezentînd blocuri logice interconectate, care corespund porților şi funcțiilor logice. [30]

6.4 Reprezentarea funcțiilor logice

Definiție: Fie $B = \{0,1\}$. O **funcție logică complet specificată** *f* cu ***n*** intrări este o mapare $f{:}B^n \rightarrow B$. Fiecare element din domeniul B^n se denumeşte **minterm** al lui *f*, iar $f^{1}(1) = \{x \in B^n \mid f(x) = 1\}$ este **onset**-ul lui *f*, iar $f^{1}(0) = \{x \in B^n \mid f(x) = 0\}$ este **offset**-ul lui *f*. Fiecare element din $\{0, 1\}^n$ este o muchie.

Definiție: O **funcție logică incomplet specificată** *f* cu ***n*** intrări este o mapare $f{:}\{0,1,*\} \rightarrow B^n$. $f^{1}(*) = \{x \in B^n \mid f(x) = *\}$ este **setul de indiferență** (don't care) al lui f.

Definiție: Un **literal** este o variabilă sau complementul acesteia. Un **cub** sau un **termen produs** este un produs sau o conjuncție de unul sau mai mulți literali astfel încît dacă *x* apare în produs, atunci *x'* nu poate apărea, şi viceversa. Un literal *a* sau *a'* reprezintă setul tuturor mintermilor pentru care variabila *a* ia valoarea 1 sau 0. Un cub reprezintă intersecția setului de mintermi reprezentați de toți literalii din el. Dacă într-un cub este prezentă o variabilă şi complementul său, atunci cubul devine identic 0. O **acoperire** *F* este un set de cuburi şi este interpretat ca o sumă booleana a elementelor sale.

Definiție: **Distanța Hamming** dintre 2 cuburi *c'* şi *c''*, notată prin *d(c', c'')*, este cardinalitatea setului de variabile x_i astfel încît ori $x_i \in c'$ şi $\overline{x_i} \in c''$ sau $\overline{x_i} \in c'$ şi $x_i \in c''$.

Intersecția a două cuburi *c'* şi *c''*, este definită doar dacă $d(c', c'') = 0$ şi $\exists\ c''' = c' \cup c''$. Este denumită intersecție deoarece acoperă intersecția setului de noduri acoperite de *c'* şi *c''*. Spre exemplu, intersecția dintre $a\overline{b}$ şi ac este $a\overline{b}c$. Intersecția unei acoperiri *F* cu un cub *c'* este un set al tuturor intersecțiilor lui $c' \in F$ astfel încît $d(c', c_i) = 0$ cu *c'*.

Un cub este un **implicant**, sau **termen produs**, al unei funcții logice *f* dacă nu acoperă nici un vîrf off-set din *f*. O acoperire **on-set** *F* a unei funcții logice *f* este un set de cuburi astfel încît fiecare cub din *F* este un implicant al lui *f* şi fiecare vîrf on-set din *f* este acoperit de către cel puțin un cub din *F*. O acoperire **off-set** R a unei funcții logice *f* este o acoperire a complementului lui *f*. Fiecare acoperire *F* corespunde unei funcții logice unice complet specificate, notată cu *B(F)*. O funcție logică poate avea multe acoperiri.

O acoperire este interpretată ca o sumă booleană a elementelor proprii, deci poate fi vazută şi ca o implementare sumă de produse bivalentă a funcției complet specificate *B(F)*.

Un implicant din *f* este **implicant prim** dacă nu este acoperit de către orice alt singur implicant al lui *f*. Un cub *c'* este o acoperire on-set *F* a unei funcții logice *f* care poate fi expandată împotriva off-set-ului lui *f* prin eliminarea literalilor din el în timp ce nu acoperă orice virf off-set. Rezultatul **expansiunii** nu este unic, dar este totdeauna un implicant prim din *f*. O acoperire *F* este **redundantă** dacă un anumit cub din ea este redundant, altfel este **iredundantă**. Într-o acoperire primă şi iredundantă *F* fiecare cub are cel puțin un vîrf relativ esențial care este on vîrf on-set care nu este acoperit de nici un alt cub din *F*.

Cofactorul f_{x_i} al unei funcții logice *f* care depinde de literalul x_i este funcția logică obținută prin evaluarea lui *f* la $x_i = 1$. În mod similar, $f_{\overline{x_i}}$ este obținut prin evaluarea lui *f* la $x_i = 0$. Cofactorul lui *f* are următoarea proprietate pentru fiecare variabilă x_i:

$$f = x_i \times f_{x_i} + \overline{x_i} \times f_{\overline{x_i}} \text{ (descompunerea Shannon)}$$

Cofactorul c_x al unui cub c în funcție de literalul x_i este nedefinit dacă $\overline{x_i} \in c$ sau $c - \{x_i\}$ în caz contrar. Cofactorul F_{x_i} al unei acoperiri F în funcție de literalul x_i este setul de cuburi din F pentru care cofactorul care depinde de x_i este definit, cofactorizat în funcție de x_i.

Dacă F este o acoperire a unei funcții logice f, atunci $B(F_{x_i}) = f_{x_i}$.

O funcție logică f este **monoton crescătoare** pentru o variabila x_i dacă $f(x_i = 0, \beta) = 1 \Rightarrow f(x_i = 1, \beta) = 1$ pentru $\forall \beta \in \{0,1\}^{n-1}$, astfel dacă se mărește valoarea variabilei x_i de la 0 la 1, nu va scădea niciodată valoarea lui f de la 1 la 0. În mod similar, o funcție logică este **monoton descrescătoare** pentru o variabilă x_i dacă $f(x_i = 1, \beta) = 0 \Rightarrow f(x_i = 0, \beta) = 0$ pentru $\forall \beta \in \{0,1\}^{n-1}$. O funcție logică este **unară** pentru variabila x_i dacă este monoton crescătoare sau monoton descrescătoare în x_i. Altfel, funcția f este **binară** în x_i. O acoperire F este unară pentru o variabilă x_i, dacă variabila x_i apare doar în o singură fază (x_i sau $\overline{x_i}$) în cuburile sale (respectiv monoton crescătoare sau monoton descrescătoare). O funcție unară are o acoperire unică primă şi iredundantă, care este setul tuturor implicanților săi primi. Mai mult, dacă F este o acoperire unară a lui f, atunci f este o funcție unară.

***Definiție**:* O **sumă de produse** (SP) este o suma booleană sau o disjuncție de cuburi. O sumă de produse reprezintă uniunea setului de mintermi reprezentați de cuburile din el.

***Definiție**:* O formă factorizată este definită recursiv după cum urmează:
- un literal este o formă factorizată
- suma a două forme factorizate este o formă factorizată,
- produsul a doua forme factorizate este o formă factorizată.

Definiție**:* O funcție logică f cu ***n intrări şi ***k*** ieşiri este o mapare $f{:}B^n \rightarrow B^k$.

Definiție**:* O rețea booleană ***N este o reprezentare a unei funcții logice cu ieşiri multiple. Este un graf aciclic orientat, intrări primare PI(***N***), ieşiri primare PO(***N***) şi noduri interne intermediare IN(***N***).

Intrările primare nu au arce de intrare iar **ieşirile primare** nu au arce de ieşire. Y_i este o variabilă asociată fiecărui **nod intern** i iar f_i este reprezentarea unei funcții logice. Partea logică a fiecărui nod este reținută de obicei sub formă de suma de produse. Există un arc de la nodul i la nodul j din graf, dacă în reprezentarea lui f_j j foloseşte explicit y_i sau y_i'. În acest caz i se numeşte **fanin**-ul lui j, iar j este **fanout**-ul lui i. Setul fanin-urilor unui unui nod i este denumit *FI(i)*, iar setul fanout-urilor *FO(i)*. Dacă există o cale de la nodul i la nodul j atunci, se spune ca nodul i este un **fanin tranzitiv** al lui j, iar j este un **fanout tranzitiv** al lui i. Setul fanin-urilor tranzitive ale unui nod i este denumit *TFI(i)*, iar fanout-ul său tranzitiv este denumit *TFO(i)*. Rețeaua condusă de nodul i este setul de arce de tipul $(i, f^{\theta}), f^{\theta} \in FO(i)$, şi se numeşte **net** i. În Fig. 13 este reprezentată o rețea booleană cu 4 intrări primare a,b,c,d, o ieşire primară z, şi nodurile interne p,q,r.

O variabilă multi-valentă este o variabilă v_i care poate lua orice valoare dintr-un set finit S_i. O funcție logică multi-valentă din un set de n variabile multi-valente v_i este o mapare din produsul cartezian al lui S_i în $\{0,1\}$. Proprietățile funcțiilor logice ordinare (unde $S_i = \{0,1\}$ pentru toti i) pot fi direct extinse la funcții logice ale variabilelor multi-valente.

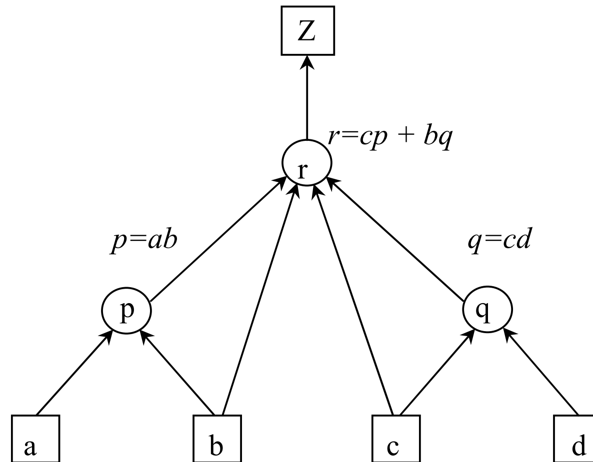


Figura.13 Rețea Booleană

6.5 Sinteza funcțiilor logice

Este procesul de citire a unei descrieri de nivel înalt a unui circuit şi de generare a unei implementări a circuitului în termeni de porți logice. Sinteza se realizează pentru o funcție obiectiv precum minimizarea ariei sau întirziere. Deoarece este un proces complex se împarte de obicei în 2 faze: optimizarea independentă de tehnologie urmată de optimizarea dependentă de tehnologie. Optimizarea logică independentă de tehnologie încearcă să genereze o reprezentare abstractă optimală a circuitului în funcție de de funcția obiectiv. Cea mai folosită reprezentare a circuitului este cea de rețea booleană şi cea mai folosită măsură este numărul de literali ai rețelei în formă factorizată, care este suma tuturor nodurilor interne ale rețelei de literali din reprezentarea în forma factorizată a fiecărui nod. Faza dependentă de tehnologie încearcă să implementeze rețeaua booleană optimizată utilizînd o librărie de porți logice predefinite în timp ce optimizează funcția obiectiv.

6.5.1 Optimizarea logică independentă de tehnologie

Principalul obiectiv în aceasta fază este de a găsi subfuncții care pot fi utilizate în comun de funcții multiple din nodurile booleene. Operația de bază în generarea subfuncțiilor este operația de **diviziune**.

***Definiție**:* O **expresie algebrică** este o reprezentare sub formă de sumă de produse a unei funcții logice care este minimală în raport cu conținutul cubului singular *(SCC)*, adică nici un cub nu este conținut în expresia altui cub.

***Definiție**:* Produsul a 2 expresii algebrice *f* şi *g*, *fg* este o sumă $\sum c_i d_j$, unde $\{c_i\} = f$, $\{d_j\} = g$, realizată iredundant în raport cu *SCC*, de exemplu $ab + a = a$. Dacă *f* şi *g* au suport disjunct, atunci este un produs algebric. Altfel este un produs boolean.

***Definiție**:* Fiind date 2 funcții f şi g, o operație de **diviziune** generează un cît q şi un rest r astfel încît $f = gq + r$. Dacă gq este un produs algebric, atunci este denumit **diviziune algebrică**. Altfel este o **diviziune booleană**.

Deşi diviziunea booleană este mai puternică, în sensul că poate genera mai puțini literali decît o diviziune algebrică în expresia $gq + r$, ea este totuşi mai dificil de obținut din punct de vedere computațional. Din aceste motive, în optimizarea logică, este utilizată de obicei diviziunea algebrică.

***Definiție**:* Fiind date 2 expresii algebrice f şi g, o diviziune f/g este denumită **diviziune slabă,** dacă în expresia $f = gq + r$:

-gq este algebrică
-r are cît mai puține cuburi posibile

Dacă ***n*** este numărul de termeni produs în f şi g, atunci diviziunea slabă poate fi realizată în $O(n \log n)$.

Definiție**:* O expresie este liberă de cuburi ***(cube-free) dacă nici un cub nu divide expresia exact. Spre exemplu, $ab + c$ este liberă de cuburi, în timp ce $ab + ac$ şi abc nu sunt. Conform definiției, o expresie liberă de cuburi trebuie să aibă mai mult de un singur cub deoarece un literal dintr-un cub divide cubul exact.

***Definiție**:* **Divizorii primari** ai unei expresii f sunt setul de de expresii $D\{f\} = \{f/c | c \text{ este cub}\}$.

***Definiție**:* **Kernel**-urile unei expresii f sunt setul de expresii $K\{f\} = \{g \mid g \in D(f) \text{ iar } g \text{ este liberă de cuburi}\}$.

Definiție**:* Un cub ***c utilizat pentru a obține kernel-ul $k = f/c$ este denumit **co-kernel** al lui k, iar cu $C(F)$ se notează setul de co-kernel-uri ale lui f.

Diviziunea este folosită ca operație de bază în operațiile de optimizare logică în rețele booleene care includ descompunere, extracție, resubstituție şi eliminare. Descompunerea este operația logică care exprimă o funcție logică dată sub forma de subfuncții mai simple. Un nod asociat cu fiecare subfuncție este adăugat în rețeaua booleană doar dacă nu există în prealabil. De exemplu, funcția

$$f = abc + abd + \overline{acd} + \overline{bc}d$$

poate fi descompusă în:

$$\begin{aligned} f &= xy + \overline{xy} \\ x &= ab \\ y &= c + d \end{aligned}$$

unde x şi y sunt adăugați ca noduri noi dacă nu există deja noduri cu aceeaşi funcție.

Extracția este definită ca operația care identifică subexpresiile comune din diferite funcții logice în rețeaua booleană. Nodurile asociate cu subexpresiile sunt create doar dacă ele nu există deja. Spre exemplu, după ce se extrag f, g şi h de mai jos

$$f = (a + c)cd + e$$
$$g = (a + b)e$$
$$h = cde$$

se obține

$$f = xy + e$$
$$g = xe$$
$$h = ye$$
$$x = a + b$$
$$y = cd$$

unde x şi y sunt adăugați ca noduri noi dacă este necesar. Resubstituția este operația care re-exprimă o funcție f în funcție de altă funcție g, unde ambele funcții deja există în rețeaua booleeană. Spre exemplu dacă

$$g = a + b$$
$$f = ac + bc$$

după resubstituția lui g în f se obține

$$f = gc$$

Eliminînd o funcție g în f este operația logică care re-exprimă o funcție f fară a utiliza în mod explicit funcția g. Ca rezultat, nodul boolean g este eliminat din fanin-ul funcției f. Deşi la prima vedere, eliminarea pare să intre în contradicție cu alte operații logice, ea este special introdusă cu scopul de a obține o optimizare independentă de tehnologie în afara minimului local. Operațiile logice descrise sunt utilizate pentru a restructura rețeaua booleană. O altă operație care minimizează funcția logică stocată în fiecare nod boolean este denumită **simplificare** sau **minimizarea nodurilor**, şi utilizează tehnicile de minimizare logică binivel aplicate fiecărui nod. Oricum, nodurile nu sunt minimizate independent unul de celalalt deoarece este necesară o anumita flexibilitate, care există datorită faptului că fanin-urile unui nod boolean ***n*** sunt legate unul de celălalt de către nodurile rețelei din fanin-ul tranzitiv al lui ***n***. Prin urmare fanin-urile nu pot fi libere să ia orice combinație de valori. Setul de combinații de valori, pe care nu le pot avea fanin-urile lui ***n***, formează *indiferențele de satifiabilitate (SDC)* ale lui ***n***. De asemenea, pentru unele combinații de valori pe care le iau intrările primare, valoarea evaluată la nodul ***n*** poate să nu fie observată la ieşirile primare. Cu alte cuvinte ieşirile primare rămin neschimbate după schimbarea valorii din nodul ***n***. Acest set de combinații de valori este denumit *indiferențele de observabilitate (ODC)* ale lui ***n***. În plus, circuitul sintetizat este de obicei doar un modul dintr-un sistem. Pentru unele combinații de valori pe care le iau ieşirile primare, comportamentul sistemului ramîne neschimbat. Acest set de combinații de valori este denumit *indiferențe externe (XDC)*. Setul tutuor *SDC*-urilor şi *ODC*-urilor unui nod este de obicei larg şi nu poate fi calculat eficient. În acest caz un subset corespunzator ambelor subseturi *SDC* şi *ODC*, împreună cu *XDC*, este utilizat ca valori indiferente în minimizarea logică binivel a nodurilor. După restructurare şi simplificare, reteaua booleană este optimizată iar pasul următor este faza dependentă de tehnologie, care mai este denumită şi faza de mapare.

Exemplu: Pentru sinteza logică a unui automat finit, se consideră automatul generat aleator cu programul **genfsm**. Descrierea sa internă este reprezentată prin tabela sa de tranziții (STT), în format kiss după cum se poate observa în Fig. 14, sau prin graful de tranziții (STG) din Fig.15 [57]

00	s0	s1	00
01	s0	s3	00
10	s0	s1	00
11	s0	s6	01
00	s1	s3	00
01	s1	s0	00
10	s1	s7	00
11	s1	s4	01
00	s2	s0	11
01	s2	s5	00
10	s2	s0	01
11	s2	s1	10
00	s3	s7	11
01	s3	s7	10
10	s3	s2	10
11	s3	s6	11
00	s4	s0	01
01	s4	s5	11
10	s4	s2	00
11	s4	s1	00
00	s5	s4	01
01	s5	s3	00
10	s5	s7	11
11	s5	s7	00
00	s6	s7	11
01	s6	s5	10
10	s6	s5	00
11	s6	s6	11
00	s7	s1	00
01	s7	s4	11
10	s7	s7	01
11	s7	s2	01

Figura 14. Reprezentarea prin STT a automatului

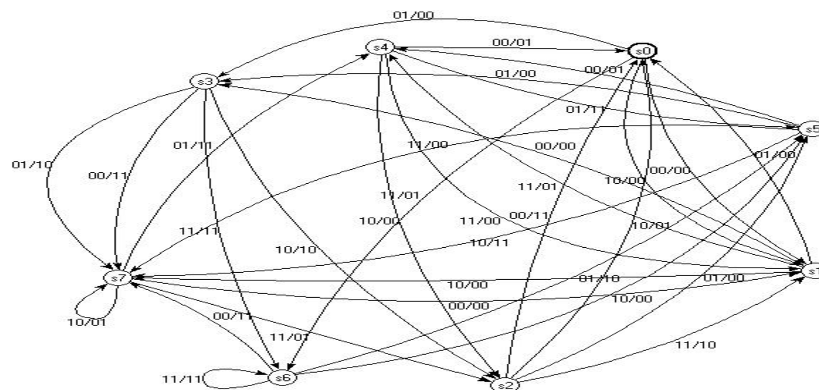


Figura 15. Reprezentarea prin STG a automatului

De asemenea în tabelul 5 este reprezentată tabela de tranziții a automatului sub forma:

Tabelul 5. Reprezentarea prin STT a automatului

x_1x_0 / Q	$Q_{n+1}/(z_1z_0)_n$			
	00	**01**	**11**	**10**
s0 = 000	s1/00	s3/00	s6/01	s1/00
s1 = 001	s3/00	s0/00	s4/01	s7/00
s2 = 010	s0/11	s5/00	s1/10	s0/01
s3 = 011	s7/11	s7/10	s6/11	s2/10
s4 = 100	s0/01	s5/11	s1/00	s2/00
s5 = 101	s4/01	s3/00	s7/00	s7/11
s6 = 110	s7/11	s5/10	s6/11	s5/00
s7 = 111	s1/00	s4/11	s2/01	s7/01

În tabelul 6 se calculează ieşirile automatului şi stările interne la pasul următor

Tabelul 6. Reprezentarea prin STT a stărilor interne şi a ieşirilor

$(x_1x_0)_n$ / $(y_1y_0)_n$	$(y_2y_1y_0)_{n+1}$				$(z_1z_0)_n$			
	00	**01**	**11**	**10**	**00**	**01**	**11**	**10**
s0 = 000	001	010	101	001	00	00	01	00
s1 = 001	010	000	110	100	00	00	01	00
s2 = 011	000	111	001	000	11	00	10	01
s3 = 010	100	100	101	011	11	10	11	10
s4 = 110	000	111	001	011	01	11	00	00
s5 = 111	110	010	100	100	01	00	00	11
s6 = 101	100	111	101	111	11	10	11	00
s7 = 100	001	110	011	100	00	11	01	01

Se realizează în continuare pentru fiecare variabilă de stare şi de ieşire în parte, optimizarea Veigh-Karnaugh pentru implementarea cu bistabile de tip “D”:

$(x_1x_0)_n$ / $(y_2y_1y_0)_n$	$(y_2)_{n+1}$				$(y_1)_{n+1}$				$(y_0)_{n+1}$			
	00	**01**	**11**	**10**	**00**	**01**	**11**	**10**	**00**	**01**	**11**	**10**
000	0	0	1	0	0	1	0	0	1	0	1	1
001	0	0	1	1	1	0	1	0	0	0	0	0
011	0	1	0	0	0	1	0	0	0	1	1	0
010	1	1	1	0	0	0	0	1	0	0	1	1
110	0	1	0	0	0	1	0	1	0	1	1	1
111	1	0	1	1	1	1	0	0	0	0	0	0
101	1	1	1	1	0	1	0	1	0	1	1	1
100	0	1	0	1	0	1	1	0	1	0	1	0

Tabelul 7. Reprezentarea prin STT a valorii stărilor următoare

$(x_1x_0)_n$ \ $(y_2y_1y_0)_n$	z_1				z_0			
	00	**01**	**11**	**10**	**00**	**01**	**11**	**10**
000	0	0	0	0	0	0	1	0
001	0	0	0	0	0	0	1	0
011	1	0	1	0	1	0	0	1
010	1	1	1	1	1	0	1	0
110	0	1	0	0	1	1	0	0
111	0	0	0	1	1	0	0	1
101	1	1	1	0	1	0	1	0
100	0	1	0	0	0	1	1	1

Tabelul 8. Reprezentarea prin STT a valorii ieşirilor

După ce se realizează optimizarea prin metoda clasica Veigh-Karnaugh se obțin următoarele ecuații pentru ieşiri şi pentru stările următoare:

$$(y_2)_{n+1} = y_2\overline{y}_1y_0 + y_2y_0x_1 + \overline{y}_1y_0x_1 + \overline{y}_2\overline{y}_1x_1x_0 + \overline{y}_2y_1\overline{y}_0\overline{x}_1 + \overline{y}_2y_1\overline{y}_0x_0 + \overline{y}_2y_1\overline{x}_1x_0 + y_1\overline{y}_0\overline{x}_1x_0 + y_2y_0\overline{x}_1\overline{x}_0 +$$
$$y_2\overline{y}_1\overline{x}_1x_0 + y_2\overline{y}_1x_1\overline{x}_0$$

$$(y_1)_{n+1} = y_2\overline{x}_1x_0 + \overline{y}_1\overline{y}_0\overline{x}_1x_0 + y_2y_1y_0\overline{x}_1 + y_2\overline{y}_1\overline{y}_0x_0 + y_1\overline{y}_0x_1\overline{x}_0 + \overline{y}_2\overline{y}_1y_0\overline{x}_1\overline{x}_0 + \overline{y}_2\overline{y}_1y_0x_1x_0 + \overline{y}_2y_1y_0\overline{x}_1x_0 +$$
$$y_2\overline{y}_1y_0x_1\overline{x}_0$$

$$(y_0)_{n+1} = y_1\overline{y}_0x_1 + \overline{y}_1\overline{y}_0\overline{x}_1\overline{x}_0 + \overline{y}_2\overline{y}_1\overline{y}_0x_1 + \overline{y}_2y_1y_0x_0 + y_2y_1\overline{y}_0x_0 + y_2\overline{y}_1y_0x_0 + y_2\overline{y}_1y_0x_1 + y_2\overline{y}_1x_1x_0$$

$$(z_1) = \overline{y}_2y_1\overline{y}_0 + \overline{y}_2y_1\overline{x}_1\overline{x}_0 + y_1\overline{y}_0\overline{x}_1x_0 + \overline{y}_2y_1x_1x_0 + y_2\overline{y}_1y_0\overline{x}_1 + y_2\overline{y}_1y_0x_0 + y_2\overline{y}_1\overline{x}_1x_0 + y_2y_1y_0x_1\overline{x}_0$$

$$(z_0) = y_1\overline{x}_1\overline{x}_0 + \overline{y}_1x_1x_0 + \overline{y}_2y_1y_0\overline{x}_0 + y_2y_1y_0\overline{x}_0 + y_2y_1\overline{y}_0\overline{x}_1 + y_2y_0\overline{x}_1\overline{x}_0 + y_2\overline{y}_1\overline{y}_0x_0 + y_2\overline{y}_1\overline{y}_0x_1 + \overline{y}_2y_1\overline{y}_0x_1x_0$$

După sinteza ecuațiilor obținute se realizează implementarea la nivel RTL din Fig. 16.

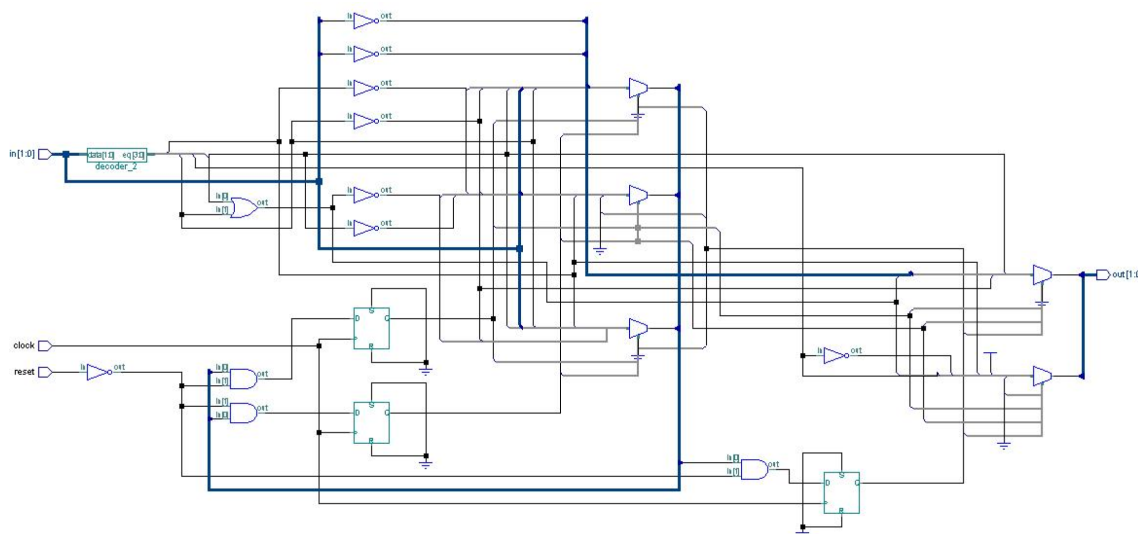


Figura 16. Implementarea fizică la nivel RTL a automatului

6.5.2 Optimizarea logică dependentă de tehnologie

Rețeaua booleană optimizată trebuie implementată utilizînd un set de porți logice predefinite, proiectate şi caracterizate cu atenție într-un pas anterior. Acest set de porți este referit ca o **librărie de porți logice**. Fiecare poartă are un cost care reprezintă suprafața sau întîrzierea sa. Prin urmare, costul poate fi calculat atunci cînd o poartă considerată se potriveşte cu un set de noduri în rețeaua booleană. Pentru a reduce complexitatea procesului de mapare a nodurilor cu un număr arbitrar de fanin-uri în porțile din librărie, rețeaua booleană este mai întîi descompusă în porți logice fundamentale precum porți *NAND*, *NOR*, sau inversoare. Rețeaua booleană descompusă este denumită **graf de expunere**. Acest pas este de cele mai multe ori denumit **descompunere tehnologică**. Fiecare poartă din librărie este reprezentată ca o rețea booleană, unde fiecare nod este de asemenea reprezentat în funcție de cîteva porți logice fundamentale. Grafurile acestor porți sunt denumite **grafuri şablon**. O acoperire a grafului de expunere este o colecție de grafuri şablon astfel încît fiecare nod din graful de expunere este conținut în unul sau mai multe grafuri şablon. Acoperirea este constrînsă în aşa fel încît fiecare ieşire primară este o ieşire a grafului de expunere, şi fiecare intrare necesară pentru un graf şablon este o intrare sau ieşire primară a altui graf şablon din acoperire. Găsirea unei acoperiri în graful de expunere este de obicei **denumită mapare tehnologică**. Problema mapării tehnologice este NP-completă, ceea ce înseamnă că sunt utilizate metode euristice de a rezolva problema. Cele mai folosite metode euristice partiționează graful de expunere în arbori şi este utilizată apoi o abordare dinamică pentru a găsi maparea optimală.

6.6 Optimizarea logică secvențială

Optimizarea logica secventiala a constituit subiectul unui studiu intensiv pe parcursul mai multor decenii de cercetare. Circuitele secventiale pot fi specificate prin modele de limbaje HDL sau sintetizate prin intermediul tehnologiilor VLSI. In ambele cazuri, circuitul secvential poate prezenta un aspect comportamental sau unul structural, sau o combinate din ele. Circuitele sincrone secventiale sunt de cele mai multe ori modelate de o componenta de circuit combinational si registri, dupa cum se poate observa din Fig. 17.

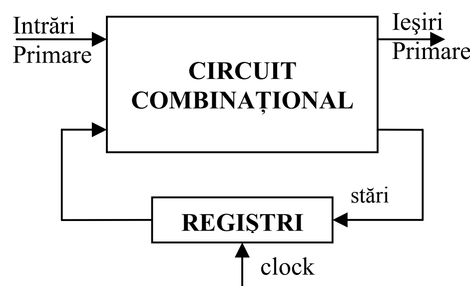


Figura 17. Diagrama bloc a implementării unui automat finit

Dacă comportamentul circuitelor logice combinaționale poate fi exprimat cu ajutorul funcțiilor logice, comportamentul circuitelor secvențiale poate fi prelucrat prin trasarea secvențelor de intrare-ieşire. O modalitate convenabilă de a exprima comportamentul circuitului este prin intermediul modelelor automatelor finite, cum ar fi diagramele de tranziție, din Fig 15. Reprezentările pe baza diagramelor de tranziție sunt la nivel comportamental.

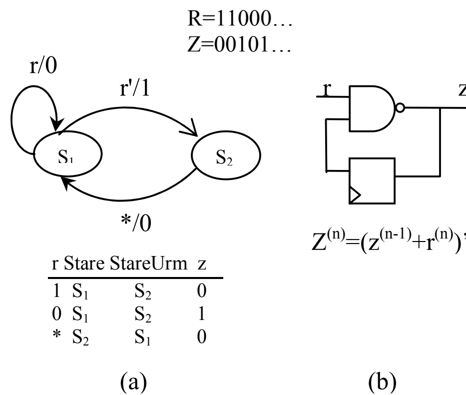


Figura 18. Modele de circuite secvențiale: (a) diagrama de tranziție; (b) rețea logică sincronă

Metodele clasice de optimizare secvențială utilizează diagrame de tranziție sau tabele de tranziție. În timp ce au fost propuse o multitudine de tehnologii de optimizare, această reprezentare a automatelor finite duce lipsa unei corelări directe între manipularea stărilor şi suprafața corespunzătoare ocupată şi respectiv între variațiile de timp sau întîrzieri. În general, tehnologiile de optimizare au scopul de a reduce în complexitate modelul care corelează reducerea suprafeței, dar nu realizează bine îmbunătățirea performanței.

O metodă de reprezentare alternativă a comportamentului circuitului se poate realiza prin intermediul expresiilor logice în termeni de variabile temporale. În cazul circuitelor combinaționale, este de cele mai multe ori convenabil de a exprima comportamentul de intrare/ieşire prin intermediul unui set de expresii locale cu dependențe mutuale. Acest lucru conduce la reprezentarea circuitului în termeni de **rețele logice sincrone** care exprimă interconectarea modulelor combinaționale şi regiştrilor, după cum se poate obs. în Fig. 18 (b). Rețelele logice sincrone prezintă un comportament structural atunci cînd modulele combinaționale corespund porților logice. Ele sunt aspecte hibride structurale/comportamentale ale circuitului atunci cînd modulele sunt asociate cu funcții logice. Unii algoritmi de optimizare recenți pentru circuitele secvențiale utilizează reprezentarea rețelelor de porți logice pentru retiming. În acest caz există o corelație directă între transformările circuitului şi îmbunătățirea suprafeței sau a performanțelor acestuia.

Diagramele de tranziții de stări pot fi transformate în rețele logice sincrone prin **codificarea stărilor** şi pot fi extrase din rețelele logice sincrone prin **extragerea stărilor**. Codificare stărilor defineşte reprezentarea stărilor în termeni de variabile de stare, astfel permițînd descrierea în termeni de rețele. Codurile stărilor neutilizate sunt condiții de **indiferență** care pot fi utilizate pentru optimizarea rețelei. Principalul scop în extragerea stărilor este determinarea stărilor valide (cele vizibile după starea de reset) dintre acele identificate de către toate atributele de polaritate ale variabilelor de stare. Proiectarea sistemelor prin optimizarea secvențială îmbunătățeşte metodele de optimizare în ambele domenii de reprezentare.

6.7 Minimizarea stărilor

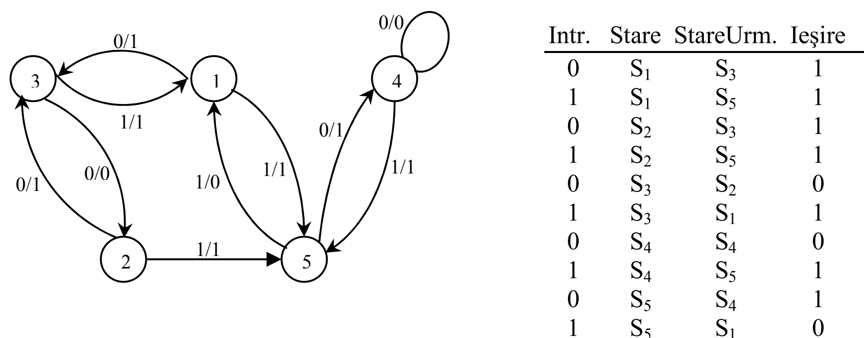
Problema minimizării stărilor presupune reducerea numărului de stări ale automatelor. Acest lucru conduce la reducerea mărimii grafului de tranziție. Reducerea stărilor poate fi corelată cu reducerea numărului de elemente de memorie. Atunci cînd stările sunt codificate cu un număr minim de biți, numărul de regiştri este dat de log_2 din numărul de stări. Reducerea stărilor presupune de asemenea o reducere a numărului de tranziții, rezultînd o reducere a porților logice. Minimizarea stărilor poate fi definită informativ ca o derivare a unui automat finit către unul cu un comportament similar care prezintă un număr minim de stări.[63]

6.7.1 Minimizarea stărilor pentru automate finite complet specificate

Atunci cînd sunt considerate automate complet specificate, funcția de tranziție δ şi funcția de ieşire λ sunt specificate pentru fiecare pereche (intrare,stare) în $X \times S$. Două stări sunt echivalente dacă secvența de ieşire a automatului finit inițializat pentru cele două stări coincide pentru orice secvență de intrare. Echivalența este verificată prin utilizarea rezultatelor teoremei următoare:

Teoremă: Două stări ale unei maşini finite sunt echivalente dacă şi numai dacă, pentru orice intrare, ele au ieşiri identice şi stările următoare corespunzătoare sunt echivalente.

Întrucît echivalența este simetrică, reflexivă şi tranzitivă, stările pot fi partiționate în clase de echivalență. O astfel de partiție este unic identificată. O implementare a unui automat finit cu un număr minim de stări este aceea în care fiecare stare reprezintă doar o singură clasă, sau echivalent unde există doar cîte o stare în fiecare clasă. Minimizarea numărului de stări ale unei maşini complet specificate presupune în acest caz calcularea claselor de echivalență. Acestea pot fi derivate printr-un rafinament iterativ al unei partiții din setul de stări. Fie Π_i , i = 1,2, … , n_s , care denotă partițiile. Pentru început, blocurile partiției Π_1 conțin stări ale căror ieşiri sunt identice pentru fiecare intrare, adică satisfac condiția de echivalență. Apoi blocurile de partiții sunt rafinate iterativ prin împărțiri succesive cu condiția ca toate stările din orice bloc Π_{1+1} au stările următoare în acelaşi bloc al lui Π_i pentru orice intrare posibilă. Atunci cînd iterația converge, spre exemplu $\Pi_{i+1} = \Pi_i$ pentru o anume valoare a lui i, blocurile corespunzătoare sunt clasele de echivalență căutate. Convergența este întotdeauna obținută în cel mult n_s iterații. În cazul limită în care sunt necesare n_s iterații, obținem o partiție unitate, unde toate blocurile au cîte o stare fiecare. Complexitatea algoritmului este $O(n^2_s)$.



Intr.	Stare	StareUrm.	Ieşire
0	S_1	S_3	1
1	S_1	S_5	1
0	S_2	S_3	1
1	S_2	S_5	1
0	S_3	S_2	0
1	S_3	S_1	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_1	0

Figura 19. Diagrama de stări. Tabelul de stări

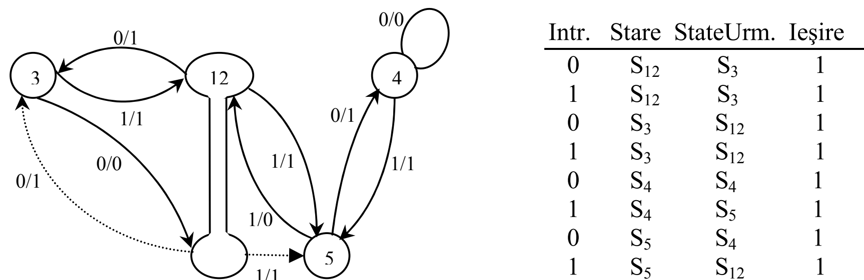
Pentru exemplul reprezentat în Fig. 19, setul de stări poate fi partiționat mai întîi în funcție de valoarea ieşirilor după cum urmează:

$$\Pi_1 = \{\{s_1 , s_2\} , \{s_3 , s_4\} , \{s_5\}\}$$

Apoi este verificat fiecare bloc din Π_1 pentru a putea observa dacă stările următoare corespunzătoare sunt într-un singur bloc din Π_1 pentru orice intrare. Următoarele stări din s_1 şi s_2 sunt identice. Următoarele stări din s_3 şi s_4 sunt în blocuri diferite. Prin urmare, blocul $\{s_3,s_4\}$ trebuie spart, iar partiția Π_1 va deveni de forma:

$$\Pi_1 = \{\{s_1 , s_2\} , \{s_3\}, \{s_4\} , \{s_5\}\}$$

Atunci cînd se verifică blocurile din nou, se găseşte că nu mai există nici un rafinament posibil, întrucît stările următoare din s_1 şi s_2 sunt identice. Prin urmare există patru clase de stări compatibile. Se denotă $\{s_1,s_2\}$ ca s_{12} în tabelul şi diagrama minimala din Fig. 20:



Intr.	Stare	StateUrm.	Ieşire
0	S_{12}	S_3	1
1	S_{12}	S_3	1
0	S_3	S_{12}	1
1	S_3	S_{12}	1
0	S_4	S_4	1
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_{12}	1

Figura 20. Diagrama de tranziție minimală. Tabela de tranziție minimală

În metoda descrisă anterior, rafinarea partițiilor este realizată prin căutarea tranzițiilor stărilor aparținînd blocului luînd în considerare celalte stări. *Hopcroft* a sugerat o metodă de rafinare a partițiilor în care sunt prelucrate tranzițiile din stările blocului considerat. Fie un bloc al unei partiții A $\in \Pi_i$ şi, pentru fiecare intrare, subsetul de stări ale căror stări următoare sunt din A, care sunt notate cu P. Atunci orice bloc B $\in \Pi_i$ este considerat echivalent cu A daca ori $B \subseteq P$ sau $B \cap P = \varnothing$. Dacă nici una din aceste condiții nu este satisfacută, blocul B este împărțit în $B' = B \cap P$ şi $B^2 = B - B \cap P$, iar cei doi împărțitori devin parte a lui Π_{i+1}. Dacă nici un bloc nu necesită o împărțire, atunci partiția defineşte o clasă de stări echivalente.

Exemplu: Fie tabelul din Fig. 20. Setul de stări poate fi partiționat mai întîi în funcție de ieşiri, la fel ca în cazul anterior:

$$\Pi_1 = \{\{s_1 , s_2\} , \{s_3 , s_4\} , \{s_5\}\}$$

Apoi este verificat fiecare bloc din Π_1. Fie A = $\{s_5\}$ şi intrarea 1. Stările ale căror stare următoare este s_5 sunt setul $P = \{s_1 , s_2 , s_4\}$, care este un subset al setului P şi nu necesită împărțiri următoare. Blocul $B = \{s_3 , s_4\}$ nu este un subset din P şi $B \cap P = \{s_4\}$. Prin urmare blocul este spart ca $\{\{s_3\} , \{s_4\}\}$, cu structura următoare,care defineşte 4 clase de stări echivalente:

$$\Pi_1 = \{\{s_1 , s_2\} , \{s_3\}, \{s_4\} , \{s_5\}\}$$

Observaţie: *Metoda Hopcroft* este importantă întrucît are un comportament asimptotic îmbunătăţit. Este de notat faptul că atunci cînd un bloc este considerat că împarte pe celelalte, nu este necesar să fie reconsiderat decît atunci cînd este împărţit. În acest caz, ambii împărţitori vor avea aceleasi rezultate atunci cînd sunt consideraţi la rîndul lor să împartă alte blocuri. Prin urmare, se poate considera cel mai mic dintre ele pentru următoarele iteraţii. Acest lucru este cheia în demonstrarea faptului că algoritmul poate fi executat în $O(n_s log\ n_s)$ paşi.

6.7.2 Minimizarea stărilor pentru automate finite incomplet specificate

În cazul automatelor incomplet specificate, funcţia de tranziţie δ şi funcţia de ieşire λ nu sunt specificate pentru anumite perechi de intrări/ieşiri. În mod similar, condiţiile de **indiferenţă** denota tranziţiile şi ieşirile nespecificate. Ele modeleaza faptul că anumite patern-uri de intrare nu pot apărea în anumite stări sau faptul că anumite ieşiri nu sunt observabile din anumite stări în unele condiţii de intrare.

O secvenţă de intrare este denumită aplicabilă dacă nu conduce la o tranziţie nespecificată oarecare. Două stări sunt compatibile dacă secvenţele de ieşire ale automatului finit iniţializat în cele două stări coincide atunci cînd ambele ieşiri sunt specificate şi pentru orice secvenţă de intrare aplicabilă. Următoarea teoremă este valabilă pentru automatele finite incomplet specificate:

Teoremă: Două stări ale unui automat finit sunt compatibile dacă şi numai dacă, pentru orice intrare, funcţiile de ieşire corespunzătoare se potrivesc atunci cînd ambele sunt specificate şi stările următoare corespunzătoare sunt compatibile atunci cînd ambele sunt specificate.

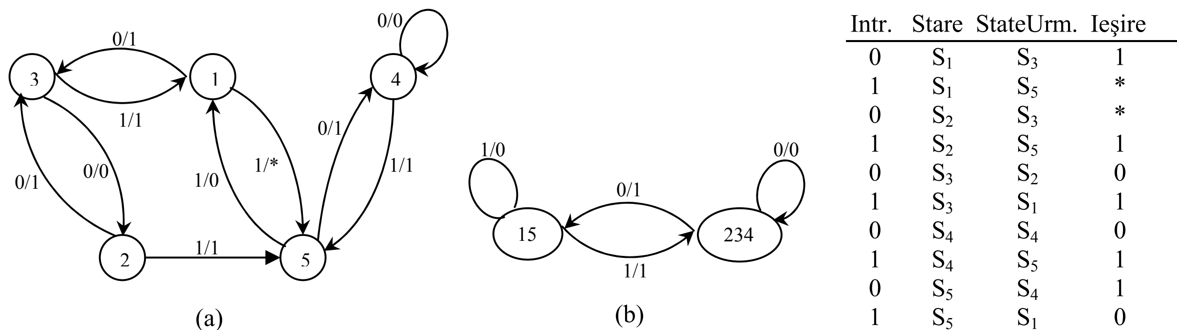
Această teorema formează baza procedurilor iterative pentru determinarea claselor de stări compatibile care sunt duplicatul celor utilizate pentru calcularea stărilor echivalente. Cu toate acestea există două diferenţe majore faţă de cazul automatelor complet specificate.

Prima, compatibilitatea nu este o relaţie de echivalenţă, deoarece compatibilitatea este o relaţie reflexivă şi simetrică dar nu una tranzitivă. O clasă de stări compatibile este apoi definită astfel încît toate stările sale sunt perechi compatibile. Maximul claselor de stări compatibile nu formează o partiţie de seturi de stări. Deoarece clasele se pot suprapune, pot exista soluţii multiple asupra problemei.Rigiditatea problemei provine din acest fapt.

A doua, selecţia unui număr adecvat de clase de compatibilitate pentru a acoperi setul de stări este complicat datorită implicaţiilor dintre clase, deoarece compatibilitatea a două sau mai multe stări poate necesita compatibilitatea altora. Deci selecţia unei clase de compatibilitate pentru a fi reprezentate de către o singură stare poate implica faptul că alte clase trebuie de asemenea luate în consideraţie. Un set de clase de compatibilitate are proprietatea de închidere atunci cînd toate clasele de compatibilitate implicate sunt în acel set sau sunt conţinute de către clasele din acel set.

Exemplu: Fie automatul finit din Fig. 21 (a), descris de tabelul de tranziţii din Fig. 21. Pentru simplicitate este considerată incomplet definită doar funcţia de ieşire λ. După cum se poate

observa mai întîi, înlocuirea articolelor *(*indiferent)* cu valori de 1 ar conduce la un alt automat finit complet specificat. Din păcate există un număr exponențial de automate finite



Intr.	Stare	StateUrm.	Ieşire
0	S_1	S_3	1
1	S_1	S_5	*
0	S_2	S_3	*
1	S_2	S_5	1
0	S_3	S_2	0
1	S_3	S_1	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_1	0

Figura 21. (a) Diagrama de stări. (b) Diagrama de stări minimală. Tabelul de stări

complet specificate în corespondența cu alegerea valorilor * (indiferente). Se consideră compatibilitatea perechilor de stări. Perechea $\{s_1,s_2\}$ este compatibilă. Perechea $\{s_2,s_3\}$ este compatibilă în raport cu compatibilitatea $\{s_1,s_5\}$. Perechea $\{s_1,s_3\}$ nu este compatibilă. Acest lucru demonstrează lipsa de tranzitivitate a relației de compatibilitate. În tabelul următor este descrisă lista de perechi compatibile şi incompatibile:

	Perechi	Perechi Implicate
Compatibile	$\{s_1 , s_2\}$	
Compatibile	$\{s_1 , s_5\}$	$\{s_3 , s_4\}$
Compatibile	$\{s_2 , s_4\}$	$\{s_3 , s_4\}$
Compatibile	$\{s_2 , s_3\}$	$\{s_1 , s_5\}$
Compatibile	$\{s_3 , s_4\}$	$\{s_2 , s_4\}$, $\{s_1 , s_5\}$

Clasele de compatibilitate maximă sunt următoarele:

Clase	Clase implicate
$\{s_1 , s_2\}$	
$\{s_1 , s_5\}$	$\{s_3 , s_4\}$
$\{s_2 , s_3 , s_4\}$	$\{s_1 , s_5\}$

Minimizarea numărului de stări ale unui automat finit incomplet specificat constă în selectarea unui număr suficient de clase de compatibilitate care satisfac proprietatea de închidere asfel încît stările sunt acoperite. Deci problema minimizării stărilor poate fi formulată ca o problema de acoperire şi poate fi rezolvată exact, local sau global, optimal, prin algoritmii corespunzători. Stările unei implementari minimale corespund în acest caz claselor selectate şi numărului lor la cardinalitatea unei acoperiri minimale. Setul de clase compatibile maximal satisface întotdeauna proprietatea de închidere. Prin urmare calculul produce întotdeauna o soluție realizabilă şi nu este necesară o verificare a implicațiilor produse. Din păcate, cardinalitatea poate fi mai mare decît acoperirea minimă sau chiar mai

mare decît cardinalitatea setului original de stări. Acoperirile minime pot implica clase de compatibilitate care nu sunt în mod necesar minimale.

6.8 Codificarea stărilor

Problema codificării sau asignării stărilor constă în determinarea reprezentării binare a stărilor automatului finit. În cazul cel mai general, problema codificării stărilor este complicată de alegerea tipului de registru utilizat pentru memorare, de tip D, T, JK, din care regiştrii D sunt cei mai utilizați.

Codificarea afectează suprafața şi performanțele circuitului. Cele mai cunoscute tehnici de codificare a stărilor au ca scop reducerea complexității circuitului. Complexitatea circuitului este legată de numărul de biți de stocare n_b utilizați pentru reprezentarea stărilor (lungimea codificării) şi de mărimea componentei combinaționale. Măsura celei din urmă este foarte diferită atunci cînd se consideră implementarea bivalentă sau multivalentă a circuitului. Din aceste considerente, tehnologiile de codificare a stărilor au fost dezvoltate independent.

6.8.1 Codificarea stărilor pentru circuite bivalente

Complexitatea circuitului unei reprezentări *sumă de produse* este legată de numărul de intrări, ieşiri şi termeni produs. Pentru implementări bazate pe PLA-uri, aceste numere pot fi utilizate pentru a calcula rapid suprafața circuitului şi lungimea fizică a celei mai lungi căi de propagare, care este în corelație cu întîrzierea căii critice. Numărul de intrări şi ieşiri ale componentei combinaționale este de două ori suma dintre lungimea codării stărilor plus numărul de intrări şi ieşiri primare. Numărul de termeni produs de considerat este mărimea unei reprezentări *sumă de produse* minimale.

Alegerea codificării afectează atît lungimea codificării cît şi mărimea unei acoperiri bivalente. Există $2^{n_b}!/(2^{n_b}-n_s)!$ codificări posibile, iar alegerea celei mai bune dintre ele este o sarcină dificilă. Mărimea reprezentării *sumă de produse* este invariantă la permutarea şi complementarea biților codați. Prin urmare, numărul de coduri relevante poate fi definit astfel:

$$Nc = \frac{\left(2^{n_b}-1\right)!}{\left(2^{n_b}-n_s\right)! \cdot n_b!}$$

Cea mai simplă codificare este codificareă *One-hot*, unde fiecare stare este codificată prin un bit corespunzător setat pe valoarea 1, ceilalți fiind egali cu 0, asftel încît $n_b = n_s$. Codificarea *One-hot* necesită un număr excesiv de intrări şi ieşiri şi s-a demonstrat că nu minimizează mărimea reprezentării *sumă de produse* a componentei combinaționale corespunzătoare. În primele etape ale codificării stărilor a fost pus accentul pe coduri de lungime minimă, utilizînd $n_b = [\log_2 n_s]$ biți pentru reprezentarea setului de stări *S*. Cele mai multe metode euristice clasice pentru codificarea stărilor sunt bazate pe un criteriu de reducere a dependențelor. Au existat şi abordări bazate pe algoritmi genetici pentru evoluarea unui set de soluții optimale pentru codificarea stărilor unei probleme.[51] Problema abordată este de a codifica stările astfel încît variabilele de stare să prezinte un număr cît mai mic de dependențe de cele reprezentînd stări anterioare. Dependențele reduse sunt slab corelate cu minimalitatea într-o reprezentare *sumă de produse*. În anii 1980 a fost introdusă minimizarea simbolică, cu scopul de a rezolva problema codificării stărilor.

Minimizarea unei reprezentări simbolice este echivalentă cu minimizarea dimensiunii unei forme sumă de produse legată de toate codurile care satisfac constrîngerile corespunzătoare. Întrucît există programe de minimizare simbolică sau euristică, la fel ca şi programe de codificare, ele pot fi aplicate rapid problemei codificării stărilor. Modelul automatelor finite necesită soluția atît la problema codificării intrărilor cît şi la problema codificării ieşirilor. Datorită reacției inverse a automatelor finite, se impune necesitatea de consecvență. Perechi de simboluri, intrări şi ieşiri, care corespund aceloraşi stări trebuie să prezinte acelasi cod. Cu alte cuvinte, setul de stări trebuie să fie codificat în timp ce satisface simultan constrîngerile de intrare şi de ieşire.

Exemplu: Fie automatul finit descris în următorul tabel de tranziție:

Intr.	Stare	StareUrm.	Ieşire
0	S_1	S_3	1
1	S_1	S_5	*
0	S_2	S_3	*
1	S_2	S_5	1
0	S_3	S_2	0
1	S_3	S_1	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_1	0

O acoperire simbolică minimală arată astfel:

*	$s_1s_2s_4$	s_3	0
1	s_2	s_1	1
0	s_4s_5	s_2	1
1	s_3	s_4	1

cu următoarele constrîngeri de acoperire:

- s_1 şi s_2 acoperă s_3;
- s_5 este acoperit de către toate celelalte stări

Matricile de codificare corespunzătoare devin:

$$A=\begin{bmatrix}1&1&0&1&0\\0&0&0&1&1\end{bmatrix}\qquad B=\begin{bmatrix}0&0&1&0&1\\0&0&1&0&1\\0&0&0&0&1\\0&0&0&0&1\\0&0&0&0&0\end{bmatrix}\qquad E=\begin{bmatrix}1&1&1\\1&0&1\\0&0&1\\1&0&0\\0&0&0\end{bmatrix}$$

E este o codificare care satisface simultan ambele seturi de constrîngeri, unde fiecare linie reprezintă o stare. O acoperire codificată a componentei combinaționale a automatului finit este:

*	1**	001	0
1	101	111	1
0	*00	101	1
1	001	100	1

În timp ce minimizarea simbolică şi codificarea constrinsă furnizează un spaţiu de soluții pentru problema asignării stărilor pentru circuitele bivalente, se pot observa anumite limitări ale acestei abordări. Soluția de lungime minimă compatibilă cu un set de constrîngeri poate necesita un număr de biți mai mare decît [$\log_2 n_s$]. În practică, rezultatele experimentale au demonstrat că doar cîțiva biți în plus sunt necesari pentru a satisface toate constrîngerile. În plus, constrîngerile pot fi relaxate pentru a satisface limitările lungimii codificării. Prin urmare, implemtarea circuitului propus poate fi căutată printr-o echilibrare a numărului de Intrări/Ieşiri şi a numărului de termeni produs.

Exemplu: Fie un automat finit a cărei acoperire simbolică minimală este:

00	$s_1 s_2$	s_3	100
01	$s_2 s_3$	s_1	010
10	$s_1 s_3$	s_2	001

Nu există constrîngeri ale codificării ieşirii. Cu toate acestea, satisfacerea constrîngerilor codificării intrărilor necesită cel puțin 3 biți. O codificare posibilă este: s_1 = 100; s_2 = 010; s_3 = 001. Prin urmare poate fi construită o acoperire cu cardinalitatea 3 şi n_b = 3. PLA-ul corespunzător ar avea 3 linii şi 11 coloane.

Alternativ, se poate alege să nu satisfacă o anumită constrîngere pentru a obține o codificare pe o lungime de 2 biți. Presupunînd că se împart implicanții simbolici top în 2, numiți 00 s_1 s_3 100 şi 00 s_2 s_3 100, atunci este posibilă următoarea codificare pe 2 biți:s_1 = 00; s_2 = 11; s_3 = 01. În acest caz se obține o acoperire cu cardinalitatea 4 şi n_b = 2. PLA-ul corespunzător ar avea 4 linii şi 9 coloane.

6.8.2 Codificarea stărilor pentru circuite multivalente

Măsura suprafeței totale este legată de numărul de biți de codificare (regiştri) şi de numărul de literali din rețeaua logică. Întîrzierea corespunde lungimii căii critice din rețea. Pentru date, au fost dezvoltate doar metode euristice şi de optim global pentru calcularea codificării stărilor care optimizează suprafața.

Cea mai simplă metodă de abordare este de a calcula o asignare optimală a stărilor pentru un model bivalent şi apoi de a restructura circuitul cu algoritmi de optimizare combinațională. În ciuda faptului că alegerea codurilor stărilor este realizata în timp ce se ia în considerare un model diferit, rezultate experimentale au demonstrat rezultate surprinzător de bune. O posibilă explicație este faptul că multe automate finite au tranziții înşelătoare şi funcții de ieşire care pot fi implementate cel mai bine în cîteva stagii.

Dificultatea codificării stărilor pentru modele logice multivalente este cauzată de marea varietate de transformări disponibile pentru a optimiza o rețea logică şi datorită problemei estimării literalilor. Prin urmare, tehnicile de codificare au fost cosiderate în conexiune cu o transformare logică particulară. Devadas, a propus o metodă de codificare euristică care privilegiază extragerea cuburilor comune; metoda a fost apoi optimizată prin extragerea subexpresiilor optimale şi legarea lor mai apoi cu codificarea.[32] Este convenabil de a păstra distanța dintre codurile corespuzătoare mică, deoarece aceasta coreleaza cu dimensiunea unui cub comun care poate fi extras.

Exemplu: Pentru un automat finit dat, cu un set de stări S = $\{s_1, s_2, s_3, s_4, s_5\}$, se consideră două stări s_1 şi s_2 cu o tranziţie în aceeaşi stare s_3 cu intrările *i* respectiv *i'*. Se presupune utilizarea unei codificări pe 3 biţi. Se codifică ambele stări cu coduri adiacente, denumite 000 şi 001. Acestea corespund la cuburile *a'b'c'* şi *a'b'c*, unde {a,b,c} sunt variabilele de stare. Apoi tranziţia poate fi scrisă ca *i'a'b'c'+ ia'b'c*, sau echivalent *a'b'*(*i'c'+ ic*). Nu s-ar fi putut extrage nici un cub dacă s_2 ar fi fost codificat ca 111, şi s-ar fi putut extrage un cub mai mic dacă s-ar fi ales 011.

Problema de codificare este modelată de un graf complet de pondere a marginilor, unde muchiile sunt o corespondenţă unu la unu cu stările. Marginile denotă proximitatea codului dorit pentru perechile de stări şi este determinat prin scanarea sistematică a tuturor perechilor de stări. Codificarea stărilor este determinată de îngrădirea acestui graf într-un spaţiu boolean de dimensiuni apropiate. Deoarece îngrădirea grafului este o problemă intractabilă, au fost utilizaţi algoritmi euristici pentru a determina o codificare unde distanţe de perechi de coduri se corelează în mod rezonabil de bine cu marginile (cu cît este mai mare marginea cu atît este mai mică distanţa).

Există în continuare două complicaţii majore. Prima, atunci cînd se consideră coduri de două stări cu tranziţii în aceeaşi stare următoare, mărimea cubului comun poate fi determinată de distanţa codului, dar numărul de extrageri de cuburi posibile depinde de codificarea stărilor următoare. Prin urmare, suprafaţa cîştigată nu poate fi legata direct de tranziţii, şi marginile sunt doar o indicaţie imprecisă a cîştigului posibil per ansamblu prin extragerea cuburilor. În al doilea rînd, extragerea cuburilor comune interacţionează cu fiecare.

Pentru a face faţă acestor dificultăţi, au fost propuşi doi algoritmi euristici de către Devadas. Aceştia utilizează un graf complet de pondere a marginilor, marginile fiind determinate în mod diferit. În primul algoritm, denumit „*orientat fan-out*", perechile de stări care au tranziţii în aceeaşi stare următoare au primit margini cu pondere superioară (pentru a atinge coduri apropiate). Marginile sunt calculate de o formulă complexă care ia în calcul pattern-urile de ieşire. Această abordare încearcă să maximizeze mărimea cuburilor comune din funcţia codificată pentru starea următoare. În al doilea algoritm, denumit „*orientat fan-in*", perechile de stări cu tranziţii care vin din aceleaşi stări primesc margini cu pondere superioară. Din nou o regulă complexă determină marginea în timp ce ia în calcul patern-urile de intrare. Această strategie încearca să maximizeze numărul de cuburi comune din funcţia codificată pentru starea următoare.

Exemplu: Se consideră tabelul din exemplul anterior şi algoritmul orientat fan-out. Mai întîi este construit un graf complet de 5 muchii şi apoi sunt determinate ponderile marginilor, considerate ponderi binare pentru simplicitate. Prin urmare o pondere mare este considerată 1. Se ia, spre exemplu, perechea de stări $\{s_1, s_2\}$, unde fiecare stare are o tranziţie către s_3. Ponderea 1 este asignată marginii $\{v_1, v_2\}$. Prin scanarea tuturor perechilor de stări, marginile $\{\{v_1, v_2\}, \{v_2, v_4\}, \{v_1, v_4\}, \{v_4, v_5\}, \{v_3, v_5\}\}$ pimesc ponderea unitate, în timp ce marginile rămase primesc ponderea 0. Codificarea reprezentată prin următoarea matrice reflectă proximităţile necesare specificate de către ponderi:

$$E = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Prin inlocuirea codurilor din tabelă şi ştergerea acelor linii care nu contribuie la tranziția codificată şi la funcția de ieşire, se obține:

1	110	111	1
0	000	001	0
1	000	011	1
0	011	110	1
0	001	110	1
1	001	001	0

Fie variabila de intrare *i*, variabilele de stare *a, b, c* şi funcțiile de tranziție codificate f_a, f_b, f_c. Expresia $f_a = iabc' + i'a'bc + i'a'b'c$ poate fi rescrisă sub forma $f_a = iabc' + i'a'c(b+b') = iabc' + i'a'c$. Cubul comun $i'a'c$ este legat de codurile lui $\{s_4 , s_5\}$, care sunt adiacente. Cubul $i'a'c$ nu este în conjuncție cu o expresie, deoarece în acest caz particular intrările primare se potrivesc în corespondență cu tranzițiile de la $\{s_4 , s_5\}$ luat în considerație.

Presupunînd că s_5 are codul 101, sau echivalent $ab'c$, atunci $f_a = iabc' + i'a'bc + i'ab'c$, care poate fi rescris ca $f_a = iabc' + i'c(a'b + ab')$. Expresia pentru f_a este de dimensiune mai mare decît cea anterioară. Aceasta este datorită distanței mai mari a codurilor lui $\{s_4 , s_5\}$.

Este important de menționat că în timp ce extragerea cuburilor reduce numărul de literali, multe alte transformări interactive pot obține acelaşi obiectiv. Prin urmare, tehnica de codificare exploatează doar o metodă pentru reducerea complexității unei rețele logice multivalente.

6.9 Hazardul circuitelor

Hazardul reprezintă o deviere posibilă a funcției de ieşire de la comportamentul specificat inițial. Există două categorii de baza ale hazardului: combinațional şi secvențial. La rîndul său, hazardul combinațional poate fi clasificat în static şi dinamic.

Definiție: Hazardul **static** se referă la mai multe tranziții de tipul 0-1-0 (sau 1-0-1) în timp ce comportamentul specificat trebuie să fie o valoare statică de 0 sau 1. Hazardul de tip 0-1-0 este denumit hazard de tip 0, iar cel de tip 1-0-1 este denumit hazard de tip 1.

Definiție: Hazardul **dinamic** se referă la tranziții multiple de la 0 la 1 sau invers în timp ce comportamentul specificat trebuie să fie o singură tranziție de la 0 la 1 sau invers.

Hazardurile mai pot fi de asemenea clasificate în funcție de natura lor în două categorii: hazarduri logice şi hazarduri funcționale (Eichelberger).

Definiție: Hazardurile logice (sau funcționale) se referă la hazarduri dinamice şi statice în funcții care nu răspund monolitic la toate schimbările de la intrare. Hazardurile funcționale pot fi eliminate doar prin alterarea comportamentului specificat.

Definiție: Un „glitch” se referă la manifestarea unui hazard.

Definiție: Atunci cînd două sau mai multe variabile de stare se schimbă în acelaşi timp, se spune că circuitul are un hazard secvențial sau un hazard de „curse”. Dacă ieşirea sau stările circuitului depind de efectele unei „curse” (ordinea schimbărilor), atunci „cursa” este critică.

Definiție: Un hazard „esențial” se referă la o cursă între intrări şi variabilele de stare care pune circuitul într-o stare finală incorectă.

Hazardurile esențiale au loc atunci cînd într-un circuit o variabilă de stare se schimbă înaintea schimbării unei mărimi de intrare, sau atunci cînd schimbarea a 3 mărimi de intrare diferite conduce la aceeaşi stare finală ca şi cum s-a schimbat doar o singură mărime de intrare.

Definiție: Un semnal este stabil dacă acel semnal precum şi toate semnalele din fanin-ul său tranzitiv au atins valorile stărilor de echilibru.

Hazardul dinamic este cel mai greu de eliminat atunci cînd este permis mai multor semnale de a îşi schimba comportamentul în acelaşi timp. Dacă hazardul static este eliminat din circuit, atunci va fi eliminat automat şi hazardul dinamic. Cea mai simplă metodă de a elimina hazardul static este utilizarea unei diagrame Karnaugh (Fig. 22). Hazardurile de timp pot produce eşecuri intermitente sau aleatoare în funcționarea corectă a circuitului. Tipul de eşec în funcționarea circuitului depinde de intrările circuitului şi de cît de rapid îşi schimbă starea. O altă metodă de eliminare a hazardului de timp pentru funcționarea corectă a circuitului în timpul utilizării sale este de a recalcula tactul de ceas pentru obținerea semnalelor finale. Recalcularea clock-ului nu elimină „glithch”-urile, în schimb elimină eşecurile în funcționarea circuitului. Această soluție este de obicei aleasă de producătorii care sunt nesiguri de sursa de apariție a „glitch”-urilor, sau cum să stopeze apariția lor pe parcursul dezvoltării circuitului. Soluționarea problemei prin utilizarea diagramelor Karnaugh rezultă în utilizarea mai multor componente, pe cînd recalcularea perioadei de tact necesită utilizarea unui bistabil în plus.

În arhitectura calculatoarelor, un hazard este o problemă potențială care poate apărea într-un procesor cu arhitectură pipeline. Există de obicei trei tipuri de hazarduri: hazard de date, hazard de branşă (subdomeniu) şi hazard structural. Într-un procesor cu o arhitectură pipeline instructiunile sunt executate în mai multe stagii, astfel încît în orice moment sunt executate mai multe instrucțiuni în paralel, iar ordinea de execuție a instrucțiunilor nu poate fi efectuată în ordinea dorită. Un hazard are loc atunci cînd două sau mai multe din aceste instrucțiuni simultane (posibil în afara ordinii de execuție) intră în conflict.

AB	00	01	11	10
s=0	0	1	1	0
s=1	0	0	1	1

Figura 22. Soluționarea hazardului static utilizînd diagrame Karnaugh

Un hazard este definit ca orice tranziție a unui semnal care nu a fost prevazută de proiectant. Motivele apariției unui hazard pot fi influențate de către:

- distribuția delay-urilor (timpilor de răspuns) între porțile unui circuit
- funcția logică implementată de proiectant

Timpul de răspuns al unei porți logice depinde de procesul de fabricație şi de factori legați de mediul de lucru precum temperatura, expunerea la radiațiile ultraviolete, şi altele. Acest lucru face dificil pentru un proiectant ca să se asigure că circuitul nu va prezenta hazard. Hazardul poate apărea în circuitele secvențiale la elementele de memorie, dar tactul de ceas este prevăzut astfel încît să prevină apariția hazardului între blocurile secvențiale şi cele combinaționale. Pe parcursul proiectării unui circuit sincron trebuie acordată o atenție sporită rutării liniilor de ceas (clock), astfel încît acestea trebuiesc distribuite de la pad-urile (sursele) de intrare către elementele de memorie dintr-un chip. Propagarea semnalului electric pe parcursul liniilor de semiconductor lungi conduce la degradarea semnalului.

Acest tip de probleme care apar şi trebuiesc rezolvate în proiectarea circuitelor sunt atribuite tăierii semnalului de tact sau „clock-skew". Circuitele asincrone nu mai prezintă probleme legate de „clock-skew" întrucit în proiectarea acestora nu mai sunt necesare semnale de ceas care să trebuiască rutate pe tot chip-ul. [110]

O problemă importantă în proiectarea circuitelor VLSI este puterea disipată pe suprafața circuitului. Pad-urile unui circuit de obicei permit disiparea puterii în proporție de peste 50%. Restul puterii este disipată de evaluarea părții logice din regiunile combinaționale, în liniile de tact conducătoare şi datorită curenților de scurgere. În circuitele asincrone, puterea este disipată în comutarea porților precum şi în circuitele adiționale necesare pentru a detectarea completă a un semnal. Puterea disipată poate fi clasificată în:

- pasivă, cauzată de curenți de scurgere, predominantă în aplicațiile portabile
- dinamică, cauzată de activitatea de comutare, predominantă în circuitele de calcul performante

Puterea disipată depinde de asemenea de tehnologia utilizată (CMOS, ECL, BiCMOS, etc) precum şi de tehnica de proiectare.

Avantajele proiectării circuitelor sincrone sunt:

- Hazardul care apare în interiorul circuitului nu afectează ieşirile
- Modularitatea regiunilor combinaționale datorită izolării relative furnizate de către elementele de memorie care intervin între blocurile combinaționale
- Uşurința de testare. Tehnologii ca boundary-scan şi Jtag pot fi utilizate pentru a testa şi detecta erorile de fabricație
- Sunt relativ insensibile la întîrzierile care apar în porțile logice. Dacă întîrzierile din porți nu respectă limitele impuse, circuitul poate să nu funcționeze la frecvența dorită, dar poate funcționa la o frecvență mai joasă

Dezavantajele circuitelor sincrone sunt:

- Suprafața ocupată este mai mare, datorită prezenței elementelor de memorie precum şi rutării semnalelor de ceas
- Trebuie acordată o atenție sporită în distribuirea semnalelor de ceas pe parcursul circuitului

Capitolul 7

Metode de estimare a efortul logic în proiectarea circuitelor

În proiectarea circuitelor complexe sunt necesare cunoştinţe avansate despre poziționarea geometrică, redimensionarea tranzitoarelor, gradul de încărcare capacitivă, densitatea de integrare, optimizarea circuitelor, etc. Problemele care apar în cadrul acestui „flow” de proiectare sunt variate şi de diverse complexități. O problema importantă este optimizarea rețelelor logice arbitrare astfel încît să permită un timp de raspuns cît mai mic fără a fi necesare simulările pentru detectarea tuturor erorilor din faza de proiectare.

Studiul efortului logic în optimizarea circuitelor a permis calculul numărului de stagii logice necesare pentru a găsi cea mai rapidă implementare a unei funcții logice. Metoda descoperă dimensiunile potrivite ale tranzistoarelor în fiecare etapă pentru a obține cele mai bune performanțe per ansamblul circuitului. De asemenea furnizează un model care poate fi aplicat în etapele incipiente ale proiectării pentru a alege între structuri alternative diferite fără a face eforturi de simulare intensive.

Metoda atribuie un efort logic fiecărei funcții logice. Pentru efortul logic al unui inversor este atribuită ca exemplu valoarea 1. Pentru orice altă funcţie logică, efortul logic descrie nivelul de eficiență curentă față de cea a inversorului la producerea ieşirii curente, pentru o cantitate echivalentă a capacității de intrare. Efortul unei funcții logice depinde în principal de topologia circuitului şi într-o mai mică măsura de proprietățile electrice a procesului de fabricație folosite în proiectarea sa. În tehnologia CMOS, efortul logic pentru fiecare intrare a unei funcții logice comune cu 2 intrări variază de la 4/3 pentru funcția *NAND* pînă la 4 pentru funcția *XOR*. Efortul logic pentru funcții cu mai mult de 2 intrări este în general mai mare. Eforturile de la fiecare nivel logic pot fi combinate pentru a calcula efortul logic al întregii rețele logice. În cazul în care mai multe dispozitive logice pleacă de la o sursă comună, efortul total implică suma eforturilor dispozitivelor conduse. Circuitele complexe, cu un efort logic general mai mic, pot fi făcute să meargă mai rapid decît circuitele logice echivalente cu un efort logic mai mare.

7.1 Estimarea timpului de răspuns pentru o poartă logică

Metoda analizei efortului logic reformulează un model convențional simplu al timpului de răspuns într-o poartă logică *CMOS* şi introduce de asemenea un număr de noțiuni noi. Întîrzierea într-o poartă logică poate fi exprimată ca suma a 2 componente:

- o parte fixă denumită întîrzierea parazitică, p
- o parte proporțională cu încărcarea de pe ieşirea porții, denumită efortul de aşteptare, f

Expresia pentru timpul de aşteptare este $d = f + p$. (1)

Expresii precum aceasta exprimă timpul de aşteptare în unități de timp τ. Efortul de întîrziere depinde parţial de încărcare şi partial de proprietățile porții logice care conduce încărcarea. Se introduc 2 termeni legați de aceste efecte: efortul logic g, precum şi efortul electric, h. Efortul timpului de aşteptare a porții logice este dat de produsul celor 2 cantități, notat cu $f = g \times h$. (2)

Funcția efortului logic capturează efectul topologiei porții logice şi este independentă de mărimea tranzistoarelor din circuit. Efortul electric descrie modalitatea în care mediul electric al porții logice afectează performanța, precum şi modalitatea în care mărimea tranzistoarelor din poarta logică determină capacitatea de încărcare a conductorului. Efortul electric este definit prin: $h = C_{out} / C_{in}$, (3)

unde C_{out} este capacitanta care încarcă poarta logică şi C_{in} este capacitanta care este prezentată de către poarta logica la unul din terminalele sale de intrare.

Combinînd ecuațiile (1) şi (2) se obține ecuația de bază care modelează întîrzierea dintr-o poartă logică, în unități de timp τ astfel:

$$d = g \times h + p, \quad (4).$$

Ecuația (4) arată clar că atît efortul logic cît şi efortul electric contribuie la întîrzierea timpului de răspuns în acelaşi fel. Se poate observa că p şi g sunt independente de mărimea tranzistoarelor din poarta logică, în timp ce h se referă direct la mărimea tranzistoarelor. Efortul logic g exprimă efectele topologiei circuitului legate de întirziere indiferent de încărcare sau mărimea tranzistoarelor. Eforul logic este util deoarece depinde doar de topologia circuitului. Valorile efortului logic pentru cîteva porți logice CMOS sunt descrise în tabelul 9.

Tip poartă	**Număr de intrări**					
Inversor	*1*	*2*	*3*	*4*	*5*	*N*
NAND		*4/3*	*5/3*	*6/3*	*7/3*	*(n+2)/3*
NOR		*5/3*	*7/3*	*9/3*	*11/3*	*(2n+1)/3*
Multiplexor		*2*	*2*	*2*	*2*	*2*
XOR (par)		*4*	*6-12*	*16-32*		

Tabelul 9. Efortul logic pentru intrări porți *CMOS* statice

Efortul electric este de obicei exprimat ca un raport al mărimii tranzistoarelor mai degrabă decît al capacităților actuale. Dacă se presupune că toate tranzistoarele au aceeasi lungime minimă, atunci capacitanta unei porți de tranzistor este proporțională cu mărimea ei. Deoarece majoritatea porților logice conduc alte porți logice, amîndouă mărimile C_{in} şi C_{out} pot fi exprimate în termeni ai dimensiunilor tranzistoarelor. Dacă capacitanța de încărcare include capacitanțe reziduale datorită conexiunilor sau încărcărilor externe, atunci această capacitanță trebuie convertită într-o mărime a tranzistoarelor echivalentă.

Întîrzierea parazită a unei porți logice este fixă, independent de mărimea porții logice şi de capacitanța de de încărcare pe care o conduce. Această întîrziere este o formă de supraîncărcare pe care o are orice poartă logică. Contribuția principală a întîrzierii parazitice o aduce capacitanța regiunilor sursa/drena a tranzistoarelor care conduc ieşirea porții logice. În tabelul 10 sunt estimate întîrzierile parazitice pentru citeva tipuri de porți logice. Întîrzierile parazitice sunt date ca multipli ai întîrzierii parazitice pentru un inversor, notat cu p_{inv}. O valoare tipică pentru p_{inv} este 0.6 unități de întîrziere.

Gate type	**Parasitic delay**
inverter	p_{inv}
n-input NAND	np_{inv}
n-input NOR	np_{inv}
n-way multiplexer	$2np_{inv}$
XOR, XNOR	$4p_{inv}$
3-input majority	$6p_{inv}$

Tabelul 10. Estimarea întîrzierii parazitice pentru diferite tipuri de porți logice

7.2 Rețele logice cu mai multe stagii

Metoda efortului logic este aplicată în 2 moduri diferite în proiectarea rețelelor logice cu mai multe stagii. Metoda permite descoperirea a celui mai bun număr de stagii pentru utilizarea în rețea şi demonstrează faptul că se poate obține cea mai mică întîrziere generală prin egalizarea efortului întîrzierii în fiecare stagiu din calea aleasă.

Noțiunile de efort logic şi electric se pot generaliza cu uşurință pentru rețele cu mai multe stagii. Efortul logic de-a lungul unei căi este compus din multiplicarea eforturilor logice ale tuturor porților logice din calea respectivă. Se notează cu *G* calea efortului logic. Efortul electric calculat pe parcursul unei rețele este simplul raport dintre capacitanța care încarcă ultima poartă logică de pe traseu la capacitanța de intrare a primei porți din cale. Se utilizează în continuare simbolul *H* pentru a indica efortul electric de cale.

De asemenea se introduce un nou tip de efort, denumit efort de împrăştiere, pentru a evalua numărul de fanout-uri dintr-o rețea logică. În mod normal fanout-ul este tratat ca o formă de efort electric. Atunci cînd o poartă logică conduce multe încărcări de pe traseu, se sumează capacitanțele lor pentru a obține un efort electric. Atunci cînd avem de a face cu fanout-ul într-o rețea logica, o parte din curentul conductor disponibil este direcționat pe calea pe care se face analiza. Se defineşte efortul de împrăştiere *b* la ieşirea unei porți logice ca fiind:

$$b = \frac{C_{on} + C_{off}}{C_{on}} \qquad (5)$$

unde C_{on} este definit ca şi capacitanță de intrare pentru următoarea poartă logica de pe parcursul căii analizate iar C_{off} este definit ca şi capacitatea conexiunilor de fanout care conduc în afara căii. În cazul în care fanout-ul este egal cu 1, efortul de imprăştiere este egal cu 1. Efortul de imprăştiere pentru o întreagă cale *B*, este egal cu produsul efortului de împrăştiere pentru fiecare stagiu de pe parcursul căii alese. În continuare se defineşte efortul de cale, *F*. Ecuația care defineşte efortul de cale pentru o singură poartă logică este:

$$F = G \times B \times H \qquad (6)$$

Cu toate că nu este o masură directă a întîrzierii de pe parcursul căii, efortul de cale deține cheia minimizarii intirzierilor. Se poate observa că efortul de cale depinde doar de topologia circuitului şi de încarcarea lui, dar nu depinde de mărimile tranzistoarelor utilizate în porțile logice de pe parcursul rețelei logice. Mai mult, efortul rămîne neschimbat dacă sunt adaugate sau scoate inversoarele din cale, întrucît efortul logic al unui inversor este egal cu 1.

Întîrzierea de cale, *D*, este egală cu suma întirzierilor fiecăruia din cele *N* stagii ale logicii din cale. La fel ca şi în cazul expresiilor pentru calculul intirziererii din cazul cu un singur stagiu, se poate distinge calea efortului intirzierii de cale, D_F , precum şi întîrzierea parazitică de cale, *P*:

$$D = \sum d_i = D_F + P \qquad (7)$$

Întîrzierea efortului de cale este simplu $D_F = \sum g_i h_i$ iar efortul parazitic de cale este $P = \sum P_i$. Optimizarea proiectării unei rețele logice porneşte de la un rezultat foarte simplu: întîrzierea de cale este minimizată atunci cînd fiecare stagiu din cale are acelasi efort de stagiu, *f*. Acest rezultat este obținut prin minimizarea lui *D* prin varierea lui h_i ținind cont de faptul că efortul electric de cale, *H*, este fix. Această întîrziere minimă este atinsă atunci cînd efortul de stagiu este:

$$\hat{f} = g_i \times h_i = F^{\frac{1}{N}} \qquad (8)$$

Simbolul $\hat{f}$ indică o expresie care a atins nivelul minim de întîrziere. Combinînd aceste ecuații, se obține principalul rezultat al metodei de calcul al efortului logic, care este o expresie pentru întîrzierea minimă realizabilă pe parcursul unei căi:

$$\hat{D} = N \times F^{\frac{1}{N}} + P = N \times (G \times B \times H)^{\frac{1}{N}} + P \qquad (9)$$

În urma calculării efortului logic, de imprăştiere şi electic, se poate obține o estimare a timpului de aşteptare minim realizabil într-o rețea logică. Pentru a putea egaliza efortul apărut la fiecare nivel într-o cale, trebuie aleasă marimea potrivita a tranzistoarelor pentru fiecare stagiu logic de pe parcursul căii curente.

Prin combinarea ecuatiilor precedente obținem că fiecare stagiu logic este proiectat astfel încît să existe relația:

$$\hat{h}_i = \frac{F^{\frac{1}{N}}}{g_i} \qquad (10)$$

Această relație este folosiă pentru a putea calcula marimile tranzistoarelor, pornind de la începutul căii şi aplicînd această ecuație la fiecare poartă logică de-a lungul căii alese.

7.3 Alegerea lungimii unei căi

Cu toate că egalizarea efortului generat de fiecare stagiu dintr-o cale minimizează întîrzierea pentru o cale dată, întîrzierea poate uneori fi redusă mai mult prin ajustarea numărului de stagii dintr-o cale.

Pentru o cale cu porți logice care conțin n_1 stagii, la care se adaugă n_2 inversoare adiționale pentru a obține o cale cu un număr total de stagii $N = n_1 + n_2$. Se presupune că cele n_1 stagii originale nu pot fi alterate cu excepția scalării întrucît ele realizează funcții logice necesare, în timp ce numărul de inversoare poate fi alterat dacă este necesar, pentru a reduce întîrzierea. Cu toate că păstrarea funcției logice corecte necesită utilizarea unui număr par de inversoare, se presupune că se poate acomoda un număr impar de inversoare prin schimbarea funcției logice dacă acest lucru este necesar. Se presupune că efortul de cale $F = G \times B \times H$ este cunoscut: eforturile logice şi de împrăştiere sunt proprietăți ale stagiilor logice n_1 care nu vor fi alterate prin adăugarea de inversoare, precum şi faptul că efortul electric este determinat de capacitățile de intrare şi de încărcare necesare. Întîrzierea minimă a N stagii este suma întîrzierilor în stagiile logice şi în stagiile de inversoare:

$$\hat{D} = N \times F^{\frac{1}{N}} + \left(\sum_{i=1}^{n_1} p_i \right) + (N - n_1) \times p_{inv} \qquad (11)$$

Primul termen este întîrzierea obținută prin distribuirea efortului în mod egal intre cele N stagii. Următorul termen reprezintă întîrzierea parazitică a porților logice, iar cel de al treilea termen reprezintă întîrzierea parazitică a inversoarelor. Prin diferențierea expresiei precedente în raport cu N şi prin setarea rezultatului egal cu zero, se obține următoarea ecuație:

$$\frac{\partial \hat{D}}{\partial N} = \frac{-F^{\frac{1}{N-1}}}{N^2} + F^{\frac{1}{N}} + p_{inv} = 0 \quad (12)$$

Se defineşte $\hat{N}$ ca soluție a acestei ecuații, unde $\hat{N}$ reprezintă numărul de stagii utilizabile pentru obținerea celei mai mici întîrzieri. Dacă se defineşte că $\rho = F^{\frac{1}{\hat{N}}}$ este efortul produs de fiecare stagiu, unde numărul de stagii este ales pentru a minimiza întîrzierea, soluția la această ecuație poate fi exprimată astfel:

$$p_{inv} = \rho(1 - \ln \rho) = 0 \quad (13)$$

Cu alte cuvinte, cel mai rapid circuit este unul în care fiecare stagiu de parcursul căii poarta un efort egal cu ρ, unde ρ este soluția ecuației precedente. Prin urmare, ρ poate fi denumit un efort de stagiu optimal.

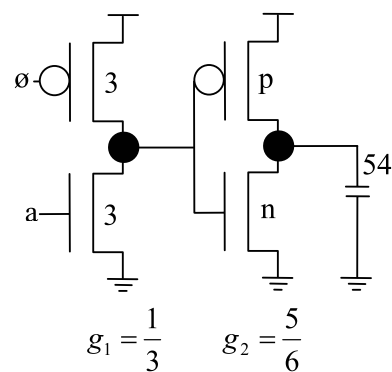
Este importantă înțelegerea relației dintre ρ şi $\hat{f}$ întrucît fiecare dintre ele specifică efortul de stagiu necesar pentru atingerea întîrzierii cele mai mici. Ecuațiile pentru $\hat{f}$ determină cel mai bun efort de stagiu atunci cînd numărul de stagii N este cunoscut.

Prin contrast, valoarea ρ, care este o constantă, independent de proprietățile unei căi, reprezintă o întîrziere de stagiu ideală, care nu poate fi atinsă într-o cale actuală.

Ecuația (13) arată că efortul optim ρ este o funcție a întîrzierii parazitice dintr-un inversor. Întirzierea este relativ independentă de numărul de stagii. Într-un circuit cu unul sau mai multe stagii în plus sau în minus va face o mică diferență, dacă sunt utilizate mărimile de tranzistoare potrivite. Diferențele mari se simt doar în cazul în care sunt necesare foarte putine stagii în proiectarea circuitului. [79]

Exemplu de calcul al efortului logic pentru un buffer.

- Constrîngeri:
  - Capacitanța de intrare maximă = 3
  - Încărcarea = 54
- Efortul logic: $G = \frac{1}{3} \times \frac{5}{6} = \frac{5}{18}$
- Efortul de împrăştiere: B = 1
- Efortul electric: $H = \frac{54}{3} = 18$
- Efortul de cale: $F = \left(\frac{5}{18}\right) \times 1 \times 18 = 5$
- Efortul de stagiu: f = $\sqrt{5}$ =2.23
- Mărimea tranzistoarelor:
  - n = 4
  - p = 6

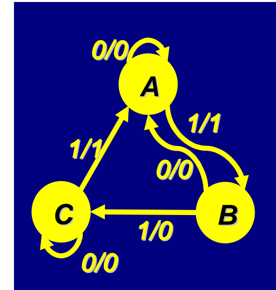


$g_1 = \frac{1}{3}$ $\quad g_2 = \frac{5}{6}$

Figura 23. Calculul efortului logic pentru un buffer

7.4 Optimizarea funcției de fanin a unui circuit logic

Algoritm de optimizare pentru Fanin



- Matricea stărilor de tranziție $S_{|s|x|s|}$:
  - Linie: una pentru starea următoare
  - Coloana: una pentru starea curentă
  - Valoare: (ne-negativă) numărul de arce care pleacă din starea s_j (coloana) către starea s_i (rînd)
- Matricea de intrare $X_{|s|x|I|}$:
  - Rînd: unul pentru starea curentă
  - Coloana: $|I|$ pentru intrare
  - Valoare: (ne-negativă) numărul de arce care intră în starea s_i (rînd) cu intrarea x_j (coloană)

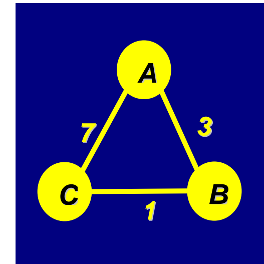
$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

- Fie N_b numărul de biți de codificare, atunci atracția dintre stările s_i şi s_j este dată de relația:

$$W_{fanin}(i,j) = N_B \times \hat{S}_i \times \hat{S}_j^T + X_i \times X_j^T$$

  - $W(1,2) = 2(1\ 1\ 1)(1\ 0\ 0)^T + (2\ 1)(0\ 1)^T = 3$
  - $W(1,3) = 2(1\ 1\ 1)(0\ 1\ 1)^T + (2\ 1)(1\ 1)^T = 7$
  - $W(2,3) = 2(1\ 0\ 0)(0\ 1\ 1)^T + (0\ 1)(1\ 1)^T = 1$

$$\begin{pmatrix} 2 & 1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}$$



Exemplu de calcul a funcției de fanin

Pentru un automat finit determinist luat ca exemplu în Fig. 24, putem calcula funcția de Fanin aplicînd teoria prezentată după cum urmează:

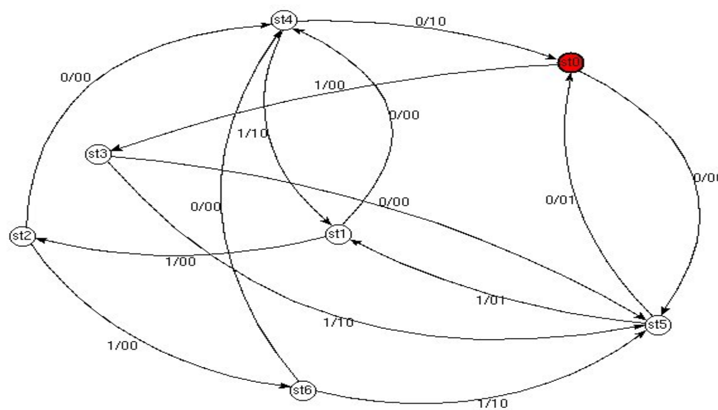


Figura 24. Graful de fluență pentru automatul finit df27.kiss2

Funcția ponderilor pentru optimizarea orientată Fanin: (i = 1, s = 7, o = 2) este:

$$W_{fanin}(i,j) = N_B \times \hat{S}_i \times \hat{S}_j^T + X_i \times X_j^T$$

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 2 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}
\times
\begin{bmatrix}
0 & 1 \\
1 & 1 \\
2 & 1 \\
3 & 1 \\
4 & 1 \\
5 & 1 \\
6 & 1
\end{bmatrix}
$$

$$\hat{S}_{|S| \times |S|} \qquad X_{|S| \times |I|}$$

Matricea ponderilor

0,1	0,2	0,3	0,4	0,5	0,6
	1,2	1,3	1,4	1,5	1,6
		2,3	2,4	2,5	2,6
			3,4	3,5	3,6
				4,5	4,6
					5,6

Tabela ponderilor

Funcția de cost este dată de formula: $Cost(fanout) = \sum W(S_{i,j})$ (2).

În continuare calculăm valorile matricii ponderilor pentru fiecare intersecție a stărilor conform cu tabelul ponderilor.

$$W(0,1) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (0\,0\,0\,0\,1\,1\,0)^T + (0\,1) \times (1\,1)^T = 3 \times 2 + 1 = 7$$
$$W(0,2) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (0\,1\,0\,0\,0\,0\,0)^T + (0\,1) \times (2\,1)^T = 3 \times 0 + 1 = 1$$
$$W(0,3) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (1\,0\,0\,0\,0\,0\,0)^T + (0\,1) \times (3\,1)^T = 3 \times 0 + 1 = 1$$
$$W(0,4) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (0\,1\,1\,0\,0\,0\,1)^T + (0\,1) \times (4\,1)^T = 3 \times 0 + 1 = 1$$
$$W(0,5) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (1\,0\,0\,2\,0\,0\,1)^T + (0\,1) \times (5\,1)^T = 3 \times 0 + 1 = 1$$
$$W(0,6) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (0\,0\,1\,0\,0\,0\,0)^T + (0\,1) \times (6\,1)^T = 3 \times 0 + 1 = 1$$
$$W(1,2) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (0\,1\,0\,0\,0\,0\,0)^T + (1\,1) \times (2\,1)^T = 3 \times 0 + 3 = 3$$
$$W(1,3) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (1\,0\,0\,0\,0\,0\,0)^T + (1\,1) \times (3\,1)^T = 3 \times 0 + 4 = 4$$
$$W(1,4) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (0\,1\,1\,0\,0\,0\,1)^T + (1\,1) \times (4\,1)^T = 3 \times 0 + 5 = 5$$
$$W(1,5) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (1\,0\,0\,2\,0\,0\,1)^T + (1\,1) \times (5\,1)^T = 3 \times 0 + 6 = 6$$
$$W(1,6) = 3 \times (0\,0\,0\,0\,1\,1\,0) \times (0\,0\,1\,0\,0\,0\,0)^T + (1\,1) \times (6\,1)^T = 3 \times 0 + 7 = 7$$
$$W(2,3) = 3 \times (0\,1\,0\,0\,0\,0\,0) \times (1\,0\,0\,0\,0\,0\,0)^T + (2\,1) \times (3\,1)^T = 3 \times 0 + 7 = 7$$
$$W(2,4) = 3 \times (0\,1\,0\,0\,0\,0\,0) \times (0\,1\,1\,0\,0\,0\,1)^T + (2\,1) \times (4\,1)^T = 3 \times 1 + 9 = 12$$
$$W(2,5) = 3 \times (0\,1\,0\,0\,0\,0\,0) \times (1\,0\,0\,2\,0\,0\,1)^T + (2\,1) \times (5\,1)^T = 3 \times 0 + 11 = 11$$
$$W(2,6) = 3 \times (0\,1\,0\,0\,0\,0\,0) \times (0\,0\,1\,0\,0\,0\,0)^T + (2\,1) \times (6\,1)^T = 3 \times 0 + 13 = 13$$
$$W(3,4) = 3 \times (1\,0\,0\,0\,0\,0\,0) \times (0\,1\,1\,0\,0\,0\,1)^T + (3\,1) \times (4\,1)^T = 3 \times 0 + 13 = 13$$
$$W(3,5) = 3 \times (1\,0\,0\,0\,0\,0\,0) \times (1\,0\,0\,2\,0\,0\,1)^T + (3\,1) \times (5\,1)^T = 3 \times 1 + 16 = 19$$
$$W(3,6) = 3 \times (1\,0\,0\,0\,0\,0\,0) \times (0\,0\,1\,0\,0\,0\,0)^T + (3\,1) \times (6\,1)^T = 3 \times 0 + 19 = 19$$
$$W(4,5) = 3 \times (0\,1\,1\,0\,0\,0\,1) \times (1\,0\,0\,2\,0\,0\,1)^T + (4\,1) \times (5\,1)^T = 3 \times 1 + 21 = 24$$
$$W(4,6) = 3 \times (0\,1\,1\,0\,0\,0\,1) \times (0\,0\,1\,0\,0\,0\,0)^T + (4\,1) \times (6\,1)^T = 3 \times 1 + 25 = 28$$
$$W(5,6) = 3 \times (1\,0\,0\,2\,0\,0\,1) \times (0\,0\,1\,0\,0\,0\,0)^T + (5\,1) \times (6\,1)^T = 3 \times 0 + 31 = 31$$

Costul final este obținut prin sumarea valorilor tuturor ponderilor, prin urmare obținem:

Cost(fanin) = 214.

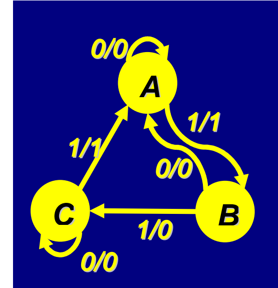
Tabelul 11: Optimizarea funcției de fanin pentru automatele finite

Nume Examplu	Nr. subFSM	Nr. intrări	Nr. ieşiri	Nr. Stări	FUNCȚIE DE COST			
					FANIN		FANOUT	
					Parțial	Total	Parțial	Total
Example-orig	0	1	1	6	115	115	39	39
Example Fanin	1	3	1	2	43	147	38	118
	2	3	1	2	53		36	
	3	3	1	2	51		44	
Shiftreg-orig	0	1	1	8	382	382	56	56
Shiftreg Fanin	1	3	1	3	79	162	60	104
	2	3	1	3	83		44	
Dk14-orig	0	3	5	7	1642	1642	1475	1475
Dk14 Fanin	1	5	5	2	1382	4414	1365	2925
	2	5	1	2	1468		732	
	3	5	1	2	1564		828	
Dk15-orig	0	3	5	4	536	536	572	572
Dk15 Fanin	1	4	5	2	478	954	421	713
	2	4	1	2	476		292	
Dk27-orig	0	1	2	7	214	214	35	35
Dk27 Fanin	1	3	2	2	74	188	47	135
	2	3	1	2	52		24	
	3	3	1	2	62		64	
Dk512-orig	0	1	3	15	5170	5170	153	153
Dk512 Fanin	1	5	3	3	314	992	199	643
	2	5	1	3	340		268	
	3	5	1	3	338		176	
Gen64-orig	0	2	4	64	2008209	2008209	40697	40697
Gen64 Fanin	1	8	4	6	51437	155126	44951	72.386
	2	8	1	6	52175		13470	
	3	8	1	6	51514		13965	
Gen256-orig	0	2	4	256	530179859	530179859	521734	521734
Gen256 Fanin	1	11	4	8	1484918	4575632	675494	1378994
	2	11	1	8	1476718		235964	
	3	11	1	8	1467394		229820	
	4	11	1	8	1466602		237716	
Gen1024orig	0	1	2	1024	136992489205	136992489205	1071762	1071762
Gen1024 Fanin	1	17	2	10	5569142	28273211	1716714	4810210
	2	17	1	10	5667209		778916	
	3	17	1	10	5613736		766896	
	4	17	1	10	5727617		780892	
	5	17	1	10	5695507		766792	
Gen10240-orig	0	1	2	10240	1373942234181120	1373942234181120	104904970	104904970
Gen10240 Fanin	1	25	2	13	869839584	6151671788	157009335	528812415
	2	25	1	13	881272182		61588992	
	3	25	1	13	890656923		64820748	
	4	25	1	13	871900774		60827276	
	5	25	1	13	876981101		61442124	
	6	25	1	13	880855222		62434588	
	7	25	1	13	880166002		60689352	

După cum se poate observa din Tabelul 11, rezultatele sunt foarte bune din punct de vedere al optimizării funcției de Fanin pentru un set de exemple luate, unele din setul de test MCNC standard, altele pentru circuite complexe, cu un număr foarte mare de stări, generate automat de către pachetul ***genfsm***. În urma optimizării funcției de cost cu ajutorul algoritmilor euristici (evolutivi) din pachetul ***fsmtool***, se obține o convergență a populației de soluții către o valoare de pîna la **218** ori mai redusă a funcției de cost pentru setul de subautomate descompuse obținute, în raport cu valoarea funcției de cost a automatului inițial, prototip.

7.5 Optimizarea funcției de fanout a unui circuit logic

Algoritm de optimizare pentru Fanout



- Matricea stărilor de tranziție $S_{|s|x|s|}$:
  - Linie: una pentru starea curentă
  - Coloana: una pentru starea următoare
  - Valoare: (ne-negativă) numărul of arce care pleacă din starea s_i (rînd) către starea s_j (coloană)
  - Matricea de ieşire $Z_{|s|x|O|}$:
  - Rînd: unul pentru starea curentă
  - Coloana: una pentru ieşire
  - Valoare: (ne-negativă) numărul de arce care pleacă din starea s_i (rînd) cu ieşirea z_j (coloană)

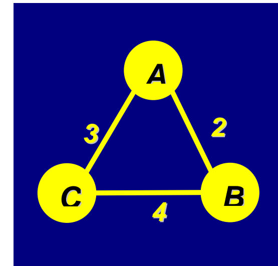
$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

- Fie N_b numărul de biți de codificare, atunci atracția dintre stările s_i şi s_j este dată de relația:

$$W_{fanout}(i,j) = N_B \times S_i \times S_j^T + X_i \times X_j^T$$

  - $W(1,2) = 2(1\ 1\ 0)(1\ 0\ 1)^T + (1)(0)^T = 2$
  - $W(1,3) = 2(1\ 1\ 0)(1\ 0\ 1)^T + (1)(1)^T = 3$
  - $W(2,3) = 2(1\ 0\ 1)(1\ 0\ 1)^T + (0)(1)^T = 4$

$$\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$



Exemplu de calcul a funcției de fanout

Pentru un automat finit determinist luat ca exemplu în Fig. 25, putem calcula funcția de Fanout aplicînd teoria prezentată după cum urmează:

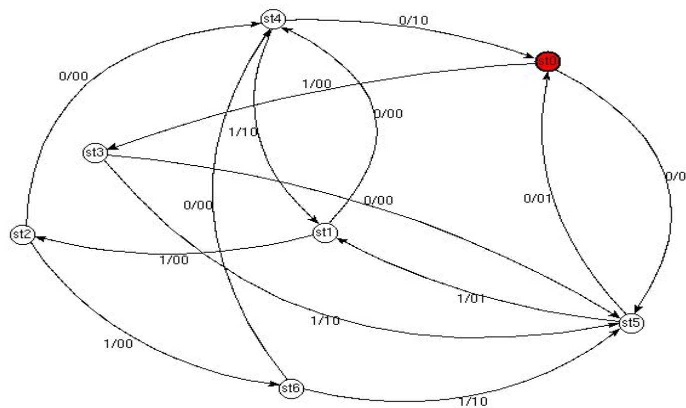


Figura 25. Graful de fluență pentru automatul finit df27.kiss2

Funcția ponderilor pentru optimizarea orientată Fanout: (i = 1, s = 7, o = 2) este:

$$W_{fanout}(i,j) = N_B \times S_i \times S_j^T + Z_i \times Z_j^T$$

$$\begin{bmatrix} 0&0&0&1&0&1&0 \\ 0&0&1&0&1&0&0 \\ 0&0&0&0&1&0&1 \\ 0&0&0&0&0&2&0 \\ 1&1&0&0&0&0&0 \\ 1&1&0&0&0&0&0 \\ 0&0&0&0&1&1&0 \end{bmatrix} \times \begin{bmatrix} 0&0 \\ 0&0 \\ 0&0 \\ 1&0 \\ 2&0 \\ 0&2 \\ 1&0 \end{bmatrix}$$

$S_{|S|\times|S|}$ $Z_{|S|\times|O|}$

Matricea ponderilor

0,1	0,2	0,3	0,4	0,5	0,6
	1,2	1,3	1,4	1,5	1,6
		2,3	2,4	2,5	2,6
			3,4	3,5	3,6
				4,5	4,6
					5,6

Tabela ponderilor

Funcția de cost este dată de formula: $Cost(fanout) = \sum W(S_{i,j})$ (2).

În continuare calculăm valorile matricii ponderilor pentru fiecare intersecție a stărilor conform cu tabelul ponderilor.

$$W(0,1) = 3\times(0\,0\,0\,1\,0\,1\,0)\times(0\,0\,1\,0\,1\,0\,0)^T + (0\,0)\times(0\,0)^T = 3\times0+0=0$$
$$W(0,2) = 3\times(0\,0\,0\,1\,0\,1\,0)\times(0\,0\,0\,0\,1\,0\,1)^T + (0\,0)\times(0\,0)^T = 3\times0+0=0$$
$$W(0,3) = 3\times(0\,0\,0\,1\,0\,1\,0)\times(0\,0\,0\,0\,0\,2\,0)^T + (0\,0)\times(1\,0)^T = 3\times2+0=6$$
$$W(0,4) = 3\times(0\,0\,0\,1\,0\,1\,0)\times(1\,1\,0\,0\,0\,0\,0)^T + (0\,0)\times(2\,0)^T = 3\times0+0=0$$
$$W(0,5) = 3\times(0\,0\,0\,1\,0\,1\,0)\times(1\,1\,0\,0\,0\,0\,0)^T + (0\,0)\times(0\,2)^T = 3\times0+0=0$$
$$W(0,6) = 3\times(0\,0\,0\,1\,0\,1\,0)\times(0\,0\,0\,0\,1\,1\,0)^T + (0\,0)\times(1\,0)^T = 3\times1+0=3$$
$$W(1,2) = 3\times(0\,0\,1\,0\,1\,0\,0)\times(0\,0\,0\,0\,1\,0\,1)^T + (0\,0)\times(0\,0)^T = 3\times1+0=3$$
$$W(1,3) = 3\times(0\,0\,1\,0\,1\,0\,0)\times(0\,0\,0\,0\,0\,2\,0)^T + (0\,0)\times(1\,0)^T = 3\times0+0=0$$
$$W(1,4) = 3\times(0\,0\,1\,0\,1\,0\,0)\times(1\,1\,0\,0\,0\,0\,0)^T + (0\,0)\times(2\,0)^T = 3\times0+0=0$$
$$W(1,5) = 3\times(0\,0\,1\,0\,1\,0\,0)\times(1\,1\,0\,0\,0\,0\,0)^T + (0\,0)\times(0\,2)^T = 3\times0+0=0$$
$$W(1,6) = 3\times(0\,0\,1\,0\,1\,0\,0)\times(0\,0\,0\,0\,1\,1\,0)^T + (0\,0)\times(1\,0)^T = 3\times1+0=3$$
$$W(2,3) = 3\times(0\,0\,0\,0\,1\,0\,1)\times(0\,0\,0\,0\,0\,2\,0)^T + (0\,0)\times(1\,0)^T = 3\times0+0=0$$
$$W(2,4) = 3\times(0\,0\,0\,0\,1\,0\,1)\times(1\,1\,0\,0\,0\,0\,0)^T + (0\,0)\times(2\,0)^T = 3\times0+0=0$$
$$W(2,5) = 3\times(0\,0\,0\,0\,1\,0\,1)\times(1\,1\,0\,0\,0\,0\,0)^T + (0\,0)\times(0\,2)^T = 3\times0+0=0$$
$$W(2,6) = 3\times(0\,0\,0\,0\,1\,0\,1)\times(0\,0\,0\,0\,1\,1\,0)^T + (0\,0)\times(1\,0)^T = 3\times1+0=3$$
$$W(3,4) = 3\times(0\,0\,0\,0\,0\,2\,0)\times(1\,1\,0\,0\,0\,0\,0)^T + (1\,0)\times(2\,0)^T = 3\times0+2=2$$
$$W(3,5) = 3\times(0\,0\,0\,0\,0\,2\,0)\times(1\,1\,0\,0\,0\,0\,0)^T + (1\,0)\times(0\,2)^T = 3\times0+0=0$$
$$W(3,6) = 3\times(0\,0\,0\,0\,0\,2\,0)\times(0\,0\,0\,0\,1\,1\,0)^T + (1\,0)\times(1\,0)^T = 3\times2+1=7$$
$$W(4,5) = 3\times(1\,1\,0\,0\,0\,0\,0)\times(1\,1\,0\,0\,0\,0\,0)^T + (2\,0)\times(0\,2)^T = 3\times2+0=6$$
$$W(4,6) = 3\times(1\,1\,0\,0\,0\,0\,0)\times(0\,0\,0\,0\,1\,1\,0)^T + (2\,0)\times(1\,0)^T = 3\times0+2=2$$
$$W(5,6) = 3\times(1\,1\,0\,0\,0\,0\,0)\times(0\,0\,0\,0\,1\,1\,0)^T + (0\,2)\times(1\,0)^T = 3\times0+0=0$$

Costul final este obținut prin sumarea valorilor tuturor ponderilor, prin urmare obținem:

Cost(fanout) = 35.

Tabelul 12: Optimizarea funcției de fanout pentru automatele finite

Nume Examplu	Nr. subFSM	Nr. intrări	Nr. ieşiri	Nr. stări	FUNCȚIE DE COST FANIN Parțial	FANIN Total	FANOUT Parțial	FANOUT Total
Example-orig	0	1	1	6	115	115	39	39
Example Fanout	1	3	1	2	59	169	38	106
	2	3	1	2	53		36	
	3	3	1	2	57		32	
Shiftreg-orig	0	1	1	8	382	382	56	56
Shiftreg Fanout	1	3	1	3	83	162	60	100
	2	3	1	3	79		40	
Dk14-orig	0	3	5	7	1642	1642	1475	1475
Dk14 Fanout	1	5	5	2	1622	5140	1049	2585
	2	5	1	2	1758		768	
	3	5	1	2	1760		768	
Dk15-orig	0	3	5	4	536	536	572	572
Dk15 Fanout	1	4	5	2	470	972	421	685
	2	4	1	2	502		264	
Dk27-orig	0	1	2	7	214	214	35	35
Dk27 Fanout	1	3	2	2	78	200	48	104
	2	3	1	2	62		32	
	3	3	1	2	60		24	
Dk512-orig	0	1	3	15	5170	5170	153	153
Dk512 Fanout	1	5	3	3	366	1115	160	518
	2	5	1	3	354		154	
	3	5	1	3	395		204	
Gen64-orig	0	2	4	64	2008209	2008209	40697	40697
Gen64 Fanout	1	8	4	6	51437	155126	44951	72.386
	2	8	1	6	52175		13470	
	3	8	1	6	51514		13965	
Gen256-orig	0	2	4	256	530179859	530179859	521734	521734
Gen256 Fanout	1	14	4	8	1752002	8876074	673775	1594715
	2	14	1	8	1746154		229708	
	3	14	1	8	1808578		231356	
	4	14	1	8	1775762		228432	
	5	14	1	8	1793578		231444	
Gen1024orig	0	1	2	1024	136992489205	136992489205	1071762	1071762
Gen1024-orig Fanout	1	21	2	10	7059521	42734095	1705542	5527514
	2	21	1	10	7202316		758524	
	3	21	1	10	7110127		761908	
	4	21	1	10	7099561		775916	
	5	21	1	10	7089826		759336	
	6	21	1	10	7172744		766288	
Gen10240-orig	0	1	2	10240	1373942234181120	1373942234181120	104904970	104904970
Gen10240-orig Fanin	1	25	2	13	922216945	6505447625	156927777	520434081
	2	25	1	13	933343406		60467852	
	3	25	1	13	937179463		61122828	
	4	25	1	13	921689988		59943696	
	5	25	1	13	932737686		61162172	
	6	25	1	13	936012316		60176308	
	7	25	1	13	922267821		60633448	

La fel ca şi în cazul optimizării funcției de Fanin pentru un set de circuite inițiale date, se observă o ameliorare a performanțelor obținute pentru automatele descompuse. Un lucru remarcabil de notat ar fi faptul că atunci cînd se obține o îmbunătățire a funcției de Fanout, se poate observa de asemenea şi o imbunătățire a costului funcției de Fanin. Acest fapt ar putea permite utilizarea unei funcții de cost compusă, din cele două criterii, pentru a obține o optimizare şi mai eficientă a unui circuit prototip, generat inițial. Avantajul principal este dat de faptul că nu există o limitare a complexității circuitului inițial. Cu toate astea, pentru un circuit de intrare mare, timpul de execuție creşte proporțional cu numărul său de stări interne.

Capitolul 8

Metode de optimizare globală în Proiectarea Circuitelor Complexe

8.1 Introducere

O dată cu maturizarea tehlonogiilor de fabricație submicronice a circuitelor, pot fi realizate circuite şi sisteme foarte complexe cu 10 pîna la 100 milioane de tranzistoare, pe o singură pastilă de siliciu, împachetate pe unul sau mai multe straturi; în plus deja se prevăd în următorii ani densități şi mai mari de integrare de miliarde de tranzistoare pe suprafața de siliciu. Obiectivele de optimizare clasice pentru circuitele submicronice de densitate mare, pe care le vom denumi în continuare sub titlul generic de **circuite complexe**, sunt performanța înalta, putere disipată redusă, suprafața de rutare cît mai mică, minimizarea timpilor de răspuns, simplificarea aranjării geometrice, precum şi mai multe din aceste obiective la un loc specificate într-o prioritate dată. Abordarile anterioare iau în considerație satisfacerea constrîngerilor în faza de după aranjarea geometrică a celulelor şi blocurilor standard, unde este disponibil un spațiu de soluții mult mai redus, ceea ce poate conduce la proiecte cu un grad redus de optimizare. Pină acum s-au remarcat puține realizări în explorarea obiectivelor de optimizare şi a constrîngerilor de pe parcursul procesului de sinteză logică, unde spațiul de soluții este mult mai larg din punct de vedere optimal. Procedura de sinteză se poate simplifica prin descompunerea circuitului în mai multe subcircuite. Descompunerea unei maşini secvențiale implică obținerea a două sau mai multe partiții ale maşinii inițiale, fiecare partiție corespunzînd unei submaşini care functionează concurent cu celelalte pentru a respecta comportamentul inițial al circuitului original, pe care îl vom denumi în continuare **circuit prototip**. Partițiile rezultate în timpul procesului de descompunere din combinarea stărilor circuitului prototip devin stări în noile subcircuite create. Metodele precedente au abordat ca domeniu de studiu tehnici specifice de descompunere. Implementarea circuitelor secvențiale ca **automate finite** care interacționează au condus de obicei la sporirea performanțelor ca rezultat al reducerii celei mai lungi căi dintre intrările şi ieşirile latch-urilor. Algoritmii de descompunere necesari sunt specifici pentru anumite tipuri de descompunere bivalente sau multivalente, descompunere paralelă, în cascadă, generală, sau topologii de descompunere arbitrară [86].

Proiectarea circuitelor complexe este în general un proces complicat şi consumator de timp care necesită cunoasterea colecțiilor de reguli specifice domeniului. Pentru a simplifica şi a creşte viteza procesului de proiectare, au fost studiate unele aplicații ale inteligenței artificiale şi calcul evolutiv,[104] care au fost aplicate problemelor de optimizare a proiectării circuitelor complexe.[50] Cercetări recente au început să demonstreze că este posibil de a proiecta circuite complexe utilizînd tehnologii de optimizare evolutive.[61]

Acest nou domeniu de cercetare s-a dezolvtat sub numele de hardware evolutiv, sau circuite cu hardware auto reconfigurabil. Avantajul principal pe care îl aduce acest domeniu este faptul că proiectarea evolutivă va permite în mod inevitabil explorarea automată a unui set de posibilităţi mult mai mari în spaţiul de proiectare, oportunităţi care sunt în afara posibilităţii de abordare a metodelor clasice, convenţionale. În orice spaţiu de căutare de dimensiuni abordabile, cea mai bună soluţie găsită într-un spaţiu aleatoriu de soluţii nu reprezintă optimul global al spaţiului de căutare. Scopul optimizării globale este de a maximiza profitul în timpul procesului de optimizare de pe parcursul proiectării circuitului.[78]

Din diversele metode de optimizare a circuitelor complexe, autorul a ales metoda de descompunere generală întrucit este cea mai largă dar şi cea mai complexă metodă. De asemenea au fost luate cîteva exemple şi au fost extrase cîteva rezultate experimentale despre optimizarea suprafeţei şi a performanţelor timpului de răspuns care ilustrează eficienţa metodologiei propuse. Apoi vor fi prezentate programele proiectate şi uneltele utilizate şi vor fi detaliate cîteva rezultate despre diverse implementări posibile; de asemenea este stabilit un posibil flux de optimizare pentru eventuale abordări ulterioare. [88][89]

8.2 Formularea Problemei

O maşină de stări finite M poate fi descrisă de o tuplă de 5 elemente $M = (S, I, O, \delta, \lambda)$, unde S este setul de simboluri de stare, I este setul de intrări primare, O este setul de ieşiri primare, $\delta : I \times S \rightarrow S$ este funcţia de stări următoare şi $\lambda : I \times S \rightarrow O$ este funcţia de ieşire (automat Mealy). Un automat finit poate fi reprezentat printr-un graf de tranziţii (**STG**) sau prin o tabelă de stări (**STT**). [73]

Definiţie: O partiţie Π pe un set de stări S este o colecţie de subseturi disjuncte din S, denumite **blocuri**, ale căror uniune de seturi este S. [43]

Definiţie: O partiţie Π pe setul de stări S ale unei maşini M se numeşte **partiţie închisă** dacă şi numai dacă două stări s şi t care sunt în acelaşi bloc al lui Π, şi pentru orice intrare $i \in I$, stările următoare $\delta(s,i)$ şi $\delta(t,i)$ sunt într-un bloc comun din Π.

Definiţie: O partiţie este denumită **partiţie generală** dacă nu este închisă. Maşina originală este denumită **maşină prototip** iar maşinile individuale care contribuie la funcţionarea generală sunt denumite **submaşini**. (Fig.26)

Definiţie: Maşina obţinută ca rezultat al descompunerii este denumită **maşina descompusă** şi este implementată ca o reţea de maşini interconectate [49]. Structura generală a unei astfel de reţele este prezentată în Fig. 27.

Partiţia zero, notată cu $\Pi(0)$, denotă o partiţie cu $|S|$ blocuri astfel încît fiecare bloc conţine exact o stare. Se poate demonstra că o maşina M poate fi descompusă într-un set de ***n*** maşini care interacţionează între ele şi care realizează aceeiaşi funcţie cu maşina M dacă şi numai dacă există un set de partiţii netriviale astfel încît să existe următoarea definiţie:

Definiţie: Există o descompunere validă a maşinii M dacă pentru un set de partiţii dat $\Pi_1, \Pi_2, \dots, \Pi_n$, produsul lor este egal cu $\Pi(0)$.

Figura 26. Circuit Secvential. Maşina Prototip.

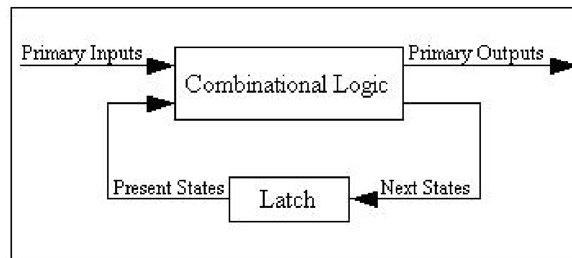
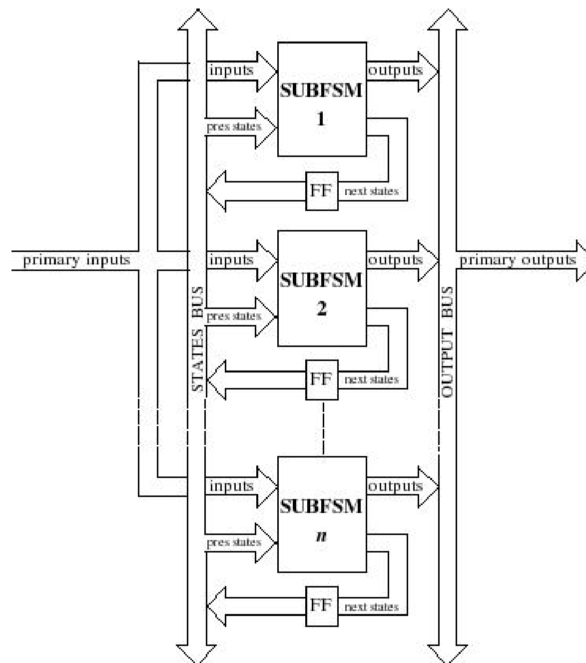


Figura 27. Topologia descompunerii generale



8.3 Implementarea spaţiului de soluţii

Metoda descompunerii generale sau arbitrare poate avea diferite topologii şi acoperă orice tip de descompunere. Principala contrîngere la ieşire rămine că orice pereche de stări a maşinii prototip trebuie să aibă coduri distincte în cadrul maşinii descompuse care rămîne neschimbată. Constrîngerile rămase depind de celelalte submaşini de la care una din submaşini primeşte informaţii despre starea prezentă. Tipurile de constrîngeri impuse se pot observa detaliat în exemplul model din Fig. 27.

Obiectivele principale în această lucrare sunt de a găsi o metodă de descompunere generală unei maşini secvenţiale aplicînd principii ale inteligenţei artificiale pentru a reduce complexitatea fiecărei submaşini în timp ce se încearca obţinerea unui număr total cît mai mic de submaşini. [17] Utilizînd metode de **optimizare globală** se pot obţine submaşini descompuse iar în decursul procedurii de descompunere sunt verificate dacă respectă proprietăţile descompunerii generale. [14]

O soluție poate fi descompusă prin combinarea mai multor partiții unic identificate. Numărul total de combinații posibile este 2^{2n}, ceea ce înseamnă că o soluție, sau un număr în baza 2^d reprezentînd soluția, poate fi reprezentată pe 2n·d poziții binare. Am calculat mai jos lungimea unei soluții reprezentate în format binar pentru diferite valori ale lui *n*:

Pentru n = 8, obținem N = 16 · (16 + 7) = 368
Pentru n = 16, obținem N = 32 · (45 + 15) = 1920
Pentru n = 64, obținem N = 128 · (296 + 63) = 45952

Pentru a ințelege mecanismul de decodificare a unei soluții, să considerăm următorul exemplu, pentru *n* = 3. Tabela de permutare (tabela permutărilor posibile) arată astfel:

$P_0 = \{ \varnothing \}$
$P_1 = \{ 1,2,3 \}$
$P_2 = \{ 1,3,2 \}$
$P_3 = \{ 2,1,3 \}$
$P_4 = \{ 2,3,1 \}$
$P_5 = \{ 3,1,2 \}$
$P_6 = \{ 3,2,1 \}$

Combinațiile de soluții posibile vor fi de forma:

$S_0 = \{ \varnothing \}$
$S_1 = \{ P_0 \}$
$S_2 = \{ P_1 \}$
$S_3 = \{ P_2 \}$
…
$S_{64} = \{ P_{63} \}$

Dacă *n* =3, atunci N = 6 · (3 + 2) = 30. O soluție va fi reprezentată pe 30 poziții binare. Pentru început se va genera un spațiu aleatoriu de soluții. Apoi se va testa dacă soluțiile inițiale reprezintă soluții viabile. De asemenea se va testa validitatea soluțiilor pentru a genera reprezentarea lor finală.[99]

Principalul interes ar putea fi mărimea maşinilor descompuse, timpul de răspuns, calea critică, sau chiar poziționarea geometrică a submaşinilor pe o anumită tehnologie specificată. Pe parcursul execuției programului ***fsmtool***, algoritmul este în principal dominat de procesul de evaluare a funcționalității şi performanței soluțiilor de optimizare globală precum şi de selecția lor pentru a găsi cele mai bune rezultate. Criteriul de selecție poate fi îmbunătățit pentru tipuri specifice de cerințe de optimizare globală pentru diverse tehnologii de implementare sau orice alt criteriu care poate fi de interes în obținerea rezultatelor.

Pe măsura ce structura partițiilor devine din ce în ce mai complexă, nu mai este eficient de a transla fiecare graf de circuit în limbajul Verilog/VHDL pentru a fi mai apoi simulat. Prin urmare, structurile vor fi stocate în memorie pină cind sunt obținute rezultatele finale. Spre exemplu, pentru un automat finit care prezintă comportamentul unui registru de deplasare, descrierea sa comportamentală poate fi realizată şi reprezentată în formatul standard industrial kisss[6] sau prin graful său de tranziții de stări (STG) din Fig. 29. [91][92]

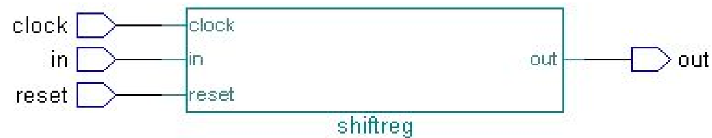


Figura 28. Schema bloc la nivel **Top** pentru maşina prototip

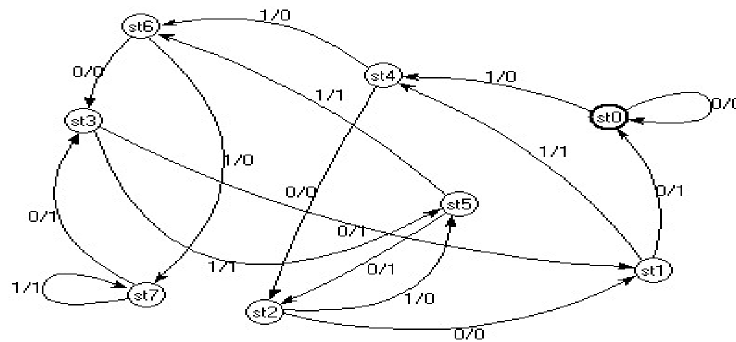


Figura 29. **STG** pentru maşina prototip [11]

Presupunînd ca avem o reprezentare la nivel top a unui registru de deplasare cunoscut ca maşina prototip şi un set de partiții generate aleatoriu cu programul ***fsmtool*** într-o etapă inițială după cum urmează:

{ (st0 st1) , (st2 st3) , (st4 st5) , (st6 st7) }
{ (st0 st2) , (st1 st3) , (st4 st6) , (st5 st7) }

În pasul următor obținem două submaşini care satisfac regulile de descompunere generală a maşinii prototip după cum se poate observa în Fig. 30 într-o vedere shematica top, sau fiecare din ele detaliată în Fig. 31 şi Fig. 32 în format de graf de tranziție **STG**.[76]

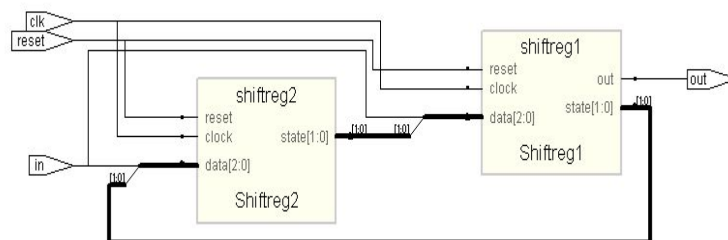


Figura 30. Schema la nivel **RTL** pentru maşina descompusă

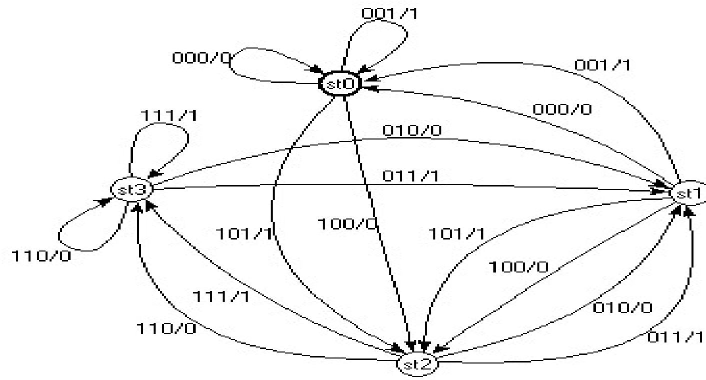


Figura 31. **STG** pentru prima submaşină

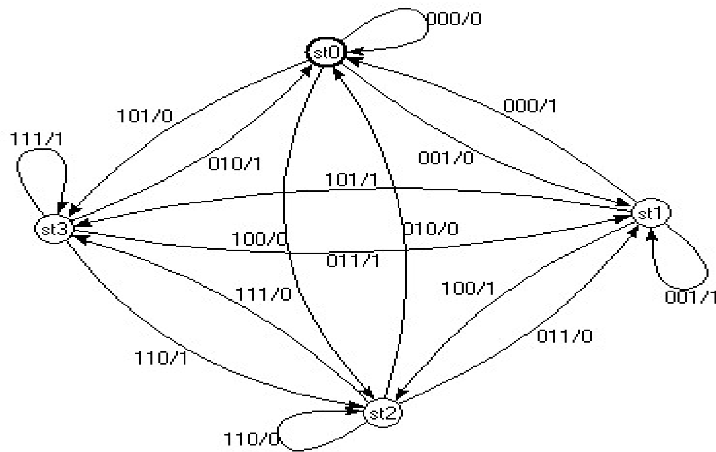


Figura 32. **STG** pentru a 2-a submaşină [12]

De asemenea trebuie verificată echivalența logică a submaşinilor utilizînd din programul **VIS** [116] , comanda *equivalence_check* pentru a verifica echivalența secvențială dintre cele două rețele flatenate ale submaşinilor prototip şi cea descompusă obținută cu ***fsmtool***. Submaşinile rezultate sunt apoi convertite cu ajutorul programului ***kiss2vl*** din reprezentarea simbolică de tip STG într-o reprezentare de nivel înalt în limbaj Verilog pentru a le utiliza ca fişiere de intrare într-un flux de proiectare în cadrul mediului integrat de dezvoltare Synplify Pro.

8.4 Rezultatele Implementării

După cum se poate observa din Tabelul 13 şi Tabelul 14, din rapoartele de sinteză, prin simularea pe familia Actel 500k, timpul de propagare scade de la 3.246 în maşina prototip la 1.568 în maşina descompusă. De asemenea, numărul de nivele logice scade de la 2 la 1. Numărul de bistabile scade de la 8 la 3 şi numărul total de celule utilizate scade de la 29 la 12. Ca o consecință la optimizarea globală, implementarea îmbunătățită obținută demonstrează faptul că frecvența estimată creşte de la 286.2 MHz pînă la 550.7 MHz.

Tabelul 13: Raportul pentru exemplul shiftreg.verilog

Cell usage	count	area	count*area
DFF	8	1.0	8.0
NOR2	4	1.0	4.0
NOR3FFT	4	1.0	4.0
NOR3	3	1.0	3.0
IB33	2	0.0	0.0
PWR	1	0.0	0.0
GL33	1	0.0	0.0
OR3	1	1.0	1.0
GND	1	0.0	0.0
OB33PH	1	0.0	0.0
INV	1	1.0	1.0
OAI21TTF	1	1.0	1.0
OR2	1	1.0	1.0
TOTAL	29		23.0

Propagation time: 3.246
Number of logic level(s): 2
Estimated frequency: 286.2 MHz

Tabelul 14: Raportul pentru exemplul shiftreg-top.verilog

Cell usage	count	area	count*area
DFF	3	1.0	3.0
NOR2FFT	3	1.0	3.0
IB33	2	0.0	0.0
PWR	1	0.0	0.0
GL33	1	0.0	0.0
GND	1	0.0	0.0
OB33PH	1	0.0	0.0
TOTAL	12		6.0

Propagation time: 1.568
Number of logic level(s): 1
Estimated frequency: 550.7 MHz

Cei patru paşi majori în utilizarea optimizării globale pentru a rezolva problema codării şirurilor de caractere cu lungime fixă sunt:

a) determinarea reprezentării soluțiilor
b) determinarea măsurii obiectivului (dimensiunea soluțiilor)
c) determinarea parametrilor şi variabilelor pentru controlul algoritmului
d) determinarea felului în care este indicat rezultatul şi criteriul de terminare a execuției algoritmului

Pentru o maşină secvențială dată, luată ca exemplu şi pentru un set specificat de partiții, se pot obține diverse tipuri de descompunere. Submaşinile rezultate echivalente au fost apoi prelucrate cu programe de sinteză standard precum Leonardo Spectrum şi Synplify Pro, unde au fost obținute diverse rezultate care sunt importante din punct de vedere al implementării tehnologice pe diverse familii de circuite şi pentru diverse cerințe de obținere a unei suprafețe optime sau a unei întîrzieri minime. Rezultate sunt obținute din prelucrarea unor seturi standard de exemple implementate pe diverse familii tehnologice de circuite. Există situații în care suprafața şi timpul de propagare sunt mai mici decît în masina prototip, dar există şi situații în care ele depăşesc proprietățile inițiale ale maşinii prototip.

Utilizînd **optimizarea globală**, în generarea automată a partițiilor, numărul de partiții obținut este redus în mod dinamic la cel mai eficient număr posibil. În timpul procesului de descompunere au fost necesare un set de programe precum: ***genfsm***, utilizat pentru generarea automată a maşinilor, ***vl2mv***, utilizat pentru a converti exemplele de la nivel verilog în format kiss, ***fsmtool***, utilizat pentru a genera seturile de partiții de stări şi pentru a realiza descompunerea efectivă a maşinilor prototip, ***kiss2vl***, utilizat pentru conversia fişierelor din format kiss în format comportamental de nivel înalt în limbaj verilog, precum şi **Vis**, pentru verificarea echivalențelor şi funcții de testare a validității submaşinilor obținute în decursul procesului de descompunere şi optimizare globală.

În tabelul 15 au fost menționate unele soluții experimentale obținute în urma aplicării metodelor de descompunere generală utilizînd AG. Ele sunt obținute în urma unui proces de sinteza complet, pornind de la un automat finit generat automat cu programul ***genfsm*** sau specificate din un set de exemple dat. Automatele de intrare sunt descrise într-un format kiss standard, ele sunt prelucrate şi reprezentate în memorie şi apoi descompuse cu ajutorul AG care au fost utilizați pentru îmbunătățirea seturilor de partiții obținute cu ***fsmtool***.
Rezultatele sunt reprezentate în format de ieşire kiss sau verilog pentru integrarea cu programe de sinteza standard. Ele depind de calitatea descompunerii, poziționarea stărilor interne în noile submaşini create, precum şi tehnologia specificată utilizată.

După cum se poate observa în fiecare coloană, există rezultate specifice pentru fiecare implementare tehnologică iar cele mai bune soluții ale descompunerii pot fi selectate apoi pentru fiecare librărie specificată. Acestea sunt exemple standard şi pot fi utilizate pentru obținerea anumitor rezultate experimentale pentru anumite tipuri de partiții aplicate maşinii prototip. Submaşinile rezultate obținute din procesul de descompunere cu ***fsmtool*** au fost apoi sintetizate cu programele Leonardo Spectrum sau Synplify Pro.

FISIER	TEHNOLOGIE	DFF	PI	PO	AREA	DELAY	PORTS	NETS	GATES	INST	ARRIVAL
Example-orig.v	**SPARTAN-XL**	**3**	**3**	**1**	**6**	**16**	**4**	**18**	**6**	**15**	**13.64**
Example-top1.v	**SPARTAN-XL**	**3**	**3**	**1**	**5**	**16**	**4**	**17**	**5**	**14**	**13.64**
Example-top2.v	**SPARTAN-XL**	**3**	**3**	**1**	**7**	**16**	**4**	**17**	**7**	**14**	**13.64**
Example-top3.v	**SPARTAN-XL**	**3**	**3**	**1**	**5**	**16**	**4**	**17**	**5**	**14**	**13.64**
Shiftreg-orig.v	**SPARTAN-XL**	**3**	**3**	**1**	**6**	**16**	**4**	**18**	**6**	**15**	**13.64**
Shiftreg-top1.v	**SPARTAN-XL**	**4**	**3**	**1**	**12**	**17**	**4**	**25**	**12**	**22**	**13.84**
Shiftreg-top2.v	**SPARTAN-XL**	**4**	**3**	**1**	**7**	**16**	**4**	**18**	**7**	**15**	**13.84**
Shiftreg-top3.v	**SPARTAN-XL**	**3**	**3**	**1**	**3**	**8**	**4**	**12**	**3**	**9**	**6.90**
Example-orig.v	**VIRTEX-II**	**3**	**3**	**1**	**6**	**7**	**4**	**16**	**6**	**13**	**6.73**
Example-top1.v	**VIRTEX-II**	**3**	**3**	**1**	**8**	**7**	**4**	**18**	**8**	**15**	**6.73**
Example-top2.v	**VIRTEX-II**	**3**	**3**	**1**	**7**	**7**	**4**	**17**	**7**	**14**	**6.79**
Example-top3.v	**VIRTEX-II**	**3**	**3**	**1**	**6**	**7**	**4**	**16**	**6**	**13**	**6.79**
Shiftreg-orig.v	**VIRTEX-II**	**3**	**3**	**1**	**5**	**6**	**4**	**15**	**5**	**12**	**6.08**
Shiftreg-top1.v	**VIRTEX-II**	**4**	**3**	**1**	**9**	**6**	**4**	**25**	**9**	**22**	**6.08**
Shiftreg-top2.v	**VIRTEX-II**	**4**	**3**	**1**	**9**	**6**	**4**	**20**	**9**	**17**	**6.03**
Shiftreg-top3.v	**VIRTEX-II**	**3**	**3**	**1**	**2**	**5**	**4**	**12**	**2**	**9**	**5.49**
Example-orig.v	**ALTERA-FLEX10k**	**3**	**3**	**1**	**9**	**8**	**4**	**18**	**9**	**15**	**8.50**
Example-top1.v	**ALTERA-FLEX10k**	**3**	**3**	**1**	**8**	**10**	**4**	**17**	**8**	**14**	**9.60**
Example-top2.v	**ALTERA-FLEX10k**	**3**	**3**	**1**	**8**	**8**	**4**	**16**	**8**	**13**	**8.50**
Example-top3.v	**ALTERA-FLEX10k**	**3**	**3**	**1**	**8**	**10**	**4**	**16**	**8**	**13**	**9.60**
Shiftreg-orig.v	**ALTERA-FLEX10k**	**3**	**3**	**1**	**8**	**8**	**4**	**17**	**8**	**14**	**8.50**
Shiftreg-top1.v	**ALTERA-FLEX10k**	**4**	**3**	**1**	**13**	**11**	**4**	**24**	**13**	**21**	**10.70**
Shiftreg-top2.v	**ALTERA-FLEX10k**	**4**	**3**	**1**	**7**	**4**	**4**	**15**	**7**	**12**	**4.40**
Shiftreg-top3.v	**ALTERA-FLEX10k**	**3**	**3**	**1**	**3**	**4**	**4**	**10**	**3**	**7**	**4.40**
Example-orig.v	**ASIC-SCL05u**	**-**	**-**	**-**	**88**	**-**	**4**	**19**	**88**	**14**	**2.80**
Example-top1.v	**ASIC-SCL05u**	**-**	**-**	**-**	**82**	**-**	**4**	**17**	**82**	**13**	**2.31**
Example-top2.v	**ASIC-SCL05u**	**-**	**-**	**-**	**85**	**-**	**4**	**18**	**85**	**13**	**2.50**
Example-top3.v	**ASIC-SCL05u**	**-**	**-**	**-**	**89**	**-**	**4**	**21**	**89**	**15**	**2.43**
Shiftreg-orig.v	**ASIC-SCL05u**	**-**	**-**	**-**	**125**	**-**	**4**	**27**	**125**	**21**	**2.34**
Shiftreg-top1.v	**ASIC-SCL05u**	**-**	**-**	**-**	**179**	**-**	**4**	**35**	**179**	**28**	**4.29**
Shiftreg-top2.v	**ASIC-SCL05u**	**-**	**-**	**-**	**38**	**-**	**4**	**10**	**38**	**7**	**0.80**
Shiftreg-top3.v	**ASIC-SCL05u**	**-**	**-**	**-**	**45**	**-**	**4**	**12**	**47**	**9**	**1.45**

Tabel 15. Rezultate experimentale obținute pentru setul de test în procesul de optimizare

Pe parcursul testării algoritmilor de descompunere s-au utilizat fişiere din setul standard de teste MCNC, precum şi fişiere generate automat cu programul ***genfsm***. Exemplele au fost apoi sintetizate şi optimizate pe mai multe seturi standard de circuite FPGA de la Xilinx, familia Spartan XL şi Virtex-II, precum şi de la Altera, familia Flex-10k. De asemenea s-a încercat şi o implementare pe setul ASIC SCL05u. Rezultatele diferă în funcție de particularitățile fiecărui set în parte, unele sunt mai bune, altele mai putin bune. Scopul este de a obține rezultate cît mai bune pentru fiecare set de circuite în parte, pentru optimizarea cît mai eficientă a numărului de unitați tehnologice utilizate în fiecare caz în parte.

Pe parcursul procedurii de descompunere a automatului prototip s-a realizat o optimizare a funcției de ieşire, în sensul că aceasta a fost calculată doar o singură dată pentru un singur subautomat, restul subautomatelor obținute au contribuit doar la menținerea corectitudinii funcțiilor de tranziție necesare păstrării echivalenței funcționale a circuitului rezultat cu circuitul prototip.

8.5 Concluzii şi implementări viitoare

În acest capitol s-au construit şi utilizat un numar de programe standard precum **genfsm**, **fsmtool** şi **kiss2vl**, pentru a implementa partea teoretică şi pentru a putea obține şi anumite rezultate experimentale şi practice. Am demonstrat potențialul şi eficiența programelor realizate de a genera o maşina de stări finite precum şi submaşinile aferente, de a aplica o procedura de **optimizare globală** asupra descompunerii generale utilizînd un set de partiții generate manual sau automat şi posibilitatea de a obține submaşini valide pentru a reduce complexitatea fiecărei submaşini în timp ce numărul de submaşini obținute este cît mai redus. Această etapă de pre-sinteză poate fi uşor integrată într-un program de sinteză standard şi prin abordarea sa generală poate fi uşor adaptat pentru diferite probleme de optimizare precum descompunerea funcțiilor logice complexe şi implementarea optimală pe familii tehnologice specifice cu un număr variabil de porți logice în structurile lor interne, de diferite mărimi şi suprafețe de implementare.

Acest proces standard poate fi îmbunătățit în viitor prin găsirea unei metode alternative de a testa rezultatele optimale găsite în urma unei etape de generare a rezultatelor. Necesitatea utilizării unui program extern standard pentru evaluarea rezultatelor poate fi înlocuită cu o funcție obiectiv interna într-un proces de îmbunătățire a calității soluțiilor obținute. Această imbunătățire va reduce timpul de execuție precum şi complexitatea procesului de optimizare.

Capitolul 9

Strategii Evolutive

9.1 Noțiuni introductive

În prezent evoluția tehnologică a ajuns la un nivel în care sinteza logică este integrată cu sinteza de nivel înalt. Problema descompunerii circuitelor complexe a fost abordată înca de la începuturile proiectării asistate de calculator a circuitelor digitale. Studiind proprietățile funcțiilor complexe devine posibil de a le reprezenta prin intermediul mai multor funcții simple echivalente. Utilizînd proprietățile lor funcționale se pot descompune circuite mari şi complexe în sisteme cu circuite mai mici care pot fi obținute într-o perioadă de timp mai scurtă şi sunt mai usor de întreținut. În cazul programelor de sinteza logică multinivel, cum ar fi MVSIS [77], pot apărea probleme specifice cum ar fi generarea unor circuite de performantă scazută atunci cînd sunt abordate funcții complexe. Prin descompunerea circuitului complex în mai multe subcircuite mai simple, procedura de sinteză poate fi simplificată. Descompunerea unei maşini secvențiale implică găsirea a două sau mai multe partiții ale maşinii originale, fiecare partiție corespunzînd unei submaşini care functionează în paralel cu celelalte pentru a respecta comportamentul inițial al circuitului. Partițiile rezultate în timpul procesului de descompunere din combinarea stărilor maşinii inițiale devin noi stări în noile submaşini create. Algoritmii de descompunere necesari sunt specifici pentru anumite topologii de descompunere de tip paralel, cascadă, generală sau arbitrară.

Proiectarea circuitelor electronice este în general un proces complicat şi durează perioade îndelungate de timp, necesitînd în acelaşi timp cunoştințe despre reguli ample specifice domeniului. Pentru a simplifica şi a scădea perioada de proiectare au fost studiate metode de calcul evolutiv care au fost mai apoi aplicate cu succes problemelor de optimizare a proiectării sistemelor complexe. [13] Cercetări recente au demonstrat că este posibil de a proiecta circuite electronice utilizînd tehnologii de optimizare evolutive. Acest nou domeniu de cercetare a devenit cunoscut sub numele de **hardware evolutiv**. Avantajul său proeminent este faptul că proiectarea evolutivă va permite în mod inevitabil explorarea automată a unui set de posibilități mult mai bogate intr-un spațiu de căutare care trece peste limitările metodelor convenționale. Acest lucru înseamnă că tehnologiile evolutive vor ajuta la simplificarea şi micşorarea perioadei de descompunere a circuitelor complexe şi pot produce soluții îmbunatățite la o problemă data.[40]

9.2 Paradigma evolutivă

Dacă dezvoltarea este o adaptare pe o scară a vieții, evoluția este o adaptare pe scara istorică. Dupa cum s-au remarcat, în 1950, oameni de ştiinţă precum Turing şi von Neumann au fost interesați în modelarea şi întelegerea fenomenelor biologice în termeni de procesări naturale ale informației. Începutul erei calculatoarelor a promovat tendința de simulare a proceselor şi a modelelor naturale şi a condus la dezvoltarea unor modele evolutive artificiale.

Prima abordare în acest sens a fost realizată în 1950 de către Fraser, a cărui muncă a fost modelarea fenomenelor biologice prin intermediul calculatorului digital. Cu doar un an mai tîrziu, Friedberg a conceput un model evolutiv într-o abordare modernă, în care el a menționat în mod explicit că este utilizat pentru programare automată. Aceste două abordări sunt considerate începuturile calculului evolutiv. În 1962, John H. Holland inventează Algoritmii Genetici, denumiți în continuare **AG** [70], la început ca sisteme clasificatoare, care sunt dezvoltați utlerior pe direcția înglobării în aceştia a mecanismelor de adaptare din lumea biologică. Printre primele aplicații ale AG se află problema controlului curgerii unui gaz dintr-o conductă în regim staționar tranzitoriu.

La mijlocul anilor 1960 au fost definite trei din cele patru mari domenii de **calcul evolutiv**[105]:

- Programare Evolutivă (EP), *Lawrence Fogel*
- Algoritmi Genetici (AG), *John H. Holland*
- Strategii Evolutive (SE), *Bienert, Rechenberg, Schwefel*

Primul care a utilizat AG pentru optimizare a fost K.A. De Jong în 1975. Holland (1975) dă un cadru teoretic pentru adaptarea implementată în AG. Principalele trăsături ale AG dezvoltați inițial sunt legate de folosirea noțiunii de populație de cromozomi şi de operatori genetici – crossover, inversiune, mutație. AG lucrează cu şiruri de biți (*0,1*). [75]

Rechenberg I. prin lucrările din 1965 şi 1973 introduce metoda de **Strategie Evolutivă**, denumită în continuare **SE**, care imită principiile evoluției naturale pentru optimizarea unei probleme cu parametri definiți în spațiul numerelor reale. La început această metoda a fost o procedura de căutare experimentală a optimului şi, ulterior, a devenit o tehnică computatională. Domeniul este dezvoltat de Schwefel (1975, 1977), Bäck şi Hoffmeister (1991). Inițial această metodă a folosit pentru populația de start doi indivizi, un părinte şi un urmaş rezultat dintr-o mutație a părintelui. Ulterior acestă schemă a fost dezvoltată cu populații din mai multi indivizi şi cu operatori de crossover.

Tehnica de **Programare Evolutivă**, denumită în continuare PE, a apărut o dată cu lucrarea lui L.J. Fogel , A.J.Owens şi M.J. Walsh din 1966, unde tratează problema căutării în spațiul soluțiilor unei maşini cu un număr finit de stări, legată de predicția prin metode evolutive a şirurilor simbolice generate cu procese Markov şi serii de timp nestaționare. Soluțiile obținute prin mutații aleatorii ale diagramei stărilor de tranziție au fost evaluate pe baza fitness-ului lor.[56]

Koza, J.R. în 1990 propune pentru găsirea celui mai potrivit program care rezolvă o anumită problema o nouă metodă numită **Programare Genetică** [66], denumita în continuare **PE**, care este de fapt un AG[52] folosit pentru urmărirea evoluției unor programe scrise în LISP, de exemplu. [24]

După o perioadă de stagnare, acest domeniu s-a maturizat într-un domeniu comun de cercetare abia în anii 80'-90' (Goldberg 1989)[41], o dată cu acceptarea termenului de calcul evolutiv ca titlu sugestiv asupra întregului domeniu de cercetare, şi la primele conferinţe de Calcul Evolutiv, denumit în continuare **CE**.[72] Inspirat din biologie, vocabularul specializat CE derivă direct din evoluţia naturală. Cu toate acestea, unii termeni şi-au mai schimbat sensul şi , mai mult, au căpătat un sens precis, specific ştiintei calculatoarelor. Pentru început, cuvîntul definitiv pentru AG, şi pentru ceilalţi algoritmi evolutivi, este **evoluţia**. Ceea ce este „evoluat" în AG este *genomul* sau *genotipul* sau *fenotipul* sau *cromozomul*. Fiecare genom este un individ al unei populaţii. Noţiunea de individ poate fi luată ca un simplu şir, sau comportamentul implicat de execuţia unui şir, prin urmare există un genotip şi un fenotip. Funcţia de *fitness* a unui individ este echivalentul în evoluţia naturală a capacităţii sale de supravieţuire. Un fitness mai ridicat îmbunătăţeşte probabilitatea de a fi selectat pentru reproducere. *Reproducerea* unui individ este o copie a genomului său, alterată în mod aleatoriu în cazul *mutaţiei*, sau alterată în funcţie de alt individ selectat în cazul *intersecţiei*. *Copiii* sau *urmaşii* unui individ sunt genomii rezultaţi din aceste alterări. Dacă toată populaţia este înlocuită de urmaşi, atunci noua populaţie este denumită *generaţia următoare*.

În mod uzual există doar două componente ale celor mai mulţi AG care sunt dependente de problemă: *codificarea problemei* şi *funcţia de evaluare* care este de obicei dată ca o parte a descrierii problemei. Mărimea spaţiului de căutare este specificată printr-un număr de poziţii binare utilizate în codificarea problemei. Pentru o codificare pe un şir de biţi de lungime L, mărimea spaţiului de căutare este 2^L şi formează un hipercub. AG specifică muchii ale acestui hipercub L-dimensional. Fiecare codificare binară este un „cromozom" care corespunde unei muchii din hipercub şi este un membru din $2^L - 1$ hiperplane diferite, unde L este lungimea codificării binare. Este de asemenea relativ uşor de observat că $3^L - 1$ hiperplane de partiţii pot fi definite peste tot spaţiul de căutare. Pentru fiecare L poziţii în şirul de biţi pot exista valorile {**, 0, 1*} ceea ce rezultă în 3^L combinaţii posibile. Stabilirea faptului că fiecare şir este un membru din cele $2^L - 1$ partiţii de hiperplane nu furnizează prea multe informaţii dacă fiecare punct din spatiul de căutare este examinat izolat. Aceasta este cauza pentru care noţiunea de căutare bazată pe populaţii de soluţii este critică pentru AG. [66]

O populaţie de puncte luate ca exemplu furnizează informaţii despre un număr mare de hiperplane. Mai mult, hiperplanele de nivel scăzut trebuiesc specificate prin numeroase puncte din populaţie. Paralelismul implicit implică faptul că mai multe hiperplane de soluţii sunt rezolvate simultan în paralel. O parte cheie a paralelismului implicit la AG derivă din faptul că sunt evaluate mai multe hiperplane atunci cînd este evaluat un şir de soluţii.[103]

Teorema Schemei furnizează un nivel mai apropiat în schimbarea nivelului de evaluare pentru un singur hiperplan de la generaţia t la generaţia $t+1$ (Holland 1975). Un şir de biţi identifică o schemă particulară dacă acel şir de biţi poate fi construit din acea schemă prin înlocuirea simbolurilor * cu valoarea potrivită a biţilor. Toate şirurile de biţi care identifică o schemă particulară sunt conţinute în partiţia hiperplanului reprezentat de către acea schemă.

9.3 Comparație între SE, PE şi AG

Toți aceşti algoritmi au la origine acelaşi model, fiecare accentuînd anumite caracteristici ca fiind cele mai importante din procesul evoluției. Cea mai definitivă diferență este dată de interpretarea rolului operatorilor genetici. PE nu foloseşte recombinarea (crossover-ul) în timp ce AG consideră aceasta ca avînd un rol principal, minimalizînd influența mutației (folosită de PE). În AG, mutațiile sunt inversiuni de biți, pur aleatorii, avînd o frecvență mică. Strategiile evolutive folosesc ambii operatori. Atît PE cît şi SE folosesc un zgomot de distribuție gaussiană, avînd media zero pentru a perturba toate variabilele obiectiv.

În plus, SE folosesc o distribuție log-normală pentru deviațiile standard ale mărimii paşilor de mutație (pentru scalarea lor) şi o distribuție normală pentru covariante.

Recombinarea parametrilor în SE este folosită pentru a realiza autoadaptarea, proces care se desfasoară în PE fără intervenția recombinării. Altfel spus, PE este o abstractizare a evoluției la nivelul populațiilor care se reproduc (specii), astfel că mecanismele de recombinare nu sunt folosite, deoarece recombinarea nu are loc între specii. În SE abstractizarea evoluției este la nivelul manifestării individului şi aici informația de autoadaptare este „pur" genetică, astfel că sunt raționale diferite forme de recombinare (crossover). Practica a arătat că eficacitatea unor astfel de operatori genetici depinde puternic de problema de rezolvat.

În AG de obicei se face o codificare a soluțiilor posibile problemei analizate în şiruri de simboluri semnnificative (de obicei de tip boolean) care reprezintă genomul (genotipul). Dacă funcția obiectiv nu este o funcție pseudo-booleană, fiecare şir va fi decodificat într-un set de variabile de decizie corespunzătoare (fenotipul) înainte de a fi evaluat fitness-ul indivizilor populației. AG folosiți pentru modelarea proceselor adaptive recurg la şiruri binare, la operatori de crossover şi mutație. PE foloseşte reprezentarea care derivă din problemă.

Toate cele trei clase de CE (canonic) dau indivizilor o durată medie de viața de o generație, adică indivizii au descendenti numai odată la un moment dat, iar părinții selectați pentru o nouă iterație sunt tratați ca noi descendenți.[96] AG şi PE insistă asupra selecției probabiliste, implementată diferit în cele două tehnici. SE folosesc un mecanism de selecție strict determinist, fiind *extinctive*, în sensul excluderii clare a indivizilor cei mai slabi. În PE unii indivizi sunt excluşi din selecție în timp ce AG folosesc *prezervarea*, în sensul că fiecare individ primeşte o probabilitate de selecție diferită de zero. Mecanismul elitist este implicit în selecşia din PE, iar în SE apare selecția $(\mu + \lambda)$-ES. SE lucrează cu un surplus de descendenți dacă $\lambda > \mu$. Aceasta este de folos în cazul lucrului cu condiții de tip inegalități, cind violarea lor duce la „moartea" descendenților respectivi. În AG elitismul se foloseşte numai dacă este exprimat explicit.

AG şi PE alocă probabilități de împerechere şi reproducere a indivizilor în raport cu valorile relative ale fitness-ului (funcției obiectiv) sau prin referinte la pozițiile relative rezultate din clasificarea după rang. Convergența globală cu probabilitatea unu, cu certitudine, este demonstrată teoretic la toate formele simplificate de CE. Rezultatele asupra ratei de convergență sunt cunoscute la SE şi la PE, pentru funcții test simple. Investigațiile teoretice din AG se bazează pe teorema schemei, care nu poate estima rata de convergență a algoritmului.

9.3.1 Rezumat al caracteristicilor

Caracteristicile principale ale CE – SE, PE, AG sunt descrise pe scurt după cum urmează:

Reprezentarea: SE, PE – valori reale , AG – valori binare.

Fitness-ul: SE – valori ale funcţiei obiectiv, PE,AG– valori scalate ale funcţiei obiectiv

Autoadaptarea: SE – deviaţii standard şi unghiuri de rotaţie, PE – coeficienţi de corelaţie, variante, AG – nu au.

Mutaţia: SE – operator principal, gaussiană, PE – operator, gaussiană, AG – operator de bază, inversare de biţi.

Recombinarea: SE – discretă şi intermediară, importantă pentru autoadaptare, PE – nu, AG – operator principal, crossover în *z* puncte, crossover uniform.

Selectia: SE – deterministă, extinctivă sau bazată pe prezervare, PE – probabilistă, extinctivă, AG – probabilistă , bazată pe prezervare.

Conditii: SE – relaţii de inegalitate arbitrare, PE – nu, AG – limite simple prin mecanismul de codificare.

Teorie: SE – rata de convergenţă pentru cazuri speciale, (1 + 1) – ES, (1 + λ) – ES, convergenţă globală pentru (μ + λ) – ES, PE – rata de convergenţă cazuri speciale (1 + 1) – ES, convergenţă globală pentru (1 + 1) – ES, AG – teoria prelucrării schemelor, convergenţă globală pentru versiunea cu selecţie elitistă.

Datele care arată diferenţe între algoritmi similari, cum sunt cei discutaţi aici, demostrează că în prezent încă nu s-au formulat pe deplin principiile fundamentale care să stea la baza CE.

În această lucrare ne vom referi în mod special la AG întrucit dintre toate tehnologiile studiate se preteaza cel mai bine asupra mapării spaţiului de soluţii pe vectori binari care conţin codificările soluţiilor problemelor propuse. Prin urmare se va studia în detaliu abordarea soluţiei din punct de vedere al implementării şi lucrului cu operatorii genetici.

9.4 Evoluția

Domeniul *vieții artificiale*, dezvoltat în contextul tehnicilor amintite anterior şi în paralel cu acestea, nu are ca obiectiv principal să modeleze cît mai exact viața biologică, ci să investigheze cum forma noastră de viață (de pe Pamînt) provine din componente lipsite de viață. Numeroase probleme stiințifice de inginerie, de finanțe sau sociale, se pot rezolva cu ajutorul algoritmilor evolutivi. Ei au fost folosiți cu succes într-o mare varietate de probleme de *optimizare*, dintre care optimizarea numerică şi optimizarea combinatorie [22]. Aplicațiile includ găsirea parametrilor unor probleme matematice, probleme de cablare complexă multistrat, programe de distribuire a locurilor de muncă sau de strategie a pregătirii personalului (compania aeriană KLM), de utilizare a forței de muncă şi de optimizare a fluxului de producție , reducerea vibrațiilor în rotoarele pentru elicoptere (studii la Bell Helicopters), optimizarea traiectoriei brațului unui robot,[106] dirijarea vehicolelor, tehnici de interogare a bazelor de date, aplicații de optimizare în sisteme de comunicații, găsirea drumului minim în retea, etc. De asemenea, maşinile care au posibilităti de invățare folosesc algoritmii evolutivi în sarcini de clasificare şi predicție.

Teoria evolutionistă (darvinistă), bazată pe selecția naturală, susține că indivizii unei specii de animale sau plante care se adaptează mai bine la mediu au probabilitatea mai mare de a supraviețui şi deci de a da urmaşi. Proprietățile fenotipului sunt cruciale pentru adaptarea la un mediu specific de viață. De-a lungul generațiilor repetate, indivizi cu caracteristici mai potrivite mediului în care trăiesc, se vor perpetua în detrimentul celor care nu se adaptează, care vor dispare. Astfel, *fitness*-ul (performanța), unui organism se poate defini în două moduri. El reprezintă probabilitatea ca organismul să trăiască pentru reproducere adică să fie *viabil* sau *fitness*-ul este funcție de numărul de descendenți ai organismului respectiv, adică legat de *fertilitate*. Deşi mediul actionează prin selecția fenotipurilor, eliminînd competitorii slabi, ceea ce selectează evoluția este genotipul.

Selecția naturală în lumea biologică este oarbă, „nu priveşte înainte”, nu calculează consecințele şi nu are un scop precizat. Acestea sunt deosebiri semnificative față de CE dezvoltat în domeniul *Inteligenței Artificiale*.[69] Astfel, tehnicile de calcul înlătură „deficiențele” existente la evoluția naturală folosind baza de cunoştințe asociată problemei de rezolvat. Este interesant de subliniat că procese (aparent) foarte simple, care stau la baza selecției naturale, conduc la niveluri de specializare şi complexitate remarcabile. Trebuie amintit că în natură procesul de evoluție are loc pe o scară de timp foarte mare şi la o populație extrem de numeroasă, în foarte multe generații, condiții care nu sunt obținute în calculele numerice.[126]

9.5 Codificarea şi transmiterea informației. Cromozomi

AG folosesc termeni similari cu cei din biologie (genetica moleculară) în sensul cel mai simplu al acestor noțiuni. ADN-ul este purtătorul de informație al vieții şi evoluției, esențial în mecanismul de apariție al unui individ, adică a *fenotipului*, pe baza planului de construcție general al speciei, adică a *genotipului*. *Ontogeneza* unui organism este dezvoltarea acestuia de la momentul apariției unui *zigot* fertilizat si pîna la moartea sa. O celulă care conține două seturi de cromozomi se numeşte diploidă, altfel se numeşte haploidă, adică posedă un set unic de cromozomi în fiecare celulă.

Fiecare cromozom conține mii de *gene*, care sunt blocuri funcționale ale ADN. Fiecare pereche de gene determină particularități specifice la descendenți, caracteristici fizice, sau mentale (culoarea ochilor, grupa de sînge), numite *fenotip*. Fiecare genă din cromozomii omologi poate avea expresii diferite, numite *alele*, care corespund valorii genelor respective.

Combinațiile posibile ale perechilor de gene, adică un set particular de gene, distribuit pe mai mulți cromozomi, reprezintă *genotipul*, respectiv „rețeta" specifică unui individ. Întregul material genetic al unei celule este denumit *genom*.

În prelucrarea informațiilor cu AG, aceştia lucrează cu **cromozomi** (soluții posibile ale problemei), care reprezintă codificarea prin şiruri de simboluri sau numere (de obicei şiruri binare) a caracteristicilor problemei de rezolvat. Subşirurile care codifică diferiți parametri ai problemei constituie genele cromozomilor. Cromozomii reprezintă genotipul (codificarea soluției), în timp ce fenotipul este soluția ca atare. O singură soluție a problemei (un cromozom) se exprimă prin termenul de „membru al populației" sau „individ din populație".

La parcurgerea unui AG se trece prin mai multe etape în procesul de reproducere, în care o populație este înlocuită prin apariția unei noi generații (populații). Se creează aleatoriu o populație inițială (de soluții posibile ale problemei de rezolvat) de indivizi (cromozomi). Fiecare cromozom poate fi considerat ca un punct în spațiul de căutare al soluțiilor posibile ale problemei. Urmează executarea unor iterații, care includ selecția celor mai bune soluții şi realizarea de recombinații, folosind operatori specifici. În acest proces, populația de cromozomi este înlocuită succesiv (la un nou ciclu) de o altă populație, pe baza unei funcții de fitness, ce asigură un scor fiecărui cromozom din populația curentă. Uneori pot exista mutații asupra unor soluții. Prin selecția soluțiilor bune din cursul iterațiilor, respectiv prin alegerea succesivă a cromozomilor, calculatorul va genera soluții din ce în ce mai bune, similar cu fenomenul de evoluție naturală, care produce indivizi din ce în ce mai adaptați mediului înconjurător. În calculul evolutiv, membrii populatiei „se nasc", se „împerechează" şi „mor" în cîteva microsecunde, astfel că schimbările majore în populație se fac foarte rapid.

9.6 Operatorii genetici

AG includ noțiunile fundamentale ale geneticii; folosesc natura ca sursă de inspirație, fără a crea însă modele care să urmeze îndeaproape mecanismele din natură; dezvoltă metode care se transpun usor în limbajul calculatoarelor. În forma cea mai simplă AG conțin trei operatori: *selecția*, *mutația* şi *variația*.

Selecția este operatorul prin care cromozomii unei populații de soluții sunt aleşi pentru reproducere. Cu cît aceşti cromozomi sunt mai potriviți ca soluții ale problemei de rezolvat, cu atît ei sunt aleşi de mai multe ori pentru reproducere. Selecția este operatorul care impune o anumită desfaşurare a procesului de evoluție, prin preferarea indivizilor mai buni pentru a supraviețui şi a se reproduce.

Crossover-ul constă în cazul AG în schimbul de „material genetic" la recombinarea dintre doi cromozomi care nu sunt în perechi, cum este cazul părintilor haploizi. Acest operator realizează un schimb de informații între indivizii populației, transmitindu-le la descendenți. Astfel, şirurile de biți (cromozomii) 011001 şi 111000 după crossover, începînd de exemplu cu poziția a patra, conduc la urmaşii 011000 şi 111001. Rata (probabilitatea) de crossover determină dacă operația are loc sau nu. Rata 1 semnifică un crossover cert, iar rata 0

arată că nu va avea loc nici un crossover. Prin crossover, indivizi cu alele bune (în sensul de utile în stabilirea soluției problemei) pentru diferite gene nu vor mai competiționa între ei şi pot rezulta descendenți cu ambele gene bune. Crossover-ul poate avea loc într-un punct, în două puncte atunci cînd se schimbă secțiunea dintre cele două puncte, în *n* puncte sau poate fi uniformă, atunci cînd se schimbă un număr mare de gene.

Mutația, este o operație unară, şi se obține prin înlocuirea unui bit (0 sau 1) sau a unui simbol al alfabetului folosit, pentru un locus ales aleatoriu, cu bitul complementar (1 sau 0) sau cu un alt simbol ales aleatoriu din alfabet. Mutația introduce inovația în populația de indivizi, prin generarea unor variante ale acestora. De exemplu, dacă şirul 110011 suferă o mutație în poziția patru rezultă şirul 110111. Mutația poate avea loc în orice poziție din şir, cu o probabilitate de obicei foarte mică. Se mai foloseste şi operatorul de *inversiune*, care între două poziții fixate din cromozom inversează ordinea genelor. Mutația de *inversiune* este diferită de mutația de *reinițializare* (unde este reinițializat un bit), deoarece inversiunea se referă la schimbarea unui bit, iar inițializarea are numai 50% sanse de schimbare a bitului. Pentru date de alt tip decît şirul de biți, operatorul mutație de reinițializare face ca genele să fie reinițializate aleatoriu. În AG mutația se face şi sub alte cîteva forme. Mutația în *trepte* se produce prin adăugarea sau scăderea unei valori, de obicei mică, la valoarea curentă a genei. Acest tip de mutație este foarte eficientă deoarece prin distribuția gaussiană sunt şanse mari de a produce modificări mici şi în mod progresiv aceste modificări conduc la o modificare majoră. Modificarile mici produc îmbunătățiri ale performanțelor soluției, iar o schimbare majoră ocazională permite ieşirea dintr-un minim local, din spațiul de căutare a soluțiilor.

În prelucrările cu şiruri binare, crossover-ul joacă un rol mult mai mare decît mutația în procesul de căutare a soluțiilor. Dacă în reprezentările binare mutația este un operator slab, în probleme cu codificări ale soluțiilor cu un alfabet extins mutația poate avea variante multiple şi devine un operator puternic. De exemplu, cu reprezentarea în numere reale se pot afişa variabile cu distribuție aleatorie (cu distribuție normală) la valorile existente şi în acest fel să apară schimburi mici sau majore în populație. Crossover-ul este un operator util atunci cînd în populație există gene scurte, semi-independende. În cazurile în care sunt gene dependente, aşezate la distanță în cromozom, crossover-ul poate duce la efecte constructive, în sensul obținerii de soluții viabile, cum este cazul optimizării unei rețele neuronale, unde grupuri de ponderi din zone diferite ale unui cromozom sunt corelate, condiționîndu-se reciproc. Utilitatea operatorului de crossover pentru o aplicație particulară poate fi estimată numai prin încercări experimentale.

Variația este un operator mai rar utilizat în care biții se schimbă astfel încît numerele codificate de şirul de biți cersc sau scad încet, modificări neîntîlnite la ceilalți operatori genetici. Dacă mutația schimbă un bit unic al şirului (cromozomului), variația modifică valoarea codificată în acel şir cu un mic increment sau decrement. Încrucişarea într-un punct schimbă fragmentele alăturate ale şirurilor pereche. Pentru încrucişări multipunctuale se schimbă între parteneri numai fragmentele între pozițiile fixate pentru crossover.

9.7 Reprezentarea soluțiilor

Pentru rezolvarea unei probleme cu AG se începe cu proiectarea unei reprezentări a soluției acestei probleme. Reprezentarea este atît de importantă pentru acest domeniu încît se poate spune că AG se pot aplica oriunde se poate găsi o exprimare potrivită a soluției problemei. O soluție poate fi orice valoare care este posibilă de interpretat pentru a fi soluția corectă sau răspunsul final al problemei. [1]

Cea mai folosită formă de reprezentare a unei soluții este un şir de caractere dintr-un alfabet prestabilit {0,1} , {0,1,*} , {0,1,…,9} , {A,B,…,Z}. În general orice tip de date (numere, şiruri, structuri) pot fi codificate ca şiruri de biți. Genele care marchează elemente ale unei soluții, sunt reprezentate prin biți sau blocuri de biți adiacenți. În cazul cromozomului constituit dintr-un şir de biți ai alfabetului binar, o alelă reprezintă caracterul 0 sau 1. Pentru un alfabet mai larg se multiplică şi numărul de alele posibile în fiecare locus al genelor.

Lungimea şirului de caractere depinde de alfabetul folosit şi de cantitatea de informații ce trebuie înmagazinată în acest şir. Un şir este analog cu un cromozom sau un set de cromozomi. Dacă se reprezintă o soluție cu un şir de 12 biți şi se utilizează alfabetul {0,1} se poate exprima un set de valori de 12 variabile sau parametri, unde fiecare bit ia o valoare binară (0 sau 1) a unui parametru. Fiecare parametru sau bit este analog unei gene. Dacă valorile parametrilor sunt mai multe decît valorile binare 0 sau 1, atunci o soluție de 12 biți poate să reprezinte valorile a 3 parametri în cazul în care fiecare parametru foloseşte 4 biți, de la 0000 la 1111 sau în zecimal de la 0 la 15. În acest caz se poate spune că soluția sau cromozomul are 3 gene.

Reprezentarea binară, folosită de obicei în AG tradiționali, prezintă o serie de neajunsuri atunci cînd se aplică la probleme numerice multidimensionale, ce reclamă o precizie ridicată. Astfel, pentru a manevra 100 de variabile în domeniul [-500 , 500], cu o precizie de 6 cifre, lungimea vectorului soluție binar este de 3000. Acesta generează un spațiu de căutare de 10^{1000}, unde AG devine ineficient în găsirea soluției. Totuşi alfabetul binar oferă numărul maxim de scheme pe bit de informație în raport cu orice altă codificare, fapt ce a determinat ca reprezentarea binară să domine dezvoltările în AG. Reprezentarea binară facilitează şi tratările teoretice ale AG, analize care sunt cele mai avansate din punct de vedere formal, matematic, din tot spectrul de metode de calcul evolutiv.[42]

Genotipul unui „individ”, în AG care folosesc şiruri de biți pentru reprezentarea cromozomilor, este dat de configuratia biților din cromozomii individuali. Pină de curînd noțiunea de fenotip nu era folosită de AG, dar dezvoltări recente au introdus această noțiune, de exemplu în codificarea prin şiruri de biți a unei rețele neuronale alături de considerarea rețelei neuronale ca atare.

Reprezentarea printr-un şir de caractere a unei soluții este uzuală în AG, dar există şi alte forme de reprezentare mai potrivite pentru aplicații concrete, cum ar fi soluția sub forma unui graf [18], a unei matrici,etc.

9.8 Blocuri de construcție şi scheme

În forma cea mai generală se poate spune că AG lucrează prin descoperirea, accentuarea şi recombinarea unor „blocuri" bune ale soluțiilor unei probleme într-o manieră de acțiune paralelă. Mai exact, dacă soluțiile sunt constituite din şiruri de biți în care există combinații ale părtilor optime ale soluțiilor bune, se obține un fitness mai mare al şirului în care sunt prezente, în raport cu şirurile care nu conțin astfel de combinații. În acest caz se pune problema asemănărilor care există între cromozomii diferiți ai unei populații. AG urmăresc ca genele bune să supraviețuiască în detrimentul genelor proaste, care sunt eliminate. Tot acest proces este oarecum similar cu metoda *divide et impera*. [2]

O formalizare a componentelor sau blocurilor optime dintr-o soluție a fost dată de Holland, care a introdus pentru aceasta noțiunea de schemă. Schema (H) este un set de biți dintr-un alfabet cu trei caractere, descrisă de *template*-uri (sabloane) compuse din cifre aparținind setului $\{0,1,*\}$, unde '*' înlocuieste oricare din celelalte caractere ale alfabetului.

Schema este o generalizare a unei părți a unui şir. H defineşte hiperplane în spatiul n-dimensional, unde n este lungimea şirului de biți. Schema descrie o submulțime a unui cromozom unde se păstrează pentru toți membrii anumite poziții fixe. H = 1**1***0 este o schemă de lungime $\lambda = 8$, care reprezintă toate şirurile ce conțin 8 biți care încep cu cifra 1, se termină cu cifra 0 şi au cifra 1 în poziția 4. Aici H are trei biți definiți (1,1,0) şi se spune că este de ordinul 3, deci ordinul $\theta(H)$ este numărul de poziții fixe ale schemei. Distanța dintre cei mai îndepărtați biți definiți, ficşi şi diferiți de * reprezintă lungimea definitorie pentru H, notată cu $d(H)$, care în acest caz este egală cu 7. Această lungime dă numărul punctelor de încrucişare situate în zona semnificativă a schemei. Şirurile care se potrivesc la şablonul H sunt numite *instanțe* ale lui H. La structura H prezentată corespund instanțele 10010000, 11111110, 11010110, etc. Nu toate submulțimile posibile ale mulțimii de şiruri de lungime n se pot defini ca scheme. În teoria AG se consideră că operatorii pentru selecție, mutație şi crossover actionează efectiv asupra componentelor de tipul schemelor.

Un şir de biți de lungime n reprezintă o instanță a 2^n scheme diferite. Cu alte cuvinte o schemă care conține n simboluri * corespunde la 2^n soluții (şiruri), deoarece * poate lua valoarea 1 sau 0. Orice populație de s şiruri de lungime n include instanțe pentru un număr de scheme diferite cuprins între 2^n şi $s \times 2^n$. Limita inferioară se atinge atunci cînd toate cele s şiruri sunt identice. În acest caz, numărul de scheme este cel corespunzător pentru un singur şir. Şirul 11 este o instanță pentru următoarele scheme: 11, 1*, *1, **. Dacă se consideră că toate soluțiile sunt diferite, în număr de s, fiecare conținînd 2^n scheme, rezultă limita superioară de $s \times 2^n$. În realitate acest numar nu este atins deoarece diferite şiruri de soluții corespund la aceeaşi schemă. Astfel o populație de două şiruri 00 şi 11, conține 7 scheme: 11, 1*, *1, **, 00, 0*, *0, iar limita superioară în acest caz este $2 \times 2^n = 8 > 7$, deoarece schema ** este conținută în şirurile 11 şi 00. Într-un set de şiruri de lungime n sunt posibile 3^n scheme diferite deoarece pentru fiecare poziție din şir sunt trei alternative: 0, 1 si *.

AG evaluează la o anumită generație fitness-ul pentru s şiruri din populație, dar conform celor de mai sus el estimeaza fitness-ul mediu, sau media fitness-urilor tuturor instanțelor posibile, ale unui număr mult mai mare de scheme. Schemele nu sunt reprezentate sau evaluate explicit de AG şi nici nu sunt calculate fitness-urile medii pentru ele. Cu toate acestea, într-un AG creşte sau descreşte numărul de instanțe corespunzătoare schemelor din populație, ceea ce corespunde la evaluări pentru scheme şi pentru fitness-ul mediu.

9.9 Considerații asupra operatorilor genetici

***Crossover*:**

Fie schemele H_1: *1***1 si H_2: ***01* care sunt conținute în sirul S_1: 110011. Cînd S_1 suferă de un crossover după poziția celui de-al doilea bit cu şirul S_2: 101110 din populație, rezultă descendenții S'_1: 111110 şi S'_2: 100011. La aceşti urmaşi schema H_1 a dispărut, deoarece primul bit definit (1 în poziția a doua din S_1) se va afla într-un şir după crossover (S'_1)., iar bitul 6 din şirul S_1 va trece la alt şir (S'_2). În schimb, schema H_2 se pastrează după crossover deoarece ambii biți definiți de această schemă (01 din pozițiile 4 şi 5 în S_1) se regăsesc după crossover în acelaşi şir (S'_2). Schema H_2 va fi distrusă dacă locul de încrucişare este după bitul 4, iar schema H_1 se va regăsi în descendenți dacă crossover-ul are loc după primul bit. Deci o schemă va supraviețui sau va fi distrusă în funcție de doi factori: modelul schemei şi poziția în şir la care are loc crossover-ul. Pentru H_1 sunt posibile 5 locuri de încrucişare (λ-1): *!1!*!*!*!1. Dacă unul din locuri cade în lungimea definitorie, $d(H)$, a şirului (5 – 1 = 4), atunci H_1 va fi distrus, întrucit el va supraviețui numai dacă are loc crossover-ul în afara acestei lungimi, adică după bitul din poziția unu. Deci probabilitatea ca H_1 să dispară la descendenți este de 4/5, cînd toate pozițiile de încrucişare sunt egal probabile.

Generalizînd, rezultă expresia probabilității ca o schemă să fie distrusă de forma: $p_d = d(H)/(\lambda-1)$. Şansa de supraviețuire a unei scheme depinde şi de forma partenerului de la operația de crossover. În cazul precedent crossover-ul S_1 cu partenerul S_2 a condus la dispariția schemei H_1. Dacă S_1: 110011 se încrucişează cu S_3: 011110, de exemplu la poziția după al doilea bit (S'_1: 111110, S'_2: 010011) sau după al treilea bit (S'_1: 110110, S'_2: 011011), H_1 va supraviețui în S'_2. Acest factor, pattern-ul şirului pertener, face să existe inegalitatea $p_d \leq d(H)/(\lambda-1)$. Şansa de supraviețuire a unei scheme depinde şi de forma partenerului de la operația de crossover. În cazul anterior crossover-ul dintre S_1 cu partenerul S_2 a condus la dispariția schemei H_1. Dacă S_1: 110011 se încrucişează cu S_3: 011110, de exemplu dacă la poziția după al doilea bit (S'_1: 111110, S'_2: 10011) sau după al treilea bit (S'_1: 110110, S'_2: 011011), H_1 va supraviețui în S'_2. Acest factor, pattern-ul şirului partener, face să existe inegalitatea: $p_d \leq d(H)/(\lambda-1)$, pentru cazul cert în care crossover-ul a avut loc.Se notează cu p_c probabilitatea ca procesul de crossover să fie aplicat unui şir. Probabilitatea ca o schemă H să fie distrusă va fi $p_d \leq p_c\ (d(H)/(\lambda-1))$. Schema H supraviețuieşte la apariția unei încrucişări aplicată unei instanțe părinte a lui H dacă unul din urmaşi este şi el o instanță a lui H, deci probabilitatea de supraviețuire va fi:

$$p_s = 1 - p_d \geq 1 - p_c \frac{d(H)}{\lambda - 1}.$$

Limita inferioară a probabilității ca H să supraviețuiască unei încrucişări este $L_c(H) \geq 1 - p_c d(H)/(\lambda-1)$. Probabilitatea de supraviețuire a şirurilor care suferă încrucişări este mai mare pentru scheme mai scurte. Numărul asteptat de instanşe ale lui H la momentul $t+1$ la acțiunea operațiilor de reproducere şi crossover, considerate acționînd independent, va fi:

$$n(H, t+1) \geq \frac{g_m(t)}{f_{med}(t)} n(H, t)\left[1 - p_c \frac{d(H)}{\lambda - 1}\right].$$

Din relația de mai sus rezultă că schemele cu valori ale fitness-ului $g_m(t)$ mari şi lungimi definitorii $d(H)$ mici sunt mult mai probabile să supraviețuiască şi să se multiplice în populațiile descendenților.

***Mutația*:**

Se notează cu p_m probabilitatea ca un bit din orice poziție a şirului să sufere o mutație. Pentru ca schema *H* să supraviețuiască după o mutație a unei instanțe *S*, trebuie ca toate pozițiile biților definiți din *H* să ramînă ficşi după mutația suferită de *S*. Astfel, schema *H*: *1***1 conținută în şirul *S*: 110011 că dispare dacă *S* suferă mutația la poziția a doua, devenind *S'*: 100011. Într-o schemă sunt $\theta(H)$ poziții fixe ale biților, conform definiției parametrului *ordinul schemei H*. Pentru fiecare poziție a biților în şir probabilitatea de supraviețuire a lor, în urma unei mutații, va fi $1\text{-}p_m$. Probabilitatea de supraviețuire a schemei *H* va fi dată de $(1\text{-}p_m)^{\theta(H)}$, unde $\theta(H)$ este ordinul lui *H*. Dezvoltînd în serie Taylor (pentru p_m <<1) avem relația aproximativă $(1\text{-}p_m)^{\theta(H)} \approx 1\text{-}\ \theta(H)p_m$. Din relațiile precedente rezultă că probabilitatea de supraviețuire a şirurilor supuse mutațiilor şi care conțin o anumită schemă este mai mare cu cît schema este de ordin mai mic.

9.10 Teorema schemei

În urma considerațiilor anterioare, se poate rezuma faptul că prin acțiunea operatorilor de reproducere, crossover şi mutație, va rezulta numărul scontat de instanțe ale schemei *H*, generate la momentul *t+1*, dat de relația:

$$n(H,t+1) \geq \frac{g_m(t)}{f_{med}(t)} n(H,t)\left(1-p_c\frac{d(H)}{\lambda-1}\right)\left[(1-p_m)^{\theta(H)}\right]$$

Această relație constituie **teorema schemei** sau *teorema fundamentală* a algoritmilor genetici şi defineşte dezvoltarea unei scheme de la o generație la alta. Teorema arată că scheme de ordin scăzut, al caror fitness mediu se situează deasupra mediei, vor avea un numar de instanțe evaluate care creste exponențial în timp, pe parcursul execuției AG. La fiecare generație numărul de instanțe ale acestor scheme, care nu se distrug şi au un fitness deasupra mediei, cresc cu factorul $g_m(t)/f_{med}(t)$. Prin aproximația făcută pentru probabilitatea de supraviețuire în urma mutației, teorema schemei se poate scrie astfel:

$$n(H,t+1) \geq \frac{g_m(t)}{f_{med}(t)} n(H,t)\left(1-p_c\frac{d(H)}{\lambda-1}-\theta(H)p_m\right).$$

Număul evaluat de teorema schemei în partea din dreapta a relației de mai sus reprezintă limita inferioară deoarece aici se iau în considerare numai efectele distructive (pentru *H*) ale încrucişării şi mutației. În realitate crossover-ul poate recombina instanțe chiar mai performante, cu scheme de ordin superior. Existența acestui proces constituie *ipoteza blocurilor de construcție*.

Teorema schemei se aplică atît la scheme cît şi la submulțimi de şiruri din spațiul de căutare. Schemele de tip şiruri scurte, cu fitness mediu înalt, sunt foarte potrivite pentru ilustrarea tipurilor de blocuri de construcție care sunt combinate efectiv în procesul de crossover. Există în literatura de specialitate o serie întreagă de critici şi semnalări referitoare la această teoremă, la limitările introduse de ea şi la implicațiile practice. (Vose, M.D – [117]).

9.11 Paralelismul implicit

AG în evaluarea la un moment dat a populației de *s* șiruri va estima implicit fitness-urile medii ale tuturor schemelor prezente în populație, fără necesități în plus de memorie, de alte resurse de procesare sau timp de calcul suplimentar. Numărul de instanțe reprezentative pentru fiecare schemă din populație va creşte sau va scădea conform teoremei schemei. Acele scheme al căror fitness estimat se află deasupra mediei cuprind un număr mai mare de experiențe, în sensul că au un număr mai mare de instanțe în populație. Această evaluare subînțeleasă se numeşte **paralelism implicit**, fără a însemna că AG rulează pe calculatoare cu procesare paralelă, deşi ei se pretează foarte bine pentru aceasta. Caracteristica de paralelism implicit este independentă de modul de codificare a soluției. Selecția în AG are ca efect influențarea gradată a procedurii de lucru în direcția obținerii de instanțe pentru scheme care au fitness-ul estimat deasupra mediei. În principiu, se poate spune că în timp acest fitness evoluează spre o acuratețe mai mare, deoarece algoritmul genetic procesează mai multe instanțe ale acelor scheme. Cu operatorii de selecție şi crossover, şirurile dintr-o populație vor tinde, după mai multe iterații, să aibă acelaşi bit într-o anumită poziție. Mutația este operatorul care asigură în aceste cazuri diversitatea pentru o poziție binară dată. [123]

Diferite tipuri de codificări folosite pentru introducerea informațiilor în cromozomi, conduc la grade diferite de paralelism implicit. Pentru a ilustra acest fapt, se compară două codificări avînd o capacitate de informație purtată, similară. O codificare binară este făcută cu un număr mic de alele, pe şiruri lungi (de ex. pe 100 biți), iar cealaltă codificare, zecimală, se face cu un numar mare de alele pe şiruri scurte, de exemplu șiruri zecimale de lungime 40. Prima codificare are un grad de paralelism mai mare decît a doua, deoarece o instanță pentru codificarea binară conține mai multe scheme decît o instanță pentru codificarea a doua, zecimală, respectiv comparînd 2^{100} cu 2^{40}.

Există mai multe metodologii de incorporare a paralelismului în AG, ceea ce conduce la existența unor categorii distincte de astfel de algoritmi. În AG *masiv paraleli* se utilizează un număr de procesoare, 2^{10} sau mai multe. În acest caz un procesor este asociat cu un individ din populație. În *modelele cu insule paralele*, algoritmul consideră că o serie de subpopulații evoluează în paralel. La aceste modele apare conceptul de *migrație*, adică deplasarea unui individ de la o subpopulație la alta şi crossover-ul între indivizi din diferite subpopulații. AG *paraleli hibrizi* au o corespondență unu la unu între procesor şi individ, la fel ca la modelul masiv paralel, dar este folosit numai un numar mic de procesoare. Acest model foloseşte şi alte tipuri de algoritmi, cum ar fi *hill-climbing*, pentru a mări performanțele sistemului.

9.12 Metode de căutare a informațiilor

Căutarea unor date memorate se referă la regăsirea informațiilor conținute în memoria calculatorului. Un exemplu clasic îl reprezintă căutarea într-o bază de date mare, de nume şi adrese ordonate după un criteriu oarecare. Căutarea binară este una din numeroasele metode eficiente de găsire a unei înregistrări solicitate.

Căutarea unei căi spre un obiectiv înseamnă a găsi un set de acțiuni prin care să se treacă de la starea inițială la obiectivul final stabilit. În domeniul Inteligenței Artificiale s-au dezvoltat mai mulți algoritmi pentru această problemă, printre care se numără căutarea în adîncime „depth-first search” şi „branch and bound”.

Cautarea pentru o soluție reprezintă o clasă de căutari mai generală decît căutarea menționată anterior. Obiectivul în acest caz este de a găsi o soluție la o problemă căutînd în spațiul soluțiilor posibile. Căutarea unei căi spre un obiectiv se poate codifica ca o soluție candidată, devenind astfel un subset al căutării pentru o soluție. AG sunt folositi în acest tip de căutare. Aceşti algoritmi reprezintă metode de căutare globală, care nu utilizează informația relativă la gradient, ceea ce permite aplicarea acestora la probleme unde intervin funcții care prezintă mai multe extreme locale.

În primul tip de căutare se pune problema găsirii unei informații existente explicit în spațiul cautat. Pentru celelalte tipuri, soluțiile posibile care sunt analizate se creează în timpul procesului de căutare. Astfel, la căutarea într-o problemă de căutare, nu se analizează tot arborele de căutare în care nodurile sunt deja memorate. În multe cazuri, similare, sunt prea numeroase datele pentru a memora întreg arborele de căutare a soluției. Abordarea se face prin construcția succesivă a arborelui de căutare, pe măsura avansării algoritmului. Ideea este de a găsi o soluție optimă prin parcurgerea numai a unei porțiuni a arborelui de căutare. Prin urmare, căutarea în spațiul soluțiilor posibile nu trebuie să se realizeze prin crearea tuturor soluțiilor posibile şi apoi evaluarea acestora. Strategia AG este de a găsi soluțiile optime sau bune, prin analiza numai a unei fracții mici din soluțiile posibile ale problemei.

Chiar dacă căutarea unei căi pentru a atinge un obiectiv se poate rezolva prin aplicarea AG, folosind soluții posibile codificate adecvat, metode mai eficiente au fost dezvoltate în Inteligența Artificială prin tehnici cu arbori sau grafuri de căutare. Totuşi aceste tehnici nu au aplicabilitate în toate problemele de căutare. Tehnica de căutare prin deplasări pe direcția scăderii gradientului pleacă de la un punct oarecare al spațiului şi face mici schimbări pentru îmbunătățirea soluției. Dacă se poate determina panta în orice punct al spațiului de căutare, această tehnică este foarte performantă şi conduce direct către un minim sau maxim. În cazul în care spațiul este „accidentat" aceste extreme pot fi locale şi soluția gasită nu este optimă.

Eliminarea acestui neajuns este realizată în tehnica determinării stohastice a direcției de scădere a gradientului, unde schimbările mici în poziție sunt realizate aleatoriu şi deplasarea are loc numai dacă se obține o îmbunătățire a soluției. Această manieră de abordare se poate aplica uneori şi la AG.

Toate metodele de căutare a unei soluții cuprind o serie de etape:

- Generarea unui set de soluții posibile (populația inițiala în cazul AG);
- Evaluarea soluțiilor propuse cu criterii de fitness;
- Selectarea pe baza rezultatelor pasului anterior a candidaților păstrați;
- Producerea de varianți cu ajutorul unor operatori aplicați soluțiilor următoare.

Căutarea cu AG prezintă diferențe fața de algoritmii de căutare obişnuiți, generali(hill-climbing, simulated annealing). În AG căutarea se face pe soluții codificate, plecînd de la o populație inițială, nu lucrînd direct pe informația primară, iar operatorii acționează cu o anumită probabilitate, nefolosindu-se reguli deterministe. AG folosiți pentru căutarea unui anumit punct sau a unui optim, de fapt caută la un moment dat o populație de puncte, nu un singur punct din spațiul de căutare. În AG se fac puține supoziții referitoare la forma spațiului de căutare astfel că ei pot fi folosiți într-o mare varietate de probleme. O proprietate importantă a AG ca metode de cautare, o reprezintă faptul că ei mențin o populație de soluții posibile, în timp ce alte metode de căutare proceseaza la un moment dat un singur punct din spațiul de căutare. Totuşi, unele tehnici care includ mai multe caracteristici ale spațiului de căutare al unor probleme particulare, pot fi mai eficiente decît metoda generala a AG.

„Uneltele” utilizate într-un AG sunt: generarea aleatorie de soluții, împerecherea soluțiilor, mutația aplicată soluțiilor şi evaluarea soluțiilor, bazată pe valoarea fitness-ului. AG este o metodă de *căutare aleatorie ghidată* sau orientată, în sensul că foloseşte întîmplarea, care depinde de şansă, dar are incorporate o serie de direcționări pentru a căuta soluții efective. AG este o căutare aleatorie ghidată probabilistic deoarece AG îşi orientează căutarea pe baza unei probabilități. Acest tip de căutare este în contrast cu metodele de căutare aleatorie oarbe, cum ar fi metoda *Monte Carlo*. Şi rețelele neuronale sunt metode de căutare aleatorie ghidate. În backpropagation valorile inițiale ale ponderilor sunt alese aleatoriu, dar ponderile capătă valori succesive orientate spre minimizarea erorii între ieşiri şi valorile țintă.

Metoda Hill-Climbing foloseşte tehnici iterative aplicate la un singur punct din spațiul de căutare. La o iterație t un nou punct este selectat dintre vecinii punctului curent. În cazul lucrului cu şiruri binare s_c este mutat sistematic fiecare bit de la stînga la dreapta cîte unul la o iterație şi se înregistrează fitness-ul şirului modificat cu un bit $f(s_n)$. Tehnica se mai numeşte şi tehnică de căutare locală sau căutarea vecinătății. Dacă noul punct are asociată o valoare mai mică pentru o problema de minimizare sau mai mare pentru gasirea unui punct de maxim, noul punct devine punctul curent ($t \leftarrow t + 1$). În caz contrar alte puncte vecine sunt selectate şi testate în raport cu punctul curent. Condiția de terminare este îndeplinită cînd nu se mai poate obține o îmbunătățire a soluției, la o nouă iterație. Această metodă găseste numai valori de optime locale care depinde puternic de fixarea punctului de start al algoritmului. Nu se poate stabili o eroare relativă pentru soluția obținută, în raport cu soluția optimă globala a problemei analizate. Pentru a micşora dependența rezultatului de alegerea punctului de start se repetă metoda de *hill-climbing* pentru un număr mare de puncte de start, selectate aleatoriu, dar aceste puncte se pot alege şi în funcție de rezultatele rulărilor precedente.

Metoda de călire simulată „simulated annealing” elimină o parte din dezavantajele metodei „hill-climbing”, soluțiile nedepinzînd de punctul de start şi acestea situîndu-se de obicei foarte aproape de punctul optim. Matematic, medoda este similară cu strategiile evolutive şi este derivată dintr-un model fizic de cristalizare. În acest caz competiționează doar doi indivizi. Metoda introduce probabilitatea p de acceptare a înlocuirii punctului curent de un nou punct. Dacă noul punct are o valoare mai bună a funcției obiectiv, atunci $p = 1$ şi acesta devine punctul curent. În celelalte cazuri p este dependent de valorile funcției obiectiv pentru punctul curent şi noul punct, depinzînd şi de un paramentru suplimentar de control, temperatura T. Cu cît coboară temperatura T, cu atît şansele de acceptare a noului punct ca punct curent sunt mai mici. Condiția de terminare verifică dacă sistemul a atins „echilibrul termic”, adică dacă distribuția noilor şiruri selectate respectă distribuția Boltzmann. Condiția de stop verifică daca la valori mici ale lui T nu se mai produc schimbări acceptate, cu alte cuvinte sistemul a „inghețat”.

Metoda *NFL* „no free lunch”, este un cadru care se adresează aspectelor de bază ale cautării, focalizîndu-se pe legătura între funcțiile de fitness şi algoritmii de căutare. Terorema *NFL* afirmă că performanța medie este aceeaşi pentru toți algoritmii de căutare. Din ea rezultă faptul că spre exemplu, dacă se compară un AG cu un alt algoritm de căutare, cum ar fi călirea simulată sau chiar o căutare aleatorie, atunci AG are performanțe mai bune pentru o anumită clasă de probleme iar ceilalți algoritmi pentru altă clasă de probleme. Acest lucru implică incorporarea în algoritmii de căutare a unor cunoştințe despre problema în studiu. *NFL* furnizează informații teoretice din interiorul procedurii de căutare. *NFL* se poate referi şi la funcții de fitness variabile în timp, el demonstrează că independent de funcția de fitness nu se poate alege cu succes între doi algoritmi dacă se bazează doar pe rezultatele lor anterioare.

9.13 Permutări

Întrucit în această lucrare sunt utilizați pe larg operatorii genetici care folosesc operații de permutări în spațiul soluțiilor, vom defini în continuare un set de operații specifice.

O permutare a unui set finit este un aranjament al elementelor sale într-un şir. Pentru un set unic *n* de obiecte, există *n!* permutări ale setului de obiecte. Există numeroase proprietăți care sunt relevante pentru manipularea reprezentării permutărilor cu algoritmi evolutivi, atît din punctul de vedere al reprezentării cît şi din punctul de vedere al perspectivei analitice.

Pe masură ce natura cercetărilor a început să aplice algoritmii evolutivi la aplicații care sunt reprezentate în mod nativ cu ajutorul permutărilor, a devenit evident faptul că aceste probleme prezintă modalități diferite de codificare față de problemele de optimizare a parametrilor tradiționale. Pentru anumite tipuri de probleme există soluții echivalente multiple. Atunci cînd este utilizată o permutare în reprezentarea unui ciclu (problema comisului voiajor), toate schimbările consecutive ale permutărilor sunt de asemenea soluții echivalente. Mai mult, toate reversurile unei permutări sunt de asemenea soluții echivalente.

Aceste situații de simetrie, pot cauza probleme pentru algoritmii evolutivi care se bazează pe recombinare. Alt tip de problemă îl constituie faptul că problemele de permutare nu pot fi procesate utilizînd aceiaşi operatori generali de mutație şi recombinare care sunt aplicați la problemele de optimizare a parametrilor. Utilizarea unei reprezentări a permutărilor poate masca diferențele foarte mari în abordarea problemelor de optimizare a parametrilor.

9.13.1 Maparea unui set de întregi în permutări

Un set de *n!* permutări poate fi mapat într-un set de întregi în diverse feluri. Whitley şi Yoo (1995) prezintă următorul algoritm care converteşte un intreg X în permutarea corespunzătoare ordinului său:

1. caută o ordine oarecare a permutării care este specificată pentru sortare
2. -sortează şi îndexează N elemente ($N \geq 1$) ale permutării de la 1 la N.
   -alege un index X pentru o permutare specifică astfel încît $0 \leq X \leq N!$
3. dacă $X=0$, alege toate elementele care ramîn din lista de permutări sortate din secvența în care apar şi opreşte.
4. dacă $X<(N-1)!$ atunci alege primul element din lista ramasă: du-te la 6., altfel continuă
5. gaseşte Y astfel încît $(Y-1)(N-1)! \leq X < Y(N-1)!$. Al Y-lea element din lista sortată este următorul element din permutare. $X = X-(Y-1)((N-1)!)$.
6. şterge elementul ales din lista de elemente sortate: $N=N-1$, du-te la 3.

Algoritmul poate fi de asemenea inversat pentru a mapa întregi către permutări, pentru permutări de lungime 3, se generează următoarea corespondență:

X = 0	index 123	X = 3	index 231
X = 1	index 132	X = 4	index 312
X = 2	index 213	X = 5	index 321

9.13.2 Inversul unei permutări

O proprietate importantă a unei permutări este faptul că poseda un invers bine definit. (Knuth 1973). Fie o permutare $a_1a_2...a_n$ a unui set {1,2,...,n}. Acest lucru poate fi scris în felul următor:

$$\begin{pmatrix} 1 & 2 & 3 & ... & n \\ a_1 & a_2 & a_3 & ... & a_n \end{pmatrix}$$

Inversul este obținut prin reordonarea ambelor rînduri astfel încît cel de al doilea rînd să fie transformat în secvența (1 2 3, ... ,n). Reordonarea la primul care apare este o consecință a reordonării rîndului secund care conține permutarea: $a_1' a_2' ...a_n'$. Knuth (1973) dă ca exemplu o permutare (5 9 1 8 2 6 4 7 3), şi arată că inversul său poate fi obținut astfel:

$$\left(\frac{5\,9\,1\,8\,2\,6\,4\,7\,3}{1\,2\,3\,4\,5\,6\,7\,8\,9}\right) = \left(\frac{1\,2\,3\,4\,5\,6\,7\,8\,9}{3\,5\,9\,7\,1\,6\,8\,4\,2}\right)$$

care conține inversul (3 5 9 7 1 6 8 4 2). Knuth arată că $a'_j = k$ dacă şi numai dacă $a_k = j$.

Inversul va fi folosit ca parte a unei funcții sau pentru funcții de mapare a permutărilor într-o formă canonică, ceea ce permite un schimb mai uşor de a modela problemele cu reprezentări de permutări.

9.13.3 Funcția de mapare

Atunci cînd se modelează algoritmi evolutivi de cele mai multe ori este util să se calculeze o funcție de transmisie $r_{i,j}(k)$ care conține probabilitatea de recombinare a şirurilor *i* şi *j* şi permite obținerea unui şir arbitrar *k*. Witley şi Yoo (1995) explică cum se calculează funcția de transmisie pentru un singur şir *k* şi apoi cum se generalizeaza rezultatele pentru toate celelalte şiruri. În acest caz, şirurile reprezintă funcția de permutare şi remapare, notată cu @, după cum urmează:

$$R_{3421.1342}(3124)=r_{3421@3124.132@3124}(1234).$$

Funcția *r* poate fi generalizată după cum urmează:

$$R_{3421.1342}(3124)=r_{uzyx.xwzy}(wxyz),$$

unde variabilele w, x, y, şi z sunt variabile care reprezintă elementele permutării. (*w=3, x=1, y=2, z=4*). Dacă *wxyz* reprezintă permutarea canonică 1234, atunci avem:

$$r_{wzyx,xwzy}(wxyz)=r_{1432.2143}(1234)=r_{3421.1342@3124}(1234).$$

De asemenea se poate lega acest operator de mapare de procesul de găsire al unui invers. Permutările din expresia:

$$r_{3421.1342}(3124)=r_{1432.2143}(1234)$$

sunt incluse ca rînduri într-o matrice. Pentru a mapa partea stîngă a expresiei precedente pentru termenii din partea dreaptă, mai întîi se calculează inversele pentru fiecare din termenii din partea stîngă.

$$\left(\frac{3\,4\,2\,1}{1\,2\,3\,4}\right)=\left(\frac{1\,2\,3\,4}{4\,3\,1\,2}\right)$$

$$\left(\frac{1\,3\,4\,2}{1\,2\,3\,4}\right)=\left(\frac{1\,2\,3\,4}{1\,4\,2\,3}\right)$$

$$\left(\frac{3\,1\,2\,4}{1\,2\,3\,4}\right)=\left(\frac{1\,2\,3\,4}{2\,3\,1\,4}\right)$$

Se colectează cele trei inversoare într-un singur vector. Se adaugă apoi (1 2 3 4) în vector şi se inversează permutarea (2 3 1 4), în acelaşi timp se rearanjează toate celelalte permutări din vector:

$$\begin{pmatrix} \frac{4\,3\,1\,2}{1\,4\,2\,3} \\ \frac{2\,3\,1\,4}{1\,2\,3\,4} \end{pmatrix}=\begin{pmatrix} \frac{1\,4\,3\,2}{2\,1\,4\,3} \\ \frac{1\,2\,3\,4}{3\,1\,2\,4} \end{pmatrix}$$

Aceasta conţine permutările (1 4 3 2), (2 1 4 3) şi (1 2 3 4) care respectă forma canonică dorită şi este legată de noţiunea de substituţie într-o formă simbolică canonică. De asemenea procesul de găsire a unei permutări p_i şi p_j poate fi inversat în următorul context:

$$r_{1432.2143}(1234)=r_{p_i \cdot p_j}(3124)=r_{p_i \cdot p_j}(3124)=r_{p_i @3124.p_j @3124}(1234)$$

9.13.4 Metode de permutare

Un atribut evident al problemelor de permutare este că operatorii de crossover simpli eşuează în generarea urmaşilor care sunt tot permutări. Davis (1985) şi Goldberg şi Lingle (1985) au definit o parte din primii operatori pentru problemele de permutare.

9.13.4.1 Aplicarea metodei de crossover propusă de Davis

Se aleg 2 permutări pentru recombinare. Se notează primul părinte ca şir de tăiere şi al doilea ca şir de umplere. Se selectează 2 puncte de crossover. Se copie sublista de elemente de permutare dintre punctele de crossover din şirul de tăiere direct în urmas, plasîndu-le în aceeaşi poziţie absolută. Pornind de la al 2-lea punct de crossover, se gaseşte următorul element din şirul de umplere care nu apare în urmaş. Pornind de la al 2-lea punct de crossover, se plasează elementul de la şirul de umplere în următoarea secţiune liberă din urmaş. Atunci cînd se ajunge la sfîrşitul şirului de umplere, se întoarce înapoi la începutul şirului.

Procedînd în acest fel, crossover-ul ordonat al lui David are proprietatea de recombinare pură (Radcliffe 1994): „atunci cînd sunt combinaţi 2 părinţi identici, atunci urmaşii sunt identici cu părinţii”. Dacă nu se copie elementele din şirul de umplere începînd cu al 2-lea punct de crossover, atunci recombinarea poate să nu fie pură.

Exemplu: Părintele 1: A B . C D E F . G H I
Crossover: _ _ _ C D E F _ _ _

Părintele 2: h d . a e i c . f b g
Elemente libere în ordine: b g h a i

Urmaş: a i C D E F b g h

Elementele din secţiunea de crossover păstrează ordinea relativ, poziţia absolută, şi adiacenţa de la părintele 1. Elementele care sunt copiate de la şirul de umplere păstreaza doar informaţia de ordine relativă de la al 2-lea părinte.

9.13.4.2 Metoda de crossover mapat partial (PMX)

Goldberg şi Lingle (1985) au introdus operatorul de crossover mapat parţial (PMX). Metoda PMX este asemănătoare cu metoda lui David de crossover ordonat. Un şir părinte este desemnat ca părintele 1, iar celălalt ca părintele 2. Două puncte de crossover sunt selectate şi toate elementele din părintele 1 dintre punctele de crossover sunt copiate direct în urmaş. Acest lucru înseamnă că metoda PMX defineşte de asemenea o secţiune de crossover în aceeaşi maniera ca şi crossover-ul ordonat:

Părintele 1: A B . C D E . F G
Selecţia crossover: _ _ C D E _ _

Părintele 2: c f . e b a . d g .

Diferenţa dintre cei 2 operatori este faptul cum se face copierea în metoda PMX din părintele 2 în secţiunile libere din urmaş după ce a fost definită o secţiune de crossover. Se notează părinţii ca P_1 şi P_2 şi urmaşii ca *OS*, iar P_{1_i} denotă al *i*-lea element din permutarea P_1.

Pentru acele elemente dintre punctele de crossover din părintele 2, dacă elementul P_{2_i} a fost deja copiat în offspring, atunci nu se ia nici o acţiune. În exemplul de mai sus, elementul *e* din parintele 2 nu necesită nici o procesare. Pentru celelalte elemente se consideră poziţia în care apar ele în secţiunea de crossover. Dacă următorul element de la poziţia P_{2_i} din părintele 2 nu a fost deja copiat în urmaş, atunci găseşte $P_{1_i} = P_{2_i}$; dacă poziţia *j* nu a fost copiată în urmaş atunci asignează $OS_j = P_{2_i}$. Următorul element din secţiunea de crossover a părintelui 2 din exemplul dat este *b* care este în aceeaşi poziţie cu *D* din părintele 1. Elementul *D* este localizat în cadrul parintelui 2 cu indexul 6 şi urmaşul OS_6 nu a fost copiat. Se copiază *b* în urmaş pe poziţia corespunzătoare. Acest lucru rezultă în :

Offspring: _ _ C D E b _ .

O problemă apare atunci cînd se încearcă plasarea elementului *A* în urmas. Elementul *A* din părintele 2 este mapat către elementul *E* din părintele 1. *E* pică în poziţia 3 în părintele 2, dar poziţia 3 a fost deja ocupată în urmaş, fiind ocupată de *C*, deci se caută elementul *C* în părintele 2. Poziţia este neocupată în urmaş, asfel încît elementul *A* este plasat în urmaş la poziţia ocupată de *C* în părintele 2. Acest lucru rezultă în:

Offspring: a _ C D E b _ .

Toate elementele din părintele 1 şi părintele 2 care cad în interiorul secţiunii de crossover au fost plasate în cadrul urmaşului. Elementele rămase pot fi plasate prin copierea directă a pozitiilor acestora din părintele 2. Aceasta conduce la următorul rezultat:

Offspring: a f C D E b g.

9.13.4.3 Metoda de crossover ordonat

Metoda crossover-ului ordonat (Syswerda 1991) precum şi cea a crosssover-ului poziţional sunt diferite de metoda lui David şi metoda PMX prin faptul că nu există un bloc învecinat care este direct pasat urmaşului. În schimb sunt selectate foarte multe elemente aleator prin poziţia absolută.

Operatorul de crossover de ordinul 2 porneste prin selecţia a *K* poziţii aleatoare în părintele 2, unde părinţii sunt de lungime *L*. Elementele corespunzătoare din părintele 2 sunt apoi localizate în părintele 1 şi reordonate astfel încît să apară în aceeaşi ordine relativa în care apar în părintele 2. Elementele din părintele 1 care nu corespund elementelor selectate din părintele 2 sunt pasate direct urmaşilor.

Exemplu: Părintele 1: A B C D E F G
  Părintele 2: C F E B A D C
  Elementele selectate: * * * .

Elementele selectate din părintele 2 sunt F, B şi A. Elementele relevante sunt reordonate în părintele 1. Apoi se reordonează A B _ _ _ F _ din părintele 1 care conţine
f b _ _ _ a _ .

Toate celelalte elemente sunt copiate direct din părintele 1. Toate celelalte elemente sunt copiate direct din părintele 1:

(f b _ _ _ a _) combinate cu (_ _ C D E _ G) conţin sirul (f b C D E a G).

9.13.4.4 Metoda de crossover poziţional

Syswerda defineşte un operator secundar denumit crossover poziţional. Utilizînd acelaşi exemplu, prima data se aleg *L-K* elemente din părintele 1 care sunt direct copiate în urmaş. Aceste elemente sunt copiate prin poziţie. Acest lucru conduce la următoarea progresie:

$$\# \# C D E \# G \rightarrow f \# C D E \# G \rightarrow f b C D E \# G \rightarrow f b C D E a G.$$

În mod evident, în acest caz, cei doi operatori generează exact acelaşi urmaş.

9.13.4.5 Metoda de crossover uniform

Crossover-ul uniform propus de Davis (1991) este identic cu crossover-ul poziţional şi crossover-ul ordonat de ordinul 2, cu excepţia faptului că sunt generaţi 2 urmaşi. Un şir de biţi este utilizat pentru a denota selecţia poziţiilor. Urmaşul 1 copie elementele direct din părintele

1 în acele poziții din şirul de biți marcate cu biții de 1. Urmaşul 2 copie elementele din părintele 2 în acele pozitii marcate cu biții de 0. Ambii urmaşi copie apoi elementele rămase de la celălalt părinte într-o ordine relativă.

9.13.4.6 Metoda recombinării muchiilor

Metoda a fost introdusă ca un operator specializat pentru problema comisului voiajor. Motivația din spatele acestui operator este că acesta poate păstra adiacența dintre elementele de permutare, întrucît costul unui tur în problema comisului voiajor este direct legata de setul de relații de adiacență (distanța dintre oraşe) care există între elementele permutate. Ca exemplu, se consideră următoarele 2 tururi ca părinți pentru recombinare:

Părintele 1: g d m h b j f i a k e c
Părintele 2: c e k a g b h i j f m d.

O listă de muchii este construită pentru fiecare oraş din turneu. Lista de muchii pentru un anume oraş a este compusă din toate oraşele din cei 2 părinți care sunt adiacenți oraşului *a*. Dacă un anume oraş este adiacent lui *a* în ambii părinți, atunci acest lucru este marcat cu un semn „+”. Algoritmul este descris după cum urmează:

a) alege un oraş aleator ca oraş curent inițial; şterge toate referințele la acest oraş din tabela de muchii
b) caută în lista de adiacență a oraşului curent; dacă există un nod marcat cu „+” atunci se merge la acel oraş în continuare. Altfel alege dintre orasele din lista de adiacenta curentă următorul oraş pentru a fi unul al carui adiacență proprie este cea mai mică. Legaturile sunt rupte aleator. Odată ce un oras a fost vizitat, sunt sterse referințele din lista de adiacență a celorlalte oraşe şi nu mai este accesibil din celelalte oraşe
c) se repetă pasul 2 pînă cînd turul este complet sau un oras a fost accesat şi nu mai deține intrări în lista de adiacență. Dacă nu au fost vizitate toate oraşele, atunci se alege aleator un nou oraş pentru a începe un nou tur parțial.

Oras	Lista de muchii
a	+k, g, i
b	+h, g, j
c	+e, d, g
d	+m, g, c
e	+k, -c
f	+j, m, i
g	a, b, c, d
h	+b, i, m
i	h, j, a, f
j	+f, i, b
k	+e, +a
m	+d, f, h

Tabelul 16: Tabelul muchiilor

9.13.4.7 Metoda conservativă maximală (MPX)

Această metoda a fost propusă de către Mühlenbein în 1991 şi este descrisă în următoarea secvenţă în pseudocod pentru operatorul de crossover conservativ maximal, MPX:

a) alege poziţia $0 \leq i <$ nr. noduri şi lungimea $b_{low} \leq k \leq b_{up}$ aleator.
b) extrage din şirul de muchii de la poziţia i pînă la poziţia $j = (i+k)$ *MOD* noduri de la partener (donor)
c) copie şirul de crossover în urmaş
d) adaugă succesiv muchii noi pîna cînd urmaşul reprezintă un tur valid

În examplul următor se ilustrează operatorul MPX:

Receptor:	G D M H B J F I A K E C
Donator:	c e k a g b h i j f m d
Segment iniţial:	_ _ k a g _ _ _ _ _ _ _ .

Se observa că G este conectat la D în receptor şi că elementele de la D pînă la I pot fi luate de la receptor fără a duplica oricare din elementele deja aflate în urmaş. Acest lucru produce urmîtorul tur parţial: _ _ k a g D M H B J F I.

În acest punct nu există nici o muchie în oricare din părinţi care să fie conectată la I sau nu a fost utilizată. În acest caz metoda MPX sare oraşele din receptor pînă cînd găseşte una care nu a fost utilizată. În acest caz ajunge în E. Acest lucru face ca să fie adăugate şi E şi C la turneu, astfel încît se obţine:

E C k a g D M H B J F I.

Se poate observa că metoda MPX nu transmite informaţii de adiacenţă de la părinţi la urmaşi precum alţi operatori de recombinare a muchiilor, deoarece utilizează mai puţin căutarea înainte pentru a evita o ruptură în construcţia rezultatului.

Operatorii prezentaţi sunt utilizati pentru păstrarea informaţiilor de adiacenţă (recombinarea muchiilor) sau informaţii despre ordinea relativă (crossover uniform sau bazat pe ordine). Operatorii trebuie să accepte şi lucrul cu poziţii ale cromozomilor.

Doi operatori care realizează acest lucru sunt: operatorul de crossover ciclic precum şi cel de fuziune (MXI).

9.14 Selecţia

În utilizarea selecţiei prin competiţie, se alege un grup q de invizivi din populaţia de soluţii. Aceştia pot fi aleşi din populaţie cu sau fără înlocuire. Acest grup face parte din competiţie: un individ învingător este determinat în funcţie de valoarea sa de fitness. Cel mai bun individ care are cea mai bună valoare a fitness-ului este de obicei ales în mod determinist, prin urmare se poate face printr-o selecţie ocazională stohastică. În ambele cazuri, cîştigătorul este inserat în populaţia următoare iar procesul este repetat de λ ori pentru a obţine o nouă populaţie. De obicei competiţia se face între doi indivizi (competiţie binară). Bineînţeles, competiţia se poate extinde la un grup arbitrar de mărime q, denumit mărimea competiţiei.

Descrierea următoare presupune că indivizii sunt selectati prin înlocuire iar individul învingător este selectat în mod deterministic:

Intrare: *Populația* $P(t) \in I^{\lambda}$, *mărimea competiției este* $q \in \{1, 2, \ldots, \lambda\}$
Ieşire: *Populația după selecția* $P(t')$

- *Competiție* $(q, a_1, a_2, \ldots, a_{\lambda})$:
- *Pentru* $i = 1$ *pină la* λ
  - $a'_i \leftarrow$ *cel mai bun individ din q indivizi aleşi aleator din* $\{a_1, a_2, \ldots, a_{\lambda}\}$;
- *returnează* $\{ a'_1, a'_2, \ldots, a'_{\lambda}\}$

Selecția prin metoda competiției poate fi implementată foarte eficient şi are o complexitate egală cu $O(\lambda)$ întrucît nu este necesară nici o sortare a populației. Algoritmul descris mai sus are o variație mare a numărului de urmaşi întrucît este efectuat un număr independent de încercări λ. În cazul selecției prin metoda competiției o scalare sau o translare a valorii funcției de fitness nu afectează comportamentul metodei de selecție. Prin urmare, în acest tip de selecție nu sunt necesare tehnicile de scalare utilizate pentru selecția proportională, ceea ce conduce la o simplificare a aplicației şi o îmbunătățire a performanțelor algoritmului evolutiv utilizat.

Mai mult, selecția prin metoda competiției este foarte potrivită pentru algoritmii evolutivi paraleli. În cele mai multe scheme de slecție, sunt necesare calcule globale pentru a calcula ratele de reproducere a indivizilor. Spre exemplu, în cazul selecției proporționale, sunt necesare valorile funcției de fitness din populație, iar în cazul selecției prin metoda rangurilor sau prin selecția prin truncare este necesară o sortare a întregii populații de soluții.

În acest caz, competițiile pot fi efectuate între ele în mod independent, astfe încît doar grupuri de q indivizi trebuie să comunice între ele.

9.14.1 Setarea parametrilor

În cazul în care $q = 1$, nu este efectuată nici o selecție, iar indivizii sunt aleşi aleator din populație. Selecția prin competiție binară este echivalentă cu selecția liniară de rang cu $\eta^{-} = \frac{1}{\lambda}$ (Blikle si Thiele, 1995), unde η^{-} exprimă numărul de urmaşi al individului cu fitnesul cel mai mic. O dată cu creşterea valorii lui q, creşte presiunea operatorului de selecție. Pentru multe aplicații cum ar fi programarea genetică, s-au folosit cu succes şi valori mai mari ale lui q.

9.14.2 Intensitatea selecției

Intensitatea operatorului de selecție este altă măsură a puterii selecției care este împrumutată de la genetica populației. Intensitatea de selecție, S, este gradul de schimbare din funcția de fitness relativă a populației datorită faptului că selecția este împărțită prin intermediul variației populației înaintea selecției σ care este notată cu $S = \frac{(u^{*} - u)}{\sigma}$, cu o funcție de fitness medie u înainte de selecție, şi u^* dupa selecție.

Pentru a elimina dependența de intensitatea de selecție din populația inițială se presupune existența unei populații inițiale distribuită Gaussian (Mühlenbein şi Schlierkamp-

Voosen, 1993). Plecînd de la această premiză, intensitatea de selecție a selecției prin metoda competiției este determinată de către următoarea ecuație:

$$S_{tour}(q) = \int_{-\lambda}^{\lambda} qx \frac{1}{(2\pi)^{1/2}} e^{-x^2} \left(\int_{-\lambda}^{\lambda} \frac{1}{(2\pi)^{1/2}} e^{-y^2/2} d_y \right)^{q-1} d_x$$

9.15 Evaluarea funcției de fitness

Majoritatea algoritmilor genetici necesită codificare, adică maparea din reprezentarea cromozomilor la structuri de domeniu (parametri). Operatorii de recombinare (crossover sau mutație) lucrează direct pe reprezentarea codificată, nu pe structurile de domeniu.

Se presupune că o problemă de optimizare este dată sub forma:

$$f = M \rightarrow \Re \qquad (1)$$

unde *M* este spațiul de căutare al funcției obiectiv *f*. Atunci funcția de evaluare a fitness-ului F este descrisă după cum urmează:

$$F : R \xrightarrow{d} M \xrightarrow{f} \Re \xrightarrow{s} \Re_{+} \quad (2)$$

$$F = s \circ f \circ d \quad (3)$$

unde *R* este spațiul reprezentării cromozomilor, *d* este o funcție de decodare, iar *s* este o funcție de scalare. Funcția de scalare *s* este de obicei utilizată în combinație cu selecția proporțională pentru a garanta valorile de fitness pozitive şi maximizarea funcției de fitness. Spre exemplu, atunci cînd se codifică o funcție obiectiv f_n cu valori reale printr-o codificare binară, atunci funcția de fitness *F* definită mai sus se redefineşte astfel:

$$F : \{0,1\}^l \xrightarrow{d_b} \Re^n \xrightarrow{f_n} \Re \xrightarrow{s} \Re_{+} \quad (4)$$

unde *l* este lungimea unui cromozom şi d_b este codificarea binară, astfel încît d_b mapează segmente de cromozom în numere reale de dimensiuni corespunzătoare. Evaluările cromozomilor sunt transformate în valori de fitness prin diverse metode. Spre exemplu, există multe scheme de codare care utilizează un set de caractere binare care poate codifica un parametru cu acelasi sens, cum ar fi un cod binar şi un cod Gray. Rezultatele experimentale au arătat că codificarea Gray este superioară celei binare doar pentru un set particular de optimizări de funcții pentru algoritmii genetici. Analizele sugerează că codul Gray elimină problema distanței Hamming care face ca unele tranziții să fie dificile pentru reprezentări binare. Prin urmare, schema de codificare este foarte importantă pentru îmbunătățirea eficienței de căutare a algoritmilor genetici.

În contrast cu algoritmii genetici, strategiile evolutive şi programarea evolutivă lucrează direct în spațiul secund *M*, astfel încît ele nu mai necesită decodificarea funcției *d*. Mai mult, de obicei nu necesită o funcție de scalare *s*, astfel încît funcția de evaluare a fitness-ului sa fie specificată complet de către ecuația (1).

Este greu de a se calcula o soluție cu o acuratețe globală pentru probleme complexe. Dificultatea provine din obiectivitatea funcției de fitness, care de obicei vine doar cu costul unor cunoştințe semnificative despre spațiul de căutare. Pentru a elimina încrederea în funcții de fitness obiective, este utilizată *competiția*.

Funcția de fitness a competitivității este o metodă de calcul a funcției de fitness care este dependentă de populația curentă, în timp ce o funcție de fitness standard returnează acelaşi fitness pentru un individ indiferent de ceilalți membri din populație.

Avantajul competitivității este că algoritmii evolutivi nu necesită o valoare a funcției de fitness exactă, întrucît majoritatea schemelor de selecție merg doar prin compararea fitness-ului, de multe ori criteriul „mai bun, mai rău” fiind de ajuns. În acest fel, metoda este mult mai eficientă din punct de vedere computațional şi mai usor de implementat cu algoritmi paraleli.

9.16 Exemplu de implementare a unui algoritm evolutiv

Implementarea în limbajul pseudocod al unui algoritm evolutiv este descrisă în exemplul de mai jos.

```
// inițializează numărătorul de ceas
t = 0;
// inițializează o populație normală aleatoare de indivizi
initializare_populatie P(t);
// evaluează funcția de fitness pentru toti indivizii din populație
evaluare P(t);
// testează pentru criteriul de terminare (timp, fitness, etc)
while (neterminat) do {
        // incrementează numărătorul de ceas
        t = t + 1;
        // selectează sub-populația pentru producerea urmaşilor
        p’ = selectare_parinti P(t);
        // recombină genele părinților selectați
        recombina P(t);
        // perturbă populația împerecheată stohastic
        mutatie P’(t);
        // evaluează noua funcție de fitness
        evaluare P’(t);
        // selectează urmaşii din funcția de fitness actuală
        P = urmasi (P, P’(t))
}
```

9.17 Circuite hardware evolutive

Circuitele hardware evolutive sunt construite pe celule reprogramabile de tip FPGA şi pot fi reconfigurate prin utilizarea tehnologiilor evolutive care se pot adapta la schimbările mediului înconjurător. Dacă apar erori hardware sau sunt necesare noi funcții hardware, atunci circuitele hardware evolutive îşi pot altera structura lor hardware pentru a se putea acomoda la astfel de schimbări.[107]

Cercetarea în domeniul circuitelor hardware evolutive a fost inițiată în mod independent în Japonia şi în Suedia după anii 90. Începînd de atunci, interesul acordat acestui domeniu a crescut considerabil. Primul workshop internațional despre circuite hardware evolutive, EVOLVE95, a fost tinut la Lausanne în octombrie 1995.

Există viziuni diferite despre circuitele hardware evolutive. Una dintre ele priveste circuitele hardware evolutive ca o aplicație a tehnicilor evolutive pentru sinteza circuitelor.

Altă abordare priveşte circuitele hardware evolutive ca un echipament hardware care este capabil de adaptare online prin reconfigurarea arhitecturii sale în mod dinamic şi autonom. Circuitele hardware evolutive reprezintă un obiect de cercetare diferit de algoritmii evolutivi, în care arhitectura hardware ramine neschimbată şi este utilizată pentru implementarea de funcţii ale algoritmilor evolutivi, cum ar fi selectia, recombinarea şi mutaţia.

Principalul avantaj al implementării hardware a algoritmilor evolutivi este de a mări viteza de execuţie a funcţiilor lor specifice. Mărirea vitezei nu implică o execuţie mai rapidă a algoritmului evolutiv deoarece nu se măreşte evaluarea funcţiei de fitness care este de cele mai multe ori partea cea mai mare consumatoare de timp a aplicaţiei evolutive.

Clasificarea circuitelor hardware evolutive se poate face după tipul de implementare al acestora: exstrinsec şi intrinsec (deGaris 1994). Circuitele hardware evolutive extrinseci simulează procesul evolutiv prin software şi descarcă cea mai bună configuraţie în hardware la fiecare generaţie o singură dată. Circuitele hardware evolutive intrinseci simulează procesul de evoluţie direct în hardware, fiecare cromozom fiind utilizat pentru reconfigurarea dinamică a circuitului hardware. Acest lucru rezultă într-un număr de reconfigurări egal cu mărimea populaţiei din fiecare generaţie. Unul din avantajele circuitelor hardware evolutive este acela că se pot obţine reprezentări simbolice ale circuitelor, care mai apoi pot fi implmenetate într-o varietate de platforme digitale.

Abordări mai recente pun accentul pe proiectarea evolutivă a circuitelor complexe în scopul proiectării de circuite hardware evoluate dinamic. Din punct de vedere al nivelului reprezentării cromozomilor, implementarea circuitelor poate fi clasificată în abordări directe şi indirecte. În abordarea directă circuitele hardware evolutive codifică biţii arhitecturali ai circuitului ca şi cromozomi, care specifică conectivitatea şi funcţiile diferitelor componente hardware la nivel redus al circuitului. Pe de altă parte, în abordarea indirectă nu se evoluează biţii arhitecturali în mod direct, ci se utilizează o reprezentare de nivel înalt, cum ar fi arbori şi gramatici, ca şi cromozomi, care sunt apoi utilizaţi pentru a genera circuite. O gramatică este reprezentată de o diagramă de producţie, care poate fi un graf orientat.

Algoritmul evolutiv cel mai utilizat pentru a evolua circuitele hardware este algoritmul genetic, în care fiecare prototip de circuit este codificat cu un şir pe un bit. Algoritmul separă în mod explicit informaţia genetică care este recombinată şi mutată (genotipul) din circuitul curent care este evaluat (fenotipul). Funcţia de fitness trebuie să evalueze fiecare pas al algoritmului evolutiv şi trebuie să fie calculată pentru fiecare membru al populaţiei de evoluat. [109]

Altă clasificare a metodelor de cercetare din domeniul circuitelor hardware evolutive poate fi o clasificare a acestora în 2 grupuri: orientate către inginerie pe de o parte, şi către embriologie pe de altă parte. Abordarea orientată către partea de inginerie se referă la dezvoltarea unei maşini care poate să îsi schimbe structura sa hardware. De asemenea încearcă să dezvolte o nouă metodă: proiectarea circuitelor hardware fără proiectanţi umani. Acest grup include activităţi în ETL, ATR, HIP Research Labs, şi Sussex University. [26]

Abordarea orientată pe partea de embriologie se referă la dezvoltarea unor maşini care se pot reproduce şi repara singure. Cercetări în acest sens se fac la Swiss Federal Institute of Technology şi ATR. Ambele se bazează pe analiza automatelor celulare bidimensionale. [15]

În circuitele hardware evolutive, reprezentările genotipurilor funcţiilor hardware sunt în final transformate în structuri hardware implementate în circuite FPGA. Există diverse tipuri de reprezentări. Spre exemplu, ideea de bază în implementarea evolutiva ETL este de a privi biţii arhitecturali ai circuitului FPGA ca genotipuri care sunt apoi manipulate de către un algoritm genetic. Algoritmul genetic caută pentru cei mai potriviţi biţi arhitecturali. O dată ce s-a găsit genotipul potrivit, atunci este mapat direct în celula FPGA.

Evoluţia hardware este denumită evoluţie la nivel de poartă logică întrucît fiecare genă poate corespunde unei porţi logice fundamentale cum ar fi o poartă *AND*.

9.17.1 Abordarea orientată pe inginerie

Începînd din 1992, ETL a desfaşurat cercetări în evoluţia la nivel de poartă logică şi a dezvoltat 2 sisteme de aplicaţii. Una din ele este un robot de sudură prototip a cărei parţi de control poate fi controlată de către un circuit hardware evolutiv atunci cînd are loc o eroare hardware. Utilizînd algoritmul evolutiv, algoritmul învaţă circuitul de implementat fără a avea nici o cunoştinţă anterioară despre circuit, atunci cînd funcţiona corect. Alta este un sistem flexibil de recunoaştere a formelor care foloseşte robusteţea reţelelor neuronale artificiale. În timp ce reţeaua neuronală învaţa funcţiile nesensibile la zgomot prin ajustarea greutăţii şi a pragului unităţilor neuronale, circuitele hardware evolutive implementează aceste funcţii direct în hardware prin învăţarea evolutivă.[53] De recent s-a trecut la evoluţia la nivel de funcţie, unde fiecare genă corespunde unei funcţii reale cum ar fi multiplicarea în virgulă mobilă şi funcţii trigonometrice. Evoluţia la nivel de funcţie poate ajunge la performanţe comparabile cu cele ale reţelelor neuronale. Pentru ca această evoluţie să fie posibilă, este proiectat şi implementat un circuit dedicat ASIC.

De asemenea se fac cercetări şi in domeniul descrierilor hardware în limbaje HDL utilizînd programarea genetică. Limbajul HDL este generat automat şi o dată ce s-a obţinut o descriere hardware, aceasta poate fi implementată imediat în circuitul hardware real. Pînă în prezent s-au obţinut cu succes circuite simple, cum ar fi sumatoarele.

9.17.2 Abordarea orientată pe embriologie

S-au făcut cercetări active la Institutul Federal de Tehnologie din Suedia care se referă la dezvoltarea unui circuit FPGA care se poate auto-reproduce şi auto-repara.

Un punct foarte interesant este acela că descrierea hardware este reprezentată prin diagrame de decizii binare (BDD) şi că aceste BDD-uri sunt tratate ca genomi de către algoritmul genetic. Fiecare bloc logic al circuitului FPGA citeşte partea genomului care descrie funcţia sa şi este reconfigurat corespunzător. Dacă un anume bloc este deteriorat, genomul poate fi utilizat pentru a realiza o operaţiune de auto-reparare: unul dintre blocurile logice de rezervă va fi reconfigurat conform descrierii blocului deteriorat.

Alt tip de cercetare în acest domeniu este evoluarea unei reţele neuronale utilizînd o maşina de tip automat celular bidimensional (maşina MIT CAM8) în scopul de a construi un creier artificial.[25] Reţeaua neuronală este formată ca un model pentru construirea unui automat celular bidimensional prin evoluarea regulilor de tranziţie a stărilor utilizînd un algoritm genetic.

Aria de cercetare în domeniul circuitelor hardware evolutive este nouă şi prezintă un puternic potenţial de cercetare în domeniul aplicaţiilor fascinante care nu au putut fi abordate pină acum datorită necesităţilor de adaptare sau datorită lipsei unui sistem cu răspuns în timp real. Aceste aplicaţii pot include comunicaţii multimedia şi procesare digitală adaptivă. Pentru a putea fi utilizate în practică, circuitele hardware evolutive trebuie să găseasca algoritmi de învăţare mai rapizi şi aplicaţii specifice.

Capitolul 10

Aplicații ale algoritmilor evolutivi în sinteza circuitelor digitale complexe

Optimizarea unui automat finit în circuitele moderne pe perioada procesului de sinteză este o problemă NP-completă. Metodele euristice sunt utilizate pentru a diviza automatele finite complexe într-o rețea de automate finite. Rețeaua de automate finite rezultată este compusă din subautomate care interacționează între ele. În această lucrare se va implementa un pachet de programe care pot fi utilizate într-un pas de pre-sinteză pentru descompunerea generalizată a automatelor finite deterministe. Acest pachet de programe va diviza un singur automat finit într-o rețea de subautomate prin reducerea complexității fiecărui subautomat în timp ce se încearcă o minimizare a numărului de submaşini obținute. Aplicarea acestui tip de descompunere generalizată pentru orice automat finit va acoperi alte metode de descompunere cum ar fi: descompunerea în cascadă, în serie, în paralel, sau descompunerea prin factorizare.

10.1 Preliminarii

Legea lui Moore prezice că dimensiunile circuitelor integrate digitale se vor dubla la fiecare 18 luni. Ca un rezultat la acest fapt, perioadele de ceas necesare se injumătățesc, ceea ce face mult mai dificil pentru un anume set de circuite cum ar fi automatele finite complexe să satisfacă perioadele de timp de răspuns. Sinteza circuitelor complexe poate rezulta în extragerea de automate finite complexe. Programele de sinteza actuale reuşesc cu greu să facă față complexității crescînde a circuitelor. Productivitatea proiectanților şi uneltele EDA abia țin pasul cu frecvențele crescătoare a perioadei de ceas şi a numărului de tranzistoare (creşterea densității de integrare).

Descompunerea funcțională a fost utilizată cu succes dintre alte metode pentru sinteza logică a platformelor FPGA. Descompunerea generală poate fi utilizată pentru a analiza şi descompune orice sistem discret, binar, multivalent, sau simbolic, din domeniul ingineriei moderne şi al stiintei. L. Józwiak este unul din cei care au formulat teoria şi metodologia descompunerii generale care are de a face cu realizarea comportamentului automatelor mari prin rețele de automate interconectate, mai mici, care operează în paralel, denumite maşini parțiale.[82] În această lucrare se va dezvolta un program care aplică metoda descompunerii generale a automatelor finite deterministe pentru a produce circuite optimizate pentru timp de răspuns şi suprafață optimă. Se va testa eficiența rezultatelor pe un set de circuite de benchmark standard precum şi pe un număr de automate finite deterministe generate automat cu ajutorul pachetului de programe.

Datorită optimizării automatul inițial va utiliza un număr mai mic de porți logice şi va putea funcționa la o frecvență mai ridicată decît circuitele prototip. Metoda descompunerii generalizate împarte un circuit complex în subcircuite mai mici care sunt conectate între ele. Metoda creează o nouă mapare a stărilor pentru fiecare subautomat pe baza funcției de fanin

sau fanout a fiecărei variabile de stare şi a intrărilor la fiecare subautomat. Problema descompunerii stărilor este de asemenea considerată un caz special al descompunerii generalizate. Un caz particular special al unei descompuneri generale este descompunerea funcțională. Aceasta are de a face cu automate finite care au o singură stare şi comportament de stări neimportante, cu funcții combinaționale. În această abordare, o funcție combinațională mare este realizată printr-o rețea de funcții mai mici.

O maşină de stări finite M poate fi descrisă de o tuplă de 5 elemente $M = (S, I, O, \delta, \lambda)$, unde S este setul de simboluri de stare, I este setul de intrări primare, O este setul de ieşiri primare, $\delta : I \times S \rightarrow S$ este funcția de stări următoare şi $\lambda : I \times S \rightarrow O$ este funcția de ieşire pentru un automat Mealy iar $\lambda : S \rightarrow O$ este funcția de ieşire pentru un automat Moore. Un automat finit poate fi reprezentat printr un graf de tranzitii (**STG**) sau prin o tabelă de stări (**STT**). Automatele Moore sunt un caz special al automatelor Mealy. Prin urmare, setul de metode şi teorii utilizate pentru automatele Mealy pot fi de asemenea aplicate şi pentru automatele de tip Moore. În continuare se vor aplica metodele de descompunere pe automatele de tip Mealy. În cazul automatelor incomplet specificate, funcția de stări următoare este de forma $\boldsymbol{\delta} : \boldsymbol{I} \times \boldsymbol{S} \rightarrow \boldsymbol{2^S}$, iar funcția de ieşire este de forma $\boldsymbol{\lambda} : \boldsymbol{I} \times \boldsymbol{S} \rightarrow 2^{\boldsymbol{O}}$.

Pachetul de programe citeşte descrierea automatului finit în format de intrare Kiss şi generează un rezultat în format Kiss sau cod verilog. Stările unui automat finit sunt listate în automatul prototip (original) ca nume de stări simbolice. Fiecare nume de stare simbolică are o valoare binară corespunzătoare. Metoda de descompunere va schimba numărul de stări şi va re-codifica valorile stărilor. Minimizarea simbolică realizează minimizarea logică inainte de recodificarea stărilor subautomatelor. Minimizarea simbolică a fost implementată de către DeMicheli în KISS (1985). O parte din neajunsurile din KISS sunt rezolvate în succesorul său, NOVA, care are vine cu o abordare mai eficientă şi flexibilă în rezolvarea constringerilor, reprezentînd problema ca o problemă de îmbunătățire a grafului de soluții.

Definiție: O partiție Π pe un set de stări S este o colecție de subseturi disjuncte din S, denumite **blocuri**, ale căror uniune de seturi este S.

Definiție: O partiție Π pe setul de stări S ale unei maşini M se numeşte **partiție închisă** dacă şi numai dacă două stări s şi t care sunt în acelaşi bloc al lui Π, şi pentru orice intrare $i \in I$, stările următoare $\delta(s,i)$ şi $\delta(t,i)$ sunt într-un bloc comun din Π.

Definiție: O partiție este denumită **partiție generală** dacă nu este închisă. Maşina originală este denumită **maşina prototip** iar maşinile individuale care contribuie la funcționarea generală sunt denumite **submaşini**. (Fig. 33)

Definiție: Maşina obținută ca rezultat al descompunerii este denumită **maşină descompusă** şi este implementată ca o rețea de maşini interconectate . Structura generală a unei astfel de rețele este prezentată în Fig. 33.

Partiția zero, notată cu $\Pi(0)$, denotă o partiție cu $|S|$ blocuri astfel încît fiecare bloc conține exact o stare. Se poate demonstra că o maşina M poate fi descompusă într-un set de ***n*** maşini care interactionează între ele şi care realizează aceeaşi funcție cu masina M dacă şi numai dacă există un set de partiții netriviale astfel încît să existe următoarea definiție:

Definiție: Există o descompunere validă a maşinii M dacă pentru un set de partiții dat $\Pi_1, \Pi_2, \ldots, \Pi_n$, produsul lor este egal cu $\Pi(0)$.

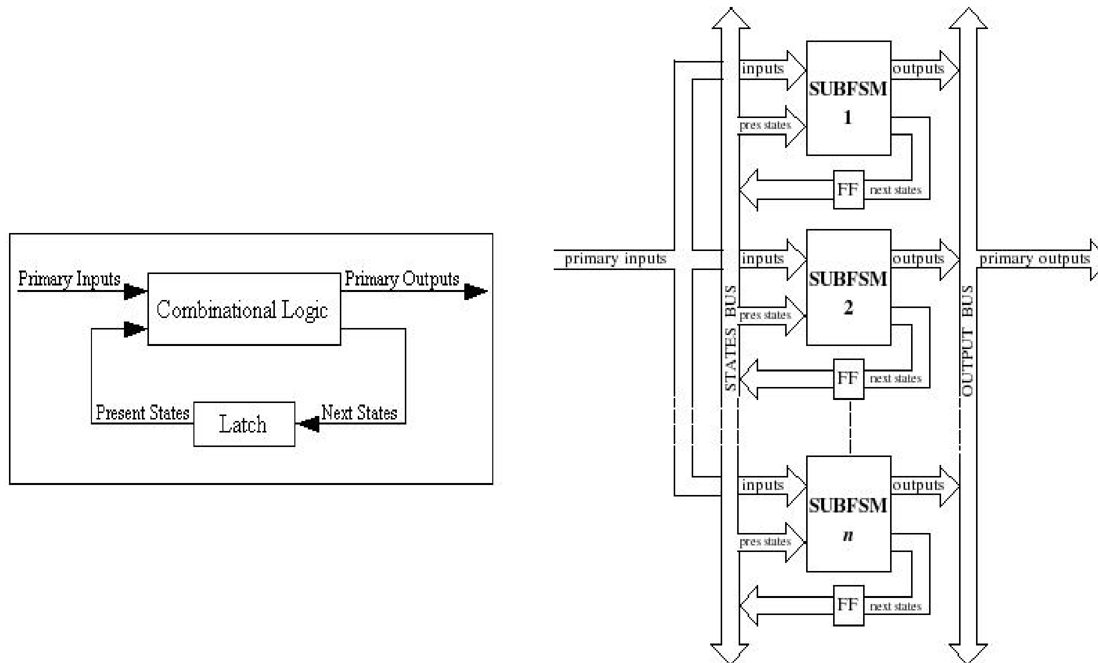


Figura 33. Circuitul Secvențial. Maşina Prototip. Topologia descompunerii generale

10.1 Metode de reprezentare a soluțiilor

Înainte de aplicarea unui algoritm evolutiv, este necesară o metodă de codificare a soluțiilor potențiale ale problemei într-o formă în care pot fi procesate pe calculator. Una dintre aceste abordări este de a codifica soluțiile ca şiruri binare: secvențe de 0 şi 1, unde digitul corespunzator fiecărei poziții reprezintă valoarea unei variante ale soluției. O altă abordare, similară, este de a codifica soluțiile ca şiruri de numere întregi sau zecimale, unde fiecare poziție reprezintă un caz particular al soluției. Această abordare permite obținerea unei precizii mai bune şi al unui nivel al complexității mai mare decît metoda comparativă, considerată restrictivă, care utilizează numai numere binare, şi este adeseori „mai apropiată intuitiv de spațiul problemei" (Flemming şi Purshouse 2002, p. 1228).

Altă metodă de abordare este de a reprezenta indivizii dintr-un algoritm evolutiv ca şiruri de litere, unde fiecare literă reprezintă de asemenra un aspect specific al soluției. Un exemplu al acestei tehnici este abordarea lui Hiroaki Kitano în „codificarea gramaticală", unde algoritmului evolutiv i s-a atribuit sarcina de a evolua un set simplu de reguli reprezentate sub forma unei gramatici libere de context care a fost utilizată mai apoi pentru a genera rețele neuronale antrenate pentru diverse probleme (Mitchell 1996, p. 74).

Scopul acestor metode este de a facilita definirea operatorilor care produc modificările aleatorii ale candidaților selectați, cum ar fi: schimbarea din 0 la 1 sau invers, adăugarea sau scăderea unei valori a unui numar cu un nivel ales aleator, sau schimbarea unei litere cu alta. Spre exemplu, codificarea gramaticală a lui Kitano, menționată mai sus, poate fi scalată în mod eficient pentru a crea rețele neuronale complexe de dimensiuni mari, iar arborii utilizați de Koza în programarea genetică pot creşte şi deveni arbitrar de largi astfel încît să poată rezolva orice problemă la care pot fi aplicați.

Definiție: Un Algorithm Evolutiv (**AE**) descris sub forma AE = ($I, \Phi, \Omega, \Psi, s, \iota, \mu, \lambda$) este denumit Algoritm Genetic (**GA**) [122] dacă şi numai dacă:

(1) $I = B^l$ (spațiul binar {0, 1});

(2) $a \in I : \Phi(a) = \delta(f(\Upsilon(a)), \Theta_\delta)$, unde $\delta : \mathfrak{R} \, x \, \Theta_\delta \rightarrow \mathfrak{R}^+$ este o funcție de scalare (*C1*), şi Υ este o funcție de codificare (*C2*);

(3) $\Omega = \{ m_{\{pm\}} : I^\mu \rightarrow I^\mu , r_{\{pc, z\}} : I^\mu \rightarrow I^\mu , r_{\{pc\}} : I^\mu \rightarrow I^\mu \}$ (*C3*);

(4) $\Psi(P) = s(m_{\{pm\}} (r_{\{pc, z\}} (P)))$;

(5) $s : I^\mu \rightarrow I^\mu$, operatorul de selecție proportională, care eşantionează indivizii în acord cu funcția de probabilitate (*C4*);

(6) Condiția de terminare ι (*C5*);

(7) $\lambda = \mu$ (numărul de indivizi descendenti egal cu numărul de indivizi părinți)

În această definiție s-au utilizat: ***I*** = spațiul soluțiilor, Φ(a) = funcția de fitness, şi formele auxiliare (*C1-C5*).

(***C1***) AG necesită o funcție de scalare $\delta : \mathfrak{R} \, x \, \Theta_\delta \rightarrow \mathfrak{R}^+$ (valori reale pozitive) pentru maparea valorilor funcției obiectiv $f(\Upsilon(a))$ la valori reale pozitive.[118] Sunt mai multe metode scalare; cele mai utilizate metode sunt descrise după cum urmează:

Scalare liniara statică: $\delta(f(\Upsilon(a, \{c_0 , c_1\}) = c_0 f(\Upsilon(a)) + c_1$, unde $c_0 \in \mathfrak{R} \setminus \{0\}$, $c_1 \in \mathfrak{R}$ sunt constante.

Scalare liniara dinamică: $\delta(f(\Upsilon(a , \{c_0 , P(t)\}) = c_0 f(\Upsilon(a)) - min \{ f(\Upsilon(a_j)) \mid a_j \in P(t)\}$, unde P(t) reprezintă populația curentă şi $c_0 \in \mathfrak{R} \setminus \{0\}$.

Scalare logaritmică: $\delta(f(\Upsilon(a , \{c_0\}) = c_0 - log (f(\Upsilon(a)))$, unde funcția obiectiv ia valori pozitive şi

$c_0 > log (f(x)) \ \forall x \in \Pi [u_i , v_i] (i=1,n)$, unde ***x*** reprezintă variabilele obiect şi $\{u_i , v_i\}$ domeniul de variabile obiect.

Scalarea exponențială: $\delta(f(\Upsilon(a, \{c_0 , c_1 , c_2\}) = (c_0 f(\Upsilon(a)) + c_1)^{c2}$, unde $c_0 \in \mathfrak{R} \setminus \{0\}$, $c_1 \in \mathfrak{R}$ sunt constante.

(***C2***) AG, spre deosebire de SE şi PE, lucrează cu şiruri de biți de lungime fixă ***l***, deci spațiul indivizilor este $\boldsymbol{I = B^l}$. În cazul aplicării AG la probleme de optimizare, cu parametri care iau valori continue de forma $f : \Pi [u_i , v_i] (i=1,n) \rightarrow \mathfrak{R} (u_i < v_i)$, şirul de biți din vectorul $a = (a_1 , \dots , a_l) \in B^l$ codifică vectorul $x \in \Pi [u_i , v_i] (i=1,n)$. Acest proces se realizează prin împărțirea logică a şirului de biți în ***n*** segmente egale ca lungime, $\boldsymbol{l_x}$, astfel încît $l = n \cdot l_x$ şi fiecare segment $a_{i1} , \dots , a_{ilx} \in B^{lx}$ din ***a*** codifică variabilele obiect x_i corespunzătoare. Decodificarea unui segment se face prin conversia pozitiilor binare în valorile întregi corespunzătoare, între 0 şi 2^{lx-1}. Apoi se face maparea liniară a întregilor în intervalul de valori reale $[u_i , v_i]$, astfel obținem funcția pentru decodificare $\Upsilon^I : B^{lx} \rightarrow [u_i , v_i]$. Decodificarea numerelor din baza binară în zecimal se realizează prin următoarea formulă:

$$\Upsilon^i (a_{i1,} ..., a_{il_x}) = u_i + \frac{v_i - u_i}{2^{l_x} - 1} \left(\sum_{j=0}^{l_x - 1} a_i (l_x - j) \times 2^j \right)$$

Prin interpretarea segmentelor şirului de biți în *codul Gray* reprezentarea este mai potrivită pentru AG, îmbunătățind schimbările pe care le face operatorul de mutație (Hollstien, 1971). În reprezentarea binară, o singură mutație a celui mai semnificativ bit conduce la alterarea majoră a şirului. Pentru codul Gray acest lucru se produce numai în puține cazuri. Există multe forme ale codului Gray pentru codificarea aceloraşi şiruri de biți. Cel mai folosit este *codul Gray reflectat binar*. Prin generarea sa se pleacă cu toți biții zero şi se schimbă succesiv cel mai din dreapta bit, pentru a produce un nou şir. De exemplu,

reprezentările unor numere întregi în bazele zece, doi standard şi cod Gray sunt: 0-000-000 , 1-001-001 , 2-010-011 , 3-011-010 , 4-100-110 , 5-101-111 , 6-110-101 , 7-111-100.

Se defineşte un operator $\oplus$ care indică adăugarea de modulo 2 în **B** (spațiul binar), astfel că avem: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$. Conversia reciprocă între un şir de biți în codul standard binar $a = (a_1 , \dots , a_l) \in B^l$ şi corespondentul său în cod Gray $b = (b_1 , \dots , b_l) \in B^l$, pentru $\forall\ i \in \{1, \dots, l_x\}$, se defineşte prin următoarea formulă:

Standard $\rightarrow$ Gray: $\gamma : b_i = \begin{cases} a_i, if\ i = 1, \\ a_{i-1}, if\ i > 1; \end{cases}$ Gray $\rightarrow$ Standard: $\gamma^{-1} : a_i = \oplus_{j=1}^{i} b_j$.

Funcția de decodificare a numerelor reprezentate prin codul Gray în numere în baza zecimală are forma:

$$\Upsilon^i(b_{i1,},\dots,b_{il_x}) = u_i + \frac{v_i - u_i}{2^{l_x} - 1}\left(\sum_{j=0}^{l_x-1}(\oplus_{k=1}^{l_x-j} b_{ik}) \times 2^j\right)$$

(***C3***) Operatorul de mutație $m'_{\{pm\}} : I \rightarrow I$, folosind inversarea biților pentru fiecare poziție supusă mutației, produce un şir de biți $a' = (a_1' , \dots , a_l') = m'_{\{pm\}}(a_1 , \dots , a_l) = m'_{\{pm\}}(a)$ care satisfac , pentru $\forall\ i \in \{1, \dots, l\}$, următoarea condiție:

$a_i' = \begin{cases} a_i, & if\ \zeta_i > p_m, \\ 1 - a_i, & if\ \zeta_i \le p_m; \end{cases}$ unde $p_m \in [0 , 1]$ este probabilitatea de mutație, iar

ζ_i este o variabilă aleatoare uniformă, eşantionată altfel pentru fiecare bit.

Pentru crossover *într-un punct* se alege aleatoriu o poziție de încrucişare în şirul de biți şi se schimbă toți biții la dreapta acelei poziții între cei doi parteneri ***s*** şi ***v*** rezultînd:

$s' = (s_1, \dots , s_{\zeta-1} , s_\zeta , v_{\zeta+1} , \dots , v_l)$ si $v' = (v_1, \dots, v_{\zeta-1} , v_\zeta , v_{\zeta+1} , \dots , v_l)$.

Operatorul de crossover într-un punct este inferior din punct de vedere al performanțelor față de alți operatori de acelaşi tip. Astfel, la acest operator există o puternică dependență de poziția bitului a probabilității de schimb. Pe masură ce indicile poziției bitului creşte spre ***l*** (lungimea sirului) probabilitatea sa de schimb tinde la 1.

Crossover-ul *multipunctual*, cu rata p_c, are $z \ge 1$ pozitii incrucişate şi schimbă, între partenerii *S* şi *T* selectați aleatoriu din populație, fiecare al doilea segment al poziției de încrucişare care urmează, celelalte segmente rămînînd neschimbate.

Consideram $(\zeta''_1, \dots , \zeta''_z) \in \{1, \dots, l\text{-}1\}$ poziții de încrucişare eşantionate aleatoriu, unde $\zeta''_k < \zeta''_{k+1}$ si $\zeta''_{z+1} = l$ dacă ***z*** este impar, $k \in \{1, \dots , l\}$. Există mai multe tipuri de operatori de crossover multipunctuali, dintre care:

- crossover în ***z*** puncte $r_{\{pcz\}}$: $a'_i = \begin{cases} a_{S,i}, & i\,(\zeta''_k < i \le \zeta''_{k+1}), k \le z, k \ne 2k \\ a_{T,i}, & \text{in caz contrar;} \end{cases}$

- crossover uniform $r_{\{pc\}}$: $a'_i = \begin{cases} a_{S,i}, & if\ \zeta_i > 1/2, \\ a_{T,i}, & if\ \zeta_i \le 1/2; \end{cases}$

Au fost creați alți operatori de crossover: crossover cu puncte de încrucişare variabile, crossover distribuit, crossover punctual, operatori care includ astfel autoadaptarea în AG.

(***C4***) AG, la fel ca şi PE, folosesc pentru selecţie o regulă probabilistică de supravieţuire. *Probabilitatea de supravieţuire sau selecţie* $p_s(a) = \wp\{a \in s(P(t)) \mid a \in P(t)\}$ în metoda de selecţie proportională $s : I^{\mu} \rightarrow I^{\mu}$ se calculează cu formula:

$$p_s(a_i(t)) = \frac{\Phi(a_i(t))}{\sum_{j=1}^{\mu} \Phi(a_j(t))}$$

ţinînd cont de fitness-ul relativ al indivizilor $a_i(t) \in P(t) = \{a_1(t), \ldots, a_{\mu}(t)\}, \forall i \in \{1, \ldots, \mu\}$.

(***C5***) Terminarea algoritmului este controlată de numărul de generaţii pentru timpul de rulare t (cu o valoare maximă t_{max}) şi este dat de următoarea formulă:

$$\iota(P(t)) = \begin{cases} true \ , & if \ \ t > t_{\max} \ , \\ false \ , & otherwise \ . \end{cases}$$

Uneori terminarea programului este controlată de un parametru *b(P(t)),* care masoară diversitatea genotipului. Notînd un individ cu $a_i = (a_{i,1}, \ldots, a_{i,l})$, $\forall\, i \in \{1, \ldots, \mu\}$, parametrul *b(P(t))* al unei populaţii se calculează cu formula:

$$\iota(P(t)) = \frac{1}{1 \cdot \mu} \sum_{j=1}^{\mu} \max\left\{ \sum_{i=1}^{\mu} (1 - a_{i,j}), \sum_{i=1}^{\mu} a_{i,j} \right\} \in [0.5\,; 1.0].$$

Valorile $b \in [0.5\ ;\ 1.0]$ reprezintă procentele medii ale valorilor cele mai remarcabile în fiecare poziţie a şirului ce constituie un individ din populaţie. Valori mai mari sau mai mici ale lui *b* indică o diversitate genotipică mai mare sau mai mică. Folosind parametrul *b* criteriul de terminare al AG este de forma:

$$\iota(P(t)) = \begin{cases} true \ , & if \ \ b(P(t)) > b_{\max} \ , \\ false \ , & otherwise \ . \end{cases}$$

Aici parametrul $b_{max} \approx 0.95 - 0.99$ defineşte limita inferioară a diversităţii, necesară a fi menţinută în populaţie. Punctele de căutare în SE sunt vectori ai parametrilor obiect ***n***-dimensionali $x \in \Re^n$. Dîndu-se o *funcţie obiectiv* $f : \Re^n \rightarrow \Re$, *funcţia de fitness* Φ este în principiu identică cu *f*, adică dîndu-se vectorul $a \in I$ avem $\Phi(a) = f(x)$.

Aici *x* este variabila obiect, componentă a lui $a = (x, \sigma, \alpha) \in I = \Re^n \times A_s$, unde σ şi α sunt vectori şi $A_s = \Re^{n\sigma}_{+} \times [-\pi,\pi]^{n\sigma}$; $n_{\sigma} \in \{1, \ldots, n\}$; $n_{\sigma} \in \{0, (2n - n_{\sigma})(n_{\sigma}-1)/2\}$, $\Re^{n\sigma}_{+}$ mulţimea valorilor reale pozitive. Vectorul parametrilor obiect *x* poate avea ataşată una pînă la ***n*** deviaţii standard diferite α_i şi pîna la $n(n-1)/2$ unghiuri de rotaţie $\alpha_{ij} = [-\pi,\pi]$ cu ($i \in \{1, \ldots, n-1\}$, $j \in \{i+1, \ldots, n-1\}$), astfel că numărul maxim de parametri ai strategiei este $w = n(n+1)/2$. Pentru cazul $1 < n_{\sigma} < n$ deviaţiile standard $\sigma_1, \ldots, \sigma_{n\sigma-1}$ sunt cuplate cu variabilele obiect $x_1, \ldots, x_{n\sigma-1}$ şi $\sigma_{n\sigma-1}$ este folosit pentru variabilele rămase $x_{n\sigma}, \ldots, x_n$.

Ecuația schemei: Fie ζ(S,t) numărul de şiruri mapate prin schema *S* în generația curentă. Prin combinarea operatorilor de selecție, crossover, şi mutație, este obtinută o nouă formă de ecuație a schemei de reproducere:

$$\zeta(S,t+1) \geq \zeta(S,t)\frac{f(S,t)}{F(t)}\left[1-p_c\frac{\delta(S)}{L-1}-o(S)p_m\right]$$

Pe baza ecuației de mai sus se poate concluziona faptul că doar o valoare ridicată a fitness-ului nu este suficientă pentru a obține o rată mare de crestere. Într-adevăr, scheme de dimensiuni reduse, de ordin scăzut, peste medie primesc un număr exponential mai mare de încercări în generațiile următoare ale AG.

10.2 Seturi de funcții de test

Este foarte dificil de a indica care algoritm este cel mai potrivit pentru a rezolva o problemă particulară. Pentru a avea o orientare generală în alegerea algoritmului potrivit de căutare în spațiul soluțiilor, K.A. De Jong, în 1975, a propus 5 funcții de test care sunt des utilizate şi în prezent în cercetările din domeniul AG. H.P. Schwefel în 1977 a propus 62 de funcții pentru SE care acoperă o diversitate enormă de topologii de spații de căutare. Mai mulți autori au propus funcții de test, printre care H.H.Rosenbrock (1960), I.O.Bohachevsky (1986) şi T. Bäck în (1996). Relațiile lui De Jong sunt funcții simple, concepute pentru a distinge diferite tipuri de spații de căutare, respectiv continue/discontinue, convexe/neconvexe, unimodale/multimodale, cuadratice/necuadratice, deterministe/stohastice sau de dimeniune mică/mare. De Jong a reprezentat fiecare parametru sau numar real ca un întreg format dintr-un număr de biți, împărțit la o constantă. Expresiile pentru aceste funcții sunt redate în Tabelul 17.

T. Bäck în 1996 a propus 5 funcții (f_1, ... f_5) care sunt scalabile referitor la dimensiunea lor *n*. Setul include o funcție unimodală, continuă pentru compararea vitezei de convergența a algoritmului, o funcție în scară cu mai multe trepte de înălțimi diferite pentru a testa algoritmii evolutivi în cazul absenței oricărei informații despre gradientul local. Sunt acoperite şi funcțiile multimodale de diferite complexități. Relația $f_1(x)=\sum_{i=1}^{n}x_i^2$ (modelul sferei) defineşte o funcție continuă, puternic convexă, unimodală. Funcția treaptă este de forma $f_2(x)=\sum_{i=1}^{n}\left[x_i+0.5\right]^2$. Funcția de test continuă unimodală (funcția Ackley) este definită prin următoarea relație:

$$f_3(x)=-c_1\cdot\exp\left(-c_2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right)-\exp\left(\frac{1}{n}\cdot\sum_{i=1}^{n}\cos(c_3\cdot x_i)\right)+c_1+e,$$

unde $c_1 = 20$, $c_2 = 0.2$, $c_3 = 2\pi$.

O funcție multimodală a fost introdusă prima dată de Fletcher şi Powell, în 1963, şi are forma:

$$f_4(x)=\sum_{i=1}^{n}(A_i-B_i)^2 \text{ cu } A_i=\sum_{j=1}^{n}(a_{ij}\sin\alpha_j+b_{ij}\cos\alpha_j) \text{ şi } B_i=\sum_{j=1}^{n}(a_{ij}\sin x_j+b_{ij}\cos x_j)$$

Ultima funcție din set este $f_5(x)=\sum_{i=1}^{n}(C'(x_i)+x_i^2-1)$, unde:

$$C'(x)=\begin{cases}\dfrac{C(x)}{C(1)|x|^{2-D}}, daca\ x\neq 0,\\ 1 \qquad , daca\ x=0.\end{cases}$$

Notația utilizată cuprinde $C(x)=\sum_{j=-\infty}^{\infty}\frac{1-\cos(b^j x)}{b^{(2-D)j}}$, unde D = 1.85; b = 1.85.

Funcție	Definiție	Caracteristici
f_1	$\sum_{i=1}^{3}x_i^2$	unimodală cuadratică; $-5.12\le x_i\le 5.12$ vmg[a)] = 0 la $(x_1,x_2,x_3)=(0,0,0)$
f_2	$100(x_1-x_2)^2+(1-x_1)^2$	multimodală; $-2.048\le x_i\le 2.048$; vmg = 0 la $(x_1,x_2)=(1,1)$
f_3	$\left[\sum_{i=1}^{5}\text{int}(x_i)\right]^{b)}$	funcție discontinuă în trepte; vmg = -30 în tot intervalul $-5.12\le x_i\le 5.12$
f_4	$\left[\sum_{i=1}^{30}ix_i^4+Gauss(0,1)\right]^{c)}$	dimensiune superioară, stohastică; $-1.28\le x_i\le 1.28$; vmg = 0 la $(x_1,x_2,\dots,x_{30})=(0,0,\dots,0)$ fără considerarea zgomotului gaussian
f_5	$\dfrac{1}{\frac{1}{K}+\sum_{j=1}^{25}f_n^{-1}(x_1,x_2}$, unde $f_j(x_1,x_2)=c_j+\sum_{i=1}^{2}(x_i-a_{ij})^6$	multimodală extremă, maxime distincte; $-65.536\le x_i\le 65.536$; K = 500; $c_j=j$; vmg = 0.998 la (x_1,x_2) = (-32, -32)

Tabelul 17. Funcțiile de test De Jong [27]

[a)]vmg – valoarea minimului global;

[b)]int(x_i) – rotunjeşte numărul la întregul inferior cel mai apropiat;

[c)]Gauss(0,1) – variabila aleatorie cu o distribuție gaussiană normală, cu media 0 şi deviația standard 1.

În f_5 valorile lui a_{ij} sunt date de:

$$[a_{ij}]=\begin{bmatrix}-32 & -16 & 0 & 16 & 32 & -32 & -16 & \dots & 0 & 16 & 32\\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \dots & 32 & 32 & 32\end{bmatrix}$$

Pentru optimizarea condițiilor care apar în diferite probleme au fost propuse mai multe funcții test (Hock 1981, Floudas 1987, Schwefel 1995, Michalewicz 1996). Cîteva din acestea sunt redate mai jos. La minimizarea funcției:

$G_1(x,y) = 5x_1 + 5x_2 + 5x_3 + 5x_4 - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=1}^{9} y_i$, avem următoarele condiții:

$2x_1 + 2x_2 + y_6 + y_7 \le 10$ $\quad 2x_1 + 2x_3 + y_6 + y_8 \le 10$ $\quad 2x_2 + 2x_3 + y_7 + y_8 \le 10$

$-8x_1 + y_6 \le 0$ $\quad -8x_2 + y_7 \le 0$ $\quad -8x_3 + y_8 \le 0$

$-2x_4 - y_1 + y_6 \le 0$ $\quad -2y_2 - y_3 + y_7 \le 0$ $\quad -2y_4 - y_5 + y_8 \le 0$

$0 \le x_i \le 1$, i=1,2,3,4 $\quad 0 \le y_i \le 1$, i=1,2,3,4,5,9 $\quad 0 \le y_i$, i=6,7,8

Soluția globală este $(x^*, y^*) = (1,1,1,1,1,1,1,1,1,3,3,3,1)$, iar $G_1(x^*, y^*) = -15$.

Minimizarea funcției $G(X) = e^{x_1 x_2 x_3 x_4 x_5}$, cu condițiile neliniare:

$x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 = 10$ $\quad x_2 x_3 - 5x_4 x_5 = 0$ $\quad x_1^3 + x_2^3 = -1$

$-2.3 \le x_i \le 2.3$, i=1,2 $\quad -3.2 \le x_i \le 3.2$, i=3,4,5.

Minimul global este la X =(-1.7171743; 1.595709; 1.827247; -0.7636412; -0.7636450) iar $G_2(X) = 0.0539498478$.

Maximizarea funcției $G_3(x) = \frac{3x_1 + x_2 - 2x_3 + 0.8}{2x_1 - x_2 + x_3} + \frac{4x_1 - 2x_2 + x_3}{7x_1 + 3x_2 - x_3}$, cu condițiile:

$x_1 + x_2 - x_3 \le 1$ $\quad -x_1 + x_2 - x_3 \le -1$ $\quad 12x_1 + 5x_2 + 12x_3 \le 34.8$

$12x_1 + 12x_2 + 7x_3 \le 29.1$ $\quad -6x_1 + x_2 + x_3 \le -4.1$ $\quad 0 \le x_i$, i=1,2,3

conduce la soluția globală $x^* = (1,0,0)$ si $G_3(x^*) = 2.471428$.

10.3 Reprezentări grafice ale funcțiilor de test

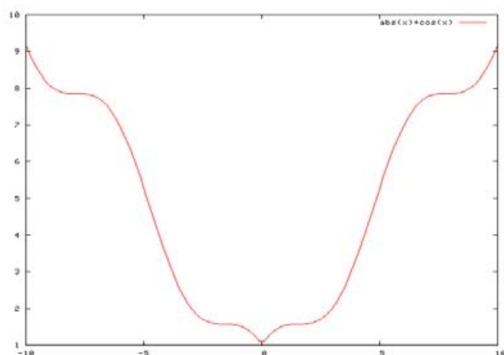


Figura 34. $|x|$ + *cos(x)*

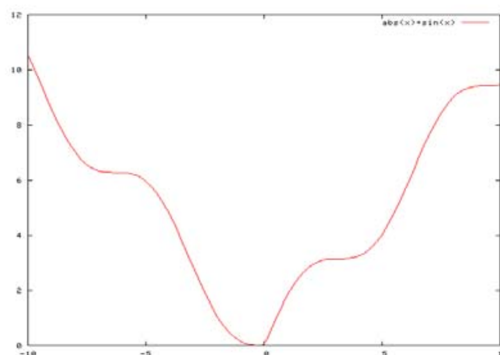


Figura 35. $|x|$ + *sin(x)*

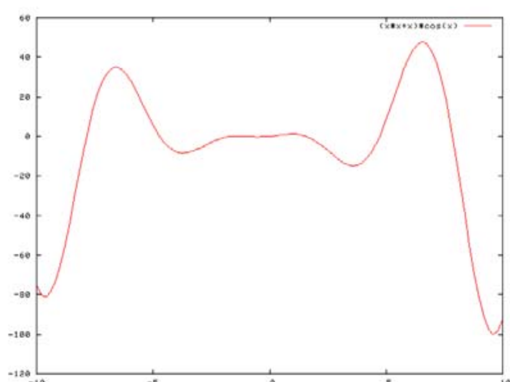


Figura 36: $(x^2 + x)\times\cos(x)$

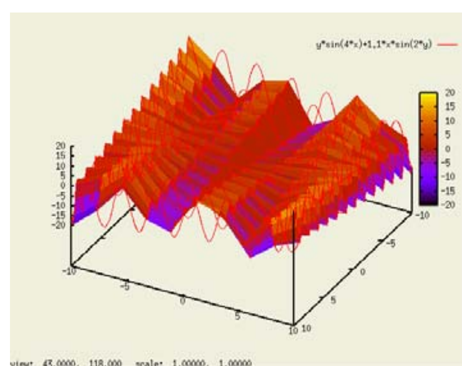


Figura 37. $x\times\sin(4x)+1.1\times y\times\sin(2y)$

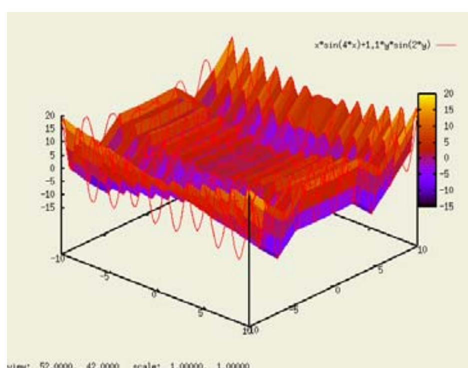


Figura 38. $y\times\sin(4x)+1.1\times x\times\sin(2y)$

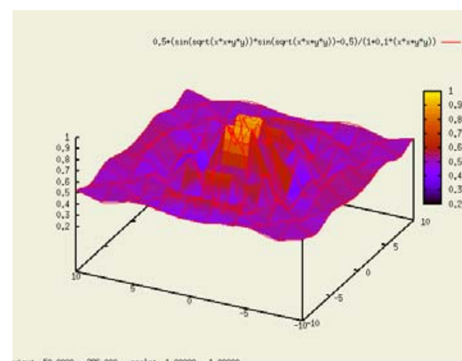


Figura 39. $0.5+\frac{\sin\sqrt[2]{x^2+y^2}-0.5}{1+0.1\times(x^2+y^2)}$

10.4 Abordări precedente

Descompunerea maşinilor secvenţiale a fost pentru prima dată tratată printr-o metodă formală de către Hartmanis şi Stearns [49]. Ei au propus două tipuri de descompunere, în paralel şi în cascadă, pe baza topologiei maşinii descompuse. Una din cele mai simple metode prin care o maşina poate fi împărţită în mai multe submaşini este **descompunerea paralelă**.

Structura submaşinilor este dată de faptul că acestea sunt alimentate cu aceeaşi secvenţă de intrare dar ele funcţionează în mod independent una de cealaltă. Nu există nici o interacţiune sau schimb de informaţii între submaşini. Această metodă are o utilitate limitată în proiectarea maşinilor de stări finite moderne întrucit abordările practice nu prezintă în mod curent o bună descompunere în paralel. Alt tip de descompunere este descompunerea în **cascadă** sau în **serie**, unde din nou fiecare submasina este condusă de aceeaşi secvenţă de intrare, dar submasinile nu mai funcţionează în mod independent una de cealaltă. O submaşina este alimentată prin intermediul unor intrări auxiliare cu informaţii despre stările interne curente ale celorlalte submaşini. Posibilitatea trecerii informaţiei de stare de la o submaşină la urmatoarea submaşină face ca descompunerea în cascadă să fie mai puternică decît descompunerea în paralel. Transmiterea informaţiei de stare este realizată serial şi o submaşina necesită informaţii atît despre starea sa internă cît şi de la stările predecesorilor ei. Ea işi trimite apoi informţia de stare doar către maşinile succesoare.

O altă formă de descompunere a fost prezentată de către Devadas şi Newton [31]. În varianta propusă de ei ambele componente ale maşinilor descompuse interacţionează una cu cealaltă. Această formă de descompunere implică identificarea subrutinelor de **factori** din maşina prototip, pentru ca apoi să extragă aceşti factori şi să-i reprezinte ca o maşina separată de **factorizare** [48]. Ocurenţele acestor factori devin apeluri către maşinile de factorizare din maşinile factorizate. Metoda nu are o funcţie de cost definită pentru optimizare şi nu garantează nimic despre calitatea descompunerii.

Descompunerea generală, sau descompunerea arbitarară, poate avea diverse tipuri de topologii şi acoperă orice tip de descompunere. Principala constrîngere la ieşire rămine faptul că orice pereche de stări din maşina prototip trebuie să aibă coduri distincte în maşina descompusă care ramîne neschimbată. Constrîngerile rămase depind de celelalte submaşini de la care o submasina particulară primeşte informaţii despre starea curentă. Tipurile de constrîngeri impuse se pot observa prin studiul topologiei din exemplul din Fig. 33

Tehnologiile microelectronicii moderne permit oportunităţi în vederea proiectării circuitelor şi sistemelor digitale foarte complexe la un cost relativ scăzut şi furnizează o mare diversitate de construcţii de blocuri logice. Aceste oportunităţi totuşi nu pot fi exploatate la maxim, datorită anumitor limitări ale metodelor de sinteză logică tradiţionale. În cazul particular al (C)PLD-urilor, FPGA-urilor şi porţilor CMOS complexe, constrîngerile sunt impuse nu de către tipul funcţiilor pe care un anumit bloc le poate îndeplini, ci de diverşi parametri structurali din construcţia blocului logic, cum ar fi numărul de intrări, ieşiri, termeni produs, bistabilele dint-un anumit bloc sau numărul de tranzistori în serie sau în paralel dintr-o anumită poartă logică, precum şi de numărul de interconexiuni dintre blocurile funcţionale.

Orice bloc funcţional este capabil să implementeze o anumită funcţie logică, în limita unor specificaţii de proiectare. Pe de altă parte, metodele de sinteză logică traditională nu iau în consideraţie constrîngerile structurale impuse. În principiu, ele sunt fiedele doar unor cazuri foarte specifice de structuri de implementări posibile, care implică sisteme complete de porţi logice de o funcţionalitate minimală (*and* + *or* + *not*, *and* + *exor* + *mux*). Ele necesită o mapare tehnologică post-sinteză pentru alte structuri de implementare.

Dacă scopul procesul de sinteză diferă semnificativ de aceste sisteme minimale (implică un mare număr de porți complexe, celule FPGA sau (C)PLD-uri), maparea tehnologică bazată pe librarii este imposibilă şi nici o mapare tehnologică nu poate garanta un rezultat bun dacă procesul de sinteza inițial a fost realizat fără a se ține cont de scopul actual.

Prin urmare, există multă cercetare în domeniul descompunereii generale (funcționale) a circuitelor combinaționale şi a a maşinilor secvențiale. Cele mai promițătoare abordări recente în analiza patern-urilor, descoperirea de cunostințe, maşini cu posibilități de invățare, baze de date, data mining, sunt de asemenea bazate pe descompunerea funcțională generală.

Pentru circuite mari, numărul de descompuneri posibile este atît de mare încît devine necesar faptul că trebuiesc construite doar cele mai eficiente descompuneri, cu funcțiile de evaluare şi mecanismele de selecție adecvate. Aceste mecanisme de evaluare şi selecție trebuie să limiteze spațiul de căutare la o mărime abordabilă în timp ce permit menținerea unei calități înalte a soluțiilor într-un spațiu limitat. Metoda descompunerii generale poate fi aplicată în multe domenii precum inginerie şi ştiința, inclusiv sinteza logică şi arhitecturală a sistemelor VLSI [87], analiza pattern-urilor, descoperirea de cunoştințe, maşini cu posibilități de învațare, sisteme de decizie, baze de date, data mining şi altele.[113] Partițiile şi seturile permit accesul la modelarea informației digitale.[61]

10.5 Metode de implementare

Principalul obiectiv în această lucrare este de a aplica metoda de descompunere generală unei maşini secvențiale, pentru a reduce complexitatea fiecărei submaşini în timp ce se încearcă menținerea unui număr total redus de submaşini. Prin utilizarea algoritmilor genetici, AG, se încearcă îmbunătățirea performanțelor submaşinilor descompuse în raport cu o funcție de cost definită [19]; în decursul procedurii de descompunere acestea sunt verificate dacă respectă proprietățile descompunerii generale. Cum se codifică o soluție potențială a unei probleme într-un cromozom potrivit este o problema cheie pentru AG. În prezent există numeroase abordări utilizate pentru reprezentarea codificată a cromozomilor pentru proiectarea structurilor hardware. Mai mult, alegerea unei metode de reprezentare potrivită a unei solutii candidate la problemă este fundamentul aplicarii AG pentru a rezolva probleme din lumea reală, aceste condiții fiind paşii necesari calculului evolutiv [20].

Prin utilizarea AG se poate defini **genotipul** ca fiind structura ADN care codifică soluția şi **fenotipul** ca fiind soluția codificată. Un **cromozom** reprezintă o soluție a problemei şi este un membru al populației de soluții. Fiecare cromozom conține mii de **gene**, care sunt blocuri funcționale ADN, fiecare genă aparținînd unor cromozomi omologi poate avea diverse expresii , denumite **alele**, care corespund unei valori specifice a genelor. [121]

Genomul reprezintă de asemenea întreaga structură genetică. În cazul curent cromozomul conține un şir de biți aparținînd unui alfabet binar şi o alela este codificată cu 0 şi 1. Fiecare cromozom poate fi considerat ca un punct în spațiul de căutare al soluțiilor posibile aparținînd problemei de descompunere generală. Datorită faptului că informația codificată este conținută pe diferite nivele, se poate face o analogie cu sistemele numerice poziționale, unde codurile sunt pe un nivel inferior de abstractizare cu numerele, şi are avantajul unei reprezentări economice. Presupunînd că ***n*** este numărul total de stări, se vor calcula mai întîi numerele sistem ca fiind toate partitiile posibile construite cu aceste stări. Numărul lor total este **n**! (numărul de permutații a ***n*** stări, deoarece fiecare partiție conține toate stările într-o ordine dată). De asemenea trebuie ținut cont de împărțirea în blocuri funcționale.

Se utilizează ***n**-1* coduri binare cu următoarea specificație: 0 pe poziția *k* înseamnă o separare între diferite blocuri de stări de pe pozițiile (*k*, *k+1*), iar 1 pe poziția *k* înseamnă că stările din poziția (*k*, *k+1*) aparțin aceluiaşi bloc logic. Din acest fapt rezultă că numărul de poziții binare necesare pentru codificarea unui „număr" este este $d = (\log_2 \mathbf{n}!) + (n-1)$ poziții binare. O soluție este compusă din combinarea mai multor partiții. În mod similar, după intersecția binară şi mutație, se testează validitatea soluțiilor următoare, pentru a putea calcula valoarea funcției de fitness.

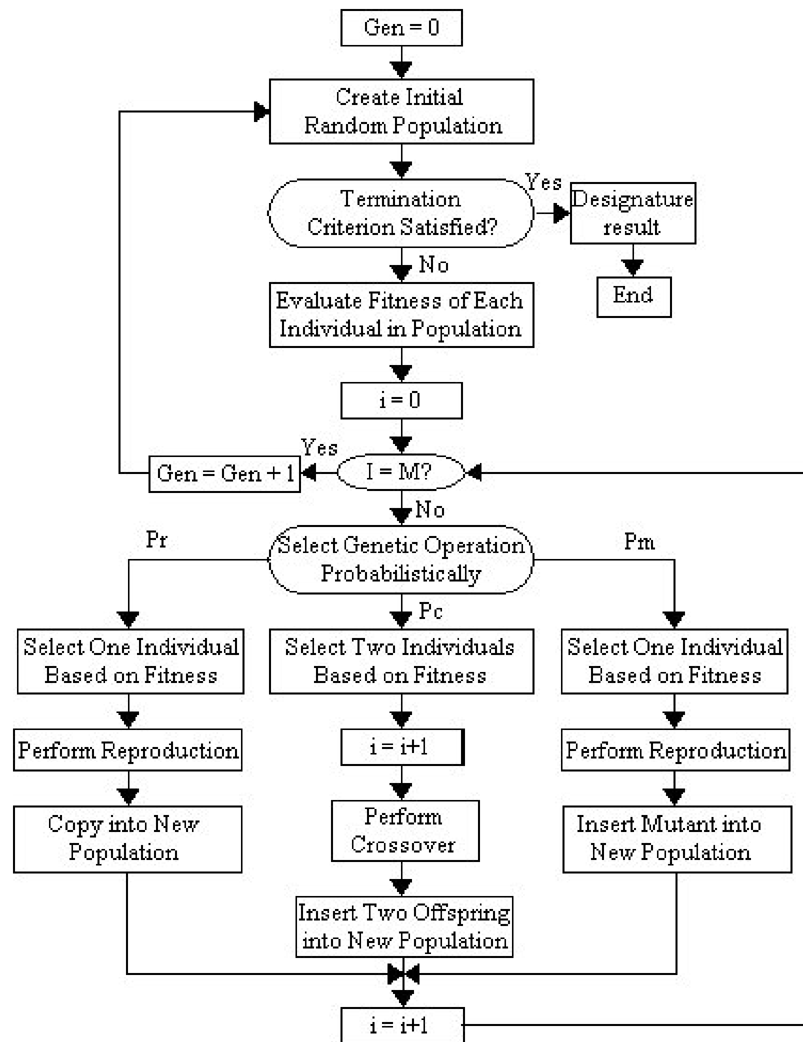


Figura 40. Descrierea funcțională pentru AG conventional

Parametrii principali pentru controlarea AG sunt mărimea populației ***M***, şi numărul maxim de generații care trebuiesc rulate ***G***. Parametrii secundari, precum Pr, Pc, Pm, sunt utilizați pentru controlul frecvenței de reproducere, intersecție şi respectiv mutație. În Fig. 40 este reprezentată o descriere functională a AG convențional care lucrează pe un şir de cromozomi de marime fixă. Indexul *i* se referă la un individ din populația de mărime *M*, iar variabila *Gen* reprezintă numărul generației curente. Fiecare reprezentare a circuitului descompus este evaluată de o funcție de fitness pentru funcționalitate şi performanță.[90]

Tabelul 18. Partiționarea automată a subautomatelor de stări

Nr. stări automat prototip	**Nr. soluții** $\log_n(n)$	**Nr. blocs** $\left[N/Nb\right]$	**Variația** $\left[N\%Nb\right]$	**Mutația 25%**	**Partiția normată**	**Listă taieturi**
4	2	2	0	0	2 2	2
5	2	2	1	0	2 3	3
6	2	3	0	0	3 3	3
7	2	3	1	0	3 4	4
8	3	2	2	0	2 3 3	3 6
9	3	3	0	0	3 3 3	3 6
10	3	3	1	0	3 3 4	4 7
11	3	3	2	0	3 4 4	4 8
12	3	4	0	1	4 4 4	4 8
13	3	4	1	1	4 4 5	5 9
14	3	4	2	1	4 5 5	5 10
15	3	5	0	1	5 5 5	5 10
16	4	4	0	1	4 4 4 4	4 8 12
17	4	4	1	1	4 4 4 5	5 9 13
18	4	4	2	1	4 4 5 5	5 10 14
19	4	4	3	1	4 5 5 5	5 10 15
20	4	5	0	1	5 5 5 5	5 10 15
21	4	5	1	1	5 5 5 6	6 11 16
22	4	5	2	1	5 5 6 6	6 12 17
23	4	5	3	1	5 6 6 6	6 12 18
24	4	6	0	1	6 6 6 6	6 12 18
25	4	6	1	1	6 6 6 7	7 13 19
26	4	6	2	1	6 6 7 7	7 14 20
27	4	6	3	1	6 7 7 7	7 14 21
28	4	7	0	1	7 7 7 7	7 14 21
29	4	7	1	1	7 7 7 8	8 15 22
30	4	7	2	1	7 7 8 8	8 16 23
31	4	7	3	1	7 8 8 8	8 16 24
32	5	6	2	1	6 6 6 7 7	7 14 20 26
33	5	6	3	1	6 6 7 7 7	7 14 21 27
34	5	6	4	1	6 7 7 7 7	7 14 21 28
35	5	7	0	1	7 7 7 7 7	7 14 21 28
36	5	7	1	1	7 7 7 7 8	8 15 22 29
37	5	7	2	1	7 7 7 8 8	8 16 23 30
38	5	7	3	1	7 7 8 8 8	8 16 24 31
39	5	7	4	1	7 8 8 8 8	8 16 24 32
40	5	8	0	2	8 8 8 8 8	8 16 24 32
41	5	8	1	2	8 8 8 8 9	9 17 25 33
42	5	8	2	2	8 8 8 9 9	9 18 26 34
43	5	8	3	2	8 8 9 9 9	9 18 27 35
44	5	8	4	2	8 9 9 9 9	9 18 27 36
45	5	9	0	2	9 9 9 9 9	9 18 27 36
46	5	9	1	2	9 9 9 9 10	10 19 28 37
47	5	9	2	2	9 9 9 10 10	10 20 29 38

După cum se poate observa în tabelul 18, algoritmul genetic efectuează o evaluare a numărului de stări a automatului prototip, dupa care se încearcă o partiționare automată a numărului de subautomate care vor fi descompuse. Din aceste partiționări se vor obține reprezentările necesare pentru construirea populației de soluții specifice fiecarei generații. Numărul de subautomate obținute pentru fiecare automat original, este dat de ecuația:

$$N_b = \log_2(n),$$

unde *n* este egal cu numărul de stări ale automatului prototip. Utilizînd această reprezentare se obține o „codificare" în scară logaritmică a numărului de stări din automatul inițial. În fiecare subautomat există un număr de stări interne specifice. Numărul de stări interne ale fiecărui subautomat este dat de numărul de „blocuri" de stări din formula $[N/N_b]$.

În cazul în care raportul, sau cîtul dintre numărul de stări din automatul original şi numărul de subautomate nu este egal cu zero, atunci obținem o variație dată de restul împărțirii lui $[N\%N_b]$. Restul va fi impărțit în mod uniform pentru fiecare bloc de partiții în parte, pentru a efectua o uniformizare a numărului de stări specifice fiecărui subautomat.

În fiecare bloc de stări există o mutație, care de fapt reprezintă un pivot mobil care se poate muta la stînga sau la dreapta numărului de stări specifice fiecarui bloc. Procentul de mutație este de 25% din valoarea numărului de blocuri din fiecare subautomat descompus. Rolul operatorului de mutație este de a permite o variație a combinațiilor obținute, în ideea de a testa cît mai multe cazuri posibile de optim local, în eventualitatea apropierii cît mai mult de cazul optim. În urma aplicării operatorilor sus-menționați, se obține pe coloana a 5-a lista de partiții normate, care vor fi luate în considerație pentru aplicarea operatorilor genetici de selecție şi recombinare. Lista finală de „tăieturi" care vor fi aplicate numărului de stări din automatul original este obținută în ultimă coloană. Ele sunt specifice fiecărui subautomat obținut, şi vor fi folosite pentru calculul indivizilor corespunzători populației de soluții.

Funcția de fitness utilizată va evalua performanța soluțiilor obținute dupa modelul din capitolul 7, legate de evaluarea funcțiilor de fanin şi fanout a unui automat finit. În funcție de valoarea fitness-ului fiecarui individ, obținut prin recombinarea indivizilor din populația anterioară, algoritmul va converge către un anumit pattern, sau set de soluții mai apropiat de funcția de cost specificată într-o etapă inițială.

Principalul punct de interes poate fi dimensiunea submasinilor descompuse, timpul de răspuns, calea critică sau chiar aranjamentele submaşinilor obținute pentru o tehnologie specificată. Timpul de execuție al algoritmului este în principal dominat de procesul de evaluare a funcționalității şi performanței populațiilor de soluții evoluate la fiecare generație.

Pe masură ce structurile de partiții devin din ce în ce mai complexe, devine ineficientă translarea grafului fiecărui subcircuit în format comportamental Verilog şi apoi simularea lui.

Prin urmare structurile sunt reprezentate în memorie pină cînd sunt obținute rezultatele finale. Constrîngerea legată de intrare în descompunerea generalizată este ca fiecare stare din automatul prototip să aibă o codificare unică. Constringerea legată de ieşire este ca fiecare codificare de stări din automatul descompus să fie de asemenea unică. Tipurile de constrîngeri pot fi observate din topologiile de exemple din Fig. 44 şi Fig. 45. Descrierea internă poate fi reprezentată în format Kiss, prin schema la nivel top (Fig. 42), sau prin graful său de tranziții (Fig. 41). Spre exemplificare, se utilizează un automat finit care are comportamentul unui registru de deplasare. Descrierea internă este reprezentată prin graful de tranziții (STG), într-un format kiss standard după cum se poate observa în Fig. 41. [98]

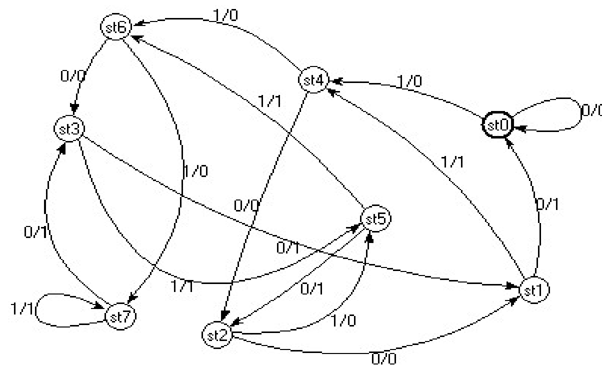
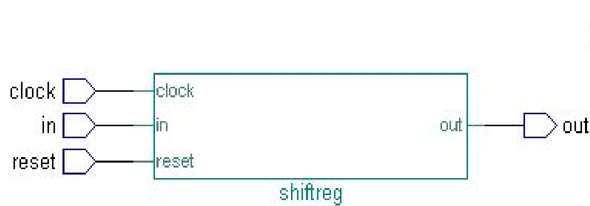


Figura 41. Graful de fluență pentru automatul prototip



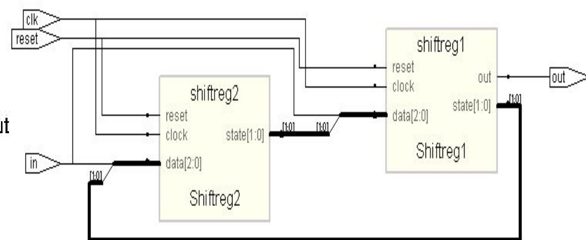
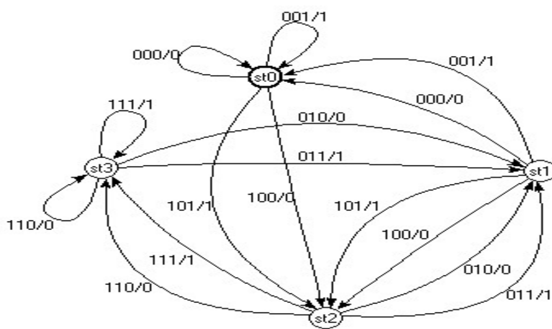


Figura 42. Schema top automat prototip

Figura 43. Schema top automat descompus



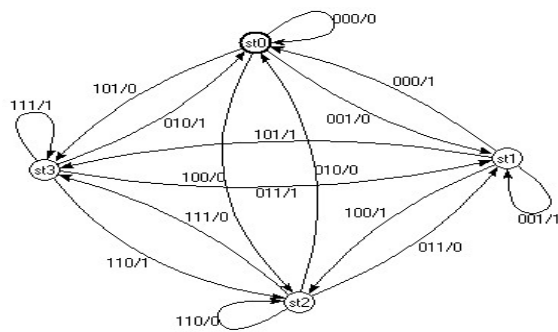


Figura 44. Graf de fluență subautomat 1

Figura 45. Graf de fluență subautomat 2

Presupunînd că avem o reprezentare la nivel de top al registrului de shiftare cunoscut ca automat prototip şi un set de partiții generate automat cu ajutorul pachetului *fsmtool* într-o etapă inițială dupa cum urmează:

{ (*st0 st1*) , (*st2 st3*) , (*st4 st5*) , (*st6 st7*) }

{ (*st0 st2*) , (*st1 st3*) , (*st4 st6*) , (*st5 st7*) }

În acest caz se obțin două subautomate care satisfac regulile descompunerii generale pentru automatul prototip după cum se poate vedea în Fig. 42 într-un plan RTL general de top, sau fiecare dintre ele detaliat în Fig. 44 şi Fig. 45.

Ne vom referi la acest tip de topologie ca la topologia descompunerii generale, întrucît reprezintă o descompunere arbitrară. În această topologie, fiecare subautomat M_i are 3 seturi de intrări simbolice:

1. un set de intrări primare I_{P_i}; aceste intrări pot fi considerate simbolice sau binare, în funcție de specificatiile inițiale
2. un set de intrări de stare I_{S_i}; aceste intrări sunt derivate de la stări ale altor subautomate; pentru un automat care corespunde unei partiții închise, I_{S_i} =0.
3. un set de stări interne, S_i.

Codificarea fiecărui subautomat poate fi arbitrară atît timp cît fiecare stare internă are un cod binar distinct. Intrările primare sunt disponibile pentru fiecare subautomat. Stările unui subautomat sunt disponibile pentru celelalte subautomate dacă acest lucru este necesar. Ieşirile sunt generate direct de către subautomate individuale.

Informația de stare este interschimbată de către subautomatele interconectate care nu sunt independente unul de celălalt. În exemplul precedent se consideră un automat prototip M cu o intrare principală şi o ieşire principală, şi 8 stări interne $S = (st_0, st_1, st_2, st_3, st_4, st_5, st_6, st_7)$. Se descompune acest automat prototip în 2 subautomate M_1 şi M_2 după cum urmează:

$$M_1: \{ (st_0\ st_1) , (st_2\ st_3) , (st_4\ st_5) , (st_6\ st_7) \}$$
$$M_2: \{ (st_0\ st_2) , (st_1\ st_3) , (st_4\ st_6) , (st_5\ st_7) \}$$

Aceste două partiții satisfac proprietățile descompunerii legale întrucit produsul lor este egal cu partiția unitate $\Pi(0)$. Blocurile de stări ale celor 2 partiții ortogonale sunt apoi codificate astfel:

$$M_1: \{ x_1 , x_2 , x_3 , x_4 \} = \{ 00, 01, 10, 11 \}$$
$$M_2: \{ y_1 , y_2 , y_3 , y_4 \} = \{ 00, 01, 10, 11 \}$$

Noile stări create $x_{1...4}$ si $y_{1...4}$ reprezintă blocurile de stări codificate din automatul prototip. În acest caz, fiecare subautomat necesită un număr de $\log_2(n) = 2$ bistabile, unde n = 4 este numărul de stări. Automatul corespunzator M_1 are alfabetul de intrare următor:

$$\sum \times \pi_B = \{(0, y_1),(1, y_1),(0, y_2),(1, y_2),(0, y_3),(1, y_3),(0, y_4),(1, y_4)\},$$

iar automatul M_2 are alfabetul de intrare următor:

$$\sum \times \pi_A = \{(0, x_1),(1, x_1),(0, x_2),(1, x_2),(0, x_3),(1, x_3),(0, x_4),(1, x_4)\}.$$

Funcțiile de stare următoare δ_{M_1} şi δ_{M_2} sunt definite pe următoarea expresie:

$$\delta_M : \sum \times \pi_{M_2} \times \pi_{M_1} \to \pi_{M_1}$$

În cazul descrierii registrului de shiftare, vom obține următoarele valori pentru funcția de stare δ_M :

$\delta_{M1}((0, y_1), x_1) = x_1$ $\delta_{M2}((0, x_1), y_1) = y_1$

$\delta_{M1}((1, y_1), x_1) = x_3$ $\delta_{M2}((1, x_1), y_1) = y_3$

$\delta_{M1}((0, y_1), x_2) = x_1$ $\delta_{M2}((0, x_2), y_1) = y_1$

$\delta_{M1}((1, y_1), x_2) = x_3$ $\delta_{M2}((1, x_2), y_1) = y_3$

$\delta_{M1}((0, y_2), x_1) = x_2$ $\delta_{M2}((0, x_1), y_2) = y_1$

$\delta_{M1}((1, y_2), x_1) = x_4$ $\delta_{M2}((1, x_1), y_2) = y_3$

$\delta_{M1}((0, y_2), x_2) = x_2$ $\delta_{M2}((0, x_2), y_2) = y_1$

$\delta_{M1}((1, y_2), x_2) = x_4$ $\delta_{M2}((1, x_2), y_2) = y_3$

$\delta_{M1}((0, y_3), x_3) = x_1$ $\delta_{M2}((0, x_3), y_3) = y_2$

$\delta_{M1}((1, y_3), x_3) = x_3$ $\delta_{M2}((1, x_3), y_3) = y_4$

$\delta_{M1}((0, y_3), x_4) = x_1$ $\delta_{M2}((0, x_4), y_3) = y_2$

$\delta_{M1}((1, y_3), x_4) = x_3$ $\delta_{M2}((1, x_4), y_3) = y_4$

$\delta_{M1}((0, y_4), x_3) = x_2$ $\delta_{M2}((0, x_3), y_4) = y_2$

$\delta_{M1}((1, y_4), x_3) = x_4$ $\delta_{M2}((1, x_3), y_4) = y_4$

$\delta_{M1}((0, y_4), x_4) = x_2$ $\delta_{M2}((0, x_4), y_4) = y_2$

$\delta_{M1}((1, y_4), x_4) = x_4$ $\delta_{M2}((1, x_4), y_4) = y_4$

Funcțiile de ieşire f_{M_1} şi f_{M_2} sunt definite de către următoarea expresie:

$$f_M : \sum \times \pi_{M_1} \times \pi_{M_2} \to \lambda$$

În cazul descrierii registrului de shiftare, vom obține următoarele valori pentru funcția de ieşire f_M :

$f_{M1}(0, x_1, y_1) = 0$ $f_{M2}(0, y_1, x_1) = 0$

$f_{M1}(1, x_1, y_1) = 0$ $f_{M2}(1, y_1, x_1) = 0$

$f_{M1}(0, x_2, y_1) =1$ $f_{M2}(0, y_1, x_2) = 1$

$f_{M1}(1, x_2, y_1) = 1$ $f_{M2}(1, y_1, x_2) = 1$

$f_{M1}(0, x_1, y_2) = 0$ $f_{M2}(0, y_2, x_1) = 0$

$f_{M1}(1, x_1, y_2) = 0$ $f_{M2}(1, y_2, y_1) = 0$

$f_{M1}(0, x_2, y_2) = 1$ $f_{M2}(0, y_2, x_2) = 1$

$f_{M1}(1, x_2, y_2) = 1$ $f_{M2}(1, y_2, x_2) = 1$

$f_{M1}(0, x_3, y_3) = 0$ $f_{M2}(0, y_3, x_3) = 0$

$f_{M1}(1, x_3, y_3) = 0$ $f_{M2}(1, y_3, x_3) = 0$

$f_{M1}(0, x_4, y_3) = 1$ $f_{M2}(0, y_3, x_4) = 1$

$f_{M1}(1, x_4, y_3) = 1$ $f_{M2}(1, y_3, x_4) = 1$

$f_{M1}(0, x_3, y_4) = 0$ $f_{M2}(0, y_4, x_3) = 0$

$$f_{M1}(1, x_3, y_4) = 0 \qquad f_{M2}(1, y_4, x_3) = 0$$

$$f_{M1}(0, x_4, y_4) = 1 \qquad f_{M2}(0, y_4, x_4) = 1$$

$$f_{M1}(1, x_4, y_4) = 1 \qquad f_{M2}(1, y_4, x_4) = 1$$

După cum se poate observa din Fig. 44 şi Fig. 45, graful stărilor de tranziție pentru un automat descompus poate conține muchii paralele. Dacă se înlocuiesc toate muchiile paralele directe care încep cu o stare st_1 şi care se termină la o stare st_2, se obține un digraf $G(V,E)$, unde V este setul de stări iar E este numărul de muchii. Acest graf poate conține bucle proprii. Numărul de muchii din digraful obținut poate fi considerat ca o funcție de cost. Cu cît este mai mare numărul de noduri din digraf, cu atît este mai mare numărul de termeni din funcțiile stărilor următoare. Acest lucru implică un timp mai mare de răspuns, un număr mai mare de porți logice şi prin urmare o implementare finală a circuitului mai complexă. Unul din principalele obtiective este marimea automatelor descompuse, timpul de răspuns, calea critică sau chiar aranjamentul subautomatelor descompuse pentru o anumită familie de circuite de tip FPGA.[97] Pe parcursul execuției pachetului de programe *fsmtool*, algoritmul este în principiu dominat de procesul de evaluare a funcționalității şi performanței soluțiilor descompuse obținute.[93]

Cei patru paşi majori în utilizarea optimizării globale pentru a rezolva problema codării şirurilor de caractere cu lungime fixă sunt:

a) determinarea reprezentării soluțiilor

b) determinarea măsurii obiectivului (dimensiunea soluțiilor)

c) determinarea parametrilor şi variabilelor pentru controlul algoritmului

d)determinarea felului în care este indicat rezultatul şi criteriul de terminare a execuției algoritmului

După cum se poate observa, în practică, AG sunt surprinzători de rapizi în căutarea efectivă în spatii complexe, puternic neliniare, multidimensionale. Acest lucru este cu atît mai surprinzător cu cît AG nu cunosc dinainte nimic despre domeniul problemei parcurse sau despre măsura funcției de fitness. AG furnizează o modalitate de a continua căutarea în spațiul de căutare prin testarea unor noi puncte diferite care au demonstrat anterior un fitness peste medie, căutînd soluții în părțile promițătoare ale spațiului de căutare pe baza informațiilor disponibile de la testarea explicită a unui număr particular de indivizi conținuți în populația curentă.

Pentru o masină luată ca exemplu şi pentru un set de partiții specificat, se pot obține diverse tipuri de descompunere. Submaşinile echivalente rezultate au fost sintetizate cu programe standard precum Leonardo Spectrum şi Synplify Pro unde au fost obținute rezultate diverse care sunt importante din punctul de vedere al implementării tehnologice pe familii specifice de circuite şi pentru anumite necesități de suprafață optimă ocupată şi timp de răspuns. Rezultatele obținute sunt de la seturi de exemple standard implementate pe diverse librării tehnologice. După cum se poate observa, există situații cînd suprafața şi de timpul de propagare sunt mai mici decît în maşina prototip, dar de asemenea există situații cînd ele depăşesc proprietățile maşinii inițiale. Prin utilizarea AG în generarea automată optimală a partițiilor, numărul obținut de partiții este redus dinamic la cel mai eficient număr posibil. Pe parcursul procesului de descompunere au fost necesare unele programe precum: ***genfsm*** utilizat pentru generarea automată a automatelor finite, ***vl2mv*** utilizat pentru a converti exemplele de la nivel comportamental verilog la nivel de registri de stare în format kiss, ***fsmtool*** utilizat pentru a genera setul de stări de partiționare şi pentru descompunere, şi de asemenea pentru a realiza operațiile de descompunere, ***kiss2vl*** utilizat pentru conversia de la

format kiss în format comportamentatl verilog de nivel înalt, precum şi programul ***VIS*** [7] pentru testarea echivalenței funcționale a maşinilor obținute şi testarea lor.

Tabelul 19. Raport pentru SPARTAN-XL

Mcnc files	*PI*	*PO*	*sub FSM*	*spartan-XL*			
				DFFs	*Area(FGs)*	*Delay(ns)*	*Freq(Mhz)*
Shiftreg-orig.v	*1*	*1*	*1*	*3*	*6*	*16*	*85.9*
Shiftreg-top1.v	*1*	*1*	*2*	*4*	*12*	*17*	*65.6*
Shiftreg-top2.v	*1*	*1*	*2*	*4*	*7*	*16*	*85.9*
Shiftreg-top3.v	*1*	*1*	*3*	*3*	*3*	*8*	*83.2*
Shiftreg-top4.v	*1*	*1*	*2*	*4*	*13*	*17*	*68.6*
Dk14-orig.v	*3*	*5*	*1*	*3*	*32*	*23*	*63.8*
Dk14-top1.v	*3*	*5*	*3*	*3*	*30*	*23*	*63.8*
Dk14-top2.v	*3*	*5*	*3*	*3*	*31*	*23*	*57.1*
Dk15-orig.v	*3*	*5*	*1*	*2*	*19*	*23*	*64.6*
Dk15-top1.v	*3*	*5*	*2*	*2*	*15*	*20*	*81*
Dk15-top2.v	*3*	*5*	*2*	*2*	*16*	*20*	*64.6*
Dk27-orig.v	*1*	*2*	*1*	*3*	*6*	*16*	*85.9*
Dk27-top1.v	*1*	*2*	*3*	*3*	*8*	*16*	*85.9*
Dk27-top2.v	*1*	*2*	*3*	*3*	*6*	*16*	*87.4*
Dk512-orig.v	*1*	*3*	*1*	*4*	*16*	*19*	*68.6*
Dk512-top1.v	*1*	*3*	*2*	*4*	*15*	*20*	*78.5*
Dk512-top2.v	*1*	*3*	*3*	*6*	*27*	*21*	*54.9*

Tabelul 20. Raport pentru Xilinx-XC4000XL

mcnc files	*PI*	*PO*	*sub FSM*	*XILINX-XC4000XL*			
				DFFs	*Area(FGs)*	*Delay(ns)*	*Freq(Mhz)*
Shiftreg-orig.v	*1*	*1*	*1*	*3*	*6*	*15*	*80.1*
Shiftreg-top1.v	*1*	*1*	*2*	*4*	*12*	*17*	*59.4*
Shiftreg-top2.v	*1*	*1*	*2*	*4*	*7*	*15*	*89.8*
Shiftreg-top3.v	*1*	*1*	*3*	*3*	*3*	*10*	*100.7*
Shiftreg-top4.v	*1*	*1*	*2*	*4*	*13*	*16*	*70.5*
Dk14-orig.v	*3*	*5*	*1*	*3*	*32*	*22*	*58.2*
Dk14-top1.v	*3*	*5*	*3*	*3*	*30*	*22*	*58.2*
Dk14-top2.v	*3*	*5*	*3*	*3*	*31*	*22*	*56.8*
Dk15-orig.v	*3*	*5*	*1*	*2*	*19*	*22*	*61.1*
Dk15-top1.v	*3*	*5*	*2*	*2*	*15*	*19*	*74*
Dk15-top2.v	*3*	*5*	*2*	*2*	*16*	*19*	*61.1*
Dk27-orig.v	*1*	*2*	*1*	*3*	*6*	*15*	*80.1*
Dk27-top1.v	*1*	*2*	*3*	*3*	*8*	*15*	*80.1*
Dk27-top2.v	*1*	*2*	*3*	*3*	*6*	*14*	*82.3*
Dk512-orig.v	*1*	*3*	*1*	*4*	*16*	*17*	*70.5*
Dk512-top1.v	*1*	*3*	*2*	*4*	*15*	*20*	*70.5*
Dk512-top2.v	*1*	*3*	*3*	*6*	*27*	*20*	*61.1*

După terminarea procesului de descompunere a automatului prototip, s-a testat funcționalitatea subautomatelor descompuse obținute cu programul SIS şi s-a verificat echivalența lor funcțională cu ajutorul comenzii „*seq_verify*" din programul VIS. Această comandă verifică echivalența secvențială a două rețele flatenate: reteaua prototip şi rețeaua de subautomate. Pentru calculul rezultatelor experimentale s-au utilizat ca exemple de test fişiere din setul de benchmark *MCNC* pentru a testa funcționalitatea ***fsmtool***-ului. Rezultatele sunt afişate în tabelele 19-22.

```
sis> read_kiss <inputfile.kiss2>
sis> state_assign (sa)
sis> print_stats (ps)
sis> print_latch (pl)
sis> read_library „synch.genlib"
sis> map_s
sis> print_map_stats (pms) [6]
```

```
vis> read_verilog <inputfile.v>
vis> flatten_hierarchy (fh)
vis> print_network_stats (pns)
vis> print_hierarchy_status (phs)
vis> print_models (pm) [7]
```

Tabelul 21. Raport pentru VIRTEX-II

mcnc files	*PI*	*PO*	*sub FSM*	*VIRTEX-II*			
				DFFs	*Area(FGs)*	*Delay(ns)*	*Freq(Mhz)*
Shiftreg-orig.v	*1*	*1*	*1*	*3*	*5*	*4*	*512.6*
Shiftreg-top1.v	*1*	*1*	*2*	*4*	*9*	*4*	*421.3*
Shiftreg-top2.v	*1*	*1*	*2*	*4*	*7*	*4*	*512.6*
Shiftreg-top3.v	*1*	*1*	*3*	*3*	*2*	*3*	*551.9*
Shiftreg-top4.v	*1*	*1*	*2*	*4*	*11*	*4*	*412.8*
Dk14-orig.v	*3*	*5*	*1*	*3*	*29*	*6*	*308.2*
Dk14-top1.v	*3*	*5*	*3*	*3*	*33*	*6*	*276.6*
Dk14-top2.v	*3*	*5*	*3*	*3*	*36*	*6*	*272.9*
Dk15-orig.v	*3*	*5*	*1*	*2*	*17*	*5*	*335*
Dk15-top1.v	*3*	*5*	*2*	*2*	*14*	*5*	*332*
Dk15-top2.v	*3*	*5*	*2*	*2*	*16*	*5*	*327.5*
Dk27-orig.v	*1*	*2*	*1*	*3*	*7*	*4*	*408.5*
Dk27-top1.v	*1*	*2*	*3*	*3*	*9*	*4*	*408.5*
Dk27-top2.v	*1*	*2*	*3*	*3*	*6*	*4*	*493.8*
Dk512-orig.v	*1*	*3*	*1*	*4*	*16*	*5*	*358.2*
Dk512-top1.v	*1*	*3*	*2*	*4*	*13*	*4*	*412.8*
Dk512-top2.v	*1*	*3*	*3*	*6*	*29*	*5*	*258.2*

Tabelul 22. Raport pentru Altera Flex-10k

mcnc files	*PI*	*PO*	*sub FSM*	*ALTERA FLEX-10K*			
				DFFs	*Area(FGs)*	*Delay(ns)*	*Freq(Mhz)*
Shiftreg-orig.v	*1*	*1*	*1*	*3*	*8*	*7*	*96.4*
Shiftreg-top1.v	*1*	*1*	*2*	*4*	*13*	*9*	*81.7*
Shiftreg-top2.v	*1*	*1*	*2*	*4*	*7*	*4*	*145.2*
Shiftreg-top3.v	*1*	*1*	*3*	*3*	*4*	*3*	*145.2*
Shiftreg-top4.v	*1*	*1*	*2*	*4*	*16*	*8*	*88.5*
Dk14-orig.v	*3*	*5*	*1*	*3*	*36*	*12*	*67.6*
Dk14-top1.v	*3*	*5*	*3*	*3*	*37*	*10*	*75.9*
Dk14-top2.v	*3*	*5*	*3*	*3*	*30*	*12*	*67.6*
Dk15-orig.v	*3*	*5*	*1*	*2*	*22*	*9*	*83.4*
Dk15-top1.v	*3*	*5*	*2*	*2*	*18*	*8*	*88.5*
Dk15-top2.v	*3*	*5*	*2*	*2*	*20*	*9*	*83.4*
Dk27-orig.v	*1*	*2*	*1*	*3*	*8*	*6*	*88.5*
Dk27-top1.v	*1*	*2*	*3*	*3*	*9*	*6*	*114.2*
Dk27-top2.v	*1*	*2*	*3*	*3*	*7*	*7*	*96.4*
Dk512-orig.v	*1*	*3*	*1*	*4*	*24*	*11*	*72.2*
Dk512-top1.v	*1*	*3*	*2*	*2*	*18*	*8*	*81.7*
Dk512-top2.v	*1*	*3*	*3*	*2*	*20*	*9*	*60*

Prin utilizarea metodelor externe de reprezentare a datelor cu ajutorul librăriei matematice ***GMP*** pentru implementarea subautomatelor descompuse, se pot aborda în mod virtual automate finite prototip cu un număr foarte mare de stări, în timp ce se poate menține un timp rezonabil de execuție a algoritmilor de descompunere. Alocarea dinamică a reprezentării interne a datelor („big-numbers”) este limitată doar de capacitatea fizică a memoriei RAM diponibile pe maşina de test.

Toate circuitele au fost descompuse utilizînd metoda descompunerii generale descrise în capitolul 3. Fiecare tabel conține implementări ale setului de exemple luate pentru cîte o familie tehnologică specifică (Xilinx, Altera). Primele 4 coloane din fiecare tabel descriu numele circuitului, numărul de intrări şi ieşiri primare, şi de asemenea numărul de subautomate finite obținute pentru fiecare circuit prototip. Ultimele 4 coloane de sub fiecare nume de tehnologie folosită descriu numărul de circuite bistabile de tip *D* folosite, suprafața şi întîrzierea pentru fiecare implementare top a automatului descompus, precum şi frecvența estimată pentru tehnologia aleasă.

Pe parcursul procesului de optimizare, setul de stări partiționate poate fi generat manual sau automat cu ajutorul pachetului ***fsmtool***. Pe parcursul procesului de sinteză, se

obțin rezultate superioare pentru toate librăriile în cazul automatului prototip *Shiftreg2-top2.v* precum şi *Shiftreg-top3.v* deoarece partițiile de stări prezintă o abordare superioară pentru grupul de stări selectate. Aceasta rezultă într-un număr redus de stari interne ceea ce implică o complexitate redusă a fiecărui subautomat obținut. *Dk512-top1.v* precum şi *Dk512-top2.v* prezintă un număr minim de bistabile tip *D* în maparea pe libraria Altera Flex-10k, dar un număr maxim de bistabile de tip *D* pentru celelalte librării. Maparea pe Libraria Vertex-II permite obținerea unei implementări la o frecvență mai înaltă, în timp ce librăria Spartan-XL permite o implementare într-o frecvență estimată mai redusă.

10.6 Concluzii şi implementări ulterioare

În această lucrare au fost proiectate şi testate un număr de programe standard precum ***genfsm***, ***fsmtool*** şi ***kiss2vl***, pentru a implementa partea teoretică cu scopul de a obține rezultate experimentale şi practice. A fost demonstrat potențialul şi eficiența programelor realizate de a genera o maşina de stări finite precum şi submasinile aferente, de a aplica o metodă de descompunere generală prin utilizarea unui set de partiții evoluate genetic şi posibilitatea de a obține submaşini valide pentru a reduce complexitatea fiecărei submaşini în timp ce numărul de submaşini obținute este menținut cît mai redus. Această etapă de pre-sinteză poate fi uşor integrată într-un program de sinteză standard şi prin abordarea sa generală poate fi adaptată imediat pentru diferite probleme de optimizare precum descompunerea funcțiilor logice complexe şi implementarea optimală pe familii tehnologice specifice cu un număr variabil de porți logice în structurile lor funcționale interne, de diferite mărimi şi suprafețe de implementare.

Acest proces standard poate fi imbunătățit în viitor prin găsirea unei metode alternative de a testa rezultatele optimale găsite în urma etapei de generare a rezultatelor. Necesitatea utilizării unui program extern standard pentru evaluarea rezultatelor poate fi înlocuită cu o funcție obiectiv internă într-un proces de îmbunătățire a calității soluțiilor obținute. Această îmbunătățire va reduce timpul de execuție precum şi complexitatea procesului dificil de optimizare.

Din exemplele alese care au fost sintetizate şi mapate pe diferite familii de circuite se poate concluziona faptul că metoda de descompunere generală poate fi aplicată în mod facil cu succes în algoritmii euristici pentru a obține rezultate superioare în procesul de sinteză a automatelor finite complexe, în probleme de codificarea stărilor, optimizarea suprafeței, reducerea căii critice, optimizarea frecvenței estimate pentru implementarea unui circuit ASIC sau FPGA.

Întrucît circuitele VLSI moderne au devenit în general sisteme foarte ample şi complexe, sunt necesare programe moderne de optimizare pentru a servi ca unelte de proiectare puternice şi eficiente. Autorul propune ca pachetul de programe ***fsmtool*** să se înscrie în categoria de noi unelte alternative de proiectare a circuitelor de ordinul zecilor şi sutelor de milioane de tranzistoare care vor apărea în următorii ani, ca o metoda alternativă de proiectare. În acest nou concept operatorul uman devine un factor secundar în proiectarea circuitelor ample. Optimizarea locală, bazată pe reguli, va deveni un factor secundar în raportul critic cunoscut sub numele de „*timing-closure*”. Optimizarea globală utilizează metode euristice de găsire a soluțiilor. Acest lucru nu garantează găsirea celor mai bune soluții posibile, în schimb oferă avantajul găsirii unei soluții cît mai apropiate de soluția optimală pentru criteriul propus. Acest lucru conferă un avantaj enorm în proiectarea circuitelor complexe în privința calității rezultatelor obținute, raportat la timpul de proiectare.

Capitolul 11

Concluzii finale. Contribuții originale. Obiective de viitor

11.1 Concluzii finale

Pe masură ce obiectivele cercetării din domeniul algoritmilor evolutivi a continuat să prolifereze, aplicațiile care utilizează tehnologii evolutive au migrat către sectorul comercial, dezvoltarea lor fiind alimentată de creşterea exponentială a puterii de calcul precum şi de dezvoltarea Internetului. În prezent, calculul evolutiv este un domeniu prosper, iar algoritmii genetici sunt considerati „solutii ale problemelor zilnice" (Haupt and Haupt 1998, p. 147) în domenii de studiu diverse precum predictia pieței de stocuri şi probleme de planificare, inginerie spațială, proiectarea microchip-urilor, biochinie şi biologie moleculară, precum şi planificarea zborurilor la aeroport sau a eficientizării liniilor de asamblare. Puterea evoluției a atins virtual orice domeniu de activitate, împărțind soluțiile la problemele apărute în mod transparent într-o multitudine de modalități posibile, iar noi metode de utilizare continuă să fie descoperite pe măsura ce cercetarea se dezvoltă. La baza algoritmilor evolutivi se află mai mult decît regula de bază a teoriei darwiniste[67], întrucit şansa aleatoare de variație, cuplată cu legea selecției, este o tehnică de rezolvare a problemelor cu o putere imensă şi prezintă o aplicabilitate aproape nelimitată în rezolvarea oricărui tip de problemă. Cheia succesului o reprezintă alegerea cît mai eficientă a reprezentării soluțiilor pentru problema propusă, astfel încît algoritmii evolutivi să fie cît mai natural de aplicat pentru a îsi putea demostra abilitătile deosebite în căutarea paralelă intrinsecă prin secționarea subspațiilor de stări din populațiile de soluții generate pe parcursul rezolvării problemei. Cei mai mulți algoritmi, clasici, sunt seriali, şi prin urmare aceştia vor putea explora spațiul de soluții al unei probleme doar într-o singură direcție, iar dacă soluția pe care o descoperă este suboptimală din punctul de vedere al fitness-ului, atunci nu este nimic de făcut decît să se abandoneze soluția gasită şi de a relua algoritmul de la început. Întrucît algoritmii evolutivi prezintă urmaşi multipli, aceştia pot explora spațiul de soluții în direcții multiple în acelaşi timp. În cazul în care una din căile alese se nimereşte a fi una gresită, atunci ei o pot elimina cu uşurință pentru a îşi putea continua lucrul cu soluțiile care prezintă un fitness mai bun, dîndu-le acestora o sansă mai mare de „supraviețuire" de fiecare dată cînd corespund criteriilor de căutare alese.

Datorită paralelismului care permite evaluarea implicită a mai multor scheme în acelaşi timp, algoritmii evolutivi sunt foarte bine cotați pentru rezolvarea problemelor unde spațiul soluțiilor potențiale este cu adevarat imens, prea vast pentru a căuta exhaustiv în orice perioadă rezonabilă de timp. Aceste tipuri de probleme pot fi clasificate ca probleme neliniare. Într-o problemă liniară, fitness-ul fiecarei componente este independent, deci prin urmare orice îmbunătățire adusă la orice parte va rezulta într-o îmbunătățire a sistemului ca intreg. Neajunsul acestui tip de probleme este că sunt foarte puține probleme reale care să corespundă acestui criteriu. Neliniaritatea este modelul în care schimbarea unei componente poate avea efecte în val asupra întregului sistem, şi unde schimbările multiple care sunt în detrimentul indivizilor pot conduce la imbunătățiri mult mai mari ale funcției de fitness atunci cînd sunt combinate. Neliniaritatea rezultă într-o explozie combinatorică, spatiul pentru şiruri binare de 1.000 de digiți poate fi căutat în mod exhaustiv prin evaluarea a doar 2.000 de posibilități dacă problema este liniară. În cazul în care problema este neliniară, o cautare

exhaustivă necesită evaluarea a peste 2^{1000} posibilități, ceea ce rezultă într-o evaluare a cărei complexitate va creşte exponențial cu dimensiunea spațiului de reprezentare a soluțiilor.

În mod fericit, paralelismul implicit pe care îl prezintă algoritmii evolutivi permit parcurgerea acestui numar enorm de posibilitați, prin metodele specifice, găsind cu succes rezultate foarte bune şi optime într-o perioadă scurtă de timp dupa tăierea directă doar a unor mici regiuni din vastul domeniu de căutare al funcției de fitness.

O altă caracteristică notabilă a algoritmilor evolutivi este faptul că pot fi utilizați în probleme în care fereastra funcției de fitness este complexă, este discontinuă, există zgomot, se schimbă pe parcursul timpului sau prezintă multe optime locale. Majoritatea problemelor practice prezintă un spațiu de soluții vast, imposibil de căutat cu metode exhaustive. Provocarea în acest caz devine evitarea optimelor locale, a soluțiilor care sunt mai bune decît cele similare cu ele, dar nu la fel de bune ca altele diferite din spațiul de căutare. Mulți dintre algoritmii de cautare pot fi prinşi într-un optim local, aceştia considerînd faptul că au atins o soluție cu un fitness mai ridicat drept soluția finală a problemei de rezolvat.

Algoritmii evolutivi au dovedit că sunt foarte eficienți în evitarea optimelor locale şi descoperirea optimelor globale ale unei soluții chiar şi în cazul funcțiilor de fitness complexe şi greu de evaluat. Cu toate acestea, un algoritm evolutiv nu furnizează întotdeauna cea mai bună soluție posibilă a unei probleme, dar poate garanta întotdeauna că a găsit cel puțin una din soluțiile cu fitness-ul foarte bun, şi care se găseşte în partea superioară din reprezentarea grafică a clopotului lui Gauss.

O calitate deosebită a algoritmilor evolutivi, care face diferențierea cea mai clară dintre ei şi algoritmii clasici, este faptul că algoritmii evolutivi nu au cunoştință despre specificațiile problemelor pe care trebuie sa le rezolve. În loc să utilizeze informații specifice domeniului, cunoscute anterior, pentru a îşi ghida fiecare pas şi de a face schimbări evidente pentru a îmbunătăti soluția cu un anumit scop, cum ar face un operator uman foarte calificat în acest scop, algoritmii evolutivi sunt „ceasornicari orbi" (Dawkins, 1996), aceştia efectuează schimbări aleatoare în interiorul soluțiilor candidate şi apoi utilizează funcția de fitness pentru a determina dacă aceste schimbări produc o imbunătățire. Meritul aceste tehnici este faptul că permite algoritmilor evolutivi să pornească în căutarea solutiilor cu o „minte deschisă", întrucit deciziile sunt bazate pe căutare aleatorie, orice căi de căutare posibile sunt teoretic deschise; prin contrast, orice strategie de rezolvare a unei probleme care se bazează pe cunostințe anterioare trebuie să înceapă în mod inevitabil prin eliminarea multor căi considerate ineficiente, prin urmare eliminînd orice solutie nouă care poate înca exista (Koza et al. 1999, p. 547). Algoritmii evolutivi nu sunt afectați de preconcepțiile bazate pe convingerile stabilite „cum ar trebui făcute lucrurile", sau „nu poate fi posibil să meargă". În mod similar, orice tehnică care se bazează pe cunoştințe anterioare va cădea atunci cînd aceste cunostințe nu sunt disponibile. Cu toate acestea, algoritmii evolutivi nu sunt afectați de „ignoranța" (Goldberg, 1989, p. 23).

Algoritmii evolutivi au o aplicabilitate directă în proiectarea circuitelor complexe, întrucit numărul mare de stări posibile din blocurile componente ale microprocesoarelor actuale a crescut exponential şi va continua să crească în următorii ani. Una din aplicațiile posibile sunt în cadrul proiectării utilizînd circuite de tip FPGA (field programable gate array) , care sunt niste tipuri speciale de circuite ce prezintă o arie de celule logice, fiecare dintre ele putînd avea funcționalitatea unei porți logice, conectate prin interconexiuni flexibile care pot lega celulele. Ambele funcții pot fi controlate prin software prin încărcarea unui program special în circuit şi care poate apoi altera din mers funcțiile oricărei dintre dispozitivele hardware. Aceste dispozitive pot fi exploatate în conjuncție cu principiile evolutive, pentru a produce circuite prototip, autoadaptabile pentru o anumită funcție (reconfigurarea funcțiilor logice, recunoaşterea vocii, etc). Spre exemplu s-a obținut un prototip de circuit pentru recunoaştere vocală care recunoaşte şi răspunde la comenzi vocale într-o maniară de o

simplitate aproape imposibil de realizat de către un operator uman, utilizînd doar 37 porti logice, fapt imposibil de explicat logic. Dr. A. Thompson a reuşit să genereze şiruri aleatoare de biți de 0 şi 1 care au fost utilizați pentru configurarea circuitului FPGA, prin selectarea celor mai potriviți indivizi din fiecare generație, care au fost apoi supuşi operatorilor de reproducere şi mutație. Scopul a fost evoluarea unui dispozitiv care poate distinge între tonuri de diferite frecvențe (1-10 KHz) şi apoi să distingă între cuvinte vorbite „start” si „stop”. Circuitul a evoluat fără orice coordonare inteligentă, dintr-un set de stări complet aleatoare şi nefuncționale, într-un set complex de stări, eficiente şi optimale.

Îmbunătățirile tehnologice au dus la posibilitatea de a realiza structuri digitale avînd o complexitate din ce în ce mai ridicată, în acelaşi timp obținîndu-se o reducere a dimensiunilor circuitelor în care acestea sunt integrate, precum şi reducerea consumului de energie şi mărirea vitezei de prelucrare a informațiilor. Pentru a putea aborda structurile numerice complexe, a trebuit să se găsească noi moduri de reprezentare ale acestora şi de asemenea, a unor noi metode de manipulare a acestor reprezentări. Diagramele de decizii binare sunt una dintre metodele de reprezentare eficiente a funcțiilor booleene şi a altor funcții ce pot fi reduse la funcții logice.

Unul din domeniile de aplicatie ale reprezentărilor şi a tehnicilor implicite este cel al reprezentării mulțimilor de stări ale automatelor finite şi implementării algoritmilor de manipulare a acestor multimi de stări. Datorită faptului că majoritatea metodelor utilizate în sinteza automatelor finite au la bază operații de manipulare a multimilor de stări, rezultă că acestea pot fi implementate folosind reprezentările şi metodele de manipulare implicite. Implementarea eficientă a acestor tehnici presupune realizarea unor operatori impliciti care să fie eficienți din punct de vedere al timpului de execuție şi al memoriei consumate.

În capitolul 4 s-a menționat că descompunerea şi compunerea automatelor finite sunt etape care se întilnesc des în practica, ele fiind elemente importante în procesul de sinteză al unui automat finit. Aceste probleme se impun mai ales atunci cînd există anumite restricții, fie impuse de suportul fizic în care este implementat automatul, şi care se referă la dimensiunile maxime ale automatelor ce pot fi implementate, fie la problemele legate de distribuirea semnalelor de ceas şi a reducerii legăturilor dintre partea de control şi cea de prelucrare a datelor.

În capitolul 5 sunt prezentate în detaliu metodele de reprezentare folosite în acest sens precum şi modul în care acestea pot fi utilizate pentru realizarea unor algoritmi care sa poată manipula mulțimi de obiecte de dimensiuni foarte mari. Aceste proprietăți au făcut ca reprezentările folosind diagramele de decizie binară să poată fi folosite în rezolvarea oricărei probleme care poate fi redusă la problemele ce vizează reprezentări prin funcții booleene şi manipularea acestora. Pentru reprezentările explicite, obiectele sunt reprezentate intern în memorie ca o listă de obiecte a cărei dimensiune este proportională cu numarul de obiecte considerat, în timp ce pentru reprezentările implicite, obiectele sunt reprezentate folosindu-se anumite proprietăți comune, ceea ce face ca dimensiunile reprezentării să nu mai fie proporționale cu numărul de obiecte.

Descompunerea automatelor finite presupune găsirea unei partiții pe multimea stărilor automatului şi poate fi făcută ținînd seama de diferite proprietăți pe care le are această mulțime. Au fost studiate proprietățile algebrice legate de existența partițiilor avînd proprietatea de substituție. Mulțimea acestor partiții are o structură de latice a cărei cunoaştere este utilă, atît pentru realizarea descompunerilor în serie si paralel dar şi în cazul descompunerii generalizate care le cuprinde pe toate la un loc.

În capitolul 6 s-au prezentat etapele de sinteză a circuitelor digitale complexe; aceste etape sunt strîns legate. Realizarea unui mediu de proiectare asistată de calculator pentru sinteza circuitelor complexe presupune implementarea unui număr mare de algoritmi care să rezolve toate problemele care se pun în cazul sintezei unei astfel de structuri. Acest lucru

poate fi mai uşor realizat dacă se ține cont şi de existența unor programe, disponibile în rețeaua universitară, deja realizate, şi care ajută în realizarea unor anumite etape ale procesului de sinteză şi verificare a rezultatelor. Prin urmare s-a ales de către autor posibilitatea de a realiza un mediu de proiectare asistată de calculator a circuitelor digitale complexe prin utilizarea unor structuri de circuit deja existente, utilizate ca date de intrare, sau posibilitatea generării de noi circuite, aleatoare, în scopul obținerii unor rezultate experimentale, de test, pentru validarea funcționării practice a anumitor algoritmi de optimizare implementați de către autor. Funcționalitatea mediului de proiectare poate fi extinsă prin integrarea cu celelalte programe de proiectare, atît din rețeaua universitară cît şi comerciale, datorită utilizării formatului de reprezentare a datelor obtinute în format Verilog, care este cunoscut ca limbaj standard industrial în proiectarea circuitelor digitale. Bineințeles, s-a facut şi un studiu asupra efortului logic facut în proiectarea circuitelor, în capitolul 7, şi tot aici s-a descris şi funcționarea a doi algoritmi de calcul a funcției de fitness pentru optimizarea unui automat finit legat de funcția de fanin, respectiv funcția de fanout a circuitului.

In capitolul 8 s-a facut un studiu asupra metodelor globale de optimizare şi a aplicabilității acestora în cadrul optimizării automatelor finite deterministe, iar în capitolul 9 s-au propus diverse metode de implementare a acestor algoritmi pentru obținerea unor rezultate elocvente în această privintă. Rezultatele au fost calitative, întrucît s-au obținut pentru automate finite cu un numar de peste 10.000 de stari, imbunătățiri ale funcției de fitness cu pînă la 218 ori față de funcția de fitness initială, neoptimizată.

11.2 Contribuții originale

Lucrarea elaborată reprezintă rezultatul unor cercetari în domeniul teoriei şi proiectării automatelor finite în scopul realizării unui sistem care să permită sinteza circuitelor digitale complexe utilizînd algoritmi de proiectare evolutivi, ca aplicații ale inteligenței artificiale în proiectarea circuitelor.

Rezultatele acestei activitați constituie o serie de contribuții proprii precum:

- Introducerea unor tipuri noi de reprezentări utilizînd diagramele de decizie binară pe baza folosirii de noi reguli de reducere
- Introducerea unor noi metode euristice de descompunere a automatelor finite prin utilizarea relației de echivalentă la intrări şi ieşiri definită pe multimea de stări a automatului prototip
- Proiectarea programului ***fsmtool*** care utilizează tehnologiile definite şi permite realizarea operațiilor de descompunere a automatelor finite prin metoda descompunereii generalizate; în plus programul are capacitatea de a îsi autogenera fişierele de intrare sau de a le prelua ca date de intrare; prin utilizarea metodelor euristice rezultatele descompunerii pot fi imbunatațite calitativ în funcție de anumite criterii specificate ca parametri de intrare
- Implementarea programului ***kiss2vl*** utilizat pentru conversia reprezentarii unui automat finit de la nivel de tabel de tranziție (Kiss2) în format portabil industrial (Verilog), pentru integrarea cu alte programe similare şi verificare facilă cu orice program specific a rezultatelor obtinute cu programul ***fsmtool***
- Implementarea programului ***genfsm*** utilizat pentru generarea automată a circuitelor de test a algoritmilor evolutivi proiectați
- Definirea şi implementarea unor algoritmi de calcul ai funcției de fitness (optimizarea funcțiilor de Fanin/Fanout/Area) pentru un circuit digital complex în scopul reducerii complexității acestuia pină la un nivel logaritmic (peste 10.000 stari)

- Implementarea de operatori evolutivi pentru a permite evoluarea şi convergența populațiilor de soluții inițiale în scopul găsirii unui optim global
- Codificarea internă a soluțiilor, utilizînd un algoritm de menținere a granularității fenotipului în raport cu complexitatea genotipului, prin detectie automată
- Prin utilizarea bibliotecii de funcții GMP („big numbers"), s-a putut extinde complexitatea automatelor prelucrate la o valoare virtuală infinită, singurele limitări au rămas doar cele legate de dimensiunea memoriei fizice disponibile (fără paginarea memoriei pe disc din motive de performantă) şi puterea de procesare a algoritmilor
- Validarea integrală a corectitudinii metodei de descompunere a automatelor finite, precum şi utilizarea programului extern, VIS, folosit pentru verificarea corectitudinii circuitelor rezultate
- Posibilitatea generării rezultatelor obținute în urma optimizarii circuitelor complexe direct în format Verilog (inclus intern în funcționarea programului), ceea ce a permis reutilizarea codului precum şi testarea rapidă a performanțelor rezultatelor obținute, cu orice program existent în domeniul proiectării circuitelor CAD

În structura lucrarii, capitolele 4, 7, 8, 10 şi 11 conțin contribuții proprii. Rezultatele obținute au fost publicate sub forma de lucrări la diverse conferințe naționale şi internaționale sau au fost publicate în jurnal. Exemple de rezultatele experimentale există în anexa A.

11.3 Obiective de viitor

În această lucrare autorul propune şi deschide începutul unei activități care va cuprinde domeniul sintezei circuitelor numerice complexe, precum şi noi metode de abordare a dimensiunii din ce în ce mai mari a functionalității logice a circuitelor, prin utilizarea metodelor euristice precum algoritmii evolutivi, ca aplicații ale Inteligenței Artificiale în proiectarea circuitelor.

După cum s-a putut observa pe parcusrsul lucrării, metodele euristice nu sunt de natură exactă, de aceea nu pot garanta găsirea celei mai bune soluții posibile, dar promit o apropiere cît mai eficientă de soluția optimă din spațiul de soluții, care corespunde criteriului ales. Algoritmii euristici nu pot înlocui algoritmii clasici, existenți, dar îi pot complementa în rezolvarea unor probleme dificile, de natura neliniară, sau complexitate polilogaritmică.

Algoritmii definiți şi utilizați pe parcursul acestei lucrari pot fi imbunătățiți şi eficientizați din punctul de vedere al calității rezultatelor, timpului de convergență, optimizarea funcției de fitness precum şi adăugarea de noi criterii de optimizare pentru circuitele complexe, dimensiunea memoriei utilizate (pentru un număr imens de stări interne devine necesară paginarea memoriei fizice pe dispozitive externe), adaugărea de noi metode şi algoritmi în procesul de sinteză, astfel încît să nu mai fie necesară prezența altor programe externe pentru a verifica corectitudinea rezultatelor precum şi preluarea şi convertirea lor în alt format compatibil cu următorul pas de sinteză.

Evoluția este un proces de rezolvare a problemelor care încă este într-o stare incipientă de ințelegere şi exploatare. Cu toate acestea, este deja prezent în viața cotidiană, delimitînd noi tehnologii şi imbunătățind viața de zi cu zi, iar aceste imbunătățiri în viitor vor continua să se diversifice; în lumea stiințifică se spune deja că viitorul va fi „evoluția". Nici unul dintre nenumaratele avantaje ale utilizării algoritmilor evolutivi nu ar fi posibil fără existența unei înțelegeri detaliate a procesului evolutiv. Evoluția este o realitate, şi este în continuă desfăşurare, depinde de oameni cum înțeleg acest lucru şi de cum îl pot folosi în scopul abordării cît mai eficiente a problemelor din ce în ce mai complexe cu care ne confruntăm în toate domeniile de activitate.

Referințe

[1] ALMAINI A.E.A., MILLER J.F., THOMSON P., BILLINA S., *State assignment of Finite State Machines using a Genetic Algorithm*, Dept. Of Electrical and Computer Engineering, Napier University, Edinburgh, 1995
[2] ALMAINI, A.E.A., ALI B., KALGANOVA T., *Evolutionary Algorithms and their use in the Design of Sequential Logic Circuits*, Genetic Programming and Evolvable Machines, 5, 11-29, 2004, Kluwer Academic Publishers, 2004
[3] BIRKHOFF G. , *Lattice Theory*. Amer. Math. Soc. Colloquium Publ. 25, revised edition, New York, 1948
[4] BOOLE G. , *The mathematical analysis of logic*. Cambridge 1847
[5] BRAYTON R.K., HACHTEL G.D., McMULLEN C.T., SANGIOVANNI-VINCENTELLI A., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984
[6] BRAYTON R.K., SANGIOVANI-V.A. , SIS: *A system for sequential circuit synthesis*, Electronic Research Laboratory, Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 4 May 1992
[7] BRAYTON R.K., VIS: A *system for verification and synthesis*. Technical Report UCB/ERL M95/104, Electronics Research Lab, Univ. Of California, December 1995
[8] BRAYTON R.K., *Logic Synthesis*, EE219B, EECS Dept. UC Berkeley, 1998
[9] BRAYTON R., HACHTEL G., McMULLEN C., SANGIOVANNI-VINCENTELLI A., *Logic minimization algorithms for VLSI synthesis*, Kluwer Academic Publishers, 1984
[10] BRZOZOWSKI J.A., LUBA T. , *Decomposition of Boolean Functions Specified by Cubes*, University of Waterloo, Canada, Warsaw University of Technology, Poland
[11] BRYANT R., *Graph based algorithm for Boolean function manipulation*, IEEE Transactions on Computers, vol. 35 (no. 8), pp 667-691, 1986
[12] BRYANT R.E., *On the complexity of VLSI implementations and graph representations of Boolean functions with applications to integer multiplication*, IEEE Transactions on Computers, C-40:205-213, February 1991
[13] BURKE E.K. and J.P. NEWALL. "*A multistage evolutionary algorithm for the timetable problem*." IEEE Transactions on Evolutionary Computation, vol.3, no.1, p.63-74 (April 1999)
[14] CALISTRU C.N., PRUTEANU C., *Modern Control Optimization Strategies for Low Cost Automation*, accepted in Jurnal on The 4th WSEAS SIMULATION, MODELLING AND OPTIMIZATION (ICOSMO 2004), ISBN 960-8457-02-5, September 13-16, 2004, Izmir, Turkey
[15] CAPCARRÈRE M. , *Cellular Automata and other Cellular Systems: Design and Evolution*, PhD. Thesis No. 2541, March 1st, 2002
[16] CARLOS L.Q. , STRUM M. , GERPAR and GERTAB2. *Two FSM Decomposition Optimization Tools*, Laboratorio de Microelectronica LME-EPUSP, Brasil
[17] CHELLAPILLA, K. , FOGEL D. , "*Evolving an expert checkers playing program without using human expertise.*" IEEE Transactions on Evolutionary Computation, vol.5, no.4, p.422-428 (August 2001)
[18] CHEN D., AOKI T., HOMMA N., Toshiki T., Higuchi T., *Graph-Based Evolutionary Design of Arithmetic Circuits*, IEEE Transactions on Evolutionary Computation, Vol. 6, No. 1, Febr. 2002
[19] COELLO C. , "*An updated survey of GA-based multiobjective optimization techniques*." ACM Computing Surveys, vol.32, no.2, p.109-143 (June 2000)
[20] COELLO C., CHRISTIANSEN A.D., ARGUIRE A.H., *Automated Design of Combinational Logic Circuits by Genetic Algorithms*, Proceedings of the International Conference in Norwich, U.K., 1997, Springer Wien New York

[21] COUDERT O., MADRE J.C., *A new method to compute prime and essential prime implicants of boolean functions. Advanced research in VLSI and parallel systems*, pp 113-128, The MIT Press, T. Knight and J. Savage Editors, March 1992
[22] CROITORU C., *Tehnici de bază în optimizarea combinatorie*, Ed. Univ „Al. I. Cuza”, Iaşi, 1992
[23] CROITORU C., *Introducere în algoritmica Grafurilor*, Editura Tehnică, Bucureşti, 2001
[24] DE GARIS H., *Genetic Programming: Evolutionary Approaches to Multistrategy Learning*, chapter 21, Morgan Kauffman, 1994
[25] DE GARIS H., *An artificial brain – ATR's CAM-brain project aims to build/evolve an artificial brain with a million neural net modules inside a trilion cell cellular automata machine*, New Generation Computing, vol. 12, Berlin: Springer, 1994, pp 215-221
[26] DE GARIS H., *One-Chip Evolvable Hardware: 1C-EHW*, Springer-Verlag Wien New York, Proceedings of the International Conference in Norwich, U.K., 1997
[27] DE JONG K., *The analysis of the behavior of a class of genetic adaptive systems*, PhD. Dissertation Department of Computer Science, 1975, University of Michigan
[28] DEMICHELI G., BRAYTON R., SANGIOVANNI-VINCENTELLI A., *Optimal state assignment for finite state machines*, IEEE Trans. on Computer Aided Design, July 1985
[29] DEMICHELI G., *Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros*, IEEE Transactions on Computer Aided Design, October 1986
[30] DEMICHELI G., *Synthesis and Optimization of Digital Circuits*, Stanford University, McGraw-Hill, 1994
[31] DEVADAS S., NEWTON A. R., *Decomposition and factorization of sequential finite state machines*, IEEE Trans. on CAD , Vol 8, No. 11, November 1989, pp. 1206-1217
[32] DEVADAS S., NEWTON A.R., SANGIOVANNI-VINCENTELLI A., MA H.T., *Synthesis and optimization procedures for fully and easily testable sequential machines*, Proceedings of the International Conference on Computer Aided Design, Novermber 1987
[33] DEVADAS S., NEWTON A.R., SANGIOVANNI-VINCENTELLI A., MA H.T., *MUSTANG: state assignment of finite state machines targeting multi-level logic implementations*, IEEE Transactions on Computer Aided Design, December 1988
[34] DEVADAS S., *General decomposition of sequential machine: Relationships to state assignment*, Proceedings of the Design Automation Conference, pp. 314-320, June 1989
[35] DILWORTH R.P., *The arithmetical theory of Birkhoff lattices*. Duke Math. J. 8 (1941) 286-289
[36] DOLOTTA, McCLUSKEY, *The conding of internal states of sequential machines*, IEEE Transactions, EC 13, pp 549-562, 1964
[37] DRECHSLER R., BECKER B., *Binary Decision Diagrams, Theory and Implementation*, Kluwer Academic Publishers, Boston, 1998, ISBN 0-7923-8193-9
[38] DRECHSLER R., BECKER B., GŐCKEL N., *A genetic algorithm for variable ordering of OBDDs*, International workshop on Logic Synthesis, Granlibakken, CA, May 1995
[39] EDWARDS S., *States Graphical Maniputation Tool for FSM*, March 1995
[40] FONSECA C., FLEMING P. , "*An overview of evolutionary algorithms in multiobjective optimization*." Evolutionary Computation, vol.3, no.1, p.1-16 (1995)
[41] GOLDBERG D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989
[42] GOLDBERG D.E., DEB K., THIERENS D., 1992b, *Toward a better understanding of mixing in genetic algorithms*, Proceedings of 4th International Conference on Genetic Algorithms (San Diego, CA, July, 1991), ed. R Belew and L Booker, pp 190-195
[43] HABA C.G., *Contributions to synthesis of digital circuits*, PhD. Thesis, "Gh. Asachi " Technical University, Iaşi, Romania

[44] HABA C.G., HUTANU C., *Computer Aided Synthesis of Digital Circuits using Binary Decision Diagrams*, (in Romanian), Tehnica-Info Ed., Chişinău, 2002
[45] HABA C.G., BAHRIN V., PRUTEANU C., *Indiphaso e-learning system - access to software tools for digital design computer simulation of the dynamics of virus spreading*, Journal of Medical Informatics & Technologies, Vol. 9 October 2005-12-14, (ISSN 1642-6037), PP. 313-320, Poland
[46] HABA C.G., BAHRIN V., PRUTEANU C., *INDIPHASO e-Learning System - Access to Software Tools for Digital Design*, Proceedings of Distance Learning Workshop 2005, 19-21 October, Ustron, Poland, pp.1-8, ISBN 83-922374-0-4
[47] HABA C.G., *Finite State Machine Decomposition using implicit techniques*, Proceedings of Sintes 9 International Symposium on Systems Theory, Robotics and Process Informatics, June 4-6, 1998, Craiova, România
[48] HABA C.G., *A method for FSM decomposition using factorization* , Politechnic Institute Journal IASI, XLV(IL), s. III, Fasc. 5, 1999
[49] HARTMANIS J., STEARNS R.E., *Algebraic Structure Theory of Sequential Machines*, Prentice Hall, Englewood Cliffs, N.J., 1966
[50] HINTERDING R., MICHALEWICZ Z., EIBEN A.E., *Adaptation in evolutionary computation: a survey*, Proceedings of the 4th IEEE Conference on Evolutionary Computation, 1997, NJ:IEEE, pp 65-9
[51] HAUPT R., HAUPT S.E. , *Practical Genetic Algorithms*. John Wiley & Sons, 1998
[52] HOLLAND J. H., "*Genetic algorithms*." Scientific American, July 1992, p. 66-72
[53] HOLLAND J. H., *Adaptation in Natural and Artificial Systems. An introductory analysis with applications to Biology, Control and Artificial Intelligence*, MIT Press, Cambridge, Massachusetts, 1992
[54] HOPCROFT J.E., *A n log n algorithm for minimizing states in finite automata*. Tech Report Stanford University, CS 71/190, 1971
[55] HOPCROFT J.E., ULLMAN J.D., *Introduction to Automata theory*, Languages and computation, Addison-Wesley Publishing Company, 1979
[56] HORN J., *Finite Markov chain analysis of Genetic Algorithms with niching*, Proceedings of the 5th International Conference on Genetic Algorithms, pp 110-117, 1993
[57] HOZA F., GÂLEA D., *"Structura Calculatoarelor Numerice"*, Vol II, Ed. Tehnopres, 2001, ISBN 973-8048-37-0, Tempus INCOT 12123-97.
[58] GERBAUX L., SAUCIER G., *Automatic synthesis of large Moore sequencers*, the VLSI Journal 13, 1992
[59] INDIPHASO, *E-LEARNING SYSTEM FOR THE DESIGN OF EMBEDDED HARDWARE-SOFTWARE APPLICATIONS*, Project supported under CNCSIS grant No. 33371/29.06.2004 Theme 26 CNCSIS Code 516
[60] JERRAYA A.A., PARK I., O'BRIEN K., *AMICAL: an interactive high level synthesis environment*, EDAC-EUROASIC Conference, Paris, France, 22-25 February 1993
[61] JÓZWIAK L., CHOJNACKI A. : *Application of Information Relationship Measures to Logic Synthesis*, Proc. CSSP98 - 9th Annual Workshop on Circuits, Systems and Signal Processing, Mierlo, The Netherlands, November 1998
[62] JUCAN T., ANDREI S., *Limbaje Formale si Teoria Automatelor*, Ed. Univ. „Al. I. Cuza", Iasi, 1997
[63] KAM T., VILLA T., BRAYTON R., SANGIOVANNI-VINCENTELLI A., *A fully implicit algorithm for exact state minimization*, Proceedings of Design Automation Conference, pp 684-690, June 1994
[64] KOHAVI Z., *Switching and Finite Automata Theory*, McGraw-Hill Book Company, New York, second edition, 1978

[65] KOUTSOFIOS E., NORTH S.C., *Drawing graph with dot (user manual)*, AT&T Laboratories, Murray Hill, NJ, November 1996
[66] KOZA J.R., *Genetic Programming. On the Programming Computers by Means of Natural Selection*, The MIT Press, Cambridge, Massachusetts, London, England, 1992
[67] KOZA J.R., FOREST B., DAVID A., KEANE M., *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann Publishers, 1999
[68] LIN B., *Synthesis of VLSI designs with symbolic techniques*, PhD. Thesis, University of California, Berkeley, November 1991
[69] MALITA M., *Bazele inteligentei artificiale. Logici propozitionale*, Editura Tehnica, Bucuresti, 1987
[70] MAN K.F., TANG K.S. and KWONG S., *Genetic Algorithms. Concepts and Designs*, Springer, 1999
[71] McCLUSKEY E., *Logic Design Principles*, Prentice-Hall, Englewood Cliffs, NJ, 1986
[72] MENON A., *Frontiers of Evolutionary Computation*, ProductSoft Inc., Pittsburg, Pennsylvania, USA, eBook ISBN: 1-4020-7782-3, Print ISBN: 1-4020-7524-3, 2004
[73] MINATO S., *Fast Generation of Irredundant Sum-of-Product Forms from Binary Decision Diagrams*, Sasimi 1992, pp 64-73, Japan, 1992
[74] MINATO S., *Fast Weak-division method for implicit cube representation* – Synthesis and simulation meeting and International exchange, SASIMI 93, Nara, Japan, 1993
[75] MITCHELL M., *An Introduction to Genetic Algorithms*. MIT Press, 1996
[76] MUNTEAN I. , *Finite Automata Synthesis*, Editura Tehnica, Bucuresti, 1977
[77] MVSIS, MVSIS Group, *Multi-level Logic Synthesis*, Electrical Engineering and Comp. Sciences Dept, University of California, Berkeley, CA 94720, august 2001
[78] OHIGASHI H., HIGUCHI T., *Hardware Implementation of Computation in Genetic Algorithms*, Electrotechnical Laboratory Technical Report, 1995
[79] PEDRAM M., MACII E., SOMENZI F., *High-Level Power Modeling, Estimation and Optimization*, Proceedings of DAC, pp. 504-511, 1997
[80] PEDRAM M., LAI Y.T., VRUDHULA S.B.K., *BDD based decomposition of logic functions with application to FPGA synthesis*, 30th ACM/IEEE Design Automation Conference, 1993
[81] PARTHA S. R. , SOWMYA A. , *Functional Decomposition of Composite Finite State Machines*, University of New South Wales, Australia
[82] PERKOWSKI M., BURNS M.,JOZWIAK L, and GRYGIEL S., ``*An Efficient and Effective Approach to Column-Based Input/Output Encoding in Functional Decomposition"* Proceedings of 3rd International Workshop on Boolean Problems, pp. 19-29, Freiberg University of Mining and Technology, Institute of Computer Science, September 17-18, 1998.
[83] PERKOWSKI M. , *Digital design automation : finite state machine design*, dept. of electrical engineering, Portland State Univeristy
[84] PRANAV A., DEVADAS S., NEWTON A.R., *A unified approach to the decomposition and re-decomposition of sequential machines*, Proceedings of the Design Automation Conference, pp. 601-606, June 1990
[85] PRANAV A., GOSH A., DEVADAS S., NEWTON A.R., *Implicit state transition grapths: application to logic synthesis and test*, Proceedings of the International Conference on Computer Aided Design, pp 259-264, November 1990
[86] PRANAV A., SRINIVAD D., NEWTON R., *Optimum and Heuristic Algorithms of an Approach to Finite State Machine Decomposition*, IEEE Transactions on computer aided design, vol.10, no. 3, march 1991
[87] PRANAV A., *Synthesis of sequential circuits for VLSI design*, PhD. Thesis, University of California, Berkeley, 1992

[88] PRUTEANU C., *Complex Circuit Design. Finite Automata Decomposition*, Proceedings on The 7th International Conference on Development and Application Systems, 27-29 May 2004, pg. 294-298, Suceava, România
[89] PRUTEANU C., GÂLEA D., HABA C.G., CALISTRU C.N., *Global Optimization in Complex Circuits Design*, Proceedings on The 4th WSEAS SIMULATION, MODELLING AND OPTIMIZATION (ICOSMO 2004), September 13-16, 2004, Izmir, Turkey
[90] PRUTEANU C., GÂLEA D., *An Approach to Finite Automata Decomposition Using Genetic Algorithms*, Proceedings on ECIT 2004 – Third European Conference on
Intelligent Systems and Technologies, (ECIT 2004), ISBN 973-7994-77-9, Iaşi, România, July 21-23, 2004
[91] PRUTEANU C., GÂLEA D., HABA C.G., *Global Optimization in Complex Circuits Design*, Proceedings on 7th IEEE International Symposium on Signals, Circuits, Systems (ISSCS 2005), vol.2, ISBN 0-7803-9029-6, IEEE Catalog Number: 05EX1038, July 14-15, 2005, pg. 785-788, Iasi, Romania
[92] PRUTEANU C., GÂLEA D., HABA C.G., *Application of Finite State Machine General Decomoposition Method with Optimization*, Proceedings on The 8th International Conference on Development and Application Systems (DAS 2006), ISBN 973-666-194-6, 25-27 May 2006, pg. 325-332, Suceava, Romania
[93] PRUTEANU C., GÂLEA D., HABA C.G., *An Extrinsic Evolvable Hardware Approach to Logic Synthesis Optimization*, Proceedings on The 8th International Conference on Development and Application Systems (DAS 2006), ISBN 973-666-194-6, 25-27 May 2006, pg. 277-280, Suceava, Romania
[94] PRUTEANU C., PRUTEANU A.S., CALISTRU C.N., *Web Control on Internet Radio Streaming*, Proceedings on 8th International Symposium on Automatic Control and Computer Science, ISBN 973-621-086-3, October 22 - 23, 2004, Iasi, Romania
[95] Roche Emmanuel , *Factorization of Finite-State Tranducers, Cambridge*, February 1995
[96] RUDOLPH G., *Convergence analysis of canonical Genetic Algorithms*, IEEE Transactions on Neural Networks, 5(1):96-101, 1994
[97] RUPESH S.S., MADHAV P.D., NARAYANAN H., *Decomposition of Finite State Machines for Area, Delay Minimization*, Indian Institute of Technology, dept. of electrical engineering
[98] SIS, *A system for sequential circuit synthesis*, University of California, Berkeley, CA, 1992
[99] SOMENZI F. , *CU Decision Diagram Package* , Dept. of Electrical Eng., Colorado Univ. Boulder
[100] STEFAN G. , *Digital systems and Circuits*, Editura Tehnica, Bucuresti, 2000
[101] SZÁSZ G., *Introduction to Lattice Theory*, Thrid Revised and Enlarged Edition, Akademiai Kiado, Budapest, 1963
[102] SWAMY G., BRAYTON R., McGEER P., *A fully implicit Quine-McCluskey procedure using BDD's*. Tech. Report No. UCB/ERL M92/127, 1992
[103] TANESE R., *Parallel genetic algorithms for a hypercube Genetic Algorithms and their Applications*, Proceedings of 2nd International Conference on Genetic Algorithms (Cambridge, MA, 1987) ed J.J. Grefenstette, pp 177-183
[104] THOMAS B., DAVID B.F., MICHALEWICZ Z., Evolutionary Computation 1, Basic Algorithms and Operators, Institute of Physics Publishing, Bristol and Philadelphia, ISBN 0 7503 0664 5, 2000
[105] THOMAS B., DAVID B.F., MICHALEWICZ Z., Evolutionary Computation 2, Advanced Algorithms and Operators, Institute of Physics Publishing, Bristol and Philadelphia, ISBN 0 7503 0665 3, 2000

[106] THOMPSON A., *Evolving electronic robot controllers that exploit hardware resources*, Proceedings of the 3rd European Conference on Artificial Life (Berlin: Springer), pp 640-656, 1995
[107] THOMPSON A., *Silicon Evolution*, Proceedings of 1st International Conference on Genetic Programming, Cambridge, MA, 1996, MIT Press
[108] TOACŞE G., NICULA D., *Electronica Digitala*, Editura Teora, Bucuresti, Romania, 1996
[109] TWARDOWSKI K., *An associative architecture for genetic algorithm-based machine learning computer*, vol. 27, Los Alamitos, CA: IEEE Computer Society, pp 27-38, 1994
[110] VALACHI A., ONOFREI V., MONOR C., COJOCARU D., - *"Proiectarea sistemelor digitale asincrone"*, Partea I - Ed. Spectrum Iasi, 2000 - Finantata de TEMPUS INCOT Jep 12132-97
[111] VALACHI A., HOZA F., ONOFREI V., SILION R., *Analiza, sinteza si testarea dispozitivelor numerice*, Ed. Nord-Est, 1993
[112] VALACHI A., BARSAN M., *Tehnici Numerice si automate*, Ed. Junimea Iasi, 1986
[113] VALENZUELA C.L. and WANG P.Y., *VLSI Placement and Area Optimization using a Genetic Algorithm to Breed Normalized Postfix Expressions*, IEEE Transactions On Evolutionary Computing, Vol. 6, No. 4, August 2002
[114] VILLA T., SANGIOVANNI-VINCENTELLI A., *„NOVA: State assignment of Finite State Machines for optimal two level logic imlpementation”*, IEEE Transactions, 1990, C-9, pp 905-924
[115] VILLA T., *Encoding problems in logic synthesis*, PhD. Thesis, University of California, Berkeley, May 1995
[116] VILLA T., GITANJALI S., SHIPLE T., *VIS User's Manual*, The VIS Group: University of California, Berkeley, University of Colorado, Boulder, Now at Lattice Semiconductor, 1996
[117] VOSE M.D., *Generalizing the notion of schema in genetic algorithms*, Artificial Intelligence, 50, pp. 385-396, 1991
[118] WHITLEY D., *A Genetic Algorithm Tutorial*, Technical Report, CS-93-103, Nov. 10, 1993, Dept. Of Computer Science, Colorado State University
[119] WOLFGANG T., *Applied Automata Theory*, 2003
[120] YANG S., CIESIELSKI M., *Optimum and suboptimum algorithms for input encoding and its relationsship to logic minimization*, IEEE Transactions on Computer Aided Design, January 1991
[121] YOSHIYUKI I., *On Synchronized Evolution of the Network of Automata*, IEEE Transactions On Evolutionary Computing, Vol. 6, No. 2, April 2002
[122] ZAHARIA C.N., CRISTEA A., *Genetic Algorithms and Neural Networks*, Editura Academiei Romane, 2002, ISBN 973-27-0889-1
[123] ZAHARIA M.H., LEON F., GALEA D., *"Parallel Genetic Algorithms for Cluster Load Balancing"*, Proceedings of the European Conference on Intelligent Systems and Technologies, Iasi, July 2004
[124] ZAFAR H. , Maciej J. Ciesielski , *Decomposition and Functional Verification of FSMs*, University of Massachusetts, Amherst
[125] ZEIGER H.P., *Loop-free synthesis of finite state machines*, PhD. Dissertation, Dept. Of Electrical Engineering, MIT, Cambridge, MA, September 1964
[126] ZITZLER E., LOTHAR T., "*Multiobjective evolutionary algorithms: a comparative case study and the Strength Pareto approach*." IEEE Transactions on Evolutionary Computation, vol.3, no.4, p.257-271 (November 1999)

Anexa A

Rezultatele în format Verilog a fişierelor top pentru masina prototip din Fig. 28 si Fig. 29, precum şi pentru masinile descompuse din Fig. 31 si Fig. 32, obținute cu programele ***genfsm***, ***fsmtool*** şi ***kiss2vl***, realizate de către autor, sunt descrise în continuare mai jos:

```
// Verilog file written by kiss2vl on Tue
Nov 02 16:32:12 2005
module shiftreg (reset, clock, in, out);
input  reset, clock;
input  in;
output out;
reg    out;
reg    [2:0] state, nextstate;
parameter st0 = 0,
        st4 = 1,
        st1 = 2,
        st2 = 3,
        st5 = 4,
        st3 = 5,
        st6 = 6,
        st7 = 7;

initial begin
   out = 1'b0;
   state = 3'b0;
   nextstate = 3'b0;
end

always @(posedge clock)
 if (reset)
  state = st0;
 else
  state = nextstate;

always @(state or in)
 begin
  out = 1'b0;
  nextstate = st0;
  case (state)
   st0: casex (in)
       1'b0: begin
               nextstate = st0;
               out = 1'b0;
             end
       1'b1: begin
               nextstate = st4;
               out = 1'b0;
             end
      endcase
   st1: casex (in)
       1'b0: begin
               nextstate = st0;
               out = 1'b1;
             end
       1'b1: begin
               nextstate = st4;
               out = 1'b1;
             end
      endcase
   st2: casex (in)
       1'b0: begin
               nextstate = st1;
               out = 1'b0;
             end
       1'b1: begin
               nextstate = st5;
               out = 1'b0;
             end
      endcase
   st3: casex (in)
       1'b0: begin
               nextstate = st1;
               out = 1'b1;
             end
       1'b1: begin
               nextstate = st5;
               out = 1'b1;
             end
      endcase
   st4: casex (in)
       1'b0: begin
               nextstate = st2;
               out = 1'b0;
             end
       1'b1: begin
               nextstate = st6;
               out = 1'b0;
             end
      endcase
   st5: casex (in)
       1'b0: begin
               nextstate = st2;
               out = 1'b1;
```

```
          end
     1'b1: begin
            nextstate = st6;
            out = 1'b1;
          end
    endcase
 st6: casex (in)
     1'b0: begin
            nextstate = st3;
            out = 1'b0;
          end
     1'b1: begin
            nextstate = st7;
            out = 1'b0;
          end
    endcase
 st7: casex (in)
     1'b0: begin
            nextstate = st3;
            out = 1'b1;
          end
     1'b1: begin
            nextstate = st7;
            out = 1'b1;
          end
    endcase
 endcase
end
endmodule
```

```
// Verilog file written by kiss2vl on Tue Nov  2 20:05:27 2005
module shiftreg1 (reset, clock, in, out);
input  reset, clock;
input  in;
output out;
reg    out;
reg    [1:0] state, nextstate;

parameter st0 = 0,
          st2 = 1,
          st1 = 2,
          st3 = 3;

always @(posedge clock)
 if (reset)
  state = st0;
 else
  state = nextstate;

always @(state or in)
 begin
  out = 1'b0;
  nextstate = st0;
  case (state)
   st0: casex (in)
       1'b000: begin
              nextstate = st0;
              out = 1'b0;
            end
       1'b100: begin
              nextstate = st2;
              out = 1'b0;
            end
       1'b001: begin
              nextstate = st0;
              out = 1'b1;
            end
       1'b101: begin
              nextstate = st2;
              out = 1'b1;
            end
      endcase
   st1: casex (in)
       1'b000: begin
              nextstate = st0;
              out = 1'b0;
            end
       1'b100: begin
              nextstate = st2;
              out = 1'b0;
            end
       1'b001: begin
              nextstate = st0;
              out = 1'b1;
            end
       1'b101: begin
              nextstate = st2;
              out = 1'b1;
            end
      endcase
   st2: casex (in)
       1'b010: begin
              nextstate = st1;
              out = 1'b0;
            end
       1'b110: begin
              nextstate = st3;
              out = 1'b0;
```

```
          end
        1'b011: begin
          nextstate = st1;
          out = 1'b1;
          end
        1'b111: begin
          nextstate = st3;
          out = 1'b1;
          end
       endcase
  st3: casex (in)
        1'b010: begin
          nextstate = st1;
          out = 1'b0;
          end
        1'b110: begin
          nextstate = st3;
          out = 1'b0;
          end
        1'b011: begin
          nextstate = st1;
          out = 1'b1;
          end
        1'b111: begin
          nextstate = st3;
          out = 1'b1;
          end
       endcase
  endcase
 end
endmodule
```

```
// Verilog file written by kiss2vl on Tue Nov  2 20:05:59 2005
module shiftreg2 (reset, clock, in, out);
input  reset, clock;
input  in;
output out;
reg    out;
reg    [1:0] state, nextstate;

parameter st0 = 0,
          st2 = 1,
          st1 = 2,
          st3 = 3;

always @(posedge clock)
 if (reset)
  state = st0;
 else
  state = nextstate;
always @(state or in)
 begin
  out = 1'b0;
  nextstate = st0;
  case (state)
   st0: casex (in)
        1'b000: begin
          nextstate = st0;
          out = 1'b0;
          end
        1'b100: begin
          nextstate = st2;
          out = 1'b0;
          end
        1'b001: begin
          nextstate = st1;
          out = 1'b0;
          end
        1'b101: begin
          nextstate = st3;
          out = 1'b0;
          end
       endcase
   st1: casex (in)
        1'b000: begin
          nextstate = st0;
          out = 1'b1;
          end
        1'b100: begin
          nextstate = st2;
          out = 1'b1;
          end
        1'b001: begin
          nextstate = st1;
          out = 1'b1;
          end
        1'b101: begin
          nextstate = st3;
          out = 1'b1;
          end
       endcase
   st2: casex (in)
        1'b010: begin
          nextstate = st0;
          out = 1'b0;
          end
        1'b110: begin
          nextstate = st2;
```

```
            out = 1'b0;
          end
      1'b011: begin
            nextstate = st1;
            out = 1'b0;
          end
      1'b111: begin
            nextstate = st3;
            out = 1'b0;
          end
     endcase
 st3: casex (in)
      1'b010: begin
            nextstate = st0;
            out = 1'b1;
          end
      1'b110: begin
            nextstate = st2;
            out = 1'b1;
          end
      1'b011: begin
            nextstate = st1;
            out = 1'b1;
          end
      1'b111: begin
            nextstate = st3;
            out = 1'b1;
          end
     endcase
 endcase
 end
endmodule
```

```
// Verilog file written by fsmtool on Tue Sep  2 17:56:47 2005
module top (reset, clk, in, out);
input  reset, clk;
input  in;
output out;
wire   [1:0] aux1;
wire   [1:0] aux2;
shiftreg1 Shiftreg1 (.reset(reset), .clock(clk),  .data({in,aux2}),  .state(aux1), .out(out));
shiftreg2 Shiftreg2 (.reset(reset), .clock(clk), .data({in,aux1}), .state(aux2));
endmodule
module shiftreg1 (reset, clock, data, state, out);
input  reset, clock;
input  [2:0] data;
output out;
output [1:0] state;
reg    out;
reg    [1:0] state, nextstate;
parameter st0 = 0,
          st2 = 2,
          st1 = 1,
          st3 = 3;
initial begin
    out = 1'b0;
    state = 2'b0;
    nextstate = 2'b0;
end
always @(posedge clock)
 if (reset)
  state = st0;
 else
  state = nextstate;
always @(state or data)
 begin
  out = 1'b0;
  nextstate = st0;
  case (state)
   st0: case (data)
        3'b000: begin
              nextstate = st0;
              out = 1'b0;
            end
        3'b100: begin
              nextstate = st2;
              out = 1'b0;
            end
        3'b001: begin
              nextstate = st0;
              out = 1'b1;
            end
        3'b101: begin
              nextstate = st2;
              out = 1'b1;
            end
       endcase
   st1: case (data)
        3'b000: begin
              nextstate = st0;
              out = 1'b0;
            end
        3'b100: begin
              nextstate = st2;
```

```
                out = 1'b0;
              end
        3'b001: begin
                nextstate = st0;
                out = 1'b1;
              end
        3'b101: begin
                nextstate = st2;
                out = 1'b1;
              end
      endcase
   st2: case (data)
        3'b010: begin
                nextstate = st1;
                out = 1'b0;
              end
        3'b110: begin
                nextstate = st3;
                out = 1'b0;
              end
        3'b011: begin
                nextstate = st1;
                out = 1'b1;
              end
        3'b111: begin
                nextstate = st3;
                out = 1'b1;
              end
      endcase
   st3: case (data)
        3'b010: begin
                nextstate = st1;
                out = 1'b0;
              end
        3'b110: begin
                nextstate = st3;
                out = 1'b0;
              end
        3'b011: begin
                nextstate = st1;
                out = 1'b1;
              end
        3'b111: begin
                nextstate = st3;
                out = 1'b1;
              end
      endcase
  endcase
 end
endmodule
```

```
module shiftreg2 (reset, clock, data, state);
input  reset, clock;
input  [2:0] data;
output [1:0] state;
reg    [1:0] state, nextstate;
parameter st0 = 0,
          st2 = 2,
          st1 = 1,
          st3 = 3;
initial begin
    state = 2'b0;
    nextstate = 2'b0;
end
always @(posedge clock)
 if (reset)
  state = st0;
 else
  state = nextstate;
always @(state or data)
 begin
  nextstate = st0;
  case (state)
   st0: case (data)
        3'b000: nextstate = st0;
        3'b100: nextstate = st2;
        3'b001: nextstate = st1;
        3'b101: nextstate = st3;
       endcase
   st1: case (data)
        3'b000: nextstate = st0;
        3'b100: nextstate = st2;
        3'b001: nextstate = st1;
        3'b101: nextstate = st3;
       endcase
   st2: case (data)
        3'b010: nextstate = st0;
        3'b110: nextstate = st2;
        3'b011: nextstate = st1;
        3'b111: nextstate = st3;
       endcase
   st3: case (data)
        3'b010: nextstate = st0;
        3'b110: nextstate = st2;
        3'b011: nextstate = st1;
        3'b111: nextstate = st3;
       endcase
  endcase
 end
endmodule
```