UNIVERSITATEA TEHNICĂ din CLUJ-NAPOCA
FACULTATEA de AUTOMATICĂ şi CALCULATOARE
CATEDRA de CALCULATOARE

Metode de descriere a sistemelor numerice

Referat de doctorat

Conducător ştiinţific,
Prof. Dr. Ing. PUSZTAI Kalman

Doctorand,
ş.l. ing. BARUCH Zoltan

Cuprins

1. Introducere 3

1.1. Necesitatea descrierilor de nivel înalt 3

1.2. Nivele şi domenii de descriere 4

2. Modele hardware 9

2.1. Introducere 9

2.2. Clasificarea modelelor 11

2.3. Modele orientate pe stare 12
2.3.1. Automate cu stări finite 12
2.3.2. Reţele Petri 14
2.3.3. Automate cu stări finite ierarhice şi concurente 16

2.4. Modele orientate pe activitate 17
2.4.1. Grafuri ale fluxului de date 17
2.4.2. Grafuri ale fluxului de control 19

2.5. Modele orientate pe structură 19
2.5.1. Diagrame de conexiune a componentelor 19
2.5.2. Structuri de incidenţă 21
2.5.3. Reţele logice 22

2.6. Modele eterogene 24
2.6.1. Grafuri ale fluxului de control şi de date 24
2.6.2. Diagrame de structură 26
2.6.3. Automate cu stări ale programului 27
2.6.4. Modelul firelor de aşteptare 29

3. Limbaje de descriere hardware 30

3.1. Introducere 30

3.2. Tipuri de limbaje de descriere hardware 32
3.2.1. Limbaje de descriere structurale 32
3.2.2. Limbaje de descriere funcţionale 34

3.3. Caracteristici ale limbajelor de descriere hardware 37
3.3.1. Caracteristici specifice limbajelor de programare 37
3.3.1.1. Tipuri de date 37
3.3.1.2. Operatori şi instrucţiuni de asignare 37
3.3.1.3. Construcţii de control 37
3.3.1.4. Ordinea de execuţie 38
3.3.2. Caracteristici specifice unităţilor hardware 39
3.3.2.1. Definirea interfeţelor 39
3.3.2.2. Declaraţii structurale 39
3.3.2.3. Operatori la nivelul RT şi logic 39
3.3.2.4. Asincronismul 40
3.3.2.5. Ierarhia 41
3.3.2.6. Comunicaţia între procese 43
3.3.2.7. Restricţii 44
3.3.2.8. Alocarea de către utilizator 45
3.4. Formate ale limbajelor de descriere hardware 45
3.4.1. Limbaje textuale 46
3.4.2. Limbaje grafice 46
3.4.3. Limbaje tabelare 46
3.4.4. Limbaje bazate pe diagrame de timp 48
3.5. Exemple de limbaje de descriere 48
3.5.1. VHDL 48
3.5.2. HardwareC 56
3.5.3. CSP 57
3.5.4. Verilog 59
3.5.5. Statecharts 59
3.5.6. Silage 61
3.5.7. SpecCharts 62
3.6. Relaţia dintre limbaj şi arhitectură 67

4. Concluzii 74

Bibliografie 77

1. Introducere

1.1. Necesitatea descrierilor de nivel înalt

Dezvoltarea tehnologică rapidă din ultimii ani a permis proiectarea şi fabricaţia sistemelor digitale din ce în ce mai complexe. Acest lucru nu a fost posibil fără progresul în domeniul metodologiilor de proiectare şi al sistemelor de proiectare automată care asistă proiectantul în vederea aplicării acestor metodologii. În timp ce primele sisteme de proiectare asistată au fost realizate pentru verificarea proiectării (simularea logică şi verificarea amplasării şi rutării), apoi şi pentru amplasarea şi rutarea automată (pe baza unei liste de conexiuni a blocurilor funcţionale şi a unei biblioteci de celule predefinite), în ultimii ani se poate observa o dezvoltare rapidă a sistemelor de proiectare asistată, care permit specificarea sau descrierea unui sistem la diferite nivele de abstractizare în domeniul funcţional şi implementarea automată a proiectului pornind de la această descriere.

Prin verificarea automată a proiectelor şi realizarea automată a proiectării fizice s-a obţinut o reducere semnificativă a efortului de proiectare. Utilizarea mediilor de proiectare pentru specificarea sistemelor prin scheme electrice, aplicarea simulării pentru verificarea descrierilor la nivelul porţilor logice, plasarea şi rutarea automată pe baza listelor de conexiuni au determinat reducerea efortului de proiectare cu aproximativ 50%. O reducere suplimentară s-a obţinut prin posibilitatea proiectării ierarhice şi prin existenţa generatoarelor de module, de exemplu a generatoarelor pentru reţelele logice programabile. Următoarea etapă importantă a constat din introducerea tehnicilor de sinteză şi optimizare la nivel logic, care permit, de exemplu, translatarea automată a tabelelor de adevăr în reţele minimizate de porţi logice. O altă reducere semnificativă a efortului şi a duratei de proiectare se poate obţine prin utilizarea unor metode şi sisteme de sinteză care transformă o descriere de nivel înalt – sau chiar o specificaţie – într-o implementare adecvată.

Pe măsură ce complexitatea sistemelor numerice creşte, apare necesitatea unor sisteme pentru proiectarea automată la nivele de abstractizare mai înalte. Abstractizarea de nivel mai înalt va determina o reducere suplimentară a costului de proiectare. Pentru reducerea duratei de proiectare sunt necesare implementări care sunt corecte pentru prima dată, eliminându-se ciclurile de modificări şi reproiectări. Sunt necesare tehnici de sinteză care permit realizarea *corectitudinii prin construcţie*, şi chiar dacă proiectantul trebuie să ia parte la procesul de proiectare, nivelul mai înalt de abstractizare va predispune la mai puţine erori decât în cazul existenţei unor detalii de nivel mai redus. Pentru această abordare, sistemele CAD trebuie să permită verificarea atât a funcţionării cât şi a regulilor de proiectare.

Un alt concept care permite reducerea ciclului de proiectare este cel al *primei specificaţii*, care are ca scop reducerea numărului de iteraţii pentru specificarea unui produs la una

singură. Metodologia de specificaţie cu o singură iteraţie necesită modelarea cu acurateţe a procesului de proiectare şi estimarea corectă a unor indicatori de calitate, de exemplu cei de performanţă şi cost.

Pe lângă reducerea duratei ciclului de proiectare, abstractizarea de nivel înalt şi automatizarea unei părţi sau a întregului proces de proiectare are şi alte avantaje. Astfel, este posibilă explorarea mai completă a diferitelor metode de proiectare, deoarece proiectele pot fi generate şi evaluate într-un timp redus. De asemenea, dacă algoritmii de sinteză sunt performanţi, sistemele de proiectare automată pot depăşi proiectanţii de nivel mediu în ceea ce priveşte generarea proiectelor de calitate. Totuşi, verificarea corectitudinii acestor algoritmi, şi a sistemelor CAD în general, nu este o sarcină uşoară. Sistemele CAD nu pot asigura încă o calitate comparabilă cu cea a proiectantului uman pentru întregul proces de proiectare. Impedimentele principale sunt complexitatea proiectelor, care necesită o căutare eficientă în spaţiul de proiectare, şi un model detaliat, care necesită algoritmi sofisticaţi capabili să satisfacă obiective şi restricţii multiple.

Din controversa asupra soluţiilor la cele două probleme au rezultat două concepţii diferite. Adepţii primeia consideră că ierarhia proiectelor este realizată de jos în sus, din componente elementare ca tranzistoare şi porţi. Astfel, sunt necesare sisteme CAD care reţin diferite aspecte ale proiectului şi le verifică în principal prin simulare. Adepţii celei de-a doua concepţii consideră că o metodologie 'top-down', în care proiectanţii descriu proiectul, iar sistemele CAD adaugă structura electrică şi fizică detaliată, este mai potrivită pentru proiectarea viitoare a sistemelor complexe. A doua metodă se concentrează asupra definiţiei limbajelor de descriere, a modelelor proiectelor, a algoritmilor de sinteză şi a mediilor de proiectare pentru sinteza interactivă, în care intuiţia proiectanţilor poate reduce substanţial căutarea în spaţiul de proiectare.

Ambele concepţii pot fi corecte în anumite momente ale evoluţiei tehnologice. De exemplu, poate fi avantajoasă proiectarea manuală a unei celule de memorie care este multiplicată de milioane de ori. Totuşi, în acest moment al dezvoltării tehnologice, chiar sinteza care nu este optimă devine mai eficientă din punct de vedere al costului decât proiectarea manuală.

1.2. Nivele şi domenii de descriere

Pentru definirea diferitelor nivele de descriere a sistemelor numerice, *diagrama Y*, introdusă pentru prima dată de *Gajski* şi *Kuhn*, este cea mai des utilizată. Ideea de bază este că fiecare element al unui sistem numeric poate fi descris în cadrul a trei domenii diferite. Acestea sunt: *domeniul funcţional*, *domeniul structural* şi *domeniul fizic/geometric*.

În *domeniul funcţional*, se specifică funcţionarea (propusă) a sistemului, în mod ideal fără nici o referire la modul în care această funcţionare este asigurată prin implementare. Sistemul este descris ca având un set specificat de intrări şi ieşiri, şi un set de funcţii care descriu relaţia în timp dintre ieşiri şi intrări. O descriere funcţională mai cuprinde o descriere a interfeţei şi o descriere a restricţiilor impuse asupra sistemului. Descrierea interfeţei specifică porturile de I/E şi protocoalele sau relaţiile în timp între semnalele acestor porturi. Restricţiile specifică relaţiile tehnologice care trebuie să fie respectate pentru ca sistemul proiectat să poată fi verificat, testat, fabricat şi întreţinut.

În *domeniul structural*, sistemul este considerat ca o ierarhie de elemente funcţionale interconectate, ţinând cont de restricţii ca timpi de întârziere, spaţiu ocupat sau cost. O reprezentare structurală specifică implementarea sistemului, şi chiar dacă funcţionalitatea acestuia poate fi dedusă din componentele interconectate, reprezentarea structurală nu descrie funcţionalitatea în mod explicit. Uneori, o reprezentare structurală, ca de exemplu o schemă logică, poate servi ca o descriere funcţională. Pe de altă parte, anumite descrieri funcţionale, ca expresiile booleene, pot sugera o implementare, ca de exemplu o structură sub forma unor sume de produse.

În *domeniul fizic*, se detaliază implementarea sistemului, specificând caracteristicile fizice ale componentelor descrise în reprezentarea structurală. De exemplu, o reprezentare fizică poate indica dimensiunile şi poziţia fiecărei componente, ca şi caracteristicile fizice ale conexiunilor între acestea. Astfel, în timp ce reprezentarea structurală specifică conexiunile între elementele sistemului, reprezentarea fizică descrie relaţiile spaţiale dintre aceste elemente interconectate, şi alte caracteristici ca puterea consumată, căldura disipată, poziţia pinilor de intrare şi de ieşire.

Aceste domenii sunt reprezentate prin cele trei axe sub forma literei *Y* (**Figura 1**). În cadrul fiecărui domeniu, elementele pot fi descrise la diferite nivele de abstractizare. Aceste nivele sunt reprezentate ca puncte de-a lungul axelor, cu nivelele mai înalte (mai abstracte) aflate la periferia diagramei, şi nivelele mai joase aflate în apropierea centrului acesteia.

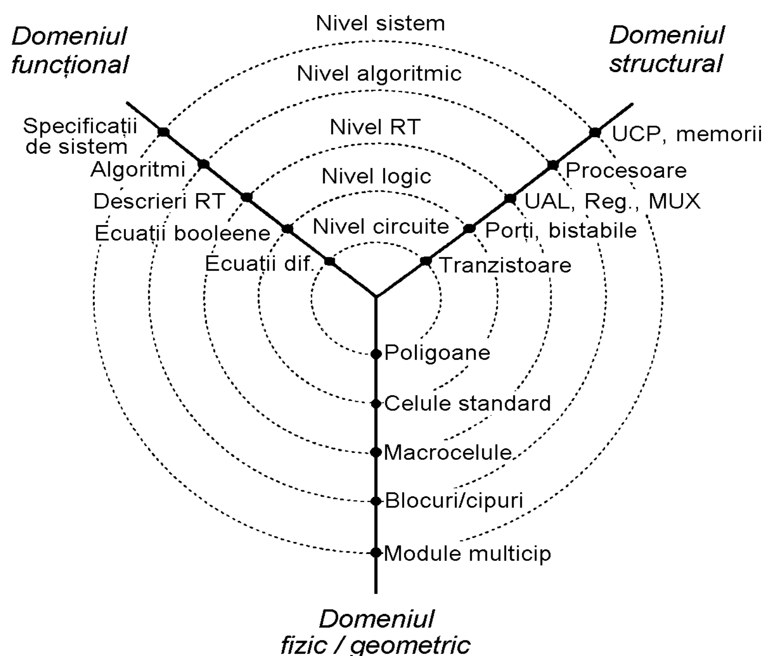


Figura 1. Domenii şi nivele de descriere.

Nivelul cel mai puţin abstract în cadrul ierarhiei este *nivelul circuitelor*. Elementele structurale care aparţin acestui nivel sunt tranzistoare, condensatoare, rezistenţe. În domeniul funcţional, tranzistorul este descris prin ecuaţii diferenţiale care specifică dependenţele între

tensiunile aplicate la terminalele sale şi curenţii care rezultă la aceste terminale. În domeniul structural, tranzistorul este utilizat pentru a construi elemente funcţionale mai complexe, de exemplu porţi logice. O poartă logică este descrisă prin interconexiunile între tranzistoare; tranzistorul este considerat ca un set de terminale, conexiunile sale interne fiind exprimate prin simbolul tranzistorului. În domeniul fizic/geometric, tranzistorul este reprezentat ca o celulă, printr-un set de poligoane sau dreptunghiuri care specifică regiunile din cadrul circuitului integrat cărora trebuie să li se aplice procesele specifice în timpul fabricaţiei (difuzie, metalizare etc.) pentru a se obţine funcţionarea descrisă în cadrul domeniului funcţional.

Următorul nivel de abstractizare este *nivelul logic*, cel al ecuaţiilor booleene şi al diagramelor de stare, în care nivelele de tensiuni sunt abstractizate cu valorile logice "1" şi "0". Elementele structurale de la acest nivel sunt porţile şi bistabilele, iar cele fizice sunt reprezentate de celule şi subcelule standard.

Următorul nivel este cel al operaţiilor asupra seturilor de valori logice care sunt grupate în cuvinte, fiind interpretate ca valori numerice asupra cărora se aplică operaţii aritmetice şi logice. Componentele principale ale acestui nivel sunt unităţile aritmetice şi de memorare realizate cu porţi şi bistabile, ca de exemplu sumatoare, circuite de multiplicare, comparatoare, numărătoare, registre, buffere de date. Deoarece activităţile din cadrul sistemului sunt descrise, la acest nivel de abstractizare, cel mai adesea prin transferul valorilor între registre, acest nivel se numeşte cel al *transferurilor între registre* (*nivel RT*), fiind numit şi *nivel microarhitectural*. Pentru descriere se mai pot utiliza diagrame, automate cu stări finite sau tabele de stare.

Pentru a descrie funcţionarea unui subsistem sau sistem, deci pentru a descrie modul în care sistemul prelucrează valorile de intrare aplicate la porturile de intrare pentru a obţine valorile de ieşire cerute, operaţiile sunt grupate în algoritmi, de unde denumirea de *nivel algoritmic*. Elementele de bază de la acest nivel sunt procesoarele, memoriile, controlerele, interfeţele şi circuitele integrate specifice aplicaţiilor (*ASIC*).

În multe cazuri, un sistem complex este compus din subsisteme. Fiecare subsistem poate fi descris printr-un proces secvenţial (deşi implementarea poate utiliza paralelismul); aceste procese se execută concurent, utilizând diferite metode de comunicare pentru interacţiunea dintre ele. Acest nivel de abstractizare se numeşte *nivel sistem*. La acest nivel elementele structurale cu care se operează sunt unităţile centrale de prelucrare, sistemele de memorie şi magistralele. Sistemele se pot descrie la nivel funcţional utilizând diferite metode, printr-un limbaj natural, un limbaj de descriere hardware sau limbaj de programare.

Elementele de proiectare care se utilizează în diferitele nivele de abstractizare sunt indicate în **Tabelul 1**.

Până acum în cadrul domeniului funcţional s-au considerat numai diferitele nivele de abstractizare care se referă la date. Există nivele de abstractizare similare care se referă la timp şi la operaţiile de control. Timpul este considerat ca fiind continuu la nivelul circuit, discret (sub forma ciclurilor de ceas) de la nivelul logic la cel algoritmic, la nivelul sistem utilizându-se modele mai abstracte ale timpului, de exemplu cicluri de operaţii.

La nivelul circuit nu există noţiunea controlului. La nivelul logic controlul există, dar nu este distins în mod clar faţă de alte activităţi din cadrul circuitului. La nivelul transferurilor între registre există stări abstracte de control; pentru fiecare stare de control se specifică condiţia care trebuie testată, transferurile între registre care trebuie executate şi următoarea stare de control în care se trece. La nivelul algoritmic există structuri de control, de exemplu bucle,

ramificații, apeluri de proceduri. În final, la nivelul sistem controlul ia forma sincronizării şi comunicării între procese.

Nivel	Reprezentare funcțională	Reprezentare structurală	Reprezentare fizică
Sistem	Specificații de sistem Procese Limbaje de descriere	Unități centrale Sisteme de memorie Magistrale	Module multicip Plăci de circuite imprimate
Algoritmic	Algoritmi Seturi de instrucțiuni	Procesoare Controlere Subsisteme	Blocuri Cipuri
Transferuri între registre (RT)	Descrieri RT Automate cu stări finite	UAL Registre Multiplexoare	Macro-celule Scheme de amplasare
Logic	Ecuații booleene Diagrame de stare	Porți Bistabile	Celule standard Subcelule Module
Circuit	Ecuații diferențiale Diagrame curent-tensiune	Tranzistoare Condensatoare Rezistențe	Poligoane Contacte Trasee

Tabelul 1. Nivele de abstractizare şi tipuri de reprezentare.

Diagrama Y poate fi utilizată şi pentru definirea sau descrierea diferitelor etape de proiectare. Acestea pot fi exprimate ca tranziții între punctele de pe axele diagramei, fiind reprezentate grafic ca arce direcționate. De exemplu, tranziția de la punctul care reprezintă un subsistem de pe axa domeniului structural la punctul reprezentând un modul sau cip de pe axa domeniului fizic corespunde etapei de plasare şi rutare a macro-celulelor, în care, pe baza unei liste de conexiuni a blocurilor funcționale, se realizează amplasarea macro-celulelor şi interconectarea dintre acestea. Reconstruirea unei rețele de tranzistoare (domeniul structural) din poligoanele corespunzătoare domeniului fizic se realizează cu un program de extragere. Recunoaşterea dispozitivelor şi extragerea parametrilor corespunde unei tranziții de la domeniul structural la cel funcțional.

Pe baza acestor tranziții, se pot defini termenii de *generare*, *extragere*, *sinteză* şi *analiză* (**Figura 2**). Tranziția de la domeniul structural la cel fizic se numeşte *generare*, iar tranziția inversă se numeşte *extragere*. Tranziția de la domeniul funcțional la cel structural se numeşte *sinteză*, iar tranziția inversă se numeşte *analiză*.

Pentru deplasarea de-a lungul unei anumite axe a diagramei *Y* se utilizează următoarele definiții. O etapă reprezentată de un arc direcționat spre centrul diagramei se numeşte *rafinare*, iar o etapă reprezentată de un arc direcționat invers se numeşte *abstractizare*. Un arc care reprezintă o buclă simbolizează o transformare în cadrul unui domeniu sau nivel, numită *optimizare*. În cadrul optimizării, funcționalitatea de bază rămâne nemodificată, dar calitatea proiectului, exprimată printr-o funcție obiectivă (care ține cont de performanțe, spațiu ocupat, putere consumată etc.), este îmbunătățită.

În cele mai multe cazuri etapele de proiectare generează două tipuri de rezultate. Funcționarea unui sistem specificat la un anumit nivel de abstractizare este transformată într-o descriere structurală la acelaşi nivel. Aceasta reprezintă componenta etapei de proiectare în privința sintezei. Descrierea structurală generată astfel se referă la obiecte abstracte descrise

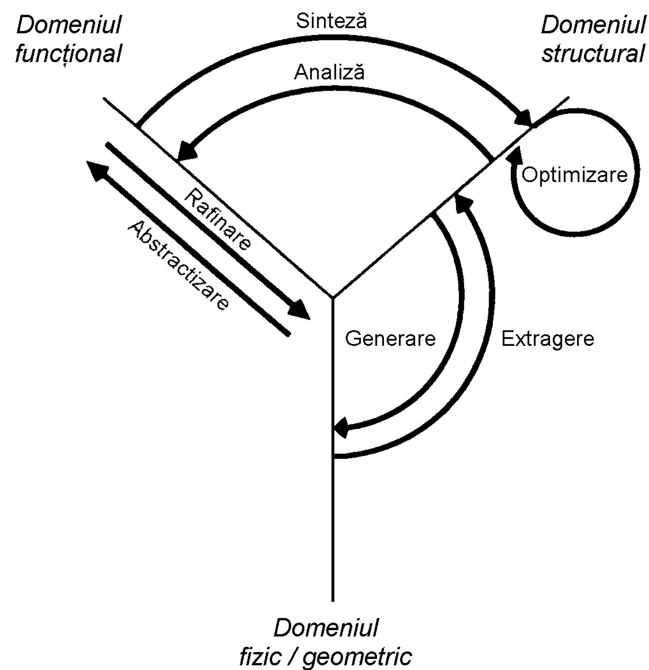


Figura 2. Tranziții în cadrul diagramei Y.

în următorul nivel de abstractizare. Pentru aceste obiecte de nivel inferior funcționarea este fie definită, fie implicită. Aceasta reprezintă componenta etapei de proiectare în privința rafinării. Descrierile funcționale ale obiectelor de nivel inferior reprezintă punctul de plecare pentru următoarea etapă de sinteză.

De cele mai multe ori, descrierile sistemelor conțin atât elemente funcționale cât şi structurale. De aceea, definiția etapelor de proiectare ca tranziții în cadrul diagramei *Y* poate deveni dificilă. Există definiții bazate pe observația că fiecare etapă în cadrul procesului de proiectare adaugă anumite informații la descrierea globală a sistemului.

2. Modele hardware

2.1. Introducere

Proiectarea unui sistem este procesul de implementare a unei specificații de funcționare a acestuia utilizând un set de componente fizice. Prima etapă în proiectarea unui sistem o reprezintă deci specificarea funcționării dorite. Pentru această specificare se pot utiliza diferite modele conceptuale.

Un *model* al unui circuit sau sistem este o abstractizare a acestuia, deci o reprezentare care pune în evidență caracteristicile sale relevante, fără detaliile asociate. Modelele sunt utilizate pentru specificarea circuitelor sau sistemelor, pentru efectuarea raționamentelor asupra proprietăților acestora sau pentru transferul informațiilor între proiectanți sau între aceştia şi sistemele de proiectare asistată de calculator.

Modelele informale, de exemplu descrierile textuale ale principiilor de funcționare ale unui sistem într-un limbaj natural, au o aplicabilitate limitată atunci când sunt utilizate sistemele CAD. În plus, aceste descrieri informale ale circuitelor sau sistemelor complexe pot fi surse de ambiguități, deoarece adesea este imposibil să se testeze completitudinea şi consistența lor. În schimb, modelele formale au o sintaxă şi semantică bine definită, de aceea acestea asigură un mijloc de a exprima informațiile despre un sistem într-un mod care poate fi interpretat neambiguu. Astfel se pot realiza sisteme automate pentru citirea, procesarea şi scrierea unor asemenea modele.

Pentru a fi util, un model trebuie să posede anumite calități. În primul rând, modelul nu trebuie să conțină ambiguități. De asemenea, trebuie să fie complet, pentru a putea descrie întregul sistem. În plus, modelul trebuie să fie inteligibil pentru proiectanți, şi să poată fi modificat cu uşurință. În fine, un model trebuie să fie suficient de natural pentru a uşura înțelegerea sistemului de către proiectanți.

Un model este un sistem formal constând din obiecte şi reguli de compoziție. Scopul unui model este de a furniza o viziune abstractă asupra sistemului. În **Figura 3** se exemplifică două modele diferite ale unui controler pentru un ascensor, pentru care se prezintă şi descrierea în limbaj natural. În **Figura 3(b)** controlerul este reprezentat ca un set de instrucțiuni, iar în **Figura 3(c)** este reprezentat ca o maşină de stare.

Fiecare din aceste modele reprezintă un set de obiecte şi interacțiunile dintre ele. Modelul maşinii de stare, de exemplu, constă dintr-un set de stări şi tranziții dintre aceste stări. Modelul algoritmic constă dintr-un set de instrucțiuni. Avantajul de a avea la dispoziție aceste modele diferite este că ele permit reprezentarea diferitelor vederi asupra unui sistem, punându-se în evidență diferitele sale caracteristici. De exemplu, modelul maşinii de stare este adecvat pentru a reprezenta comportarea în timp a unui sistem, deoarece permite expri-

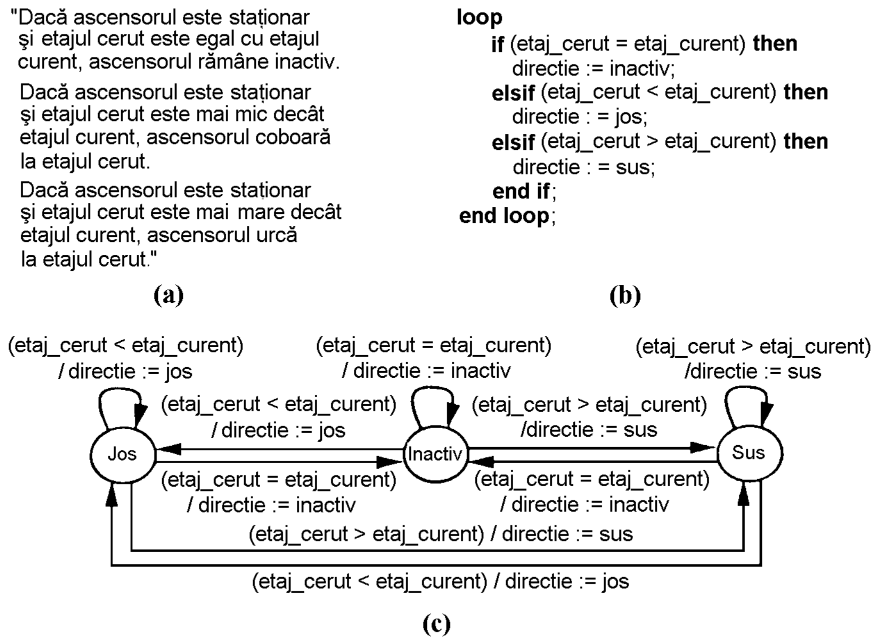


Figura 3. Modele conceptuale ale unui controler pentru ascensor: (a) descriere în limbaj natural; (b) model algoritmic; (c) mașină de stare.

marea explicită a stărilor și a tranzițiilor între stări determinate de evenimente externe sau interne. Modelul algoritmic, pe de altă parte, nu are stări explicite. Deoarece acest model poate specifica relațiile dintre intrări și ieșiri în funcție de o secvență de instrucțiuni, este adecvat pentru a reprezenta vederea procedurală asupra sistemului.

Proiectanții aleg diferite modele în diferite faze ale procesului de proiectare, pentru a pune în evidență acele aspecte ale sistemului care sunt de interes la momentul respectiv. De asemenea, sunt necesare modele diferite și pentru domenii diferite de aplicații. De exemplu, sistemele în timp real și sistemele pentru baze de date sunt modelate în mod diferit, deoarece primele se concentrează asupra comportării în timp, iar cele din urmă se concentrează asupra organizării datelor.

După ce proiectantul a ales un model corespunzător pentru a specifica funcționarea unui sistem, poate descrie în detaliu funcționarea acestuia. Procesul de proiectare nu este însă complet, deoarece un asemenea model nu descrie exact modul în care sistemul trebuie realizat. Următoarea etapă este deci *transformarea modelului într-o arhitectură*, care definește implementarea modelului prin specificarea numărului și a tipului componentelor, ca și a interconexiunilor dintre ele. În **Figura 4** se indică două arhitecturi diferite care se pot utiliza pentru implementarea modelului mașinii de stare a controlerului pentru ascensor care a fost prezentat anterior. Arhitectura din **Figura 4(a)** este o implementare la nivelul registrelor, care utilizează un registru de stare pentru a păstra starea curentă și o logică combinațională pentru a implementa tranzițiile dintre stări și valorile semnalelor de ieșire. În **Figura 4(b)** se indică o implementare la nivel de sistem, în care mașina de stare este implementată prin software, utilizând o variabilă dintr-un program pentru a reprezenta starea curentă și instrucțiuni ale programului pentru a calcula tranzițiile stărilor și valorile semnalelor de ieșire.

Anumite arhitecturi sunt mai eficiente pentru implementarea anumitor modele. În plus, tehnologia de proiectare și de fabricație au o mare influență în alegerea unei arhitecturi.

De aceea, proiectanții trebuie să ia în considerare mai multe alternative diferite de implementare în cadrul procesului de proiectare.

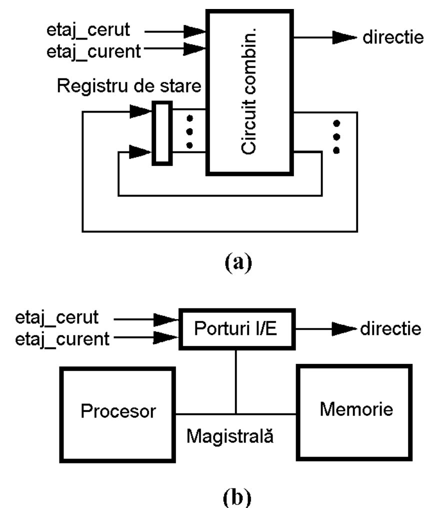


Figura 4. Arhitecturi utilizate pentru: (a) implementarea la nivelul registrelor; (b) implementarea la nivel de sistem.

Anumite arhitecturi sunt mai eficiente pentru implementarea anumitor modele. În plus, tehnologia de proiectare şi de fabricație au o mare influență în alegerea unei arhitecturi. De aceea, proiectanții trebuie să ia în considerare mai multe alternative diferite de implementare în cadrul procesului de proiectare.

Un circuit sau sistem poate fi modelat în moduri diferite în funcție de nivelul de abstractizare dorit (de exemplu, *funcțional*, *structural*, *fizic*) şi de metodele de modelare utilizate (de exemplu, *limbaje*, *diagrame*, *modele abstracte*). În ultimii ani, a crescut tendința de a utiliza limbaje de descriere hardware pentru specificarea circuitelor. Concizia acestor modele determină ca ele să fie preferabile față de diagramele logice, de stare şi cele pentru fluxul de date şi de control, chiar dacă unele modele bazate pe diagrame sunt mai puternice pentru vizualizarea funcțiilor circuitelor.

Modelele abstracte sunt modele matematice bazate pe grafuri şi algebra booleană. La nivel de sistem, funcționarea poate fi abstractizată printr-un set de operații şi dependențele dintre ele. În domeniul funcțional, un circuit secvențial este abstractizat printr-o maşină cu stări finite, care se reduce la o funcție booleană în cazul combinațional. În domeniul structural, abstractizarea se realizează sub forma unor interconexiuni între porți logice, blocuri logice sau subsisteme.

Modelele abstracte sunt suficient de puternice pentru a reține caracteristicile esențiale descrise de modelele bazate pe limbaje şi diagrame. În acelaşi timp, ele sunt suficient de simple pentru ca proprietățile transformărilor efectuate asupra circuitelor să poată fi demonstrate.

2.2. Clasificarea modelelor

În general, modelele utilizate în cadrul sistemelor de proiectare asistată de calculator se încadrează în cinci categorii:

- modele orientate pe stare
- modele orientate pe activitate
- modele orientate pe structură
- modele orientate pe date
- modele eterogene

Un *model orientat pe stare*, ca de exemplu un automat cu stări finite, reprezintă sistemul ca un set de stări şi un set de tranziții între acestea, care sunt activate de evenimente externe. Un asemenea model este cel mai potrivit pentru sistemele de control, cum sunt sistemele în timp real, la care comportarea în timp este aspectul cel mai important al proiectului.

Un *model orientat pe activitate*, ca de exemplu un graf al fluxului de date, descrie sistemul ca un set de activități legate prin dependențe de date sau de execuție. Acest model este aplicabil mai ales pentru sistemele transformaționale, cum sunt sistemele de prelucrare a semnalelor numerice, la care datele trec printr-un set de transformări cu o rată fixă.

Utilizând un *model orientat pe structură*, cum este o schemă bloc, se pot descrie modulele fizice ale sistemului şi interconexiunile dintre ele. Spre deosebire de modelele orientate pe stare şi cele orientate pe activitate, care reflectă în principal funcționarea sistemului, un model orientat pe structură se concentrează în principal asupra structurii fizice a acestuia.

Un *model orientat pe date* se va utiliza atunci când este necesar să se reprezinte sistemul ca o colecție de date între care există relații bazate pe atributele acestora, pe proprietatea de membru al unei clase etc. Acest tip de model este adecvat pentru sisteme informaționale, ca bazele de date, unde funcția sistemului este mai puțin importantă decât organizarea datelor.

În fine, proiectanții pot utiliza un *model eterogen*, care integrează mai multe caracteristici ale modelelor anterioare, în cazul unui sistem complex.

2.3. Modele orientate pe stare

2.3.1. Automate cu stări finite

Automatul cu stări finite (ASF) este modelul cel mai utilizat pentru descrierea sistemelor de control, comportarea temporală a acestor sisteme fiind reprezentată sub forma stărilor şi a tranzițiilor dintre stări.

Un model ASF constă dintr-un set de *stări*, un set de *tranziții* între stări, şi un set de *acțiuni* asociate cu aceste stări sau tranziții. Poate fi descris prin cvintuplul:

$$< S, I, O, f: S \times I \rightarrow S,\ h: S \times I \rightarrow O >$$

unde $S = \{s_1, s_2, \ldots, s_l\}$ este un set de stări, $I = \{i_1, i_2, \ldots, i_m\}$ este un set de intrări, iar $O = \{o_1, o_2, \ldots, o_n\}$ este un set de ieşiri; f este funcția stării următoare, iar h este funcția de ieşire. Fiecare ASF are o stare inițială şi un set de stări finale.

În **Figura 5** se prezintă un ASF care modelează un controler pentru ascensor într-o clădire cu trei etaje. În acest model, setul de intrări $I = \{r1, r2, r3\}$ reprezintă etajul cerut. De exemplu, *r2* înseamnă că este cerut etajul 2. Setul de ieşiri $O = \{d1, d2, n, u1, u2\}$ reprezintă direcția în care trebuie să se deplaseze şi numărul de etaje cu care trebuie să se deplaseze ascensorul. De exemplu, *d2* înseamnă că ascensorul trebuie să se deplaseze în jos cu 2 etaje, *u2* înseamnă că trebuie să se deplaseze în sus cu 2 etaje, iar *n* înseamnă că ascensorul trebuie să rămână inactiv. Din figură se observă că dacă etajul curent este 2 (deci dacă starea curentă este S_2), şi este cerut etajul 1, ieşirea va fi *d1*.

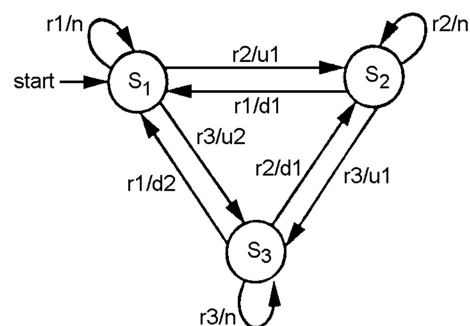


Figura 5. Modelul ASF al controlerului pentru ascensor.

Există două tipuri de modele ASF des utilizate: *bazate pe tranziţii* (Mealy) şi *bazate pe stări* (Moore), care diferă în principal prin definiţia funcţiei de ieşire h. Într-un model ASF bazat pe tranziţii, valorile ieşirilor depind de stări şi de intrări ($h: S \times I \rightarrow O$); într-un model ASF bazat pe stări, valorile ieşirilor depind numai de stări ($h: S \rightarrow O$). În **Figura 5** s-a utilizat modelul bazat pe tranziţii pentru modelarea controlerului pentru ascensor. Modelul bazat pe stări pentru acelaşi controler este prezentat în **Figura 6**, în care se indică în fiecare stare valoarea ieşirii.

Modelul ASF bazat pe stări poate necesita un număr mai mare de stări decât modelul bazat pe tranziţii. Aceasta deoarece la modelul bazat pe tranziţii pot exista arce multiple care indică la o singură stare, fiecare arc având o valoare diferită a ieşirii; la modelul bazat pe stări fiecare valoare diferită de ieşire necesită o stare proprie (**Figura 6**).

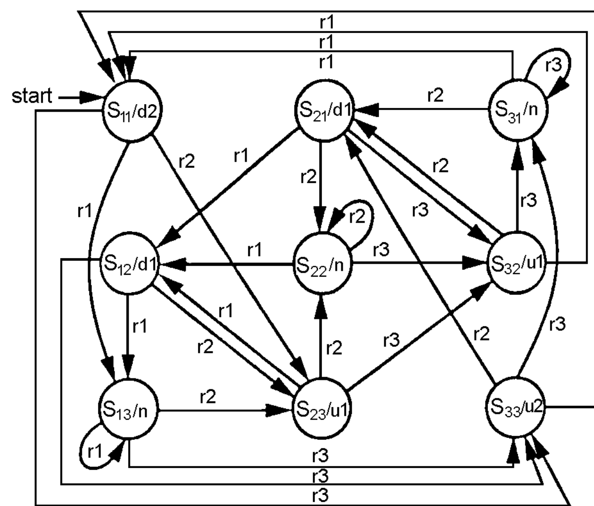


Figura 6. Modelul ASF bazat pe stări al controlerului pentru ascensor.

În cazurile în care un model ASF trebuie să reprezinte numere întregi sau flotante, dacă fiecare valoare posibilă a numerelor necesită câte o stare, modelul va necesita un număr foarte mare de stări. De exemplu, un întreg de 16 biţi poate reprezenta 2^{16} sau 65536 stări. Există o cale simplă de a reduce numărul de stări, fiind posibilă extensia unui model ASF cu variabile întregi sau flotante. Introducerea unei variabile de 16 biţi, de exemplu, va reduce numărul de stări cu 65536.

Acest tip de model ASF extins se numeşte *model ASF cu cale de date* (ASFD), fiind specificat în continuare. Se defineşte un set de variabile de memorare *VAR*, un set de expresii

$$EXP = \{ f(x, y, z, ...) \mid x, y, z, ... \in VAR \} ,$$

şi un set de asignări

$$A = \{ X \Leftarrow e \mid X \in VAR, e \in EXP \}.$$

În plus, se defineşte un set de expresii de stare sub forma unor relaţii logice între două expresii din setul *EXP*:

$$ST = \{ Rel(a,b) \mid a,b \in EXP \}.$$

Pe baza acestor definiții, un model ASFD poate fi definit prin cvintuplul:

$<S, I \cup ST, O \cup A, f, h>$

unde setul valorilor de intrare a fost extins pentru a include expresii de stare, setul valorilor de ieşire a fost extins pentru a include asignări, iar funcțiile f şi h sunt definite ca aplicații

$$S \times (I \cup ST) \rightarrow S$$

şi respectiv

$$S \times (I \cup ST) \rightarrow (O \cup A).$$

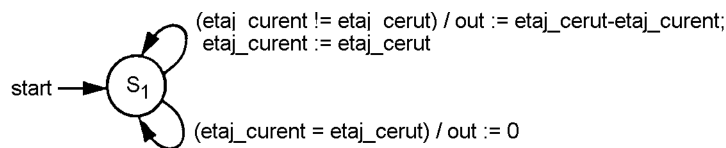


Figura 7. Modelul ASFD al controlerului pentru ascensor.

Utilizând acest model, controlerul pentru ascensor poate fi modelat cu o singură stare (**Figura 7**). Această reducere a numărului de stări este posibilă deoarece s-a introdus variabila *etaj_curent* pentru a memora valoarea etajului curent, eliminând necesitatea alocării unei stări pe etaj.

În general, modelul ASF este adecvat pentru modelarea sistemelor la care partea de control este predominantă, în timp ce modelul ASFD poate fi adecvat atât pentru sistemele la care este predominantă partea de control, cât şi pentru cele la care este predominantă partea de prelucrare. Totuşi, niciunul din cele două modele nu este adecvat pentru sistemele complexe, deoarece ele nu permit reprezentarea explicită a concurenței şi a ierarhiei. Fără posibilitatea reprezentării explicite a concurenței, un sistem complex va avea un număr foarte mare de stări. Considerăm, de exemplu, un sistem care constă din două subsisteme concurente, fiecare cu 100 de stări posibile. Dacă se încearcă reprezentarea acestui sistem cu un singur model ASF sau ASFD, trebuie să se reprezinte toate stările posibile ale sistemului, deci 100 x 100 = 10.000 de stări. În acelaşi timp, lipsa ierarhiei va determina o creştere a numărului de arce. De exemplu, dacă există 100 de stări, fiecare necesitând un arc pentru tranziția la o anumită stare pentru o anumită valoare de intrare, vor fi necesare 100 de arce, spre deosebire de singurul arc necesitat de un model care poate grupa în mod ierarhic cele 100 de stări într-o singură stare. Aceste modele au dezavantajul că odată ce se ajunge la mai multe sute de stări sau arce, ele ajung neinteligibile pentru proiectanți.

2.3.2. Rețele Petri

Modelul rețelelor Petri este un alt tip de model orientat pe stări, definit pentru modelarea sistemelor cu taskuri concurente în interacțiune. Modelul constă dintr-un set de *locații*, un set de *tranziții* şi un set de *marcaje*. Marcajele aparțin locațiilor şi circulă prin rețeaua Petri, fiind produse şi consumate ori de câte ori o tranziție este activată.

O rețea Petri se poate defini prin cvintuplul:

$< L, T, I, O, u >$

unde $L = \{l_1, l_2, \dots, l_m\}$ este un set de locații, $T = \{t_1, t_2, \dots, t_n\}$ este un set de tranziții, L și T fiind disjuncte. Funcția de intrare $I : T \rightarrow L^+$ definește toate locațiile care furnizează intrări unei tranziții, iar funcția de ieșire $O : T \rightarrow L^+$ definește toate locațiile de ieșire pentru fiecare tranziție. Funcția de marcaj $u : L \rightarrow N$ definește numărul de marcaje din fiecare locație.

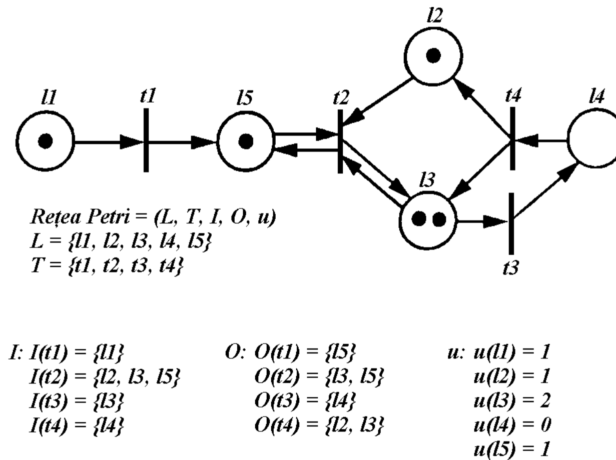


Figura 8. Exemplu de rețea Petri.

În **Figura 8** se exemplifică o reprezentare grafică și una textuală pentru o rețea Petri. În această rețea există cinci locații (reprezentate prin cercuri) și patru tranziții (reprezentate prin bare verticale). Locațiile *l2*, *l3* și *l5* furnizează intrări pentru tranziția *t2*, iar *l3* și *l5* sunt locațiile de ieșire pentru *t2*. Funcția de marcaj *u* asignează un marcaj locațiilor *l1*, *l2* și *l5*, și două marcaje locației *l3*, ceea ce se indică prin $u(l1, l2, l3, l4, l5) = (1, 1, 2, 0, 1)$.

O tranziție se poate activa numai dacă ea este validată, deci dacă fiecare din locațiile sale de intrare are cel puțin un marcaj. O tranziție este activată atunci când s-au eliminat toate marcajele de validare din locațiile sale de intrare, și s-a plasat un marcaj în fiecare locație de ieșire. În **Figura 8**, de exemplu, după activarea tranziției *t2* funcția de marcaj *u* se modifică în $(1, 0, 2, 0, 1)$.

Rețelele Petri sunt utile deoarece pot modela o varietate de caracteristici ale sistemelor. **Figura 9(a)**, de exemplu, prezintă modelarea *secvențierii*, la care tranziția *t1* se activează după tranziția *t2*. În **Figura 9(b)** se observă modelarea *ramificației*, la care sunt validate două tranziții, dar se poate activa numai una dintre ele. În **Figura 9(c)** se prezintă modelarea *sincronizării*, la care o tranziție se poate activa numai după ce ambele locații de intrare au marcaje. **Figura 9(d)** prezintă modelarea *conflictului la resurse*, două tranziții concurând pentru același marcaj, aflat în locația din centru. În **Figura 9(e)** se poate observa modelarea *concurenței*, două tranziții, *t2* și *t3*, putând fi activate simultan. Mai precis, **Figura 9(e)** modelează două procese concurente, un producător și un consumator; marcajul aflat în locația din centru este produs de *t2* și este consumat de *t3*.

Modelele cu rețele Petri se pot utiliza pentru a testa și a valida anumite proprietăți utile ale sistemelor, ca siguranța și viabilitatea. Siguranța, de exemplu, este proprietatea rețelelor Petri care garantează că numărul marcajelor din rețea nu va crește în mod nelimitat. Viabilitatea, pe de altă parte, este proprietatea rețelelor Petri care garantează o operare fără interblocare, asigurând existența a cel puțin unei tranziții care se poate activa.

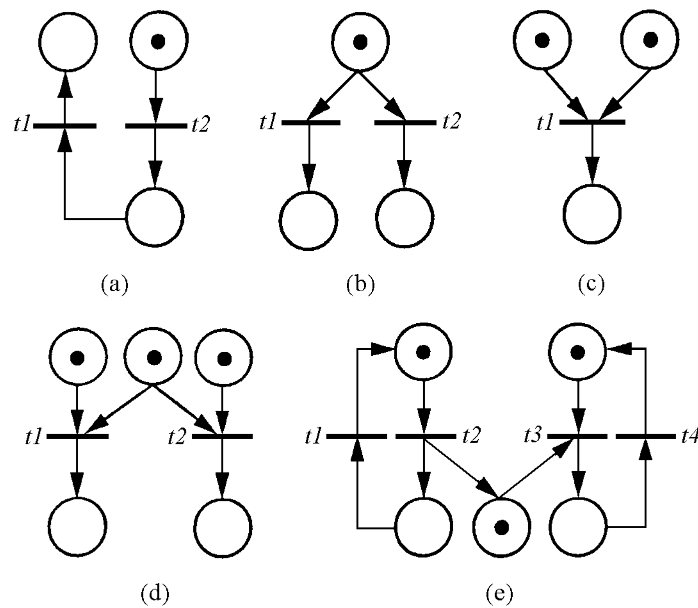


Figura 9. Rețele Petri reprezentând: (a) secvențierea; (b) ramificația; (c) sincronizarea; (d) conflictul la resurse; (e) concurența.

Deşi rețelele Petri au numeroase avantaje pentru modelarea şi analiza sistemelor concurente, au şi limitări care sunt similare cu cele ale modelului ASF: ele pot deveni neinteligibile la creşterea complexității sistemelor.

2.3.3. Automate cu stări finite ierarhice şi concurente

Modelul ASF ierarhic şi concurent reprezintă o extensie a modelului ASF, care permite reprezentarea *ierarhiei* şi a *concurenței*, eliminând creşterea exagerată a numărului de stări şi de arce care apare la descrierea sistemelor ierarhice şi concurente cu modelul ASF.

Ca şi modelul ASF, acest model constă dintr-un set de stări şi un set de tranziții. Spre deosebire de modelul ASF, în acest caz fiecare stare poate fi descompusă într-un set de *substări*, modelându-se astfel ierarhia. Fiecare stare poate fi descompusă de asemenea în *substări concurente*, care se execută în paralel şi comunică între ele prin variabile globale. Tranzițiile în cadrul acestui model pot fi structurate sau nestructurate. *Tranzițiile structurate* sunt permise numai între două stări cu acelaşi nivel ierarhic, în timp ce *tranzițiile nestructurate* pot apare între două stări oarecare, indiferent de relațiile ierarhice dintre acestea.

Un limbaj care este adaptat în mod deosebit acestui model este *Statecharts*, deoarece permite ierarhia, concurența şi comunicația între stările concurente. *Statecharts* utilizează tranziții nestructurate şi un mecanism de comunicație prin care evenimentele emise de oricare stare pot fi detectate de toate celelalte stări.

Limbajul *Statecharts* este un limbaj grafic. Se utilizează dreptunghiuri rotunjite pentru a reprezenta stările la oricare nivel, şi încapsularea pentru a exprima o relație ierarhică între aceste stări. Liniile punctate dintre stări reprezintă concurența, iar săgețile indică tranzițiile dintre stări, fiecare săgeată fiind etichetată cu un eveniment şi, opțional, cu o condiție şi/sau acțiune între paranteze.

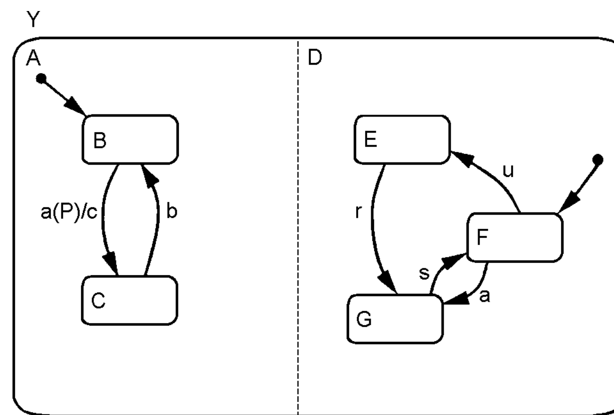


Figura 10. *Statecharts*: stări ierarhice şi concurente.

În **Figura 10** se prezintă un exemplu de sistem reprezentat în limbajul *Statecharts*. Starea *Y* este descompusă în două stări concurente, *A* şi *D*; prima constă din două substări *B* şi *C*, iar a doua cuprinde substările *E*, *F* şi *G*. Conform limbajului, dacă apare evenimentul *b* în timpul stării *C*, are loc transferul în starea *B*. Pe de altă parte, dacă apare evenimentul *a* în timpul stării *B*, are loc transferul în starea *C*, dar numai dacă este îndeplinită condiţia *P* în momentul apariţiei evenimentului. În timpul transferului din starea *B* în starea *C*, se va executa acţiunea *c*, care este asociată cu această tranziţie.

Modelul automatelor cu stări finite ierarhice şi concurente este adecvat pentru reprezentarea sistemelor de control complexe. Problema este că acest model, ca şi toate modelele orientate pe stare, se concentrează exclusiv asupra modelării controlului, şi deci poate asocia numai acţiuni foarte simple, ca asignările, cu tranziţiile sau stările sale. Ca urmare, acest model nu este adecvat pentru modelarea anumitor caracteristici ale sistemelor complexe, care necesită structuri de date complexe sau execuţia unor acţiuni complexe în fiecare stare.

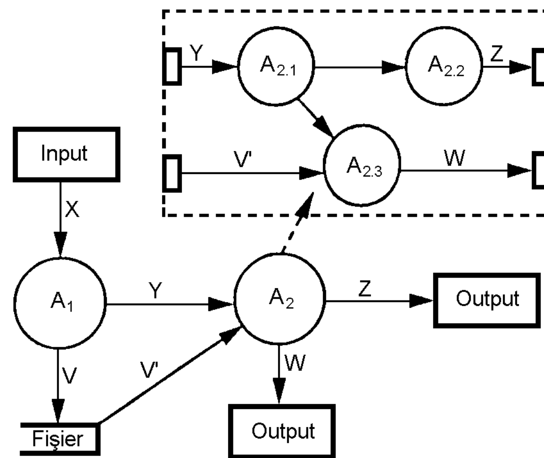
2.4. Modele orientate pe activitate

2.4.1. Grafuri ale fluxului de date

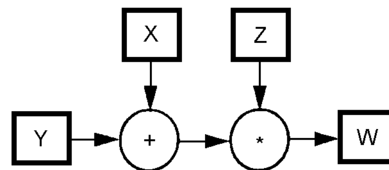
Modelele orientate pe stare sunt utilizate mai ales pentru sistemele reactive, la care starea sistemului se modifică ca răspuns la anumite evenimente externe. Spre deosebire de acestea, grafurile fluxului de date (GFD) se utilizează în principal pentru sistemele transformaţionale, la care ieşirile sunt determinate de un set de calcule efectuate asupra intrărilor sistemului. Aceste grafuri constau dintr-un set de *activităţi* (transformări) conectate printr-un set de arce care reprezintă *fluxul de date*.

Mai precis, un model GFD constă dintr-un set de noduri şi un set de arce. Există mai multe tipuri de noduri într-un graf al fluxului de date. Primul tip cuprinde nodurile de *intrare* (numite şi noduri sursă) şi nodurile de *ieşire* (noduri destinaţie), care reprezintă intrările sau ieşirile datelor. Al doilea tip cuprinde nodurile de *activitate* (nodurile de proces), care repre-

zintă activitățile care transformă sau gestionează datele. Asemenea activități pot fi descrise printr-un program, o procedură, o funcție, o instrucțiune sau o operație aritmetică. Ultimul tip de noduri este cel de *memorare*, care reprezintă diferite forme de memorare a datelor, ca înregistrările dintr-o bază de date, fişierele dintr-un sistem de operare, sau variabilele dintr-o memorie sau registru. Aceste noduri sunt interconectate prin arce direcționate care sunt etichetate cu datele care sunt transmise între cele două noduri. Acest model permite reprezentarea ierarhiei, deoarece fiecare nod de activitate poate fi reprezentat printr-un alt graf.



(a)



(b)

Figura 11. Diagramă a fluxului de date: (a) la nivel de activitate; (b) la nivel de operație.

Reprezentarea grafică a unui model GFD utilizează în general dreptunghiuri pentru nodurile de intrare şi cele de ieşire, cercuri pentru nodurile de activitate, şi dreptunghiuri deschise pentru nodurile de memorare. Fluxul datelor este reprezentat prin arce, etichetate cu datele asociate. În exemplul din **Figura 11**, sistemul constă din două activități, A_1 şi A_2, ultima fiind descompusă în activitățile $A_{2.1}$, $A_{2.2}$ şi $A_{2.3}$. În acest sistem, data X se transferă de la intrare la A_1, iar data V se calculează de A_1 şi se memorează în *Fişier*. Data V' va fi preluată apoi din *Fişier* şi se va utiliza ca intrare pentru A_2, împreună cu data Y care a fost generată de A_1. Datele Z şi W reprezintă ieşirile generate de A_2.

Un model al fluxului de date este util deoarece poate fi utilizat în diferite domenii de aplicații, sau în diferite faze de proiectare ale aceluiaşi domeniu, asociind diferite obiecte cu nodurile şi muchiile grafului. De exemplu, în domeniul procesării digitale a semnalelor, nodurile pot reprezenta variabile şi operații aritmetice, în timp ce arcele grafului pot indica de-

pendenţe ale datelor, ca în **Figura 11(b)**. În acest exemplu, operaţia '+' este dependentă de datele *X* şi *Y*, iar operaţia '*' este dependentă de data *Z* şi de ieşirea operaţiei '+'.

Un model GFD nu descrie nici o secvenţiere impusă, dincolo de dependenţele datelor existente în cadrul diferitelor activităţi. De asemenea, modelul nu conţine nici o informaţie referitoare la implementare. Din aceste motive, acest model este utilizat de multe ori în timpul fazei de specificaţie a sistemelor. Deoarece modelul GFD permite descompunerea ierarhică, se poate utiliza şi pentru specificarea sistemelor transformaţionale complexe.

2.4.2. Grafuri ale fluxului de control

Aceste grafuri sunt în multe privinţe similare cu grafurile fluxului de date, dar arcele sunt utilizate pentru a reprezenta secvenţierea sau fluxul de control. Grafurile fluxului de control sunt asemănătoare şi cu automatele cu stări finite cu căi de date, în sensul că ambele pun accentul pe aspectul de control al sistemului, dar diferă de acestea prin mecanismele care activează tranziţiile. În cazul ASF cu căi de date, tranziţiile sunt activate prin apariţia unor evenimente externe, în timp ce în cazul grafurilor fluxului de control tranziţiile sunt activate ori de câte ori o anumită activitate este terminată.

În principiu, un graf al fluxului de control constă dintr-un set de noduri şi un set de arce. Există mai multe tipuri de noduri, primul tip fiind acela al nodurilor *de început* şi *de sfârşit*, care indică punctele de început şi de sfârşit ale grafului. Al doilea tip este cel al nodurilor *de calcul*, care se utilizează pentru a defini transformări asupra datelor. Al treilea tip este cel al nodurilor *de decizie*, care se utilizează pentru controlul ramificaţiilor. Diferitele tipuri de noduri sunt interconectate prin arce direcţionate, care indică ordinea în care trebuie executate operaţiile specificate de noduri.

Graful fluxului de control este util atunci când un sistem trebuie considerat ca un set de activităţi secvenţiale, supervizate de un flux de control. Acest model este potrivit pentru sisteme cu taskuri care nu depind de evenimente externe. De asemenea, poate fi utilizat pentru a impune o ordine specifică de execuţie a activităţilor într-un graf al fluxului de date, atunci când este necesar să se modifice dependenţa naturală a datelor. Deoarece această ordine impusă de proiectant poate sugera o anumită implementare a sistemului, graful fluxului de control se utilizează în acest mod numai dacă implementarea sistemului este bine înţeleasă.

2.5. Modele orientate pe structură

2.5.1. Diagrame de conexiune a componentelor

Diagramele de conexiune a componentelor (DCC) reprezintă o clasă de modele orientate pe structură care se utilizează pentru a descrie structura fizică a unui sistem, şi nu funcţionarea acestuia. Spre deosebire de grafurile fluxului de date sau de control, care reprezintă un set de activităţi ale sistemului conectate prin dependenţe de date sau de control, diagramele de conexiune a componentelor reprezintă un set de componente ale sistemului şi interconexiunile acestora.

O diagramă de conexiune a componentelor constă dintr-un set de noduri şi un set de conexiuni dintre acestea. Nodurile reprezintă diferite componente, care sunt definite ca obiecte structurale cu un set de intrări şi ieşiri, ca porţi logice, unităţi aritmetice şi logice, proce-

soare sau chiar subsisteme. Conexiunile dintre componente pot fi reprezentate de fire sau magistrale.

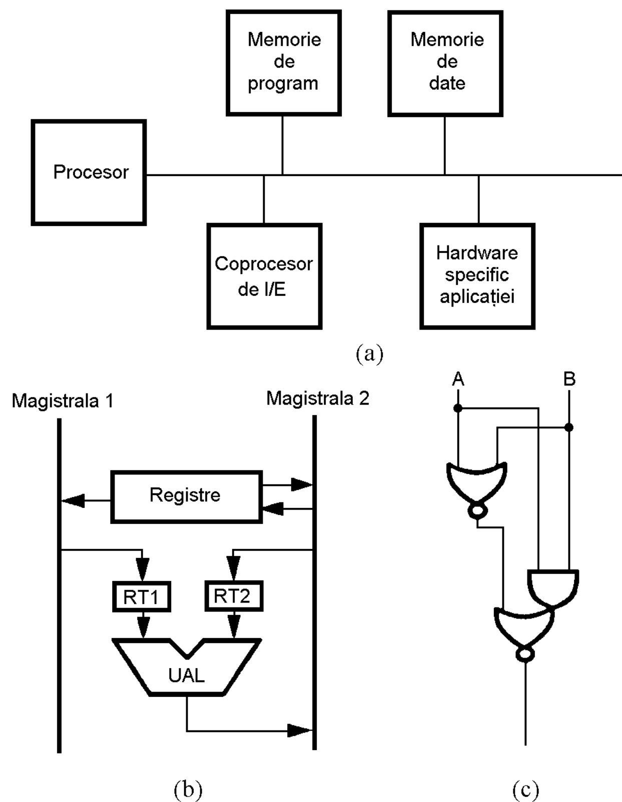


Figura 12. Modelare orientată pe structură: (a) schemă bloc a sistemului; (b) schemă la nivelul RT; (c) schemă la nivel de porți.

Deoarece acest model permite asocierea diferitelor obiecte cu nodurile şi conexiunile dintre acestea, DCC poate fi inclusă în diferite modele de reprezentare. În **Figura 12**, de exemplu, o diagramă de conexiune este instanțiată cu trei nivele diferite de abstractizare, rezultând o *schemă bloc a sistemului*, o *schemă la nivel de registre* şi o *schemă la nivel de porți*. În schema bloc, componentele sunt definite ca module la nivel de sistem, ca procesoare, memorii, sau circuite ASIC. Conexiunile dintre aceste componente sunt specificate numai parțial, deoarece această diagramă nu cuprinde informații detaliate de conexiune, ca dimensiunea magistralei sau semnalele de control. În schema la nivel de registre, componentele reprezintă unități la nivelul transferurilor între registre, ca unități aritmetice şi logice, registre, multiplexoare sau magistrale, iar conexiunile definesc modul în care datele vor fi transferate între aceste elemente aritmetice şi cele de memorie. În cadrul acestei scheme, de obicei semnalele de control nu sunt specificate. Schema la nivel de porți utilizează porți logice ca şi componente, iar conexiunile dintre aceste componente reprezintă conexiuni fizice. Conexiunile de date şi de control sunt specificate complet.

Deoarece modelul DCC este adecvat pentru a reprezenta structura sistemului, este utilizat adesea în fazele finale ale procesului de proiectare, când proiectantul trebuie să specifice implementarea sistemului.

2.5.2. Structuri de incidență

O structură de incidență constă dintr-un set de module, un set de conexiuni şi o relație de incidență între module şi conexiuni. Un model simplu pentru reprezentarea structurii este *hipergraful*, unde nodurile corespund modulelor şi arcele corespund conexiunilor. Relația de incidență este reprezentată printr-o matrice de incidență.

O altă posibilitate de a specifica o structură este de a reprezenta fiecare modul prin terminalele sale, numite *pini* (sau *porturi*), şi de a descrie relația de incidență dintre conexiuni şi pini.

De multe ori matricea de incidență este rară, şi descrierea se poate realiza mai eficient prin *liste de conexiuni*. O listă de conexiuni specifică toate conexiunile unui modul dat (listă orientată pe module), sau toate modulele unei conexiuni date (listă orientată pe conexiuni).

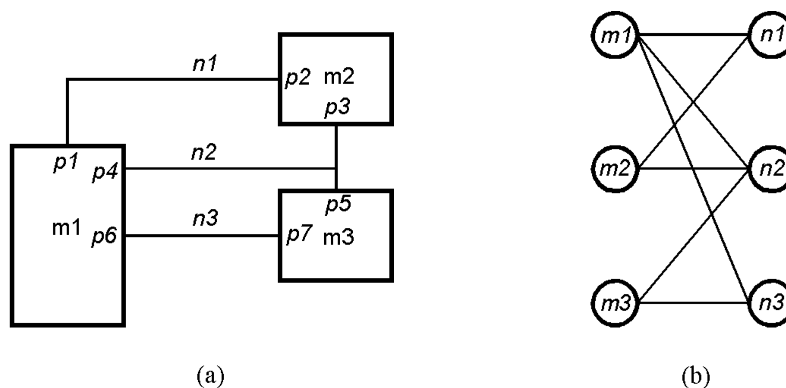


Figura 13. Exemplu de structură de incidență: (a) module, conexiuni şi pini; (b) graf bipartit.

Considerând exemplul din **Figura 13(a)**, există trei module, trei conexiuni şi şapte pini. Matricea de incidență modul-conexiune este:

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

Graful bipartit este prezentat în **Figura 13(b)**. Lista de conexiuni orientată pe module este:

m1: n1, n2, n3
m2: n1, n2
m3: n2, n3

Structurile de incidență pot fi ierarhice. Un modul terminal este o primitivă cu un set de pini. Un modul non-terminal constă dintr-un set de module, reprezentând submodulele acestuia, un set de conexiuni şi o structură de incidență care stabileşte o relație între conexiuni şi pinii modulului şi cei ai submodulelor.

Considerând exemplul precedent, presupunem că structura este ierarhică şi modulul *m2* are submodule. În **Figura 14** se prezintă detaliile modulului *m2*, care constă din submodulele *m21* şi *m22*, conexiunile *n21* şi *n22*, şi pinii interni *p21*, *p22*, *p23*, *p24* şi *p25*.

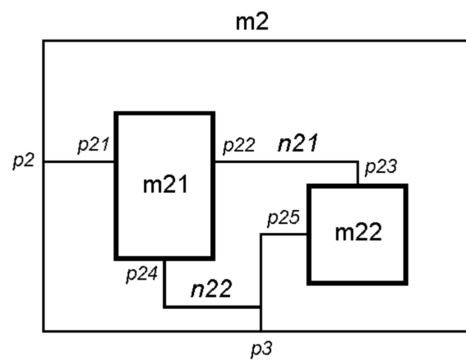


Figura 14. Exemplu de structură ierarhică: detaliile modulului *m2*.

2.5.3. Reţele logice

O *reţea logică generalizată* este o structură în care fiecare modul terminal este asociat cu o funcţie logică. Se consideră în continuare două restricţii ale acestui model: reţeaua logică combinaţională şi reţeaua logică secvenţială.

O *reţea logică combinaţională*, numită şi *reţea logică* sau *reţea booleană*, este o structură ierarhică în care:

- Fiecare modul terminal este asociat cu o funcţie logică combinaţională cu mai multe intrări şi o singură ieşire, reprezentând o *funcţie locală*.
- Pinii sunt partiţionaţi în două clase, numite intrări şi ieşiri. Pinii care nu aparţin submodulelor sunt de asemenea partiţionaţi în două clase, numite *intrări primare* şi *ieşiri primare*.
- Fiecare conexiune are un terminal distinct, numit terminal sursă, şi o orientare de la terminalul sursă la celelalte terminale. Sursa unei conexiuni poate fi o intrare primară sau o ieşire primară a unui modul de la nivel inferior. (În particular, aceasta poate corespunde cu ieşirea unei funcţii locale.)
- Relaţia indusă de conexiuni asupra modulelor este cea de ordine parţială.

Reţelele logice sunt reprezentate de obicei prin grafuri. Un *graf al reţelei logice*, $G_n(V,E)$, este un graf direcţionat, al cărui set de noduri V se află în corespondenţă directă cu intrările primare, funcţiile locale şi ieşirile primare. Setul de arce direcţionate E reprezintă descompunerea conexiunilor cu terminale multiple în conexiuni cu două terminale. De notat că graful este aciclic prin definiţie, deoarece conexiunile induc o ordine parţială asupra modulelor.

În **Figura 15(a)** se prezintă un exemplu de reţea logică, iar în **Figura 15(b)** se prezintă graful corespunzător. Graful are trei noduri de intrare, v_a, v_b şi v_c, două noduri de ieşire, v_x şi v_y, şi două noduri interne, v_p şi v_q, corespunzătoare funcţiilor logice.

În cele mai multe cazuri, reţelele logice sunt utilizate pentru a reprezenta funcţii logice cu intrări/ieşiri multiple într-un mod structurat. Reţelele logice au o funcţie logică combinaţională unică de intrare/ieşire, care se poate obţine prin combinarea funcţiilor locale pentru a exprima ieşirile primare în funcţie de intrările primare. De multe ori funcţiile de intrare/ieşire nu pot fi reprezentate în forme standard, ca sumele de produse, din cauza complexităţii lor. Acesta este un motiv pentru a se utiliza modelul reţelelor logice. De notat că acest model nu constituie o reprezentare unică a unei funcţii combinaţionale.

Modelul *rețelelor logice sincrone* este o generalizare a modelului rețelelor logice combinaționale, pentru a permite descrierea circuitelor secvențiale sincrone. În cazul acestui model, modulele terminale pot implementa funcții logice combinaționale cu intrări multiple și o singură ieșire sau elemente de întârziere sincrone. Conexiunile nu trebuie să inducă o relație de ordine parțială asupra modulelor. Totuși, subsetul conexiunilor a căror sursă nu este un element de întârziere trebuie să inducă o relație de ordine parțială asupra modulelor, pentru a modela cerința ca într-un circuit sincron să nu existe bucle combinaționale directe.

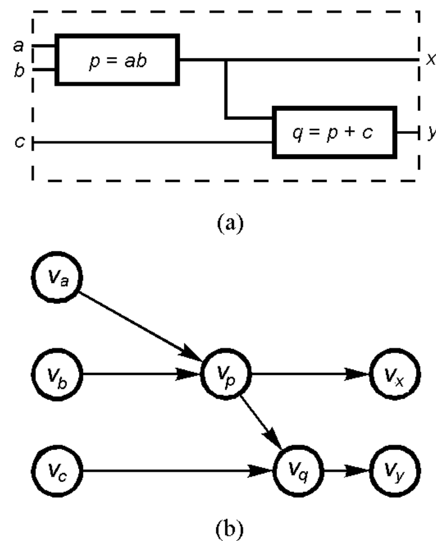


Figura 15. Exemplu de rețea logică: (a) module, intrări și ieșiri; (b) graful rețelei logice.

În timp ce funcționarea circuitelor logice combinaționale poate fi descrisă prin funcții logice, funcționarea circuitelor secvențiale poate fi descrisă prin secvențe de intrări și ieșiri, care corespund cu secvențele pe care modelele automatelor cu stări finite le pot accepta și genera. Secvențele sunt notate prin variabile cărora li se atașează momente de timp. De exemplu, $x^{(n)}$ indică variabila x la momentul de timp n. Este convenabil să existe o notație prescurtată pentru variabile, fără o dependență explicită de timp, indicând o întârziere sincronă printr-un offset față de un moment de timp de referință. Aceasta se reprezintă prin adăugarea simbolului @, urmat de valoarea offsetului, la variabila respectivă. Astfel, $x@k = x^{(n-k)}$ și $x = x@0$. Ecuațiile care utilizează notația prescurtată sunt normalizate astfel încât se presupune că partea stângă are un offset egal cu zero, deci ecuația $x^{(n+1)} = y^{(n)}$ este translatată prin $x = y@1$.

O rețea logică sincronă poate fi definită prin:

- Un set de noduri V partiționat în trei subseturi numite intrări primare V^I, ieșiri primare V^O și noduri interne V^G. Fiecare nod este asociat unei variabile.
- Un set de funcții combinaționale booleene scalare asociate cu nodurile interne. Variabilelor fiecărei funcții locale li se adaugă momente de timp, și aceste variabile sunt asociate cu intrările primare sau alte noduri interne.
- Un set de asignări ale ieșirilor primare la nodurile interne, care indică variabilele care sunt observabile din afara rețelei.

În cadrul unei rețele logice sincrone registrele se reprezintă implicit, prin ponderi pozitive asignate conexiunilor (și arcelor din graful corespunzător). Se presupune că rețelele de inerconexiune sunt divizate în seturi de conexiuni între două terminale.

De observat că o funcție locală poate depinde de valoarea unei variabile la momente diferite de timp. În acest caz modelul necesită arce multiple între nodurile respective, fiecare având o pondere corespunzătoare. Deci, o rețea sincronă este modelată printr-un multi-graf, notat cu $G_{sn}(V,E,P)$.

Un exemplu de circuit sincron și modelul corespunzător sunt prezentate în **Figura 16(a)**, respectiv **16(b)**.

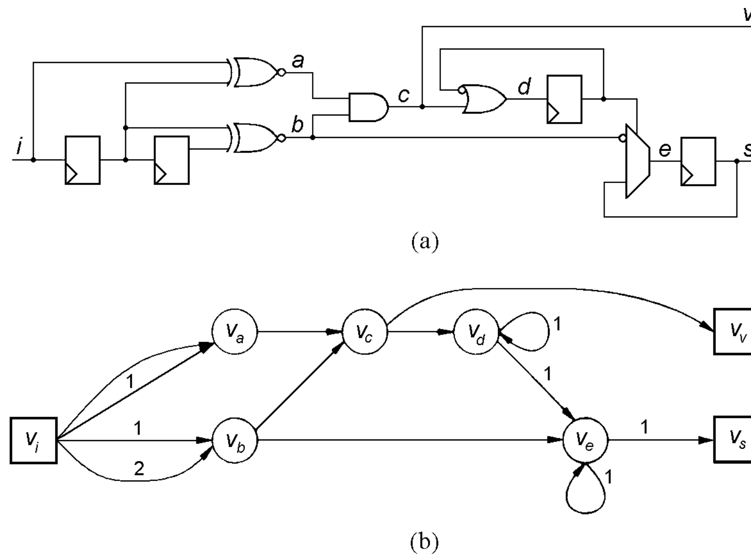


Figura 16. Exemplu de rețea logică sincronă: (a) circuit sincron; (b) graful rețelei logice.

Ca şi în cazul rețelelor combinaționale, este posibilă o reprezentare alternativă prin ecuații logice, în care apar variabile cu o dependență explicită de timp. Considerând rețeaua din **Figura 16(a)**, aceasta poate fi descrisă prin următorul set de ecuații:

$$
\begin{aligned}
a^{(n)} &= i^{(n)} \overline{\oplus}\ i^{(n-1)} \\
b^{(n)} &= i^{(n-1)} \overline{\oplus}\ i^{(n-2)} \\
c^{(n)} &= a^{(n)}\, b^{(n)} \\
d^{(n)} &= c^{(n)} + \overline{d}^{\,(n-1)} \\
e^{(n)} &= d^{(n)}\, e^{(n-1)} + \overline{d}^{\,(n)}\, \overline{b}^{\,(n)} \\
v^{(n)} &= c^{(n)} \\
s^{(n)} &= e^{(n-1)}
\end{aligned}
$$

sau, sub forma prescurtată:

$$
\begin{aligned}
a &= i\ \overline{\oplus}\ i@1 \\
b &= i@1\ \overline{\oplus}\ i@2 \\
c &= a\ b \\
d &= c + \overline{d}\ @1 \\
e &= d\ e@1 + \overline{d}\ \overline{b} \\
v &= c \\
s &= e@1
\end{aligned}
$$

2.6. Modele eterogene

2.6.1. Grafuri ale fluxului de control şi de date

Un graf al fluxului de control şi de date (GFCD) este un model eterogen conceput pentru a combina avantajele grafurilor fluxului de control şi ale grafurilor fluxului de date.

Un model GFCD conține grafuri ale fluxului de date şi un graf al fluxului de control, care indică secvențierea grafurilor fluxului de date. Astfel, modelul GFCD poate indica explicit atât dependența datelor, cât şi secvența de control a unui sistem într-o singură reprezentare.

```
CASE C IS
  WHEN 1 => X := X + 2;
            A := X + 5;
  WHEN 2 => A := X + 3;
  WHEN OTHERS => A := X + W;
END CASE;
```

(a)

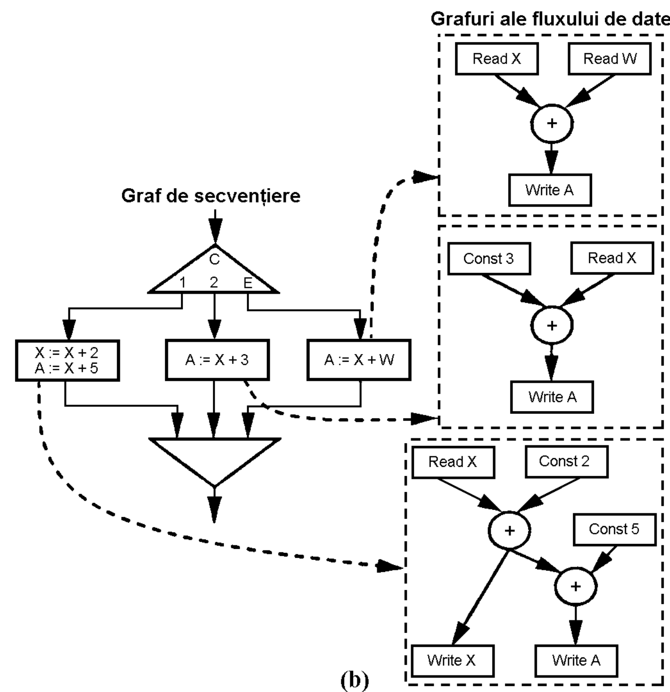


Figura 17. Graf al fluxului de control şi de date: (a) secvență de program; (b) GFCD.

În **Figura 17(b)** se prezintă o reprezentare sub forma modelului GFCD a secvenței de program din **Figura 17(a)**. Construcțiile de control, cum este instrucțiunea case, sunt reprezentate prin nodurile grafului fluxului de control, iar grupurile instrucțiunilor de asignare sunt reprezentate prin grafuri ale fluxului de date. Cele două tipuri de grafuri sunt conectate prin linii întrerupte, care indică activitățile din grafurile fluxului de date asociate cu diferitele noduri ale grafului fluxului de control.

Un model GFCD nu este însă limitat la reprezentarea construcțiilor de control şi a instrucțiunilor de asignare din limbajele de programare. Poate fi utilizat de asemenea pentru a reprezenta activitățile complexe şi acțiunile de control din cadrul unui sistem, fiind utilizat în mod frecvent la proiectarea sistemelor în timp real. Într-un asemenea sistem, graful fluxului de control poate răspunde la evenimentele interne şi externe, şi poate controla execuția activităților din graful fluxului de date prin acțiuni de control, ca *validarea* unei activități sau *invalidarea* acesteia.

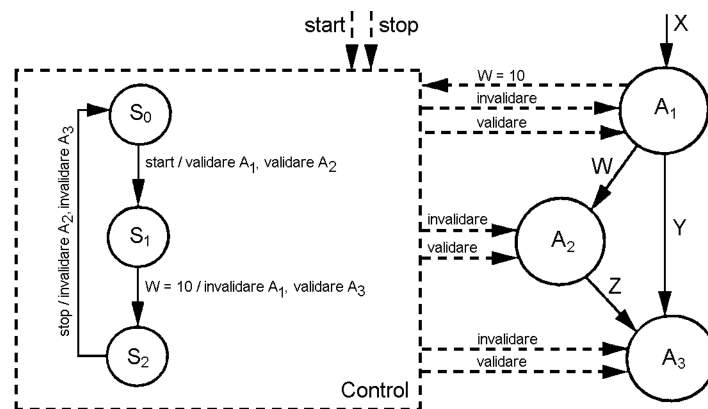


Figura 18. Exemplu de model GFCD la nivel de activitate.

În **Figura 18** se prezintă un exemplu de model GFCD la nivel de activitate. Conform acestei reprezentări, dacă graful fluxului de control se află în starea S_0 şi apare evenimentul *start*, activităţile A_1 şi A_2 vor fi validate şi sistemul va trece în starea S_1. Dacă graful se află în starea S_1 şi apare evenimentul $W=10$, activitatea A_1 va fi invalidată, activitatea A_3 va fi validată, şi sistemul va trece în starea S_2. În sfârşit, dacă sistemul se află în starea S_2 şi apare evenimentul *stop*, activităţile A_2 şi A_3 vor fi invalidate şi sistemul va reveni în starea S_0. *W*, *X*, *Y* şi *Z* reprezintă fluxul datelor între diferitele activităţi specificate în graful fluxului de date.

Avantajul principal al modelului GFCD este că se corectează incapacitatea unui graf al fluxului de date de a reprezenta controlul unui sistem, ca şi incapacitatea unui graf al fluxului de control de a reprezenta dependenţa datelor. În consecinţă, modelul este mai complet şi este adecvat pentru diferite domenii de proiectare, ca sistemele în timp real sau sinteza funcţională a circuitelor ASIC.

2.6.2. Diagrame de structură

Modelul diagramelor de structură, dezvoltat de *Yourdon* şi *Constantine*, este un alt model eterogen, elaborat pentru a specifica datele, activităţile şi controlul execuţiei activităţilor din cadrul unui sistem într-o singură reprezentare. Diagrama de structură este utilă pentru proiectanţii care lucrează cu programe.

În principiu, diagrama de structură constă dintr-un set de noduri, care reprezintă activităţi, şi un set de arce, care reprezintă apelurile de proceduri sau funcţii într-un limbaj de programare. Datele transferate între activităţi sunt indicate pe arcele dintre noduri. Controlul execuţiei activităţilor este descris printr-un set de structuri de control, ca ramificaţia, iteraţia şi apelul de procedură.

Grafic, toate activităţile din sistem sunt reprezentate prin dreptunghiuri, iar datele transferate între aceste activităţi sunt reprezentate prin săgeţi etichetate. Apelurile de proceduri sunt reprezentate prin arce, o construcţie de ramificaţie se reprezintă printr-un romb, iar o construcţie iterativă printr-o buclă.

În **Figura 19** se prezintă un exemplu de diagramă de structură. În ceea ce priveşte secvenţierea activităţilor, modulul *Main* apelează întâi modulul *Get* pentru a obţine datele *A* şi *B*. Modulul *Get* va apela la rândul său modulele *Get_A* şi *Get_B*. Modulul *Main* va

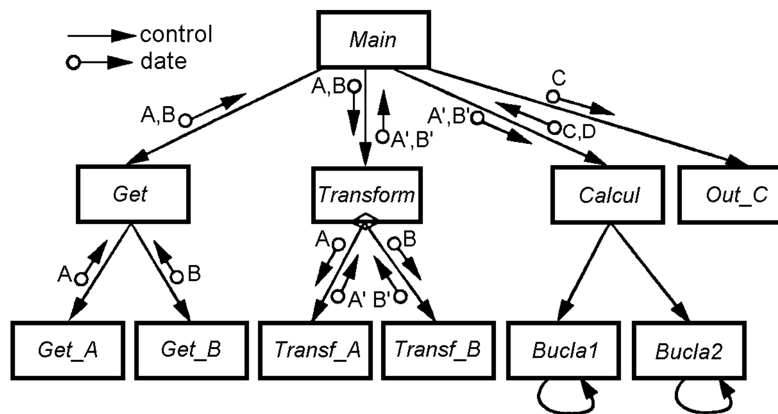


Figura 19. Exemplu de diagramă de structură.

transfera apoi datele *A* şi *B* modulului *Transform*, care va apela fie modulul *Transf_A* pentru a transforma data *A* în *A'*, fie modulul *Transf_B* pentru a transforma data *B* în *B'*, în funcţie de valorile anumitor condiţii. După obţinerea datelor *A'* şi *B'* de la modulul *Transform*, modulul *Main* va transmite aceste date modulului *Calcul*, care va apela două module iterative, *Bucla1* şi *Bucla2*. La terminarea acestor module, modulul *Calcul* va returna datele *C* şi *D* modulului *Main*, care va transmite apoi data *C* modulului *Out_C*.

Ca şi în cazul modelului GFDC, diagrama de structură poate reprezenta atât informaţiile de control cât şi datele. Trebuie menţionat că diagrama de structură nu specifică în mod complet secvenţierea execuţiei. De exemplu, nu se cunoaşte ordinea în care sunt apelate modulele *Get_A* şi *Get_B*; ordinea de execuţie a acestor module este determinată de dependenţa datelor. Pe de altă parte, modelul diagramei de structură nu furnizează construcţii care pot reprezenta un control pentru ramificaţii, iteraţii şi apeluri de proceduri. Deoarece diagramele de structură pot specifica ordinea de execuţie într-o anumită măsură, ele se utilizează în principal în etapele preliminare ale proiectării programelor secvenţiale.

2.6.3. Automate cu stări ale programului

Un automat cu stări ale programului (ASP) este un alt model eterogen care combină modelul automatelor cu stări finite ierarhice şi concurente cu modelul limbajelor de programare. Acest model constă dintr-o ierarhie de *stări ale programului*, în care fiecare stare reprezintă un mod de prelucrare distinct. În orice moment, va fi activ numai un subset de stări ale programului, cele care efectuează prelucrări.

În cadrul ierarhiei sale, modelul va consta din stări ale programului de două tipuri: compuse şi terminale. O stare compusă poate fi descompusă în substări *concurente* sau *secvenţiale* ale programului. Dacă sunt concurente, toate substările vor fi active ori de câte ori starea programului este activă, iar dacă sunt secvenţiale, substările vor fi active numai una câte una, dacă starea programului este activă.

O stare a programului descompusă secvenţial conţine un set de arce de tranziţie, care reprezintă secvenţierea între substările programului. Există două tipuri de arce de tranziţie. Primul este arcul de *tranziţie la terminare* (TT), care va fi traversat numai atunci când substarea sursă şi-a terminat prelucrările şi condiţia asociată arcului devine adevărată. Al doilea este arcul de *tranziţie imediată* (TI), care va fi traversat imediat ori de câte ori condiţia

asociată arcului devine adevărată, indiferent dacă substarea sursă şi-a terminat sau nu prelucrările.

O stare terminală se află în partea inferioară a ierarhiei şi prelucrările sale sunt descrise prin instrucţiuni ale unui limbaj de programare.

Dacă se utilizează automatul cu stări ale programului ca model, sistemul ca entitate va fi reprezentat grafic ca un dreptunghi, iar stările programului din cadrul entităţii vor fi reprezentate ca dreptunghiuri cu colţuri rotunjite. O relaţie concurentă între substările programului este indicată printr-o linie punctată între ele. Tranziţiile sunt reprezentate prin arce direcţionate. Starea iniţială este indicată printr-un triunghi, iar terminarea diferitelor stări este indicată printr-un arc de tranziţie spre punctul final, reprezentat printr-un mic pătrat în cadrul stării. Arcele TT sunt cele care încep de la un pătrat din interiorul substării sursă, în timp ce arcele TI încep de la perimetrul substării sursă.

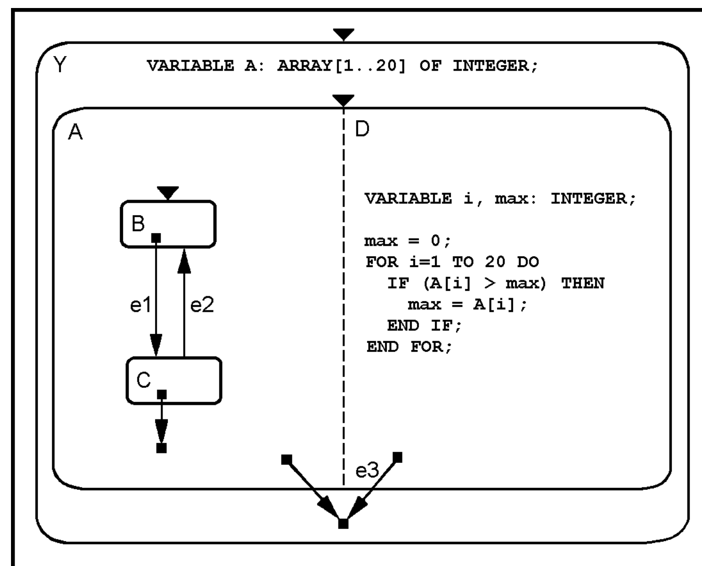


Figura 20. Exemplu de model ASP.

În **Figura 20** se prezintă un exemplu de model ASP, constând dintr-o stare rădăcină *Y*, care cuprinde două substări concurente *A* şi *D*. Starea *A* conţine două substări secvenţiale, *B* şi *C*. Stările *B*, *C* şi *D* sunt stări terminale, deşi în figură se indică numai programul pentru starea *D*. Arcele etichetate cu *e1* şi *e3* sunt arce TT, iar arcul etichetat cu *e2* este un arc TI. Configuraţia arcelor indică faptul că atunci când starea *B* se termină şi condiţia *e1* este adevărată, controlul se transferă în starea *C*. Dacă însă condiţia *e2* este adevărată în starea *C*, controlul se transferă în starea *B* indiferent dacă prelucrările efectuate de starea *C* se termină sau nu.

Deoarece automatele cu stări ale programului pot reprezenta stările, datele şi activităţile unui sistem într-un singur model, ele sunt mai potrivite pentru modelarea sistemelor care au date şi activităţi complexe asociate cu fiecare stare, decât modelele ASF ierarhice şi concurente. Un model ASP poate de asemenea elimina limitarea principală a limbajelor de programare, deoarece poate modela stările în mod explicit. Acest model permite specificarea unui sistem utilizând descompunerea ierarhică a stărilor, până când devine convenabilă utili-

zarea construcțiilor limbajelor de programare. Modelul limbajelor de programare şi modelul automatelor cu stări finite ierarhice şi concurente reprezintă două extreme ale modelului ASP. Un program poate fi considerat ca un model ASP cu o singură stare terminală conținând construcții ale limbajului. Un model ASF ierarhic şi concurent poate fi considerat ca un model ASP în care stările terminale nu conțin construcții ale unui limbaj de programare.

2.6.4. Modelul firelor de aşteptare

Acest model este diferit de modelele descrise anterior, în sensul că acestea se utilizează mai ales pentru proiectarea sistemelor, în timp ce modelul firelor de aşteptare este utilizat pentru analizarea unui sistem, de exemplu atunci când trebuie să se găsească posibilități de îmbunătățire a performanțelor sistemului.

Caracteristica modelului firelor de aşteptare este că reprezintă un sistem ca o rețea de *fire de aşteptare* şi *servere*. Cererile sunt memorate în firele de aşteptare în timp ce se aşteaptă prelucrarea lor de către servere.

Importanța acestui model este că asigură o bază pentru tipul de analize matematice necesare adesea pentru a soluționa problemele dintr-un sistem. De exemplu, dacă se cunosc anumite caracteristici ale sistemului, ca numărul de servere, tipul firelor de aşteptare, intervalul dintre două cereri consecutive, şi timpul de servire necesar pentru o cerere, modelul firelor de aşteptare permite obținerea unor informații suplimentare despre gradul de utilizare al serverelor, numărul mediu de cereri în aşteptare, rata de transfer. Proiectanții pot utiliza aceste informații pentru a localiza strangulările din sistem.

Diferitele modele pot necesita tipuri diferite de analize matematice. Anumite modele, ca cel cu un singur server şi un singur fir de aşteptare, necesită tehnici relativ simple, în timp ce altele necesită tehnici mai sofisticate, iar unele sunt aproape imposibil de analizat cantitativ.

Ca un exemplu, se analizează modelul cu un singur server şi un singur fir de aşteptare. Pentru a putea efectua analizele matematice în vederea determinării comportării sistemului, trebuie să se cunoască mai mulți parametri ai modelului. De exemplu, trebuie să se cunoască timpul dintre două cereri succesive, şi timpul necesar pentru deservirea fiecărei cereri. Este posibil să se modeleze aceşti timpi ca variabile aleatoare cu distribuții bine definite, ca în cazul în care se presupune că aceşti timpi sunt distribuiți exponențial. Într-un asemenea caz, dacă timpul mediu între două cereri este λ şi timpul mediu de deservire este μ, atunci un fir de aşteptare stabil poate fi obținut dacă $\lambda > \mu$, deoarece astfel se previne existența unui număr prea mare de cereri în aşteptare. Dacă însă $\lambda \leq \mu$, firul de aşteptare va fi instabil, putând rezulta timpi de aşteptare infiniți.

3. Limbaje de descriere hardware

3.1. Introducere

Metodologiile de proiectare actuale pentru sistemele hardware utilizează specificarea proiectelor prin scheme electrice şi apoi simularea acestora. Specificaţia iniţială este realizată de obicei în limbaj natural, fiind completată cu diagrame de stare, diagrame de timp şi scheme bloc. În acest proces, specificaţia iniţială este rafinată de către proiectant, care adaugă noi informaţii până când se obţine un proiect complet la nivelul transferurilor între registre. În timpul acestui proces de rafinare, proiectantul nu păstrează întotdeauna specificaţia iniţială sau o descriere consistentă corelată cu specificaţia iniţială. Din cauza lipsei unei descrieri formale, şi a lipsei documentaţiei în timpul procesului de proiectare, modificarea proiectului, ca şi utilizarea sistemelor de sinteză, simulare şi verificare în cadrul acestui proces, sunt dificile.

În ultimii ani, există o tendinţă de a se utiliza *limbajele de descriere hardware* pentru specificarea proiectelor, modelele bazate pe aceste limbaje fiind preferate faţă de diagramele de stare, grafurile fluxului de date şi de control, deşi unele modele bazate pe diagrame sunt mai puternice în ceea ce priveşte vizualizarea funcţiilor unui sistem. Există mai multe avantaje ale utilizării limbajelor de descriere ca limbaje de *specificaţie executabilă*, pentru descrierea sistemului la nivel conceptual.

- În primul rând, *descrierea executabilă se poate utiliza pentru simulare*, ceea ce permite proiectantului verificarea corectitudinii funcţionării propuse a sistemului. În cazul abordării tradiţionale, care începe cu o specificaţie în limbaj natural, o asemenea verificare nu este posibilă până la obţinerea unei descrieri a sistemului care permite simularea (de obicei o schemă la nivel de porţi logice).

- Al doilea avantaj este că *specificaţia poate servi ca intrare pentru sistemele de sinteză automată*, care se pot utiliza pentru obţinerea unei implementări a sistemului, reducându-se timpul de proiectare în mod semnificativ.

- În al treilea rând, o asemenea specificaţie *se poate utiliza pentru documentarea sistemului*, constituind o descriere neambiguă a funcţionării acestuia. De asemenea, aceasta poate constitui un mediu de schimb al informaţiilor între diferiţi utilizatori şi între aceştia şi diferite sisteme de proiectare.

- În sfârşit, pe măsură ce proiectele devin din ce în ce mai complexe, *limbajele de descriere pot asigura creşterea nivelului de abstractizare* pentru a se face faţă acestei complexităţi.

Chiar dacă unele limbaje de descriere s-au dezvoltat din anumite limbaje de programare, ca de exemplu *AHPL*, care se bazează pe limbajul *APL*, şi *VHDL*, care provine din limbajul *ADA*, natura specifică a circuitelor hardware determină ca aceste limbaje să fie diferite de limbajele de programare utilizate în mod curent. Astfel, semantica unei funcţii specificate într-un limbaj de descriere implică un proiect hardware care urmează a fi implementat, spre

deosebire de o funcţie care se execută pe un sistem existent. De asemenea, aceste limbaje au construcţii suplimentare pentru a se adapta la facilităţile speciale ale circuitelor.

Obiectivele multiple ale limbajelor de descriere hardware nu pot fi îndeplinite de limbajele de programare utilizate pentru specificaţiile hardware. Limbajele de programare standard au fost utilizate pentru modelarea funcţională a procesoarelor, modelare care poate fi validată prin compilarea şi execuţia modelelor. Totuşi, asemenea modele nu pot fi utilizate pentru sinteză, din cauza lipsei facilităţilor speciale necesare pentru aceste limbaje. Unele limbaje de programare au fost extinse în vederea utilizării lor pentru simulare şi sinteză. Astfel, din limbajul *C* s-au dezvoltat noi limbaje de descriere hardware, ca *ESIM* şi *HardwareC*.

Pentru a se aprecia diferenţa dintre cele două tipuri de limbaje, se vor compara în continuare cele mai importante caracteristici ale obiectelor descrise de limbajele hardware şi software.

- În primul rând, circuitele hardware pot executa operaţii cu un grad înalt de concurenţă. Din contră, programele sunt executate în mod obişnuit pe sisteme cu un singur procesor şi deci operaţiile sunt secvenţiale. Din acest punct de vedere, limbajele de descriere hardware sunt mai apropiate de limbajele de programare pentru calculatoarele paralele.
- În al doilea rând, specificaţiile circuitelor hardware necesită anumite informaţii structurale. De exemplu, interfaţa unui circuit cu exteriorul necesită definirea porturilor de intrare/ieşire şi a formatului datelor transmise prin aceste porturi. Din acest motiv, limbajele de descriere trebuie să permită atât descrieri funcţionale, cât şi structurale.
- În al treilea rând, sincronizarea operaţiilor este foarte importantă în cazul circuitelor. Pe de altă parte, sincronizarea operaţiilor din cadrul programelor este de importanţă mai redusă, cu excepţia aplicaţiilor în timp real.

Primele limbaje de descriere hardware erau utilizate pentru descrieri la nivelul porţilor logice, deoarece proiectele erau specificate şi implementate la acest nivel. Pe măsură ce nivelul de abstractizare al proiectelor devenea mai înalt, a crescut şi nivelul de abstractizare al limbajelor de descriere, rezultând o proliferare a limbajelor de descriere de nivel înalt, ceea ce a creat dificultăţi în privinţa portabilităţii descrierilor. Asemenea probleme au condus la eforturi de standardizare a limbajelor pentru proiectarea sistemelor digitale (*Conlan*, *VHDL*, *UDL/I*), cu scopul de a dezvolta un limbaj comun care se poate utiliza pentru modelare, simulare şi documentare.

Standardizarea limbajelor este dificilă, deoarece acestea trebuie utilizate pentru proiectarea diferitelor tipuri de aplicaţii, pentru diferite aspecte ale proiectării (de exemplu, simulare şi sinteză), şi pentru diferite arhitecturi. De aceea se impun cerinţe diferite pentru aceste limbaje. De exemplu, o aplicaţie în domeniul prelucrării semnalelor digitale este reprezentată cel mai eficient sub forma grafurilor semnalelor, în timp ce o interfaţă cu o magistrală este descrisă cel mai frecvent sub forma diagramelor de timp; o descriere pentru simulare necesită existenţa noţiunii timpului de simulare, în timp ce o descriere pentru sinteză trebuie să conţină construcţii care pot fi implementate prin sinteză; un proiect bazat pe automate cu stări finite necesită ca limbajul să conţină noţiunea de stare, în timp ce o descriere de sistem necesită construcţii pentru exprimarea ierarhiei şi a protocoalelor de comunicaţie.

Din aceste motive, proiectanţii necesită diferite tipuri de descrieri, sub formă tabelară, grafică sau textuală, în funcţie de tipul aplicaţiei proiectate şi de arhitectura care se va utiliza pentru implementare. Deoarece diferitele modele conceptuale au caracteristici diferite, un

anumit limbaj utilizat pentru specificație este sau nu adecvat pentru modelul respectiv după cum permite exprimarea, prin construcțiile limbajului, a tuturor caracteristicilor sau a unui număr redus de caracteristici ale modelului. Pentru ca un limbaj să poată fi utilizat în mod eficient, trebuie să existe o corelație între caracteristicile modelului conceptual şi construcțiile limbajului.

3.2. Tipuri de limbaje de descriere hardware

Limbajele de descriere hardware, ca şi limbajele de programare, pot fi clasificate în limbaje *procedurale* şi *declarative (neprocedurale)*.

Limbajele de descriere *procedurale* specifică o acțiune dorită sub forma unui algoritm, prin descrierea unei secvențe de operații efectuate de componentele unei unități. O descriere procedurală identifică fluxul informațiilor prin diferitele elemente şi natura algoritmică a proiectului. Această descriere este utilizată atunci când proiectantul este interesat de funcționarea sistemului şi mai puțin de structura exactă a acestuia. Descrierea nu este, totuşi, pur funcțională, utilizându-se elemente hardware pentru specificarea fluxului informațiilor.

Limbajele de descriere hardware au fost dezvoltate adesea împreună cu simulatoarele, iar acestea au influențat anumite opțiuni în proiectarea limbajelor. Viteza de execuție este o cerință majoră pentru simulatoare. În cazul utilizării *simulatoarelor procedurale*, timpul de simulare este considerabil mai mic față de cazul în care se utilizează simulatoare neprocedurale, aceasta deoarece simularea procedurală presupune că un element este activ numai atunci când este menționat în descriere. La alte momente de timp, nu este activă nici o operație a acestui element.

Limbajele de descriere *declarative (neprocedurale)* specifică funcționarea fiecărui element hardware şi interconexiunile dintre elemente. O descriere declarativă nu include conceptul fluxului de date, comunicația dintre elemente fiind considerată ca un transfer de valori prin conexiunile dintre acestea. Dacă se cunosc informațiile despre componente şi compoziția structurală a sistemului, acest sistem poate fi reprezentat în mod neprocedural. Avantajul este că descrierea este simplă şi concisă.

În cazul utilizării *simulatoarelor neprocedurale*, toate elementele se presupun active simultan în fiecare moment. Astfel se îmbunătățeşte fidelitatea simulării, dar timpul de simulare este mai ridicat.

Pe lângă clasificarea de mai sus, limbajele de descriere mai pot fi clasificate în limbaje cu semantică *imperativă* şi limbaje cu semantică *aplicativă*. O clasificare a limbajelor pe această bază este dificilă, deoarece adesea limbajele au caracteristici semantice din ambele categorii. Cele mai multe limbaje sunt procedurale cu o semantică imperativă.

Limbajele de descriere se clasifică şi în funcție de nivelul de descriere pentru care se utilizează (*funcțional*, *structural*, *fizic*). De exemplu, limbajele utilizate pentru descrierea la nivelul fizic sunt caracterizate prin faptul că dispun de primitive geometrice şi permit operații cu aceste primitive. Cele mai multe limbaje permit atât descrieri structurale, cât şi funcționale, deoarece specificarea circuitelor necesită de multe ori ambele tipuri de descrieri.

3.2.1. Limbaje de descriere structurale

Modelele create cu limbajele de descriere structurale descriu interconexiunile dintre componentele sistemului. Caracteristicile de bază ale limbajelor structurale le plasează în cla-

sa celor declarative, deşi unele limbaje structurale au şi caracteristici procedurale. Variabilele utilizate în cadrul limbajelor corespund porturilor componentelor.

Exemplul 3.2.1.

Se consideră un circuit semisumator. Circuitul va fi descris în limbajul *VHDL*, utilizând facilitatea de modelare structurală a acestuia.

În limbajul *VHDL*, toate proiectele sunt compuse din *entităţi*, acestea reprezentând blocurile de bază ale proiectelor. O declaraţie a unei entităţi, indicată prin cuvântul cheie ENTITY, descrie interfaţa cu exteriorul, specificând porturile şi tipul acestora. Toate entităţile care pot fi simulate au o descriere a *arhitecturii*, care specifică funcţionarea entităţii. O entitate poate avea mai multe arhitecturi; de exemplu, una din arhitecturi poate conţine o descriere structurală, şi alta o descriere funcţională. Arhitectura nu se referă la nivelul arhitectural al abstractizării.

```
ENTITY semi_sumator IS
  PORT (a, b: IN BIT; sum, carry: OUT BIT);
END semi_sumator;

ARCHITECTURE struct OF semi_sumator IS
  COMPONENT and2
    port (x, y: IN BIT; o: OUT BIT);
  END COMPONENT;
  COMPONENT xor2
    port (x, y: IN BIT; o: OUT BIT);
  END COMPONENT;
BEGIN
  u1: and2
    PORT MAP (a, b, carry);
  u2: xor2
    PORT MAP (a, b, sum);
END struct;
```

Modelul conţine două declaraţii ale altor modele, and2 şi xor2, şi două instanţieri ale acestor modele, u1 şi u2. Informaţiile suplimentare despre cele două componente and2 şi xor2 se află într-o bibliotecă de componente standard.

Pentru a realiza modele compacte ale circuitelor, se utilizează alte tipuri de variabile numite *metavariabile*. Un exemplu de metavariabilă este un index al unui tablou. Metavariabilele nu reprezintă entităţi hardware şi sunt eliminate din model în primele etape ale compilării.

Exemplul 3.2.2.

În fragmentul următor se modelează în limbajul *VHDL* un tablou de 32 inversoare între două magistrale. Se declară componenta inversor, tabloul de inversoare fiind creat prin construcţia GENERATE în care se utilizează metavariabila i.

```
ARCHITECTURE structural OF tablou_inv IS
  COMPONENT inversor
    port (i1: IN BIT; o1: OUT BIT);
  END COMPONENT;
BEGIN
  u: FOR i IN 1 TO 32 GENERATE
    inv: inversor PORT MAP (input(i), output(i));
  END GENERATE;
END structural;
```

3.2.2. Limbaje de descriere funcționale

Circuitele logice combinaționale pot fi descrise funcțional printr-un set de porturi (intrări/ieşiri) şi un set de ecuații. Limbajele *declarative* sunt cele mai potrivite pentru circuitele combinaționale, acestea fiind considerate ca o interconexiune de operatori, fiecare operator evaluând o funcție logică. Aceste modele diferă de modelele structurale prin faptul că nu există o corespondență unu la unu între expresii şi porțile logice, deoarece nu există întotdeauna o singură poartă pentru implementarea unor expresii.

Pentru descrierea circuitelor combinaționale se pot utiliza şi limbajele *procedurale*. Cele mai multe limbaje de descriere procedurale permit asignări multiple la variabile. În aceste cazuri este necesară o interpretare neambiguă a acestor asignări multiple. Se utilizează *funcții de decizie*, care returnează valoarea unui semnal dacă semnalul are mai multe drivere. Există diferite mecanisme de decizie; de exemplu, ultima asignare le poate anula pe cele anterioare. Funcțiile de decizie pot fi incluse în limbaj, sau pot fi definite de utilizator. În limbajul *VHDL*, de exemplu, utilizatorul poate defini orice tip de funcție de decizie (SAU cablat, ŞI cablat, valoare medie etc.).

Exemplul 3.2.3.

Se consideră din nou circuitul semisumator, care va fi descris în limbajul *VHDL*, utilizând facilitatea de modelare funcțională a acestuia.

```
ARCHITECTURE functional OF semi_sumator IS
BEGIN
  carry <= (a AND b);
  sum   <= (a XOR b);
END functional;
```

Operatorii `AND` şi `XOR` sunt predefiniți. Toate asignările din cadrul arhitecturii sunt executate în mod concurent.

Se consideră în continuare modelarea *circuitelor logice secvențiale*. Modelarea circuitelor combinaționale prin metode *declarative* poate fi extinsă şi la circuitele secvențiale. Aceste circuite pot fi descrise printr-un set de asignări ale unor expresii la variabile. Argumentele expresiilor sunt variabile cu întârzieri sincrone.

Exemplul 3.2.4.

În limbajul *Silage*, întârzierile variabilelor sunt notate prin caracterul `@`, urmat de o specificare a temporizării care poate fi generală. În particular, limbajul se poate utiliza pentru reprezentarea circuitelor sincrone, prin indicarea întârzierilor sincrone ca specificații de temporizare. Un filtru simplu, prezentat în **Figura 21**, poate fi descris prin expresiile următoare:

Figura 21. Filtru recursiv.

```
FUNCTION filtru (a1, a2, b1, b2, x: num)
  y: num =         /* valoarea returnata */
BEGIN
  y = interm + a2 * interm@1 + b2 * interm@2;
  interm = x + a1 * interm@1 + b1 * interm@2;
END
```

Circuitele secvenţiale pot fi modelate şi cu limbaje *procedurale*. Astfel, automatele cu stări finite pot fi descrise prin modele procedurale la care informaţia de stare este păstrată într-o variabilă. Operaţiile automatului pot fi descrise printr-o iteraţie (sincronizată cu ceasul), cu ramificaţii la fragmentele corespunzătoare stării prezente.

Exemplul 3.2.5.

Se prezintă în continuare descrierea în limbajul *VHDL* a unui automat care recunoaşte două sau mai multe valori de 1 consecutive dintr-un şir de date de intrare.

```
ARCHITECTURE functional OF recun_11 IS
  TYPE tip_stare IS (stare_zero, stare_unu);
  SIGNAL stare: tip_stare := stare_zero;
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL (clock'EVENT AND clock = '1');
    IF (in = '1') THEN
      CASE stare IS
        WHEN stare_zero =>
          stare <= stare_unu;
          out <= '0';
        WHEN stare_unu =>
          out <= '1';
      END CASE;
    ELSE
      stare <= stare_zero;
      out <= '0';
    END IF;
  END PROCESS;
END functional;
```

În acest model, semnalul `stare` este de tip enumerat şi păstrează starea automatului. Declaraţia `PROCESS` indică o zonă a arhitecturii unde toate instrucţiunile sunt secvenţiale. procesul este executat de fiecare dată când semnalul `clock` are o tranziţie de la 0 la 1. Instrucţiunea `WAIT` realizează sincronizarea modelului cu semnalul `clock`.

Limbajele de descriere funcţionale permit un anumit grad de libertate în interpretarea execuţiei în timp a operaţiilor. Sistemele de sinteză şi de optimizare utilizează această posibilitate pentru a obţine implementări optime. Pe de altă parte, rezultatele obţinute la simularea modelelor funcţionale iniţiale pot diferi de cele obţinute la simularea modelelor structurale pentru care s-a realizat sinteza, deoarece modelele iniţiale ignoră posibilităţile de optimizare. Din acest motiv, este important să se asigure mijloace de reprezentare a comportării în timp a acelor implementări care sunt compatibile cu modelul funcţional original.

Au fost dezvoltate tehnici de sinteză pentru diferite limbaje şi sisteme de proiectare care să asigure o interpretare a comportării în timp. Aceste tehnici ţin cont de construcţiile limbajului care se pot utiliza pentru sinteză, ignorând construcţiile specifice pentru simulare. Considerând automatul din exemplul 3.2.5., o tehnică de sinteză poate sincroniza operaţiile

din circuitul implementat cu instrucțiunea `wait` a modelului. Deoarece modelul are o singură instrucțiune `wait`, toate operațiile se vor executa într-un ciclu de ceas.

Exemplul 3.2.6.

Se consideră un fragment al unui model care descrie un set de instrucțiuni dintr-un calculator.

```
...
ri <= fetch(pc);
CASE ri IS
  WHEN add =>
    a <= ra + rb;
  WHEN and =>
    a <= ra AND rb;
  WHEN or =>
    a <= ra OR rb;
  WHEN xor =>
    a <= ra XOR rb;
END CASE;
pc <= pc + 1;
...
```

Funcția `fetch` returnează valoarea registrului de instrucțiuni `ri`. Există trei operații care se execută serial: încărcarea registrului de instrucțiuni `ri`, setarea registrului acumulator `a` cu rezultatul operațiilor executate între registrele `ra` şi `rb`, şi incrementarea contorului de program `pc`. O analiză a dependenței datelor arată că incrementarea contorului de program se poate executa în paralel cu una din celelalte operații.

Presupunem că pentru sinteză se adaugă o instrucțiune `wait` înaintea primei operații. Se poate presupune atunci că cele trei operații sunt sincronizate cu ceasul şi a doua operație urmează după prima, din cauza dependenței datelor. Deosebirea față de exemplul automatului este posibilitatea ca cele trei operații să nu fie terminate într-un singur ciclu de ceas. Durata operațiilor poate fi necunoscută până la executarea sintezei. Presupunând că în varianta implementată fiecare operație necesită un ciclu de ceas, la suprapunerea operațiilor acestea se vor executa în două cicluri de ceas. Pe de altă parte, un simulator va executa cele trei operații într-un singur ciclu de ceas.

Pentru a se evita diferențele între execuția în timp a simulării funcționale şi a implementării, se pot adăuga instrucțiuni `wait` suplimentare, de exemplu ca în fragmentul următor.

```
...
WAIT UNTIL (clock'EVENT AND clock = '1');
ri <= fetch(pc);
WAIT UNTIL (clock'EVENT AND clock = '1');
CASE ri IS
  WHEN add =>
    a <= ra + rb;
  WHEN and =>
    a <= ra AND rb;
  WHEN or =>
    a <= ra OR rb;
  WHEN xor =>
    a <= ra XOR rb;
END CASE;
pc <= pc + 1;
...
```

3.3. Caracteristici ale limbajelor de descriere hardware

3.3.1. Caracteristici specifice limbajelor de programare

Limbajele de descriere hardware s-au dezvoltat în cele mai multe cazuri din limbajele de programare; în consecință, cele mai multe limbaje de descriere prezintă caracteristici specifice limbajelor imperative, ca mecanisme de abstractizare a datelor, operatori funcționali pentru transformarea datelor, instrucțiuni de asignare pentru modificarea valorilor variabilelor, construcții de control, şi construcții pentru specificarea ordinii de execuție a operațiilor.

3.3.1.1. Tipuri de date

Specificarea tipului datelor dintr-un sistem reprezintă definirea formatului (de ex., a numărului de biți), a tipului (de ex., boolean, întreg, în virgulă mobilă), şi a reprezentării (de ex., cu semn sau fără semn, în complement față de 2) tuturor variabilelor dintr-o descriere funcțională. Prin această specificare se asigură descrieri concise şi inteligibile, şi se permite detectarea erorilor semantice de către compilator, de exemplu la asignarea unei valori de 16 biți la un registru de 8 biți.

Verificarea strictă a tipului datelor permite efectuarea unor teste de consistență în timpul procesului de proiectare, dar încarcă în acelaşi timp compilatorul limbajului cu sarcina efectuării a mai multor teste. În cazul minimal, variabilele vor fi caracterizate prin dimensiunea lor în biți, sistemul de proiectare presupunând un tip implicit (de ex., un vector de biți) şi o reprezentare implicită a datelor (de ex., în complement față de 2).

3.3.1.2. Operatori şi instrucțiuni de asignare

Transformarea datelor în cadrul descrierii unui proiect este efectuată de diferite tipuri de operatori ai limbajului. Aceştia pot fi clasificați în operatori aritmetici, booleeni, logici, de acces la tablouri şi de asignare. Cele mai multe limbaje de descriere hardware au un număr suficient de operatori predefiniți. Anumite limbaje permit redefinirea semanticii unor operatori existenți.

Construcția de bază a limbajelor pentru specificarea transformării datelor este instrucțiunea de asignare.

3.3.1.3. Construcții de control

Pentru specificarea funcționării în timp a unei descrieri este necesară descrierea secvențierii instrucțiunilor de asignare. Se utilizează construcții de control ca `if-then-else`, `case` şi `loop` pentru a specifica execuția condițională şi repetitivă.

În exemplul următor (**Figura 22**), descrierea ciclului de citire, decodificare şi execuție a instrucțiunilor unui procesor simplu în limbajul *ISPS* utilizează construcțiile de control `repeat` pentru buclare, `decode` şi `if` pentru execuția condițională. Prima instrucțiune din bucla `repeat` încarcă instrucțiunea curentă în registrul *RI*. Instrucțiunea `decode` selectează acțiunea care trebuia executată în funcție de valoarea câmpului *f* (primii trei biți din *RI*).

```
Mark1 :=
begin
  M[0:8191]<0:31>,
  RI<0:15>,
  f = RI<0:2>,
  s<0:12> = RI<3:15>,
  A<0:31>,
  CicluInstr(main) :=
  begin
    repeat
    RI = M[PC]<0:15> next
    decode f =>
    begin
      #0 := PC = M[s]
      #1 := PC = PC + M[s]
      #2 := A = A - M[s]
      #3 := M[s] = A
      #4,#5 := A = A + M[s]
      #6 := if A<0 => PC = PC + 1
      #7 := stop()
    end next
    PC = PC + 1
    end
  end
end
```

Figura 22. Descrierea procesorului Mark1 în limbajul *ISPS*.

3.3.1.4. Ordinea de execuție

În cadrul unui sistem, ordinea de execuție a diferitelor operații poate fi specificată implicit, fiind cea implicită a limbajului de descriere utilizat, sau explicit, prin construcții specifice ale limbajului. În exemplul anterior, toate instrucțiunile scrise în limbajul *ISPS* sunt executate în paralel, cu excepția cazului în care se utilizează construcții de control sau cuvântul cheie `next`, care forțează execuția secvențială a instrucțiunilor consecutive.

```
P1: PROCESS (clk)
BEGIN
  A <= B;
  B <= A;
END PROCESS P1;
```

(a)

```
B1: BLOCK (clk'EVENT AND
           CLK = '1');
BEGIN
  A <= GUARDED B;
  B <= GUARDED A;
END BLOCK B1;
```

(b)

Figura 23. Ordinea de execuție în limbajul *VHDL*: (a) secvențială; (b) paralelă.

Ordinea secvențială în cadrul limbajelor de descriere poate fi specificată implicit, prin dependența datelor, sau explicit, prin construcții de control. De exemplu, în **Figura 23(a)** se prezintă un proces descris în limbajul *VHDL*, în care instrucțiunile de asignare sunt executate secvențial. Prima instrucțiune asignează semnalului *A* valoarea semnalului *B*. Următoarea in-

strucțiune asignează semnalului *B* noua valoare a semnalului *A*. La sfârşitul procesului *P1*, ambele semnale *A* şi *B* vor avea vechea valoare a semnalului *B*.

Paralelismul este o caracteristică universală a unităților hardware, necesitând semantici implicite ale limbajelor şi construcții explicite pentru exprimarea acestuia. De exemplu, în limbajul *VHDL*, toate instrucțiunile de asignare din cadrul unui bloc sunt executate în paralel. În **Figura 23(b)**, partea dreaptă a fiecărei instrucțiuni de asignare este evaluată în paralel (utilizând vechile valori ale semnalelor), înainte de a asigna rezultatul semnalului din partea stângă. Deoarece blocul are o expresie de validare, evaluarea semnalelor precedate de cuvântul cheie `GUARDED` se efectuează atunci când expresia de validare este adevărată. Ca urmare, după frontul crescător al semnalului de ceas valorile semnalelor *A* şi *B* vor fi interschimbate. De notat că şi în cazul construcțiilor secvențiale ale limbajului, ca cea din **Figura 23(a)**, paralelismul este implicit între operațiile care nu au dependențe de control sau de date.

3.3.2. Caracteristici specifice unităților hardware

Construcțiile limbajelor de programare standard permit abstractizări funcționale, dar nu permit exprimarea proprietăților specifice unităților hardware în descrierea proiectelor. Pentru aceasta sunt necesare construcții suplimentare pentru definirea interfețelor, specificarea parțială a structurii proiectelor, specificarea operatorilor la nivelul transferurilor între registre şi la nivelul logic, a asincronismului, a ierarhiei, a comunicației între procese, a restricțiilor de proiectare şi a alocării de către utilizator.

3.3.2.1. Definirea interfețelor

Deoarece proiectul descris trebuie să obțină intrări din exterior şi să furnizeze rezultatele prelucrărilor ca ieşiri, trebuie definite porturile sale de intrare şi de ieşire. Definirea porturilor se referă la dimensiune (de ex., numărul de biți), mod (de ex., intrare, ieşire sau intrare/ieşire) şi caracteristici hardware (de ex., dacă portul este bufferat sau cu trei stări).

În **Figura 24** se prezintă o specificație în limbajul *DSL* a unui circuit de exponențiere, în care declarațiile din secțiunea `INTERFACE` specifică porturile şi atributele lor.

3.3.2.2. Declarații structurale

Declarațiile structurale permit specificarea registrelor, acumulatoarelor, numărătoarelor şi a altor structuri hardware care se vor utiliza ca variabile în cadrul limbajului de descriere. Aceste declarații sunt utile atunci când este necesară o anumită structură parțială, proiectată în prealabil, şi trebuie utilizate în mod explicit pentru specificarea funcționării proiectului. De exemplu, procesoarele au în mod normal registrele arhitecturale (ca numărătoarele de program sau registrele generale) fixate înaintea descrierii funcționării procesorului; aceste registre arhitecturale sunt declarate ca structuri fizice şi pot fi utilizate ca variabile în cadrul descrierii funcționale.

3.3.2.3. Operatori la nivelul RT şi logic

În plus față de operatorii aritmetici şi booleeni ai unui limbaj de programare, este necesară existența unor primitive corespunzătoare unităților hardware la nivelul transferurilor între registre (RT) şi la nivelul logic. Exemple de operatori pentru nivelul RT sunt cei de in-

crementare şi decrementare pentru variabile. Ca exemple de operatori logici la nivel de bit se amintesc cei pentru deplasare, rotire şi operații logice, şi operatorii pentru extragerea şi concatenarea şirurilor de biți.

```
CIRCUIT expon;

INTERFACE
vcc          : 12 V;
gnd          : GND;
in (15..0)   : INPUT FANIN 1;
out (15..0)  : OUTPUT FANOUT 10;
enable       : INPUT FANIN 1;
clk          : CLOCK FANIN 1;

POWER       100 mW;
VOLTAGE     12.00 V;
TECHNOLOGY  CMOS;
AREA        30 sqmm;
FREQUENCY   0 TO 500 kHz;
CLOCKBASE   clk;

PERFORMED FUNCTION
  out := #exp(in);
  CONTROL
    (enable := 0 CYCLES = 1);
    (enable := 1 CYCLES = 48);
END;

VAR x,y : FIXED (8,8);
    i   : LOGICAL (4..0);

APPLICATIVE
  out := y;
  IF enable = 0 THEN x := 1,
                     y := 1
                ELSE Start calc;
  FI;
END APPLICATIVE;

IMPERATIVE calc;
  (FOR i:= 0 TO 16 DO
     x := x * inp/i;
     y := y + 3;
   OD CYCLES = 3);
END IMPERATIVE;

END.
```

Figura 24. Specificația unui circuit de exponențiere în limbajul *DSL*.

3.3.2.4. Asincronismul

Funcționarea la nivelul RT este exprimată sub forma transferurilor între registre pentru fiecare ciclu de ceas. Totuşi, la nivelul RT circuitele prezintă şi caracteristici asincrone, sub forma semnalelor de setare, resetare şi de întrerupere. Pentru exprimarea caracteristicilor asincrone se pot utiliza construcții speciale ale limbajelor de descriere. O altă posibilitate este de a se grupa toate operațiile asincrone într-o descriere separată, cu semantica indicând faptul că operațiile asincrone sunt prioritare față de cele sincrone.

În limbajul *DSL* se utilizează a doua alternativă: secțiunea indicată prin cuvântul cheie `APPLICATIVE` descrie o funcționare asincronă, iar cea indicată prin `IMPERATIVE` descrie o funcționare sincronă. În exemplul anterior în care se descrie un circuit de exponențiere, semnalul *enable* din partea "aplicativă" determină în mod asincron dacă trebuie să înceapă calculul din partea "imperativă" *calc*, în mod sincron.

O formă mai generală a asincronismului apare în cazul proceselor comunicante care se execută sub controlul unor ceasuri diferite. În asemenea cazuri, evenimentele asincrone de la interfața unui proces pot întrerupe operațiile sincrone. Aceste tranziții se pot utiliza pentru a

defini automate cu stări finite asincrone, la care trecerea într-o nouă stare se realizează sub acţiunea unui semnal de intrare, şi nu a unui semnal de ceas. Exemple de limbaje cu asemenea facilităţi sunt *Statecharts*, *WAVES*, *SpecCharts* şi *BIF*.

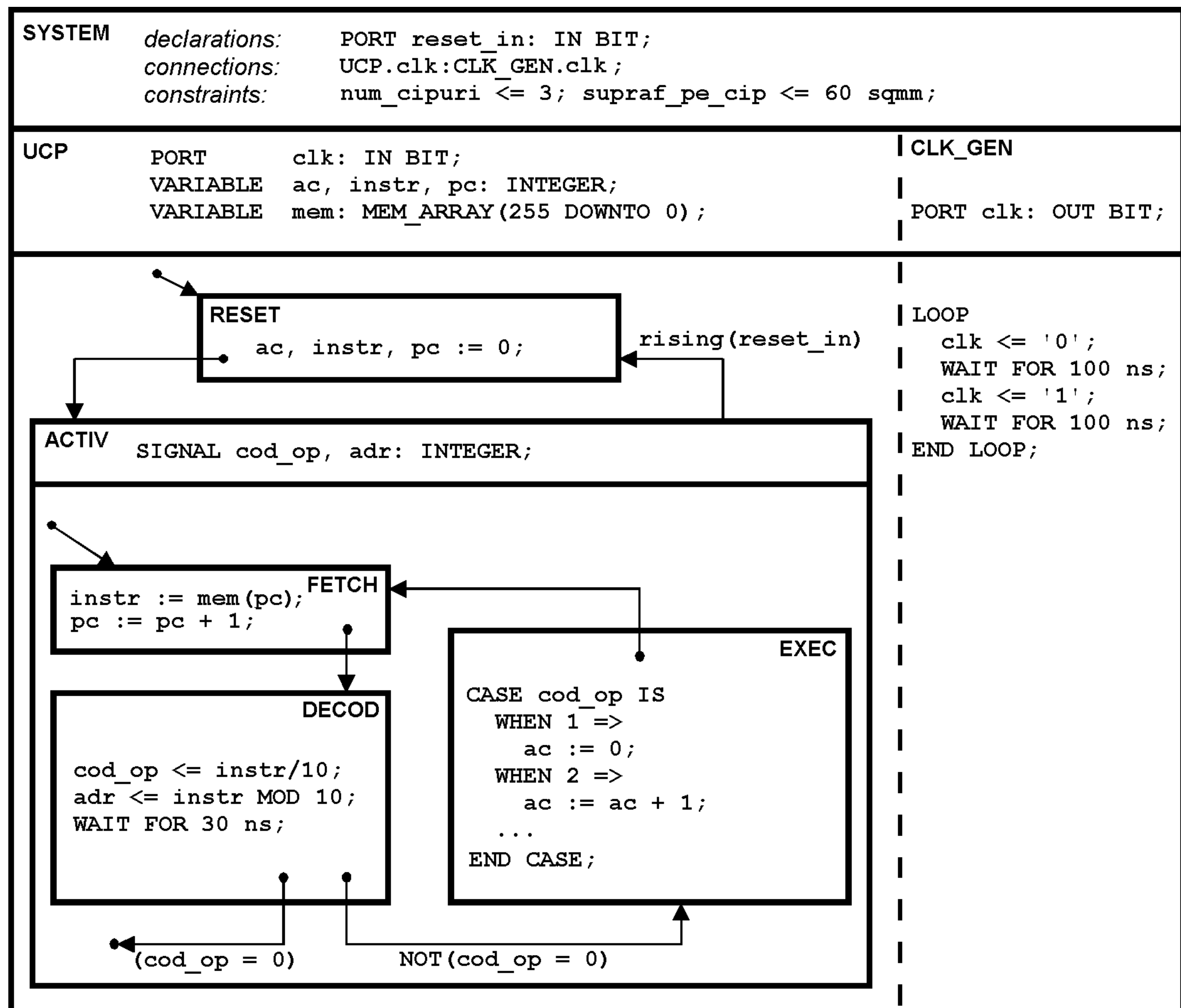


Figura 25. Ierarhia funcţională în limbajul *SpecCharts*.

În **Figura 25** se prezintă un sistem de calcul simplu descris în limbajul *SpecCharts*. Sistemul trece în mod asincron din starea *ACTIV* în starea *RESET* la frontul crescător al semnalului *RESET_IN*, indiferent de substarea în care se afla în cadrul stării *ACTIV* (*FETCH*, *DECOD,* sau *EXEC*).

3.3.2.5. Ierarhia

Pe măsură ce proiectele devin mai complexe, se utilizează descrieri ierarhice. Într-un limbaj de descriere hardware ierarhia poate avea mai multe forme: *procedurală*, *structurală*, *funcţională* şi *de proiectare*.

Pentru asigurarea *ierarhiei procedurale*, construcţiile limbajelor de programare standard utilizează funcţii şi proceduri. Această ierarhie procedurală permite descompunerea unui sistem într-un mod structurat şi permite o reprezentare concisă. În anumite limbaje de descriere hardware, o anumită funcţie poate fi înglobată într-o structură, şi poate fi utilizată ulterior

în cadrul ierarhiei structurale. În **Figura 24** se observă modul în care se realizează aceasta în limbajul *DSL*, utilizând construcția `PERFORMED FUNCTION`.

Ierarhia structurală specifică interconectarea proceselor comunicante prin semnale globale şi porturi ale proceselor. Considerând un sistem care la nivelul RT constă dintr-un set de registre şi o unitate aritmetică şi logică, o listă de conexiuni ierarhică pentru acest sistem va descrie interconexiunile componentelor la nivelul superior, fiecare componentă (de ex., UAL) fiind descompusă într-o listă de conexiuni structurală între porțile logice componente. Descrierile structurale în limbajul *VHDL*, de exemplu, permit acest tip de ierarhie.

Pentru sisteme mai complexe, aceste abstractizări ierarhice funcționale şi structurale nu mai sunt suficiente. Acest lucru este adevărat în special pentru sistemele bazate pe stări, la care modificările unor semnale de intrare pot determina tranziții imediate la diferite secvențe care descriu funcționarea. În asemenea cazuri, se utilizează *ierarhia funcțională* pentru a exprima în mod concis funcțiile complexe.

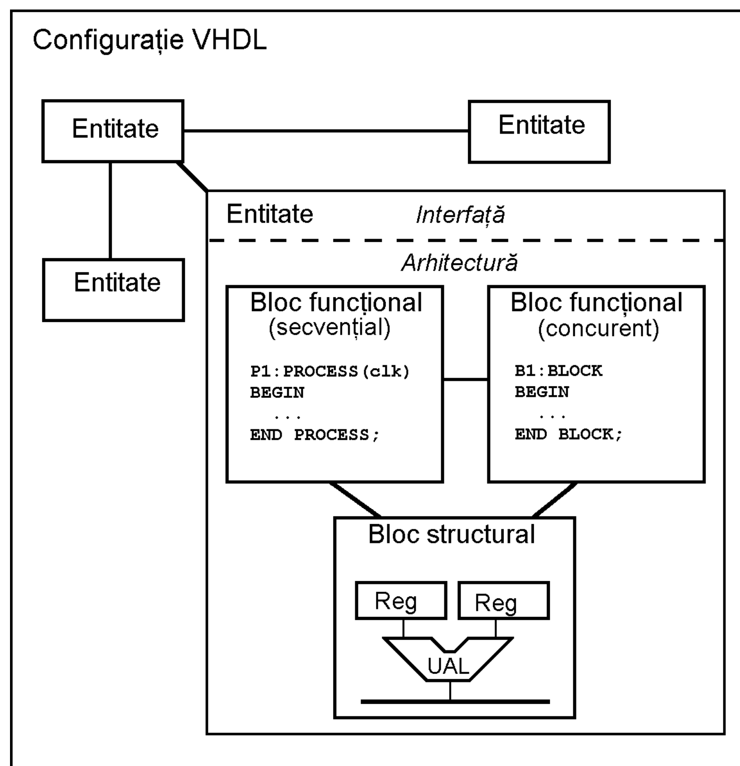


Figura 26. Ierarhia de proiectare în limbajul *VHDL*.

În **Figura 25** se prezintă un exemplu de ierarhie funcțională pentru un sistem de calcul descris în limbajul *SpecCharts*. Linia întreruptă din figură indică faptul că sistemul de calcul *SISTEM* este compus din două procese concurente, procesorul *UCP* şi generatorul de ceas *GEN_CLK*. *UCP* constă din două stări principale: *RESET*, care este starea inițială implicită a *UCP*, şi *ACTIV*, în care procesorul este operațional. Starea *ACTIV* este compusă din trei substări *FETCH*, *DECOD* şi *EXEC*, care descriu funcționarea detaliată a procesorului. Ierarhia funcțională din acest exemplu permite descompunerea unei stări în substări şi procese concurente, şi permite de asemenea descrierea asincronismului global printr-o notație care

utilizează evenimentele. De exemplu, evenimentul *rising(RESET_IN)* forţează procesul *UCP* în starea *RESET*, indiferent de substarea din cadrul stării *ACTIV* în care se află.

Ierarhia de proiectare specifică modul în care este compus întregul sistem din componente, procese comunicante etc. În **Figura 26** se indică ierarhia de proiectare reprezentată de o configuraţie din limbajul *VHDL*. Fiecare entitate de proiectare este descompusă ierarhic în blocuri, care pot fi funcţionale sau structurale

3.3.2.6. Comunicaţia între procese

La un nivel de abstractizare mai înalt, procesoarele din cadrul unei arhitecturi sunt reprezentate prin procese. În cadrul limbajelor de descriere hardware sunt necesare mecanisme care permit specificarea comunicaţiei între procesele care interacţionează. Pentru sisteme în întregime sincrone, această comunicaţie poate fi înglobată în descrierea funcţională a proceselor. În asemenea cazuri, utilizatorul poate descrie în mod explicit comunicaţia utilizând construcţiile standard ale limbajului respectiv, pentru a forţa operaţiile de citire şi scriere în cadrul ciclurilor de ceas corecte.

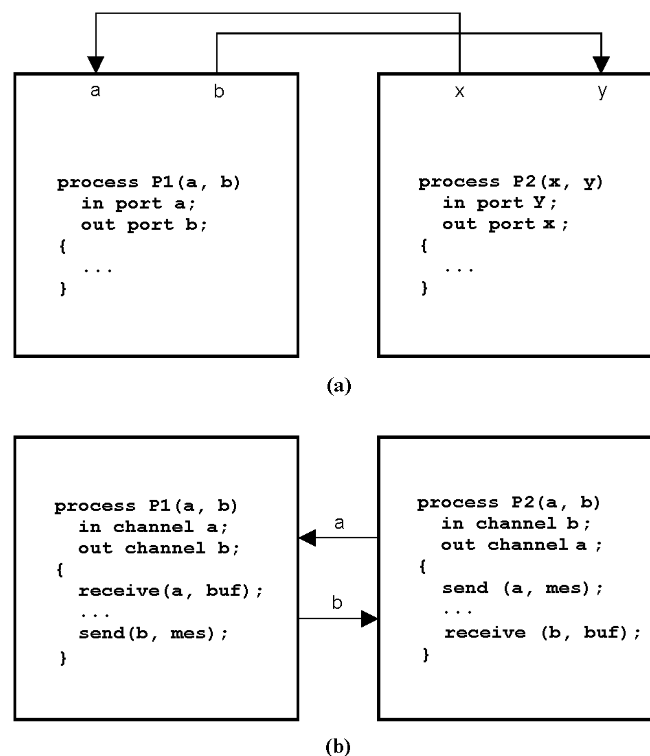


Figura 27. Sincronizarea între procese în limbajul *HardwareC*: (a) prin transmiterea parametrilor; (b) prin transmiterea mesajelor.

Dacă procesele care comunică între ele nu sunt sincrone, sunt necesare protocoale specifice pentru a obţine sincronizarea între acestea. Există două metode principale pentru realizarea acestei sincronizări: prin *medii partajate* şi prin *transmitere de mesaje*.

Sincronizarea prin *medii partajate* este realizată utilizând conexiuni sau memorii partajate care sunt accesibile simultan de procesele comunicante. În limbajul *HardwareC*, de exemplu, sincronizarea prin medii partajate este realizată prin transmiterea parametrilor utilizând porturi interconectate ale proceselor. În **Figura 27(a)** se arată modul în care data

scrisă în portul *b* al procesului *P1* este accesibilă de procesul *P2* prin portul *y*. Similar, data scrisă de procesul *P2* în portul *x* poate fi citită la portul *a* al procesului *P1*.

Transmiterea mesajelor în cadrul limbajelor de descriere este realizată prin utilizarea primitivelor pentru descrierea sincronizării între proceselor comunicante. În **Figura 27(b)** se arată modul în care poate fi descrisă în limbajul *HardwareC* sincronizarea prin transmiterea mesajelor. Canalele *a* şi *b* specifică mediul de comunicaţie între procesele *P1* şi *P2*. Primitivele de sincronizare `send` şi `receive` ale limbajului *HardwareC* sunt utilizate pentru cele două canale pentru a specifica transferul datelor între procese. Primitiva `receive(a,buf)` din procesul *P1* determină ca procesul *P1* să aştepte până când procesul *P2* transmite data prin canalul *a* cu primitiva `send(a,mes)`. Un protocol de tip cerere-confirmare poate fi exprimat într-un limbaj de descriere prin construcţii similare cu aceste primitive.

3.3.2.7. Restricţii

Restricţiile din cadrul proiectării ghidează sinteza proiectului spre realizări fezabile din punct de vedere al performanţelor, costului, testabilităţii, fiabilităţii şi a unor limitări fizice. Restricţiile de proiectare pot fi specificate separat faţă de descrierile funcţionale sau pot fi intercalate cu acestea în cadrul limbajului. *Restricţiile fizice* (de ex., numărul de pini, tehnologia sau tensiunile) sunt specificate de obicei separat faţă de descrierile funcţionale, fie în cadrul declaraţiilor, fie într-un fişier separat. De exemplu, în **Figura 24** care prezintă descrierea circuitului de exponenţiere se arată modul în care secţiunea de declaraţii a limbajului *DSL* permite specificarea diferitelor tipuri de restricţii fizice, ca puterea consumată, tensiunea de alimentare, tehnologia sau spaţiul ocupat.

Restricţiile de temporizare sunt adesea critice pentru obţinerea funcţionării corecte, ca şi pentru specificarea cerinţelor de performanţă ale proiectelor. Aceste restricţii pot fi specificate în cadrul descrierii interfeţelor, sau pot fi intercalate cu descrierile funcţionale. În ambele cazuri, restricţiile de temporizare sunt specificate de obicei ca întârzieri. În cadrul sistemelor sincrone, întârzierile sunt specificate în mod normal relativ la ceasul sistemului, ca multipli ai ciclurilor de ceas. În cadrul descrierii aceluiaşi circuit de exponenţiere, se prezintă modul în care sunt specificate în limbajul *DSL* restricţiile asupra performanţelor: în procedura imperativă `calc`, bucla `do` trebuie să se execute în cadrul a trei cicluri de ceas.

La descrierea comunicaţiei între procese, este necesar să se specifice întârzieri relative la evenimente specifice. Aceste întârzieri reprezintă valori absolute de timp necesare pentru implementarea corectă a protocolului de comunicaţie. De exemplu, într-un protocol simplu al ciclului de citire pentru o placă de memorie, trebuie să se stabilească mai întâi valoarea adresei şi să se activeze semnalul de citire al memoriei, iar apoi să se efectueze o cerere întârziată de acces la magistrala de date, dându-se timp memoriei pentru a reacţiona la cererea de citire înainte de a prelua controlul asupra magistralei de date. În **Figura 29** se prezintă un exemplu al acestui protocol de citire descris în limbajul *BIF*. În starea 1, semnalul *BusReq* este activat cu 175 ns după evenimentul *Falling(MemReq)* care a determinat trecerea în această stare, deci după stabilirea adresei *Adr* şi a semnalului *MR*.

Întârzierile cu valori absolute pot fi utilizate şi pentru a specifica restricţii asupra performanţelor, întârzieri ale căilor de date şi valori de timeout. Limbajele de simulare ca *VHDL* permit specificarea întârzierilor absolute utilizând două tipuri de întârzieri: inerţiale şi de transport. O *întârziere inerţială* reprezintă inerţia unei componente, şi necesită ca semnalul de intrare să dureze o perioadă specificată de timp înaintea activării ieşirii (de ex., timpii de seta-

re şi de menţinere pentru un bistabil). O *întârziere de transport* reprezintă întârzierea unei conexiuni; în acest caz, o modificare a intrării este propagată întotdeauna la ieşire.

3.3.2.8. Alocarea de către utilizator

Sinteza de nivel înalt reprezintă un proces de alocare şi de atribuire a entităţilor funcţionale abstracte structurilor de la nivelul transferurilor între registre. Chiar dacă acest proces poate fi efectuat complet automat, proiectanţii experimentaţi pot oferi soluţii de creştere a calităţii structurilor implementate, prin detectarea unor secţiuni critice şi îmbunătăţirea acestora. Limbajul de descriere utilizat trebuie să permită specificarea unor asemenea informaţii pentru unele soluţii parţiale sub forma alocărilor şi a asignărilor de către utilizator.

Alocările pot fi specificate ca structuri RT în declaraţiile limbajelor. Pot fi specificate patru tipuri de asignări ale utilizatorului, pentru *stări*, *registre*, *unităţi funcţionale* şi *conexiuni*. Acestea pot fi definite atât în cadrul declaraţiilor cât şi în cadrul descrierilor funcţionale.

Asignarea stărilor atribuie o operaţie din descrierea funcţională unei stări a structurii implementate; operaţia poate fi apoi notată cu starea respectivă. Anumite limbaje de descriere hardware sunt bazate pe stare (de ex., *BIF*); asignarea stărilor este realizată automat prin descrierea unei operaţii în starea corespunzătoare. Asignarea registrelor, a unităţilor funcţionale şi a conexiunilor atribuie variabile, operaţii şi instrucţiuni de asignare registrelor, unităţilor funcţionale, respectiv conexiunilor.

Aceste asignări pot fi descrise prin adnotarea variabilelor, a operaţiilor limbajului şi a instrucţiunilor de asignare prin componentele şi conexiunile alocate, utilizând construcţii speciale ale limbajelor. De exemplu, limbajele *BIF* şi *ISPS* utilizează paranteze în cadrul descrierilor pentru a indica asignarea componentelor. Limbajele *HardwareC* şi *MIMOLA* permit de asemenea alocarea şi asignarea de către utilizator.

3.4. Formate ale limbajelor de descriere hardware

Modul în care un proiect este modelat şi descris are un efect direct asupra implementării sale finale din punct de vedere al costului şi al performanţelor. O modelare necorespunzătoare va determina o implementare de calitate scăzută, indiferent de sistemul de proiectare utilizat. De asemenea, erorile conceptuale care nu sunt detectate în primele faze ale ciclului de proiectare pot conduce la eforturi costisitoare pentru a le detecta pe măsură ce proiectarea avansează.

Pentru a asista modelarea şi descrierea proiectelor, sunt necesare diferite formate ale limbajelor de descriere care sunt convenabile pentru diferiţi utilizatori şi diferite aplicaţii. La nivelele inferioare ale procesului de proiectare, descrierile grafice au fost preponderente pentru un timp îndelungat. Limbajele de descriere funcţionale au de obicei o formă textuală, deoarece aceste limbaje s-au dezvoltat din limbajele de programare. Totuşi, chiar şi pentru descrierile de nivel mai înalt o combinaţie a formelor de descriere este mai eficientă pentru specificarea proiectelor. De exemplu, interfeţele şi protocoalele sunt descrise într-un mod mai natural prin diagrame de timp; automatele cu un număr relativ redus de stări sunt descrise în mod concis prin diagrame de stare sub formă tabelară sau grafică; descrierile pur funcţionale sunt exprimate cel mai convenabil într-un format textual.

În continuare se trec în revistă formatele textuale, grafice, tabelare şi cele bazate pe diagrame de timp ale limbajelor de descriere hardware.

3.4.1. Limbaje textuale

Cele mai multe limbaje de descriere au preluat sintaxa şi o parte a semanticii de la limbajele de programare de nivel înalt ca *Pascal*, *ADA* sau *C*. Limbajele textuale pot exprima în mod succint descrierile funcţionale care conţin instrucţiuni de asignare împreună cu transformări complexe ale datelor (de ex., operatori aritmetici şi logici). Aplicaţiile cu o mare cantitate de calcule sunt descrise cel mai eficient prin utilizarea instrucţiunilor textuale de asignare.

Ca exemple de limbaje textuale de descriere hardware se amintesc *ISPS*, *Silage*, *VHDL*, *HardwareC* şi *MIMOLA*. Limbajele bazate pe logica matematică formală, ca de exemplu *HOL*, sunt exprimate de asemenea sub formă textuală.

3.4.2. Limbaje grafice

Descrierile funcţionale pot fi exprimate eficient şi sub formă grafică. În particular, ordinea de execuţie a operaţiilor, paralelismul şi fluxul de control sunt înţelese cu uşurinţă dacă se utilizează un format grafic. O organigramă funcţională este un asemenea exemplu, în care fluxul de control este exprimat sub formă grafică, iar operaţiile sunt descrise utilizând instrucţiuni textuale de asignare.

În cazul în care complexitatea sistemului este relativ redusă, limbajele bazate pe organigrame sunt utile pentru descrierea acestuia. Notaţia *ASM* şi *EXEL* sunt două exemple de asemenea limbaje. Sistemele mai complexe pot fi descrise în mod ierarhic. Limbajele *Statecharts* şi *SpecCharts* pun la dispoziţie un formalism concis pentru asemenea sisteme. În **Figura 25** se indică un exemplu de descriere grafică a unui procesor simplu utilizând limbajul *SpecCharts*. Descrierile bazate pe reţele Petri sunt de asemenea utilizate pentru descrierea grafică a proiectelor hardware; un exemplu este limbajul *GDL*.

3.4.3. Limbaje tabelare

Descrierile tabelare reprezintă o notaţie concisă pentru specificaţiile funcţionale ale modelelor bazate pe stări, în special pentru automatele cu stări finite cu cale de date, unde partea bazată pe stări a modelului poate fi exprimată în mod clar sub forma unei tabele de stări, iar operaţiile căii de date pot fi exprimate sub formă textuală, fiind adăugate părţilor corespunzătoare ale tabelului de stări. Un exemplu este limbajul *BIF* (Behavioral Intermediate Form). În acest limbaj, secvenţierea stărilor este descrisă într-un format tabelar, dar operaţiile din fiecare stare sunt descrise utilizând expresii textuale.

Figura 28 prezintă schema bloc a unui sistem compus dintr-o unitate centrală de prelucrare (UCP), un controler de magistrală şi o placă de memorie. Placa de memorie este compusă dintr-un controler de memorie şi o memorie ROM. În cadrul plăcii de memorie, comunicaţia între controlerul de memorie şi memoria ROM se realizează utilizând un semnal de adresă *Adr*, un semnal *MR* care validează memoria ROM, şi o magistrală internă de date *Data*. La nivelul sistem, se utilizează mai multe semnale şi magistrale pentru a realiza un protocol de tip „handshake”: *ABus* reprezintă liniile de adresă ale magistralei externe, *DBus* este magistrala de date, *DataRdy* indică prezenţa datelor valide pe magistrala de date a memoriei ROM, iar *BusReq* este un semnal de protocol pentru magistrala de date. Toate semnalele de protocol sunt active la nivel coborât.

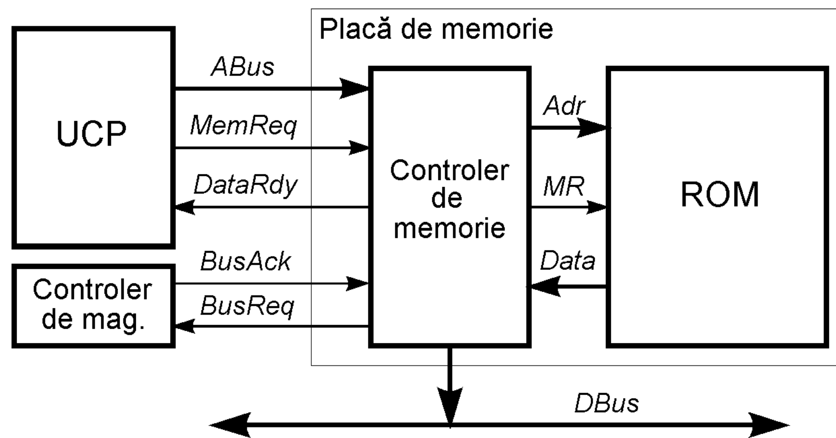


Figura 28. Schema-bloc a unui sistem format din UCP şi o placă de memorie.

Figura 29 prezintă un protocol simplu pentru un ciclu de citire al memoriei utilizând o descriere tabelară bazată pe stări în limbajul *BIF*. Ciclul de citire este iniţiat în starea 1, când semnalul *MemReq* ajunge la valoarea 0 şi identificatorul plăcii de memorie (*Id_Placa*) coincide cu biţii de adresare (16..18) ai magistralei de adrese *ABus*. În acest moment, memoria este validată, registrul intern de adresă este încărcat cu valoarea de pe magistrala de adrese, şi placa de memorie preia controlul asupra magistralei de date *DBus* prin setarea semnalului *BusReq* la zero. Evenimentul *Falling(BusAck)* determină o tranziţie în starea 2, în care placa de memorie transmite data adresată *Data* pe magistrala *DBus*. Protocolul este completat în starea 3, în care placa de memorie este dezactivată.

Stare prezentă	Cond.	Val	Acţiuni	Stare următ.	Eveniment
0		T	DBus = 'X'; DataRdy = 1; BusReq = 1;	1	Falling (MemReq)
1	ABus (18..16) == Id_Placa	T	MR = 0; Adr = ABus; BusReq (delay 175 ns) = 0;	2	Falling (BusAck)
		F		1	Falling (MemReq)
2		T	BusReq = 1; DBus = Data; DataRdy = 0;	3	Rising (MemReq)
3		T	MR = 1; Adr = 'X';	0	Rising (BusAck)

Figura 29. Ciclul de citire pentru placa de memorie, descris în limbajul *BIF*.

3.4.4. Limbaje bazate pe diagrame de timp

Protocolul pentru citirea memoriei descris în secțiunea precedentă poate fi exprimat şi sub forma diagramelor de timp. Aceste diagrame sunt utilizate în mod frecvent pentru descrierea interfeţelor, a protocoalelor şi a restricţiilor de timp asociate. Diagramele de timp pot reprezenta în mod grafic modificările semnalelor (evenimente ca *rising* sau *falling*), pot indica secvenţierea evenimentelor, şi pot pune în evidenţă relaţiile temporale între evenimente.

Au fost dezvoltate mai multe limbaje bazate pe editoare ale diagramelor de timp, de exemplu *Waves* şi *XWAVE*.

3.5. Exemple de limbaje de descriere

3.5.1. VHDL

Limbajul *VHDL* (*VHSIC Hardware Description Language*) se bazează pe limbajul de programare *ADA*. Dezvoltarea limbajului *VHDL* a fost iniţiată de către Departamentul Apărării al S.U.A., cu intenţia de a crea un limbaj standard pentru proiectul *VHSIC* (*Very High Speed Integrated Circuit*). Scopul acestui proiect a fost de a produce următoarea generaţie de circuite integrate. În procesul dezvoltării circuitelor integrate complexe proiectanţii au constatat că sistemele de proiectare pe care le aveau la dispoziţie nu erau corespunzătoare, fiind bazate pe proiectarea la nivelul porţilor logice. A apărut necesitatea unei noi metode de descriere, fiind propus un nou limbaj de descriere, *VHDL*.

Scopul noului limbaj propus era în primul rând de a descrie circuitele complexe care erau proiectate. În al doilea rând, se dorea un limbaj standard care să permită distribuirea proiectelor între toţi participanţii la proiectul *VHSIC* într-un format standard. În anul 1986 limbajul a fost propus pentru standardizare în cadrul IEEE, iar după o serie de revizuiri şi modificări a fost adoptat ca standardul IEEE 1076-1987.

Entitate şi arhitectură

Principala abstractizare în limbajul *VHDL* este *entitatea de proiectare*, care se utilizează pentru identificarea şi reprezentarea unei singure părţi a unui proiect, care execută o funcţie specifică şi are intrări şi ieşiri bine definite. Un model *VHDL* constă din cel puţin o entitate de proiectare, care este separată într-o declaraţie de entitate şi cel puţin o *arhitectură*. Declaraţia de entitate conţine descrierea interfeţei modelului cu exteriorul, descrierea atributelor, şi alte descrieri comune tuturor arhitecturilor entităţii. O entitate poate avea mai multe arhitecturi. O arhitectură reprezintă o implementare posibilă a modelului. Pot exista diferite implementări care rezultă din diferite variante de proiectare sau din diferite nivele de abstractizare în cadrul procesului de proiectare.

Limbajul *VHDL* este un limbaj de descriere multi-nivel, şi permite diferite tipuri de descrieri ale unui model hardware: *funcţionale*, *structurale* şi de tipul *fluxului de date*, fiind posibilă utilizarea unei combinaţii a acestor descrieri în cadrul unui model. Descrierea funcţională reprezintă o descriere algoritmică a funcţionării sistemului hardware. O descriere structurală conţine o listă a unor blocuri interconectate, fiecare bloc reprezentând un subsistem. Al treilea tip de descriere se utilizează pentru reprezentarea fluxului datelor din

cadrul sistemului. Un exemplu tipic este o descriere a transferurilor între registre, unde datele sunt transferate între registre şi obiecte complexe similare.

Pentru ilustrarea diferitelor tipuri de descrieri, se prezintă ca exemplu un numărător modulo 10. Declaraţia de entitate este identică în toate cazurile, fiind urmată de definirea arhitecturii specifice tipului de descriere respectiv. Se prezintă în **Figura 30** descrierea funcţională.

```
ENTITY Numarator_E IS
  PORT (clk: IN BIT; cont: OUT INTEGER);
END Numarator_E;

ARCHITECTURE Numarator_func OF Numarator_E IS
BEGIN
  PROCESS
    VARIABLE numvar: INTEGER := 0;
  BEGIN
    cont <= numvar;
    WAIT UNTIL (clk = '1') AND NOT(clk'STABLE);
    IF (numvar = 9) THEN
      numvar := 0;
    ELSE
      numvar : = numvar + 1;
    END IF;
  END PROCESS;

END Numarator_func;
```

Figura 30. Descrierea funcţională a unui numărător modulo 10 în limbajul *VHDL*.

Definiţia entităţii specifică porturile de intare şi de ieşire. Numărătorul este descris ca având două porturi externe: *clk*, de tipul `BIT`, şi *cont*, de tipul `INTEGER`. `BIT` este un tip enumerat predefinit (`'0'`, `'1'`).

Definiţia arhitecturii constă, în general, dintr-o parte declarativă şi o parte care conţine instrucţiuni. Partea declarativă este vidă în acest exemplu. Corpul arhitecturii conţine o instrucţiune `PROCESS`, care este construcţia limbajului prevăzută pentru descrierea funcţionării unei componente hardware sau chiar a unui sistem. Această construcţie poate conţine şi o parte declarativă.

Descrierea structurală a numărătorului corespunde unei liste de conexiuni între componentele acestuia. În partea declarativă a arhitecturii, se definesc componentele utilizate (*Reg_E*, *Sum_E* şi *Cmp_E*), şi porturile formale ale acestora. Componentele sunt instanţiate apoi în corpul arhitecturii (*RegistruNum*, *Sumator* şi *Comparator*). Interconexiunile dintre componente sunt specificate prin asocierea porturilor acestora, utilizând clauza `PORT MAP`. De exemplu, semnalul *num_out* este asociat cu porturile formale *o* şi *a* ale componentelor *RegistruNum* şi respectiv *Sumator*, specificând astfel că ieşirea componentei *RegistruNum* este conectată la una din intrările componentei *Sumator* (**Figura 31**).

```
ENTITY Numarator_E IS
  PORT (clk: IN BIT; cont: OUT INTEGER);
END Numarator_E;

ARCHITECTURE Numarator_struct OF Numarator_E IS

  COMPONENT Reg_E
    PORT (d: IN INTEGER; clk: BIT; o: OUT INTEGER;
          clear: IN BIT);
  END COMPONENT;

  COMPONENT Sum_E
    PORT (a, b: IN INTEGER; o: OUT INTEGER);
  END COMPONENT;

  COMPONENT Cmp_E
    PORT (i0, i1: IN INTEGER; o: OUT INTEGER);
  END COMPONENT;

  SIGNAL unu: INTEGER := 1;
  SIGNAL noua: INTEGER := 9;
  SIGNAL num_in, num_out, sum_out: INTEGER;
  SIGNAL clear: BIT;

BEGIN
  RegistruNum: Reg_E
    PORT MAP (num_in, clk, num_out, clear);
  Sumator: Sum_E
    PORT MAP (num_out, unu, sum_out);
  Comparator: Cmp_E
    PORT MAP (noua, num_in, clear);

  cont <= num_out;

END Numarator_struct;
```

Figura 31. Descrierea structurală a unui numărător modulo 10 în limbajul *VHDL*.

Arhitectura de tipul fluxului de date conține o declarație locală a semnalului *nums* şi o construcție BLOCK (**Figura 32**).Execuția tuturor instrucțiunilor din cadrul acestei construcții este inițiată în paralel. Instrucțiunile de asignare a valorilor la semnale se bazează pe ecuațiile booleene corespunzătoare. O asemenea instrucțiune se execută în două etape. Expresia din partea dreaptă a ecuației este evaluată după fiecare modificare a valorii unei variabile sau semnal, şi valoarea rezultată va fi asignată semnalului după întârzierea specificată. Orice modificare a valorii expresiei în acest interval de timp nu are (în general) nici un efect asupra unei asignări anterioare. Excepția apare dacă întârzierea ultimei asignări este mai mică sau egală cu cea a precedentei asignări şi asignarea valorii precedente nu a fost încă executată.

Cele trei dimensiuni ale timpului în limbajul *VHDL* sunt reprezentate sub forma a trei axe din **Figura 33**. Modelul de simulare este cel al unui simulator bazat pe evenimente, astfel că axa timpului real reflectă avansul timpului sub forma evenimentelor discrete.

```
ENTITY Numarator_E IS
  PORT (clk: IN BIT; cont: OUT INTEGER);
END Numarator_E;

ARCHITECTURE Numarator_data OF Numarator_E IS

  SIGNAL nums: INTEGER := 0;
BEGIN
  BLOCK ((clk = '1') AND NOT(clk'STABLE))
  BEGIN
    nums <= GUARDED 0 WHEN (nums = 9)
            ELSE nums + 1;
  END BLOCK;
  cont <= nums;

END Numarator_data;
```

Figura 32. Descrierea de tipul fluxului de date a unui numărător modulo 10 în limbajul *VHDL*.

Întârzierile delta de pe axa a doua a timpului asigură tratarea cazurilor în care instrucțiunile de asignare au întârzieri zero. Fiecare asignare a unei valori la un semnal este împărțită într-o activitate de inițiere şi o activitate de execuție. Execuția are loc cel puțin cu o întârziere delta după inițiere. Ambiguitățile determinate de asignările multiple la un semnal trebuie soluționate prin definirea unei *funcții de decizie* de către utilizator. Se prezintă în continuare un exemplu în care apar instrucțiuni de asignare cu întârzieri zero.

```
BEGIN                      -- bloc în care toate instrucțiunile
                           -- sunt inițiate în mod concurent
  semnal_a <= semnal_b;    -- asignare cu întârziere zero
  semnal_b <= semnal_a;    -- asignare cu întârziere zero
END
```

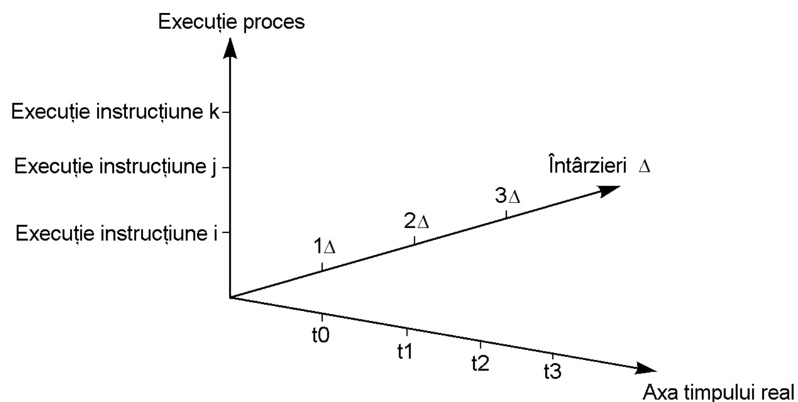


Figura 33. Cele trei dimensiuni ale timpului în limbajul *VHDL*.

Dacă execuția blocului este inițiată la momentul t_0, ambele instrucțiuni de asignare sunt executate. Semantica limbajului *VHDL* cere ca evaluarea ambelor expresii din partea dreaptă a instrucțiunilor şi asignarea valorilor să aibă loc în cadrul ciclului de simulare $t_0+\Delta$. Ca rezultat, valorile celor două semnale vor fi interschimbate. Acest mecanism asigură ca

execuția unui program *VHDL* să fie independentă de implementările particulare ale simulatoarelor.

În general, întârzierile delta sunt utilizate pentru a ordona anumite evenimente în timpul simulării, în particular a celor cu întârzieri zero. Dacă aceste evenimente nu sunt ordonate în mod corespunzător, rezultatele pot fi diferite între diferite execuții.

Limbajul *VHDL* face distincția între *întârzierile inerțiale* şi cele *de transport*. Primul tip de întârziere este implicit şi ține cont de inerția circuitelor, astfel că modificarea unui semnal de ieşire necesită ca semnalele de intrare corespunzătoare să fie menținute un anumit timp. Al doilea tip, indicat prin cuvântul cheie `transport`, este utilizat pentru a descrie modificarea semnalelor fără a ține cont de inerție.

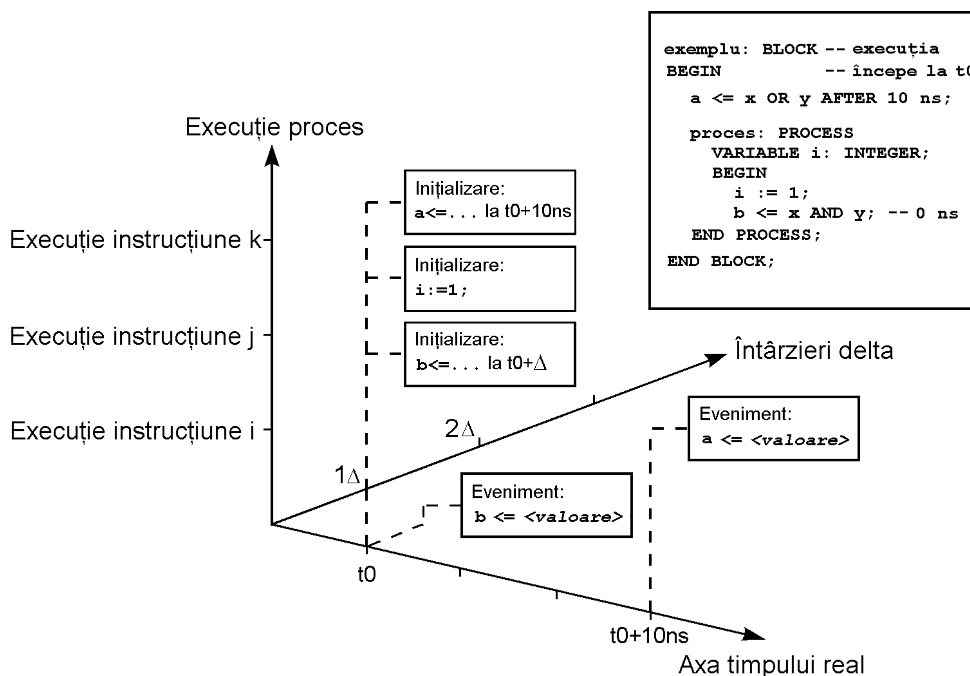


Figura 34. Modelul timpului în limbajul *VHDL*.

A treia axă a timpului (**Figura 34**) reflectă execuția instrucțiunilor în cadrul proceselor. De exemplu, un proces descrie un algoritm fără a specifica întârzieri; asignarea unor valori variabilelor, de exemplu, nu consumă timp. Într-un model *VHDL*, nu există variabile globale. În caz contrar, ar trebui specificat modul în care se execută asignările multiple la o variabilă în acelaşi timp, pentru a evita execuția dependentă de simulator a unui program *VHDL*.

Configurații

Configurațiile se utilizează pentru a descrie entitățile de proiectare care formează un model hardware complet. O configurație specifică arhitectura utilizată pentru modelarea unei entități la fiecare nivel de proiectare şi asociază instanțieri de componente entităților. În **Figura 35** se prezintă o declarație a unei configurații pentru entitatea *Numarator_E* descrisă anterior.

```
LIBRARY work;
CONFIGURATION Config_Numarator OF Numarator_E IS
USE work.ALL;
  FOR Numarator_struct
    FOR RegistruNum: Reg_E
      USE ENTITY reg_2;
    END FOR;
    FOR Sumator: Sum_E
      USE ENTITY sum_2;
    END FOR;
    FOR Comparator: Cmp_E
      USE ENTITY cmp_12;
    END FOR;
  END FOR;
END Config_Numarator;
```

Figura 35. Exemplu de configurație în limbajul *VHDL*.

Configurația declarată specifică faptul că pentru entitatea *Numarator_E* se utilizează arhitectura *Numarator_struct*, iar pentru instanțierile componentelor numărătorului se utilizează entitățile *reg_2*, *sum_2*, respectiv *cmp_12* din biblioteca *work*. Conexiunile porturilor pentru instanțierile componentelor nu sunt specificate.

Avantajul configurațiilor este că nu este necesară recompilarea întregului proiect dacă se doreşte utilizarea unei alte arhitecturi pentru o entitate sau o altă entitate pentru o componentă, fiind suficient să se recompileze noua configurație.

Subprograme şi pachete

Subprogramele constau din *proceduri* şi *funcții*, fiind similare cu procedurile şi funcțiile din limbajele *ADA* sau *Pascal*. O procedură poate returna mai multe argumente, şi poate avea parametri de intrare, de ieşire sau de intrare/ieşire. O funcție returnează o singură valoare, toți parametrii acesteia fiind parametri de intrare. Există proceduri şi funcții *concurente*, care se află în afara unei declarații de proces sau a unui alt subprogram, sau proceduri şi funcții *secvențiale*, care există întotdeauna în cadrul unei declarații de proces sau a unui alt subprogram.

Pachetele sunt utilizate în acelaşi mod ca în limbajul *ADA*. Scopul principal al unui pachet este de a grupa elemente care pot fi partajate între mai multe unități de proiectare. Declararea unei date în cadrul unui pachet permite ca data să fie utilizată de alte entități. Un pachet constă din două părți: o secțiune de declarații şi corpul pachetului.

Secțiunea de declarații defineşte interfața pachetului, în acelaşi mod în care entitatea defineşte interfața modelului. Această secțiune poate conține declarații pentru subprograme, tipuri, constante, semnale globale, componente, atribute, clauza `use`.

Corpul pachetului defineşte subprogramele care sunt declarate în secțiunea de declarații şi specifică valorile constantelor a căror nume şi tip a fost declarat în aceeaşi secțiune. De asemenea, corpul pachetului mai poate conține şi declarații care vor fi locale, ca declarații de subprograme, tipuri, constante, clauza `use`.

```
PACKAGE valori_globale IS
  CONSTANT t_cresc:    TIME := 10 ns;
  CONSTANT t_descresc: TIME := 15 ns;
  TYPE bit_7 IS ('0', '1', 'X', 'L', 'H', 'Y', 'Z');
  TYPE bit_7vect IS ARRAY (NATURAL RANGE <>) OF bit_7;
  FUNCTION bit_val (val: bit_7) RETURN BIT;
  FUNCTION bit_7val (val: BIT) RETURN bit_7;
  FUNCTION decizie (surse: bit_7vect) RETURN bit_7;
END valori_globale;

...

PACKAGE BODY valori_globale IS
  FUNCTION bit_val (val: bit_7) RETURN BIT IS
    CONSTANT biti: bit_7vect := "0100000";
  BEGIN
    RETURN biti (bit_7'POS(val));
  END;
...
END valori_globale;
```

Figura 36. Exemplu de declarație `PACKAGE` în limbajul *VHDL*.

Pachetul `valori_globale` din **Figura 36** conține definiții de constante care pot fi utilizate pentru specificarea unor întârzieri, şi definirea unei logici cu 7 valori.

Tipuri

Limbajul *VHDL* dispune de următoarele categorii de tipuri: întregi, reale, enumerate, fizice (de ex., curent, tensiune, timp), tablouri, înregistrări, pointeri (tipuri de acces), fişiere. Unele din aceste tipuri sunt predefinite, existând şi posibilitatea definirii tipurilor de către utilizatori.

Prin declarații de *subtipuri* se pot defini subseturi ale unor tipuri. În mod obişnuit, declararea subtipurilor se utilizează pentru a adăuga restricții la tipurile existente. În **Figura 37** se prezintă exemple cu definiții de tipuri.

În limbajul *VHDL*, *ierarhia structurală* este asigurată prin utilizarea blocurilor şi a declarațiilor de instanțiere a componentelor. Limbajul asigură două nivele de *ierarhie funcțională*. La nivelul superior, specificația poate fi descompusă într-un set de procese care se execută concurent, asigurând astfel şi *concurența* la nivel de taskuri. Al doilea nivel constă din descompunerea secvențială a proceselor în proceduri. Prin instrucțiunile de asignare pentru semnale, se realizează concurența la nivelul instrucțiunilor.

Comunicația între procese poate fi realizată printr-un model cu memorie partajată, care utilizează semnale a căror valoare este asignată de un proces şi care pot fi utilizate de alte procese.

Sincronizarea poate fi asigurată prin două metode. Prima constă din asocierea unei liste de semnale unui proces. Astfel se asigură ca la apariția unui eveniment asupra oricărui semnal din listă, procesul va începe să se execute. De exemplu, considerăm un proces *P*, definit astfel:

```
TYPE logic IS ('X', '0', '1', 'Z');
TYPE unsigned IS ARRAY (NATURAL RANGE <>) OF logic;
TYPE time IS RANGE implementation_defined
  UNITS
    fs;
    ps  = 1000 fs;
    ns  = 1000 ps;
    us  = 1000 ns;
    ms  = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr  = 60 min;
  END UNITS;
TYPE sub_real IS -100.0 TO +100.0;
TYPE coord IS
  RECORD
    x: INTEGER;
    y: INTEGER;
  END RECORD;
SUBTYPE val_num IS INTEGER RANGE 0 TO 15;
```

Figura 37. Declarații de tipuri în limbajul *VHDL*.

```
P: PROCESS (start, x)
   BEGIN
     ...
   END PROCESS;
```

Conform acestei definiții, procesul *P* va fi suspendat până când se modifică valoarea unuia din semnalele *start* sau *x*, ceea ce permite sincronizarea execuției procesului *P* cu alte procese care conțin semnalele *start* sau *x*.

A doua metodă de sincronizare utilizează o instrucțiune `wait`, ceea ce va suspenda procesul până când se detectează apariția unui eveniment asupra unuia din semnalele specificate, sau apariția unei condiții specificate. De exemplu, următoarea instrucțiune `wait` va determina execuția procesului numai dacă apare un eveniment asupra semnalelor *x* sau *y*, sau dacă *start = 1*:

```
WAIT ON x, y UNTIL (start = '1');
```

Specificarea *temporizării* în limbajul VHDL este limitată la specificarea temporizărilor funcționale, ca de exemplu la utilizarea clauzei `after`:

```
q <= i0 AFTER 20 ns;
```

În mod similar, se poate utiliza o clauză de tip timeout, pentru a specifica timpul maxim de așteptare printr-o instrucțiune `wait`, de exemplu:

```
WAIT ON start FOR 100 ns;
```

Specificarea celui de-al doilea tip de caracteristici de temporizare, cel al restricțiilor de temporizare, nu este posibilă în mod direct în limbajul *VHDL*, deşi asemenea restricții pot fi specificate indirect prin utilizarea atributelor.

Limbajul *VHDL* nu permite exprimarea unor caracteristici ale sistemelor numerice. De exemplu, nu există construcții pentru terminarea unui proces ca răspuns la apariția unei excepții. *Excepțiile* pot fi reprezentate doar parțial, prin utilizarea expresiilor de validare

asociate cu blocurile sau cu instrucțiunile de asignare. Astfel, o expresie de validare care este asociată cu un bloc va controla instrucțiunile de asignare la semnale din cadrul blocului. De asemenea, limbajul nu permite exprimarea *tranzițiilor între stări*. Exprimarea unei ierarhii funcționale complete, în care concurența poate fi specificată la orice nivel al ierarhiei, nu este posibilă în cadrul limbajului.

3.5.2. HardwareC

HardwareC este un limbaj de descriere hardware orientat pe sinteză. Deşi se bazează pe limbajul de programare *C*, are construcții şi semantici suplimentare pentru descrierea unităților hardware. Semantica declarativă a limbajului permite definirea modulelor structurale şi a interconexiunilor dintre acestea, iar semantica procedurală a acesteia permite specificarea funcționării în timp.

În limbajul *HardwareC*, la nivelul superior specificația unui sistem constă dintr-un bloc, care este apoi descompus într-o interconexiune de blocuri structurale şi procese concurente care comunică între ele. Blocurile corespund interconexiunilor structurale a unor entități de proiectare, aceste blocuri şi interconexiunile dintre ele specificând o *ierarhie structurală*.

Concurența la nivelul taskurilor se specifică prin intermediul proceselor, fiecare dintre acestea specificând un algoritm ca un set de operații secvențiale, care pot fi descrise prin utilizarea unui subset al construcțiilor de programare ale limbajului *C*. Poate fi specificată şi *concurența la nivelul instrucțiunilor*, în mod explicit, utilizând instrucțiunea paralelă compusă. De exemplu:

```
<
 x = b + c;
 y = p - q;
>
```

Această construcție specifică execuția simultană a calculelor asociate cu asignările la variabilele *x* şi *y*.

Comunicația între procese poate fi specificată utilizând fie modelul cu memorie partajată, fie modelul cu transmitere de mesaje. *Transmiterea prin porturi*, de exemplu, va presupune existența unui mediu partajat, ca o memorie sau liniile de interconexiune. Porturile se definesc în cadrul proceselor comunicante, şi se utilizează instrucțiuni explicite pentru citirea şi scrierea acestor porturi. Protocolul care descrie comunicația se poate specifica în cadrul descrierii procesului, ca de exemplu, în **Figura 38(a)**, unde procesul *main* transmite valoarea *n* procesului *factorial* prin portul *p_n*, şi recepționează rezultatul prin portul *p_r*.

Transmiterea mesajelor utilizează construcții explicite pentru transferul datelor şi sincronizare, prin declararea unor canale de comunicație între procesele sau blocurile comunicante. Prin utilizarea acestor construcții, proiectantul trebuie să specifice numai datele care trebuie transferate prin canale, sinteza protocolului de comunicație şi a circuitului corespunzător fiind realizată automat de către sistemele de sinteză. În **Figura 38(b)**, de exemplu, procesul *main* utilizează canalul *c1* pentru transmiterea valorii *n*, şi recepționează rezultatul prin canalul *c2*.

Pentru sincronizarea a două procese, se poate utiliza comunicația prin transmiterea mesajelor. Limbajul *HardwareC* dispune de o construcție `msgwait`, care detectează mesajele în curs de transmisie care sunt în aşteptare. Prin utilizarea acestei construcții, un proces poate

fi pus în aşteptare până la recepţionarea unui semnal, sub forma unui mesaj de un singur bit, de la un alt proces.

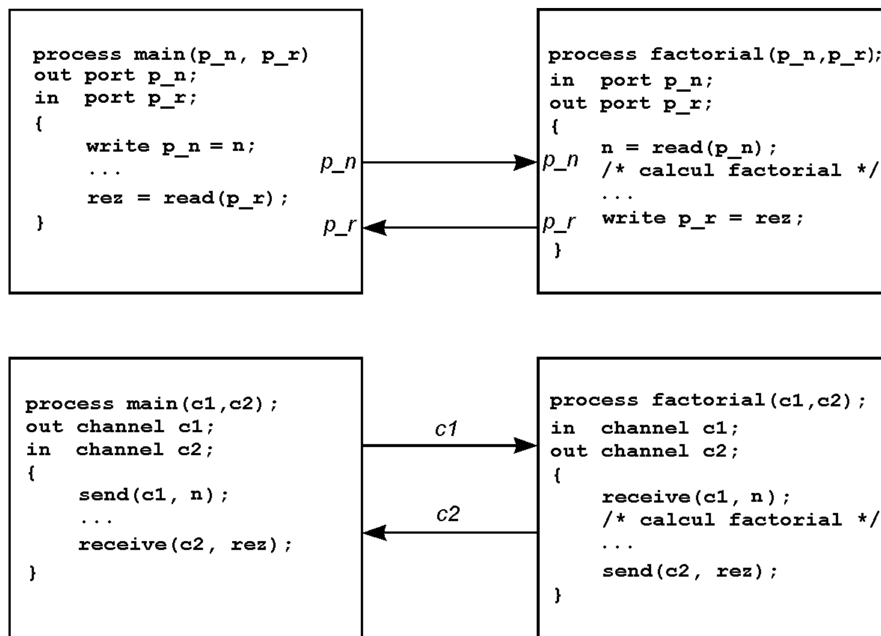


Figura 38. Comunicaţia între procese în limbajul *HardwareC*: (a) transmiterea prin porturi; (b) transmiterea prin canale.

Limbajul *HardwareC* permite specificarea descrierilor parametrizate, numite şabloane, pentru diferite modele (blocuri, procese, proceduri şi funcţii). Aceste şabloane pot fi instanţiate prin specificarea unor valori întregi ca parametri formali, şi se pot utiliza pentru descrierea unor componente de bibliotecă, ca sumatoare sau circuite de înmulţire, parametrii formali reprezentând numărul de biţi, numărul de intrări, etc. Proiectantul poate particulariza un şablon ca un mijloc de a specifica informaţii de legătură care pot fi utilizate de sistemele de sinteză. De exemplu, o anumită operaţie de adunare poate fi asociată cu o instanţiere specifică a unui sumator, permiţând specificarea de către proiectant a partajării resurselor la nivelul descrierii proiectului.

În limbajul *HardwareC* se pot specifica *restricţii de temporizare* între două instrucţiuni, prin asocierea marcajelor sau etichetelor cu fiecare instrucţiune, utilizând apoi marcajele pentru specificarea restricţiilor. În plus, pot fi specificate *restricţii ale resurselor*, indicând numărul de instanţieri ale unui model dat care se vor putea utiliza pentru sinteză.

Limbajul nu permite însă specificarea descrierilor de tipul fluxului de date, a tranziţiilor între stări şi a excepţiilor.

3.5.3. CSP

Limbajul *CSP* (*Communicating Sequential Processes*) a fost propus de *C.A.R. Hoare* pentru a elimina limitările limbajelor de programare tradiţionale în cazul programelor executate pe calculatoarele multiprocesor. *CSP* permite specificarea unui program ca un set

de procese concurente, utilizând construcţii care simplifică specificarea comunicaţiei şi a sincronizării între aceste procese. Pe lângă utilizarea sa ca limbaj de programare, *CSP* este utilizat şi pentru descrierea sistemelor hardware.

Un program *CSP* constă dintr-o listă de *comenzi*. Prin utilizarea comenzii *parallel*, un proces poate genera subprocese la orice nivel al ierarhiei, asigurând astfel *ierarhia funcţională*. Toate procesele din cadrul comenzii `parallel` vor fi executate în mod concurent, comanda fiind terminată numai dacă toate procesele sale au fost terminate. În plus, fiecare proces este la rândul său o listă de comenzi, şi fiecare comandă poate conţine propriile comenzi paralele.

În limbajul *CSP*, fiecare proces poate fi descris prin utilizarea construcţiilor de programare. Subrutinele sunt implementate ca şi corutine, ceea ce înseamnă că o subrutină este implementată ca un proces care se execută în mod concurent cu procesul apelant. Se pot simula subrutine recursive prin utilizarea unui tablou de procese, fiecare element reprezentând un nivel de recursivitate.

Construcţiile de control sunt implementate prin utilizarea comenzii `guarded`, care conţine o listă de expresii de validare sau condiţii, şi o listă de comenzi care vor fi executate numai dacă toate condiţiile din listă sunt evaluate ca adevărate. Pentru a specifica execuţia unei singure comenzi dintr-o listă, se pot utiliza comenzi *alternative*. Astfel, o instrucţiune `if` din limbajul *C*:

```
if (a > b) max = a;
      else max = b;
```

poate fi reprezentată în limbajul *CSP* prin utilizarea următoarei comenzi alternative:

```
[a > b → max := a[]a ≤ b → max := b]
```

Este posibil ca, la utilizarea unei comenzi alternative, să existe o situaţie în care mai multe comenzi din listă sunt validate. În asemenea cazuri, va fi selectată şi executată o comandă în mod arbitrar, permiţând o funcţionare nedeterministă.

În limbajul *CSP*, nu există variabile globale. Din acest motiv, *comunicaţia* între procesele concurente poate fi realizată numai prin transmiterea mesajelor, specificând în mod explicit comenzile `input` şi `output`. Comunicaţia între două procese are loc numai dacă sunt îndeplinite toate condiţiile următoare:

1. Comanda `output` a primului proces specifică al doilea proces ca destinaţie a datelor care trebuie transmise.
2. Comanda `input` a celui de-al doilea proces specifică primul proces ca sursă a datelor care trebuie recepţionate.
3. Tipul destinaţiei (în care se recepţionează datele) din comanda `input` coincide cu tipul expresiei din comanda `output`.

Acest tip de comunicaţie prin utilizarea comenzilor `input/output` reprezintă singurul mecanism de sincronizare în limbajul *CSP*.

Limbajul *CSP* are o serie de limitări, deoarece nu dispune de construcţii pentru specificarea structurii, a tranziţiilor între stări, a temporizării, a descrierilor de tipul fluxului de date, sau a tratării excepţiilor.

3.5.4. Verilog

Limbajul *Verilog* a fost dezvoltat inițial pentru specificarea şi simularea sistemelor numerice de către firma cu acelaşi nume. În 1990, limbajul a devenit public, fiind utilizat pe scară largă ca un limbaj de descriere.

Limbajul are mai multe avantaje pentru proiectanți, unul din acestea fiind faptul că permite reprezentarea *ierarhiei structurale*, sistemul fiind specificat ca o ierarhie de *module* interconectate. Fiecare din aceste module poate fi descris în unul din două moduri, fie prin utilizarea altor module de nivel inferior, fie prin specificarea funcționării sale ca un program.

Ierarhia funcțională poate fi de asemenea reprezentată în limbajul *Verilog*, în sensul că un proces de la orice nivel al ierarhiei poate genera subprocese concurente prin construcția `fork/join`, sau poate fi descompus într-un set de proceduri. Descrierea proceselor poate fi specificată prin construcții de programare cu sintaxa asemănătoare cu cea a limbajului *C*. Descrierile de tipul fluxului de date sunt de asemenea posibile, prin utilizarea instrucțiunilor de asignare continuă.

Comunicația poate fi implementată printr-un model cu memorie partajată, utilizând conexiunile dintre porturile modulelor, registrelor şi ale memoriilor pentru stabilirea comunicației între procese. *Sincronizarea* poate fi realizată în mai multe moduri, deoarece controlul sincronizării poate fi implementat prin utilizarea construcțiilor `fork/join`, sau a instrucțiunilor de control ale evenimentelor care detectează apariția unui eveniment. De exemplu, instrucțiunea:

```
@(negedge) clk #10 q = d;
```

determină actualizarea semnalului *q* cu valoarea *d* cu 10 unități de timp după frontul negativ al semnalului de ceas *clk*. Pentru a obține acelaşi efect, se poate utiliza următoarea instrucțiune `wait`:

```
wait (clk = 0);
     #10 q = d;
```

Specificația *temporizării* poate fi realizată prin modelarea întârzierilor pentru porți şi conexiuni. Pentru fiecare tip de întârziere, limbajul permite specificarea valorilor maxime, minime şi tipice. În plus, *Verilog* permite specificarea întârzierii care determină momentul în care vor fi actualizate valorile dintr-o instrucțiune de asignare. De exemplu, în instrucțiunea de asignare `#10 q = d`, valoarea semnalului *q* va fi actualizată cu 10 unități de timp după frontul negativ al ceasului.

Verilog permite tratarea *excepțiilor* prin utilizarea instrucțiunii `disable`, care invalidează un bloc de instrucțiuni secvențiale, şi transferă controlul la instrucțiunea care urmează după blocul respectiv. Limbajul nu permite însă specificarea tranzițiilor între stări.

3.5.5. Statecharts

Limbajul *Statecharts* a fost proiectat în primul rând pentru specificarea sistemelor reactive, cum sunt cele utilizate în aviație şi rețele de comunicație. Limbajul extinde automatele cu stări finite tradiționale prin includerea a trei elemente adiționale: *ierarhia*, *concurența* şi *comunicația*.

Pentru exemplificare, considerăm **Figura 39**, care reprezintă un circuit emiţător-receptor universal (UART). Obiectul de bază în limbajul *Statechart* este *starea*, iar tranziţiile între stări sunt determinate de o combinaţie de evenimente şi condiţii.

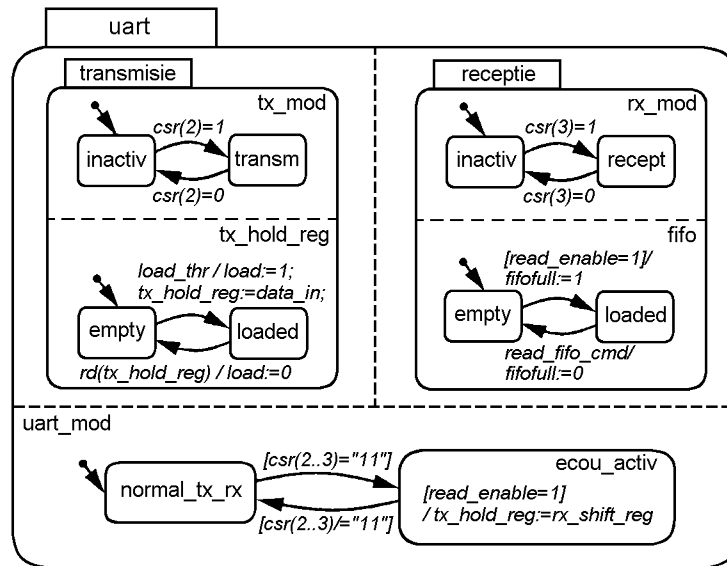


Figura 39. Specificaţia parţială a unui circuit UART în limbajul *Statecharts*.

Limbajul permite reprezentarea ierarhiei funcţionale, fiind posibilă descompunerea fiecărei specificaţii într-o ierarhie de stări. Această descompunere poate fi realizată în două moduri:

1. *Descompunere OR (secvenţială).* O stare poate fi compusă din mai multe substări secvenţiale. În **Figura 39**, de exemplu, starea *tx_mod* constă din două substări secvenţiale, *inactiv* şi *transm*.

2. *Descompunere AND (concurentă).* O stare poate fi formată din substări ortogonale, caz în care toate substările sunt active ori de câte ori starea părinte este activă. În **Figura 39**, stările ortogonale sau concurente sunt cele separate prin linii punctate, ceea ce înseamnă că starea *uart* constă din trei stări concurente: *transmisie*, *receptie* şi *uart_mod*. Descompunerea concurentă specifică o concurenţă la nivel de taskuri.

Deoarece permite descompunerea stărilor în substări secvenţiale şi concurente, limbajul *Statecharts* permite evitarea creşterii exponenţiale a stărilor care poate apare în automatele cu stări finite convenţionale. Acţiunile pot fi asociate atât cu stările cât şi cu arcele de tranziţie, şi se presupune că ele reprezintă calcule cu întârzieri zero. Acţiunile asociate cu o stare pot fi specificate fie pentru a fi executate în mod continuu cât timp sistemul se află în starea respectivă, fie pentru a fi executate la trecerea în acea stare sau la ieşirea din aceasta. Limbajul permite de asemenea specificarea tranziţiilor între stări aflate la nivele diferite ale ierarhiei.

Pentru specificarea *temporizării funcţionale*, se poate utiliza o tranziţie specială de tip timeout, care defineşte valoarea maximă şi minimă a timpului în care sistemul se poate afla în starea dorită.

Comunicația poate fi implementată printr-un mecanism de transmisie, prin intermediul căruia apariția unui eveniment, actualizarea unei variabile sau o tranziție care are loc în orice porțiune a diagramei va fi sesizată imediat în celelalte porțiuni ale acesteia. O caracteristică a limbajului *Statecharts* este versatilitatea acesteia în privința mecanismelor de *sincronizare*: sincronizarea se poate realiza prin inițializare, prin evenimente comune, sau prin detectarea datelor şi stărilor comune.

Statecharts poate specifica o funcționare nedeterministă, deoarece atunci când există două arce de la o stare care pot fi parcurse simultan, se alege unul din arce în mod nedeterminist.

Dezavantajul limbajului *Statecharts* este că nu dispune de construcții de programare, şi nici de construcții pentru specificarea structurii, completitudinii funcționale sau a descrierilor de tipul fluxului de date.

3.5.6. Silage

Limbajul *Silage* a fost dezvoltat pentru specificarea sistemelor bazate pe prelucrarea digitală a semnalelor (DSP). Descrierea la nivel înalt a acestor sisteme este realizată în mod tipic prin grafuri ale fluxului semnalelor, în care şirurile de date sunt transformate prin operatori la intervale fixe de timp, reprezentând rata de eşantionare. Aceste aplicații au un flux de control redus şi necesită un număr relativ mare de calcule. De aceea, limbajele de descriere funcțională proiectate pentru aplicațiile DSP se bazează pe primitive de nivel înalt care oferă facilități specifice prelucrării digitale a semnalelor. Alte limbaje de nivel înalt elaborate pentru aplicațiile DSP sunt *FIRST* şi *ALGIC*.

Silage este un limbaj aplicativ, în sensul că specifică doar funcțiile aplicate asupra datelor, fără utilizarea variabilelor sau a instrucțiunilor de asignare. Astfel operațiile nu produc efecte secundare. O descriere în acest limbaj constă din ecuații care sunt similare cu definițiile algoritmice, şi care descriu transformările efectuate asupra şirurilor de date pentru un singur interval de eşantionare.

Principalul avantaj al limbajului *Silage* constă în faptul că excelează în specificarea descrierii fluxului de date. Expresiile limbajului reprezintă şiruri de valori, astfel că în expresia $a + b$, de exemplu, a şi b reprezintă şiruri de numere, spre deosebire de variabilele sau elementele de tablouri întâlnite la limbajele de programare convenționale.

Un program *Silage* preia setul valorilor de intrare într-un mod sincron, şi produce rezultatele sub forma şirurilor de date. Un program *Silage* constă dintr-un set de definiții, care definesc noi valori în funcție de valorile de intrare. Ordinea definițiilor nu este semnificativă, deoarece ele nu reprezintă asignări la variabile (ceea ce ar introduce dependențe între instrucțiuni).

Funcțiile de recurență ale limbajului, în care elementele unui şir depind de valorile precedente ale şirului, permit o specificare limitată a temporizării funcționale. Operatorul de întârziere `@` este utilizat pentru a indica valori precedente dintr-un şir, ca în următoarea instrucțiune:

```
d = d@1 + 1;
```

Termenul `d@1` se referă la valoarea precedentă a operandului d, cea din timpul intervalului precedent de eşantionare. Fiecare valoare din şirul d va fi incrementată față de valoarea precedentă.

Limbajul *Silage* are constructori pentru tablouri care pot reprezenta elemente specifice ale unui vector în mod concis. De asemenea, pot fi utilizaţi operatori de reducere ca `sum` şi `max` pentru pentru a se opera asupra tablourilor întregi, iar expresiile condiţionale permit selectarea unei expresii dintr-un set de expresii. Operatorii `decimate` şi `interpolate` pot fi utilizaţi pentru reducerea sau creşterea ratei de eşantionare a unui semnal. Operatorul `decimate` permite selecţia unui număr mai redus de elemente dintr-un şir al datelor de intrare, reducând astfel rata de eşantionare. Operatorul `interpolate` are ca efect creşterea ratei de eşantionare a unui şir de date, prin inserarea unui număr mai mare de valori într-un eşantion existent.

Prin construcţia `pragma` a limbajului utilizatorul poate specifica sistemului de sinteză sau compilatorului unele sugestii de implementare. Această construcţie se poate utiliza pentru a asocia anumiţi operatori unităţilor hardware, sau pentru a indica gradul de paralelism permis pentru şirurile de date, prin asocierea funcţiilor limbajului la procesoare.

Deoarece *Silage* a fost elaborat ca un limbaj aplicativ, nu dispune de construcţii specifice limbajelor de programare obişnuite, nu permite recursivitatea sau iteraţiile cu limite dinamice. Asemenea construcţii pot fi reprezentate ca macrouri ale limbajului. De exemplu, o funcţie reprezintă gruparea unor definiţii, fiind implementată ca un macro. De asemenea, limbajul nu are variabile şi operatori de asignare, iar tranziţiile între stări, ierarhia funcţională şi gestionarea excepţiilor nu pot fi reprezentate.

3.5.7. SpecCharts

Limbajul *SpecCharts* se bazează pe modelul automatelor cu stări ale programului (ASP), şi este definit ca o extensie a limbajului *VHDL*. Obiectul de bază al limbajului este construcţia `behavior`, care corespunde direct unei stări a programului din modelul ASP. În **Figura 40** se prezintă un exemplu de descriere în limbajul *SpecCharts*.

Limbajul permite exprimarea *ierarhiei funcţionale*, un sistem fiind descris ca o ierarhie de construcţii `behavior`. Aceste construcţii pot fi compuse sau terminale.

Construcţiile `behavior` compuse sunt descompuse ierarhic într-un set de alte construcţii `behavior`, care pot fi concurente sau secvenţiale. În primul caz, toate subconstrucţiile sunt active de fiecare dată când construcţia este activă, iar în al doilea caz subconstrucţiile sunt active succesiv. În **Figura 40**, *B* şi *X* sunt construcţii `behavior` compuse. Construcţia *B* este compusă din subconstrucţiile concurente *X*, *Y* şi *Z*, iar construcţia *X* este compusă din subconstrucţiile secvenţiale *X1* şi *X2*. Într-o construcţie descompusă în subconstrucţii secvenţiale, prima subconstrucţie din listă va fi cea iniţială, la care se transferă controlul în momentul activării construcţiei părinte. În cazul subconstrucţiilor concurente, ordinea în care acestea sunt specificate nu este relevantă.

Construcţiile `behavior` terminale sunt cele care se află la baza ierarhiei, funcţionarea acestora fiind specificată prin construcţii de programare care utilizează instrucţiunile secvenţiale ale limbajului *VHDL*. În **Figura 40**, de exemplu, *X1*, *X2*, *Y* şi *Z* sunt construcţii terminale.

Limbajul *SpecCharts* permite specificarea *tranziţiilor între stări*, în sensul că se poate reprezenta secvenţierea între subconstrucţiile `behavior` prin intermediul unui set de arce de tranziţie. Un arc este reprezentat ca un 3-tuplu *<T, C, UB>,* unde *T* reprezintă tipul tranziţiei, *C* reprezintă evenimentul sau condiţia care determină tranziţia, iar *UB* reprezintă următoarea

construcție `behavior` la care se transferă controlul în urma tranziției. Dacă nu este asociată nici o condiție cu tranziția, se presupune că aceasta este `TRUE`.

```
ENTITY E IS
  PORT (P: IN INTEGER; Q: OUT INTEGER);
END E;

ARCHITECTURE A OF E IS
BEGIN
  BEHAVIOR B TYPE CONCURRENT SUBBEHAVIORS IS
    TYPE int_array IS ARRAY (NATURAL RANGE < >) OF INTEGER;
    SIGNAL m: int_array (15 DOWNTO 0);
  BEGIN
    X: (TOC, TRUE, COMPLETE);
    Y: (TOC, e3, COMPLETE);
    Z: ;

    BEHAVIOR X TYPE SEQUENTIAL SUBBEHAVIORS IS
    BEGIN
      X1: (TI, e1, X2);
      X2: (TOC, e2, COMPLETE);

      BEHAVIOR X1 TYPE CODE IS ...

      BEHAVIOR X2 TYPE CODE IS ...

    END X;

    BEHAVIOR Y TYPE CODE IS
      VARIABLE max: INTEGER;
    BEGIN
      max := 0;
      FOR j IN 0 TO 15 LOOP
        IF (m(j) > max) THEN
          max := m(j);
        END IF;
      END LOOP;
    END Y;

    BEHAVIOR Z TYPE CODE IS ...

  END B;

END A;
```

Figura 40. Exemplu de specificație în limbajul *SpecCharts*.

Ca şi modelul ASP, limbajul *SpecCharts* are două tipuri de arce de tranziție. Un arc de *tranziție la terminare* (TT, TOC - Transition On Completion) este traversat ori de câte ori construcția `behavior` sursă şi-a terminat prelucrările şi condiția asociată arcului este adevărată.

O construcție `behavior` terminală este terminată atunci când a fost executată ultima instrucțiune a acesteia, şi toate variabilele şi semnalele au fost actualizate cu valorile lor finale.

O construcție `behavior` descompusă în subconstrucții secvențiale este terminată numai atunci când efectuează o tranziție la un punct de terminare predefinit, indicat prin numele `COMPLETE` în câmpul corespunzător al arcului de tranziție. În **Figura 40**, de exemplu, con-

strucția *X* se termină numai dacă se termină subconstrucția *X2*, şi controlul se transferă de la *X2* la punctul `COMPLETE` la apariția evenimentului *e2*, după cum se specifică prin arcul `X2: TOC, e2, COMPLETE)`.

O construcție `behavior` descompusă în subconstrucții concurente este terminată atunci când toate subconstrucțiile sale (sau un subset selectat al acestora) au fost terminate. În **Figura 40**, de exemplu, construcția *B* se termină atunci când ambele subconstrucții concurente *X* şi *Y* s-au terminat şi controlul s-a transferat la punctul de terminare, după cum se specifică prin arcele `X: (TOC, TRUE, COMPLETE)` şi `Y: (TOC, e3, COMPLETE)`. De notat că terminarea construcției *B* nu este afectată de starea execuției subconstrucției *Z*. Subconstrucția *Z* este însă afectată de construcția *B*, în sensul că atunci când *B* se termină datorită tranzițiilor de la *X* şi *Y* la punctul de terminare, şi *Z* este terminat.

Un arc de *tranziție imediată* (TI) este traversat imediat ce condiția asociată devine adevărată, indiferent dacă construcția `behavior` sursă s-a terminat sau nu. De exemplu, în **Figura 40**, arcul `X1: (TI, e1, X2)` va conduce la terminarea construcției *X1* dacă apare evenimentul *e1* şi va transfera controlul construcției *X2*. Deci, un arc TI are ca efect terminarea tuturor subconstrucțiilor de nivel inferior ale construcției sursă. Arcul *timeout* este un arc TI special, care este traversat atunci când intervalul de timp asociat acestuia expiră, interval care este determinat în funcție de momentul activării construcției `behavior`.

În limbajul *SpecCharts*, doar un subset al stărilor programului este activ în orice moment de timp. Numai starea rădăcină, reprezentând întregul sistem, este activă întotdeauna. Semantica de execuție a limbajului *SpecCharts* este similară cu cea a limbajului *VHDL*, iar construcțiile `behavior` active sunt identice cu procesele din *VHDL*, cu excepția faptului că nu există o buclă implicită în care este cuprinsă construcția `behavior`. Cu alte cuvinte, construcțiile `behavior` se execută până când sunt suspendate prin instrucțiunile `wait`, nu există întârziere între două instrucțiuni `wait` succesive, şi toate semnalele sunt actualizate în intervale *delta*. Construcțiile `behavior` inactive sunt ignorate, fiind tratate ca procese *VHDL* suspendate, cu toate driverele semnalelor întrerupte.

În cazul în care o subconstrucție secvențială se termină, dar nu există nici un arc TT cu condiția adevărată, se aşteaptă până când o condiție devine adevărată. Descrierile în acest limbaj sunt deterministe, în sensul că tranzițiile au o prioritate determinată de ordinea în care ele sunt listate în descriere. Ca o regulă, arcele TI au prioritate față de arcele TT, iar arcele TI aflate la nivelul superior al ierarhiei au prioritatea maximă.

O construcție `behavior` poate conține declarații *VHDL*, ca tipuri, semnale, variabile şi proceduri, al căror domeniu este reprezentat de toate subconstrucțiile aferente.

Sincronizarea poate fi realizată prin două metode. În cazul primei metode, se utilizează instrucțiuni `wait` pentru testarea evenimentelor şi a condițiilor, ca şi în limbajul *VHDL*. Astfel, următoarea instrucțiune:

```
WAIT UNTIL (start = '0') AND (NOT start'STABLE);
```

va suspenda construcția `behavior` în care apare până la detectarea unei tranziții negative a semnalului *start*. În cazul metodei a doua, se poate utiliza un arc de tranziție TI de la o construcție `behavior` la ea însăşi, cu scopul de a sincroniza toate subconstrucțiile concurente cu stările lor inițiale.

SpecCharts permite exprimarea *ierarhiei structurale* într-un mod similar cu limbajul *VHDL*, prin faptul că sistemul sau o porțiune a acestuia poate fi încapsulată ca o *entitate*. Pentru fiecare entitate pot fi declarate porturi, şi acestea pot fi conectate cu porturile altor

entități, prin intermediul semnalelor. În **Figura 40**, de exemplu, ierarhia de construcții `behavior` este încapsulată ca o entitate *E*, declarațiile de porturi specificând interfața entității. *Ierarhia funcțională* poate fi de asemenea exprimată în mod direct, prin faptul că o construcție `behavior` poate fi formată din subconstrucții secvențiale sau concurente (**Figura 41**).

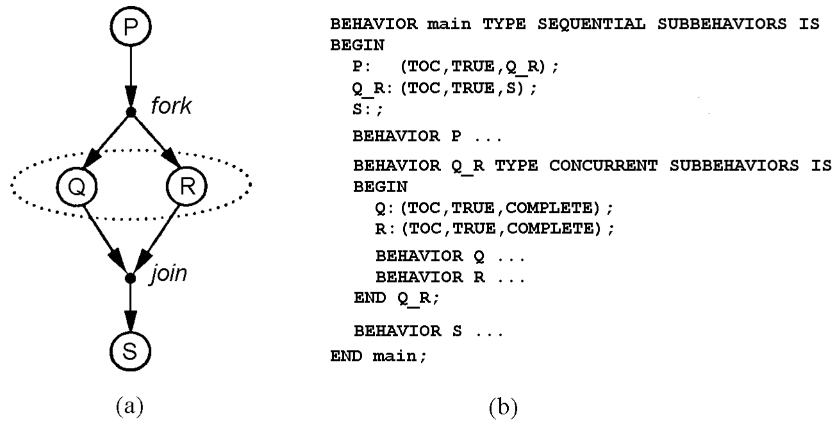


Figura 41. Descompunerea funcțională secvențială/concurentă în limbajul *Specharts*.

Comunicația în limbajul *SpecCharts* este realizată prin utilizarea variabilelor şi a semnalelor. Se pot utiliza construcții secvențiale pentru citirea şi scrierea aceleiaşi variabile, iar comunicația între construcțiile concurente este implementată prin semnale. Comunicația prin transmiterea mesajelor poate fi de asemenea specificată în acest limbaj, prin definirea unor proceduri *send/receive* pentru toate tipurile de date transmise prin canale.

Specificarea *temporizării* este realizată prin instrucțiuni `wait` şi clauze `after` în cadrul asignării semnalelor, în mod identic cu limbajul *VHDL*. O specificație suplimentară de temporizare este arcul TI *timeout*.

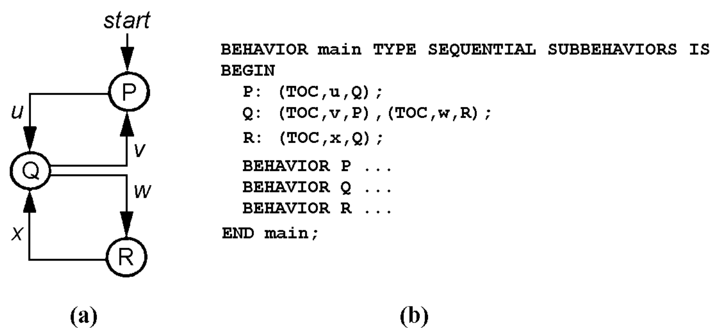


Figura 42. Specificarea tranzițiilor între stări în limbajul *SpecCharts*.

Tranzițiile între stări sunt exprimate prin arcele TT și TI (**Figura 42**). *Excepțiile* pot fi specificate direct printr-un arc TI (**Figura 43**). De notat corespondența directă dintre modelul conceptual și descrierea în limbajul *SpecCharts*.

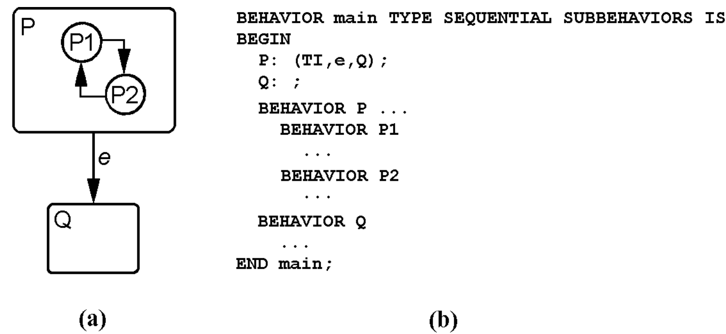


Figura 43. Specificarea excepțiilor în limbajul *SpecCharts*.

Exemplele în limbajul *SpecCharts* au fost prezentatate sub formă textuală. Tranzițiile între stări, însă, pot fi exprimate mai sugestiv sub formă grafică, ca diagrame de stare. Din acest motiv, limbajul *SpecCharts* are o *versiune grafică echivalentă*. În **Figura 44** se prezintă versiunea grafică echivalentă a specificației textuale din **Figura 40**.

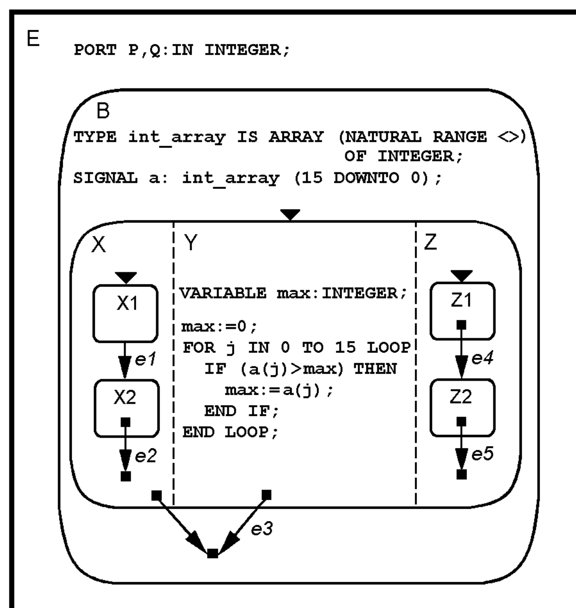


Figura 44. Versiunea grafică echivalentă a exemplului descris în limbajul *SpecCharts*.

O entitate este reprezentată printr-un dreptunghi, iar construcțiile `behavior` din cadrul unei entități sunt reprezentate prin dreptunghiuri cu colțuri rotunjite. Subconstrucțiile

concurente sunt separate între ele prin linii întrerupte. Tranzițiile sunt reprezentate prin arce direcționate. În cazul construcțiilor `behavior` descompuse în subconstrucții secvențiale, prima subconstrucție este indicată printr-un triunghi (de exemplu *X1*). Terminarea construcțiilor descompuse secvențial este indicată printr-un arc de tranziție către punctul de terminare, reprezentat printr-un pătrat în cadrul construcției (de exemplu tranziția de la *X2* etichetată cu *e2*).

Arcele de tranziție TT şi TI au reprezentări grafice diferite. Arcele TT încep de la un dreptunghi din interiorul subconstrucției sursă, ca de exemplu arcele etichetate cu *e2* şi *e3*. Arcele TI încep de la perimetrul subconstrucției sursă, ca de exemplu arcul etichetat cu *e1*.

Limbajul *StateCharts* poate fi extins în mai multe moduri. În primul rând, de multe ori este necesară descompunerea unei construcții `behavior` care a fost specificată ca un set de instrucțiuni secvențiale, într-un set de subconstrucții concurente. În asemenea cazuri, este utilă o construcție de tip `fork/join` care poate fi aplicată construcțiilor `behavior` terminale.

În al doilea rând, pentru un sistem poate fi necesară specificarea unei construcții `behavior` care execută în mod continuu un set de asignări concurente. Un exemplu este definirea unui semnal *enable* ca fiind activ cât timp există tensiune de alimentare (*power = 1*) şi semnalul de resetare este inactiv (*reset = 0*). Este utilă posibilitatea definirii semnalului *enable* în funcție de *power* şi *reset* pe toată durata funcționării sistemului, ca în următoarea instrucțiune de asignare concurentă în limbajul *VHDL*:

```
BLOCK
BEGIN
  enable <= power AND (NOT reset);
END BLOCK;
```

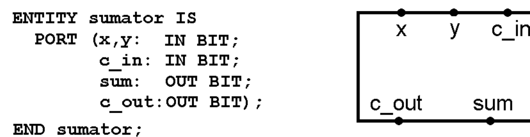
O altă extensie utilă este parametrizarea construcțiilor `behavior`. În acest caz se pot specifica sisteme de dimensiuni mari, constând din blocuri funcționale apropiate, care diferă numai printr-un număr redus de parametri. În variantele actuale, chiar dacă există *n* construcții `behavior` în cadrul sistemului, toate cele *n* trebuie specificate în întregime în descriere.

3.6. Relația dintre limbaj şi arhitectură

Din secțiunile precedente rezultă că limbajele de descriere hardware se bazează pe diferite modele de proiectare şi au diferite semantici. Pentru ca aceste limbaje să poată fi utilizate în mod eficient pentru sinteza de nivel înalt, este necesară o corespondență între modelul semantic al limbajului şi modelul arhitectural utilizat la implementare. Atunci când limbajele sunt elaborate în mod special pentru o anumită aplicație (de ex., *Silage*), o asemenea corespondență există. În cazul limbajelor elaborate pentru a acoperi un spectru larg de proiecte şi aplicații, modelul semantic al limbajului poate fi complet diferit față de modelul arhitecturii generate de sistemul de sinteză.

Această diferență între modelele semantice ale limbajelor şi modelele arhitecturale este vizibilă mai ales în cazul limbajului *VHDL*. Acest limbaj are un set variat de construcții care permit descrierea aceleiaşi specificații în mai multe moduri diferite. Toate aceste descrieri pot produce aceleaşi rezultate de simulare, dar este posibil să nu existe o relație între descrierile simulate corect şi modulele hardware generate de sistemele de sinteză de nivel înalt.

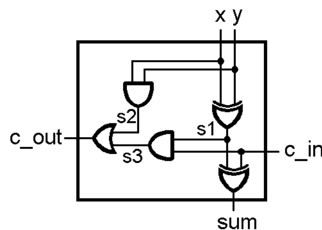
De exemplu, se consideră specificația unui sumator în limbajul *VHDL*, pentru care descrierea interfeței este prezentată în **Figura 45(a)**. În funcție de experiență, proiectantul poate elabora diferite descrieri pentru acest circuit simplu. Dacă proiectantul este experimentat în domeniul proiectării circuitelor logice, poate realiza o descriere similară cu cea din **Figura 45(b)**, deoarece sumatorul poate fi descris în mod eficient prin utilizarea operatorilor booleeni. Un proiectant cu experiență în domeniul software poate prefera o descriere similară cu cea din **Figura 46(a)**, în care se utilizează operatori aritmetici asupra variabilelor pentru a se descrie sumatorul pe baza tabelului de adevăr al acestuia.



(a)

```
ARCHITECTURE ec_bool OF sumator IS
SIGNAL s1,s2,s3: BIT;
BEGIN
  s1 <= x XOR y;
  sum <= s1 XOR c_in AFTER 3 ns;
  s2 <= x AND y;
  s3 <= s1 AND c_in;
  c_out <= s2 OR s3 AFTER 5 ns;
END ec_bool;
```

(b)



(c)

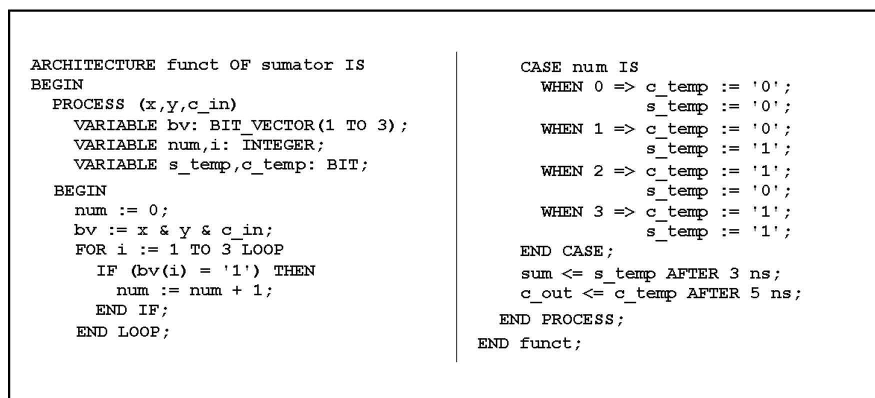
Figura 45. Descrierea de tipul fluxului de date a unui sumator în limbajul *VHDL*: (a) descrierea entității; (b) descrierea arhitecturii; (c) structura obținută în urma sintezei.

Ambele descrieri produc aceleaşi rezultate la simulare. Primul tip de descriere este mai eficient pentru sinteza automată, deoarece ecuațiile booleene sugerează implementarea pe baza operatorilor booleeni `AND`, `OR` şi `XOR`. **Figura 45(c)** indică structura hardware obținută pornind de la descrierea de tipul fluxului de date, în care fiecărui operator logic îi corespunde poarta logică respectivă.

Pe de altă parte, dacă se utilizează descrierea funcțională din **Figura 46(a)** ca intrare pentru un sistem de sinteză de nivel înalt, poate rezulta structura din **Figura 46(b)**. În structura rezultată, există circuite suplimentare pentru conversia între tipul bit şi întreg, şi se utilizează trei circuite de incrementare pentru implementarea unui simplu sumator. Aceste ineficiențe trebuie eliminate de către sistemele de sinteză. Aceasta necesită o etapă costisitoare de recunoaştere şi optimizare, ceea ce s-ar putea evita prin utilizarea stilului

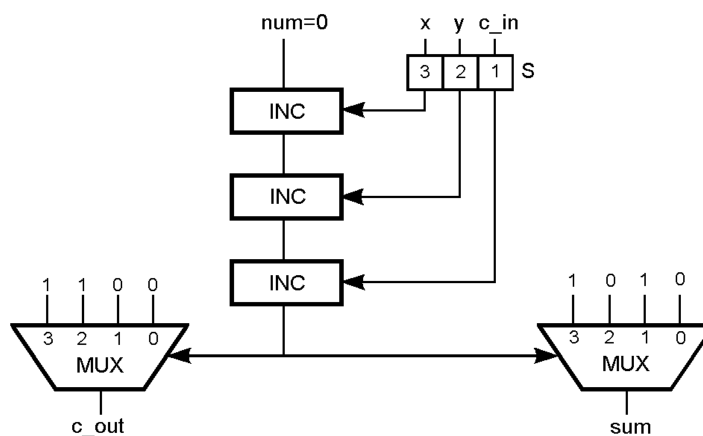
corespunzător de descriere. În plus, nu există garanția că aceste ineficiențe pot fi recunoscute şi optimizate întotdeauna.

A doua problemă apare datorită modelului timpului utilizat de către diferitele limbaje. Un limbaj bazat pe simulare modelează valorile semnalelor în diferite momente de timp. Întârzierile specificate în asemenea limbaje se referă la planificarea evenimentelor asupra acestor semnale în momentele viitoare ale timpului de simulare, execuția instrucțiunilor necesitând un timp de simulare zero. În consecință, întârzierea de la intrare la ieşire pentru un set de instrucțiuni de asignare poate fi distribuită într-un mod oarecare între instrucțiunile individuale. Chiar dacă toate descrierile generează aceleaşi rezultate corecte la simulare, aceste întârzieri au semantici ambigue pentru sinteză din cauza modurilor diferite în care ele pot fi specificate.



```
ARCHITECTURE funct OF sumator IS
BEGIN
  PROCESS (x,y,c_in)
    VARIABLE bv: BIT_VECTOR(1 TO 3);
    VARIABLE num,i: INTEGER;
    VARIABLE s_temp,c_temp: BIT;
  BEGIN
    num := 0;
    bv := x & y & c_in;
    FOR i := 1 TO 3 LOOP
      IF (bv(i) = '1') THEN
        num := num + 1;
      END IF;
    END LOOP;
    CASE num IS
      WHEN 0 => c_temp := '0';
                s_temp := '0';
      WHEN 1 => c_temp := '0';
                s_temp := '1';
      WHEN 2 => c_temp := '1';
                s_temp := '0';
      WHEN 3 => c_temp := '1';
                s_temp := '1';
    END CASE;
    sum <= s_temp AFTER 3 ns;
    c_out <= c_temp AFTER 5 ns;
  END PROCESS;
END funct;
```

(a)



(b)

Figura 46. Descrierea funcțională a unui sumator în limbajul *VHDL*: (a) descrierea arhitecturii; (b) rezultatul sintezei.

Se va ilustra această diferență între semanticile de temporizare considerând descrierile sumatorului din **Figura 45(b)** şi **Figura 46(a)**, în care se specifică o întârziere de 5 ns pentru transportul de ieşire. Întârzierea de 5 ns pentru semnalul de transport din **Figura 45(b)** este

concentrată în cadrul operatorului OR din instrucțiunea de asignare. De aceea semantica de simulare pentru acest caz sugerează că generarea semnalelor *s2* şi *s3* (operațiile AND) necesită un timp zero, în timp ce operația OR se execută în 5 ns. În contextul sintezei, această valoare a întârzierii este însă ambiguă, deoarece ea reprezintă întârzierea de la semnale intermediare la ieşirea *c_out*, şi nu întârzierea de la semnalele de intrare *x*, *y* şi *c_in* la ieşirea *c_out*. Similar, întârzierea de 5 ns pentru transportul din **Figura 46(a)** este concentrată în cadrul ultimei instrucțiuni de asignare, care nu are nici un operator, în timp ce se presupune că generarea semnalului *c_temp* se realizează într-un timp egal cu zero. Ambele descrieri au semantici ambigue pentru sinteză, din cauza modului în care sunt grupate întârzierile în instrucțiunile de asignare respective.

Se poate elabora pentru sumatorul anterior şi un model structural. **Figura 47** prezintă o descriere structurală în limbajul *VHDL* în care sumatorul este realizat prin utilizarea a două semisumatoare şi a unei porți SAU. Simularea acestei descrieri structurale este realizată prin elaborarea modelelor de simulare pentru fiecare componentă structurală. În acest context, sinteza implică implementarea structurii semisumatorului şi a porții SAU utilizând primitive de nivel inferior, urmată de plasarea şi rutarea întregului circuit.

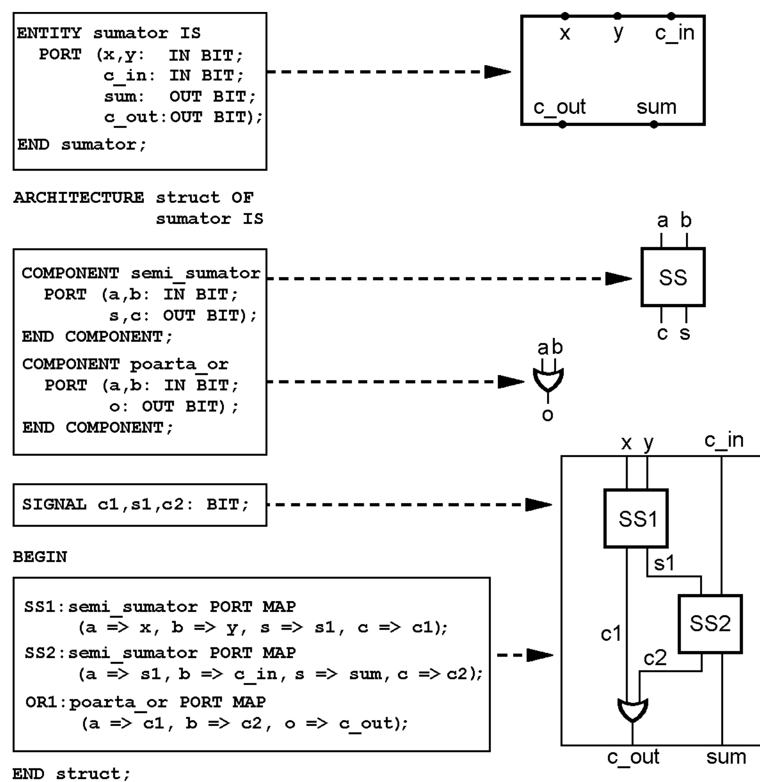


Figura 47. Descrierea structurală a unui sumator în limbajul *VHDL*.

De observat că descrierea structurală din **Figura 47** se referă explicit la componente şi la interconexiunile acestora, în timp ce descrierea de tipul fluxului de date din **Figura 45** specifică funcționarea abstractă utilizând expresii booleene. Aceasta din urmă nu descrie structura, deoarece operatorii booleeni abstracți trebuie să fie asociați cu componente fizice care execută operațiile respective.

În plus față de problemele legate de stilul descrierii care afectează calitatea sintezei şi ambiguitatea specificării întârzierilor pentru sinteză, un limbaj bazat pe simulare cum este *VHDL* poate avea construcții de limbaj care nu au implementări hardware echivalente.

Considerăm fragmentul de program *VHDL* din **Figura 48**, care descrie funcțiile de ştergere şi de numărare în sus ale unui numărător. Blocul *cnt_clr* descrie ştergerea asincronă a numărătorului atunci când semnalul *clr* este activ. Blocul *cnt_up* descrie operația de numărare sincronă atunci când numărătorul este validat şi semnalul *inc* este activ. Blocul *sel* stabileşte valoarea ieşirii *cnt* a numărătorului, prin selecția între valoarea *cnt1*, rezultată în urma ştergerii numărătorului, şi valoarea *cnt2*, rezultată în urma incrementării acestuia. Al treilea bloc este necesar numai din cauza semanticii de simulare a limbajului *VHDL*: nu se pot asigna valori semnalului *cnt* în blocuri diferite fără a scrie o funcție de decizie.

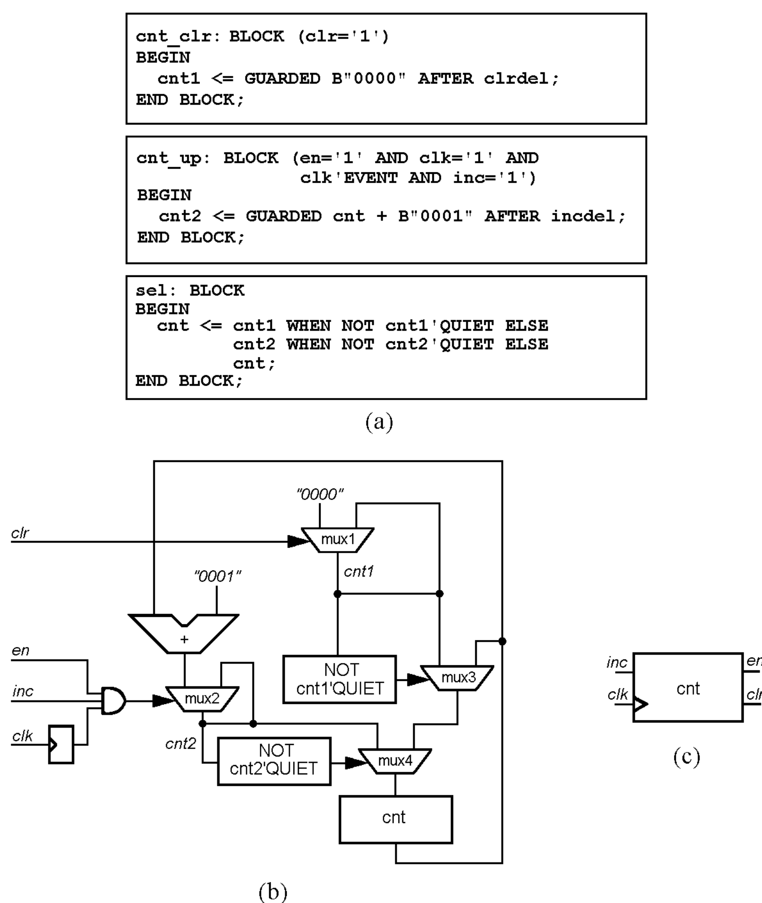


Figura 48. Descrierea unui numărător şi rezultatul inițial obținut în urma sintezei.

Această descriere este convenabilă pentru modelarea funcționării numărătorului în vederea simulării, deoarece operațiile de ştergere asincronă şi de numărare sincronă pot fi specificate separat. Acestor operații li se pot asocia valori diferite de întârziere pentru a asigura ca simularea funcționării să respecte restricțiile de temporizare cerute pentru numărător.

Totuşi, dacă se realizează sinteza din această descriere, rezultă inițial structura din **Figura 48(b)**. Această structură conține patru multiplexoare, *mux1*, *mux2*, *mux3* şi *mux4*, ca şi două blocuri pentru detectarea modificării semnalelor *cnt1* şi *cnt2*. Structura generată nu

seamănă cu cea a unui numărător standard, fiind necesare operații considerabile de optimizare pentru transformarea acestei structuri.

După cum se constată din exemplele anterioare, flexibilitatea unui limbaj de descriere permite ca funcționarea să fie descrisă în mai multe moduri. În general, nu este posibilă sinteza optimă pentru fiecare tip de descriere, dar anumite stiluri de descriere asigură o corespondență mai apropiată cu modelul arhitectural. De asemenea, atât calitatea proiectului rezultat prin sinteză, cât şi complexitatea sistemelor de sinteză utilizate sunt influențate de stilul descrierilor utilizate.

Există două soluții pentru a realiza o apropiere între modelele semantice ale limbajelor şi modelele arhitecturale. În primul rând, se pot elabora limbaje specializate prin extinderea unor limbaje existente, a căror modele semantice sunt adecvate pentru arhitectura destinație sau aplicația respectivă. *SpecCharts*, *StateMate* şi *BIF* sunt exemple de limbaje şi formalisme specializate care au la bază limbajul *VHDL*.

În al doilea rând, se poate utiliza un limbaj existent, cu restricția de a se utiliza stilurile şi principiile de modelare care garantează o sinteză eficientă. În plus, aceste stiluri şi principii de modelare îmbunătățesc documentarea proiectului şi comunicarea între proiectanți, prin producerea unui set mai redus de stiluri de descriere mai bine documentate şi înțelese. Aceste stiluri de modelare pot fi comparate cu principiile programării structurate, în scopul îmbunătățirii calității şi portabilității produselor software.

Deoarece limbajul *VHDL* este un limbaj standard cu semantica diferită de modelul hardware, se va utiliza acest limbaj pentru ilustrarea unor principii în vederea unei sinteze eficiente. Există numeroase eforturi de cercetare în scopul elaborării unor principii şi practici de modelare pentru sinteza pe baza limbajului *VHDL*. Se vor ilustra unele principii de modelare pentru câteva modele arhitecturale.

1) Modele combinaționale

Pentru un model hardware constând dintr-o rețea logică combinațională, o descriere care utilizează abstractizări funcționale ale porților logice, deci operatori booleeni ai limbajului, este cea mai adecvată. Din exemplul de modelare al sumatorului, se constată că descrierea care utilizează operatori booleeni este cea mai apropiată de modelul hardware.

Circuitele combinaționale sunt caracterizate prin concurență; deoarece nu există ceasuri care controlează secvențierea, ieşirile se pot modifica în orice moment ca urmare a modificării intrărilor. De aceea, circuitele combinaționale pot fi descrise prin construcțiile de tipul fluxului de date ale limbajului *VHDL*, utilizând instrucțiunea de asignare concurentă.

2) Modele funcționale

Aceste modele sunt caracterizate printr-o combinație de funcții sincrone şi asincrone, în care evenimentele asincrone sunt prioritare față de operațiile sincrone. Componentele tipice care pot fi descrise prin modele funcționale sunt cele secvențiale, ca registre, registre de deplasare, numărătoare şi elemente de memorie. Fiecare model funcțional este un automat cu o singură stare, care poate avea mai multe moduri de operare.

Pentru descrierea modelelor funcționale în limbajul *VHDL*, trebuie identificate mai întâi părțile modelelor care operează sincron, şi cele care operează asincron. Fiecărui semnal care apare în descrierea interfeței i se atribuie un tip în funcție de scopul acestuia (de ceas, de resetare, de control, de date). Ca urmare, funcțiile sincrone şi cele asincrone pot fi descrise într-un singur bloc, şi deoarece modelele funcționale reprezintă modele ASFD cu o singură stare, poate fi utilizată construcția `BLOCK` pentru descrierea acestor modele arhitecturale.

Ca exemplu, se descrie în limbajul *VHDL* numărătorul din **Figura 48(a)** utilizând stilul descrierii funcționale. În primul rând, se atribuie tipuri semnalelor *clr* (de resetare), *clk* (de ceas), şi *en* (de control). În continuare, se descrie funcționarea numărătorului utilizând un bloc de tipul fluxului de date (**Figura 49**). Blocul este activat numai dacă semnalul *clr* este activ (*'1'*), sau dacă *en* este activ şi apare un front crescător al semnalului *clk*. În cadrul blocului, semnalului *cnt* i se asignează valoarea *0* dacă *clr = '1'*, chiar dacă *en = '1'* şi apare un front crescător al semnalului *clk*.

```
cnt_up: BLOCK (clr = '1' OR (en = '1' AND clk'EVENT AND clk = '1'))
BEGIN
  cnt <= GUARDED
           B"0000" AFTER clrdel WHEN clr = '1' ELSE
           cnt + B"0001" AFTER incdel WHEN inc = '1' ELSE
           cnt;
END BLOCK;
```

Figura 49. Exemplu de descriere funcțională în limbajul *VHDL*.

3) Modele RT

Modelul arhitectural al transferurilor între registre corespunde modelului conceptual ASF cu cale de date. De aceea, descrierea funcțională a unui asemenea model se poate realiza pe baza stărilor, astfel încât pentru fiecare stare să se specifice condițiile care trebuie testate în unitatea de control, operațiile executate în calea de date şi starea următoare.

```
St_Fetch: BLOCK ((clk'EVENT AND clk = '1') AND (stare = s0))
BEGIN
  ri    <= M(pc);
  stare <= s1;
END BLOCK;

St_Decod: BLOCK ((clk'EVENT AND clk = '1') AND (stare = s1))
BEGIN
  CASE ri IS
    WHEN "0000" => a     <= a + 1;
                   stare <= s2;
    WHEN "0001" => a     <= 0;
                   stare <= s3;
    ...
  END CASE;
END BLOCK;
```

Figura 50. Exemplu de descriere de tip RT în limbajul *VHDL*.

VHDL nu dispune de conceptul de stare în cadrul limbajului, astfel că modelarea se poate realiza în diferite moduri. Nu toate dintre acestea determină însă o sinteză eficientă. O posibilitate de modelare a stărilor în limbajul *VHDL* este de a asocia fiecare stare cu un bloc separat. Deoarece blocurile se execută în mod concurent, este necesar să se activeze doar acel bloc care corespunde stării curente. Aceasta se realizează prin asignarea unui nume unic de stare fiecărui bloc şi actualizarea unui semnal pentru starea curentă. Un bloc de stare este

activat dacă are loc o modificare a semnalului de ceas şi dacă semnalul stării curente indică starea respectivă.

Operațiile din calea de date pot fi descrise utilizând construcții concurente. Această descriere este corespunzătoare deoarece transferurile între registre din calea de date se efectuează în paralel pe durata unei singure stări. La sfârşitul blocului de stare, semnalul stării curente este actualizat cu starea următoare.

Figura 50 prezintă un fragment în limbajul *VHDL* pentru descrierea stărilor *St_Fetch* şi *St_Decod* ale unui procesor simplu, în care fiecare bloc corespunde unei singure stări.

4. Concluzii

Se constată că pe măsura creşterii complexității sistemelor numerice, apare necesitatea unor sisteme pentru proiectarea automată la nivele de abstractizare mai înalte. Abstractizarea de nivel mai înalt determină reducerea costului de proiectare. Pentru reducerea ciclului de proiectare sunt necesare tehnici de sinteză care permit realizarea conceptelor de *corectitudine prin construcție* şi cel al *primei specificații*. În acest scop, sistemele CAD trebuie să permită verificarea funcțională şi a regulilor de proiectare, fiind necesară modelarea cu acuratețe a procesului de proiectare şi estimarea corectă a unor indicatori de calitate, de exemplu cei de performanță şi cost.

Pe lângă reducerea duratei ciclului de proiectare, abstractizarea de nivel înalt şi automatizarea unei părți sau a întregului proces de proiectare face posibilă explorarea mai completă a diferitelor metode de proiectare, deoarece proiectele pot fi generate şi evaluate într-un timp redus. De asemenea, dacă algoritmii de sinteză sunt performanți, sistemele de proiectare automată pot depăşi proiectanții de nivel mediu în ceea ce priveşte generarea proiectelor de calitate. Totuşi, verificarea corectitudinii acestor algoritmi, şi a sistemelor CAD în general, nu este o sarcină uşoară. Sistemele CAD nu pot asigura încă o calitate comparabilă cu cea a proiectantului uman pentru întregul proces de proiectare.

Pentru definirea diferitelor *nivele de descriere* a sistemelor numerice, *diagrama Y* este cea mai des utilizată. Fiecare element al unui sistem numeric poate fi descris în cadrul a trei domenii diferite: *domeniul funcțional*, *domeniul structural* şi *domeniul fizic/geometric*.

Un *model* al unui circuit sau sistem este o abstractizare a acestuia, deci o reprezentare care pune în evidență caracteristicile sale relevante, fără detaliile asociate. *Modelele informale* pot constitui surse de ambiguități, de aceea au o aplicabilitate limitată atunci când sunt utilizate pentru sistemele CAD. În schimb, *modelele formale* au o sintaxă şi semantică bine definită, de aceea asigură un mijloc de a exprima informațiile despre un sistem într-un mod care poate fi interpretat neambiguu.

Un model nu descrie exact modul în care sistemul trebuie realizat. După alegerea unui model corespunzător şi descrierea funcționării sistemului, următoarea etapă de proiectare este transformarea modelului într-o *arhitectură*, care defineşte implementarea modelului.

Un circuit sau sistem poate fi modelat în moduri diferite în funcție de nivelul de abstractizare dorit (de exemplu, *funcțional*, *structural*, *fizic*) şi de metodele de modelare utilizate (de exemplu, *limbaje*, *diagrame*, *modele abstracte*). În general, modelele utilizate în cadrul sistemelor de proiectare asistată de calculator se încadrează în următoarele categorii: orientate pe *stare*; orientate pe *activitate*; orientate pe *structură*; orientate pe *date*; *eterogene*.

Din aceste categorii, au fost prezentate următoarele modele utilizate în mod frecvent: automatele cu stări finite; rețelele Petri; automatele cu stări finite ierarhice şi concurente; grafurile fluxului de date; grafurile fluxului de control; diagramele de conexiune a componentelor; structurile de incidență; rețelele logice generalizate; grafurile fluxului de control şi de date; diagramele de structură; automatele cu stări ale programului; modelul firelor de aşteptare.

În ultimii ani, există tendința de a se utiliza *limbajele de descriere hardware* pentru specificarea proiectelor, modelele bazate pe aceste limbaje fiind preferate față de diagramele de stare, grafurile fluxului de date şi de control, deşi unele modele bazate pe diagrame sunt mai puternice în ceea ce priveşte vizualizarea funcțiilor unui sistem. Există mai multe *avantaje ale utilizării limbajelor de descriere ca limbaje de specificație executabilă*, de exemplu: descrierea executabilă se poate utiliza pentru simulare, pentru sinteza automată, pentru documentarea sistemului.

Natura specifică a circuitelor hardware determină ca aceste limbaje să fie diferite de limbajele de programare utilizate în mod curent. Astfel, semantica unei funcții specificate într-un limbaj de descriere implică un proiect hardware care urmează a fi implementat, spre deosebire de o funcție care se execută pe un sistem existent. De asemenea, aceste limbaje au construcții suplimentare pentru a se adapta la caracteristicile speciale ale circuitelor hardware.

Pe măsura creşterii nivelului de abstractizare al proiectelor, a crescut şi nivelul de abstractizare al limbajelor de descriere, rezultând o proliferare a acestor limbaje, ceea ce a creat dificultăți în privința portabilității descrierilor. Asemenea probleme au condus la eforturi de *standardizare* a limbajelor pentru proiectarea sistemelor digitale (*Conlan*, *VHDL*, *UDL/I*), cu scopul de a se dezvolta un limbaj comun care poate fi utilizat pentru modelare, simulare şi documentare.

Limbajele de descriere hardware, ca şi limbajele de programare, pot fi clasificate în limbaje *procedurale* şi *declarative (neprocedurale)*. De asemenea, limbajele de descriere mai pot fi clasificate în limbaje cu semantică *imperativă* şi limbaje cu semantică *aplicativă*.

În funcție de tipul aplicației proiectate şi de arhitectura care se va utiliza pentru implementare, proiectanții necesită diferite tipuri de descrieri, sub formă *textuală, tabelară* sau *grafică*.

Construcțiile limbajelor de programare standard permit abstractizări funcționale, dar nu permit exprimarea proprietăților specifice unităților hardware în descrierea proiectelor. Pentru aceasta limbajele de descriere hardware trebuie să dispună de construcții suplimentare pentru definirea interfețelor, specificarea parțială a structurii proiectelor, specificarea operatorilor la nivelul transferurilor între registre şi la nivelul logic, a asincronismului, a ierarhiei, a comunicației între procese, a restricțiilor de proiectare şi a alocării de către utilizator.

Au fost prezentate principalele caracteristici ale unor limbaje de descriere mai cunoscute: *VHDL*; *HardwareC*; *Verilog*; *CSP*; *Statecharts*; *Silage*; *SpecCharts*. Dintre acestea, limbajul *VHDL* are o importanță deosebită, deoarece este un limbaj standard cu o răspândire largă, şi permite realizarea proiectării la un nivel de abstractizare mai ridicat, şi nu la nivelul logic sau cel al transferurilor între registre.

Trebuie menționat că *fiecare limbaj de descriere reprezintă un instrument de proiectare şi nu o metodologie*. În domeniul limbajelor de descriere hardware este de aşteptat ca următoarea etapă să o constituie introducerea tehnicilor orientate pe obiecte. De asemenea, deoarece în realitate sistemele conțin şi părți analogice, mecanice, sau programe, limbajele şi sistemele de proiectare trebuie îmbunătățite pentru a ține cont şi de aceste aspecte. De exemplu, o soluție o reprezintă posibilitatea includerii descrierii componentelor analogice şi a programelor în descrierea sistemului.

Limbajele de descriere hardware se bazează pe diferite modele de proiectare şi au diferite semantici. Pentru ca aceste limbaje să poată fi utilizate în mod eficient pentru sinteza de nivel înalt, este necesară o *corespondență între modelul semantic al limbajului şi modelul arhitectural utilizat la implementare*. Atunci când limbajele sunt elaborate în mod special pentru o anumită aplicație (de ex., *Silage*), o asemenea corespondență există. În cazul limbajelor elaborate pentru a acoperi un spectru larg de proiecte şi aplicații, modelul semantic al limbajului poate fi complet diferit față de modelul arhitecturii generate de sistemul de sinteză, fiind necesară utilizarea unui stil de descriere corespunzător.

O altă problemă apare datorită *modelului timpului* utilizat de către diferitele limbaje. Un limbaj bazat pe simulare modelează valorile semnalelor în diferite momente de timp. Întârzierile specificate în asemenea limbaje se referă la planificarea evenimentelor asupra acestor semnale în momentele viitoare ale timpului de simulare, execuția instrucțiunilor necesitând un timp de simulare zero. De aceea, întârzierea de la intrare la ieşire pentru un set de instrucțiuni de asignare poate fi distribuită într-un mod oarecare între instrucțiunile individuale. Chiar dacă toate descrierile generează aceleaşi rezultate la simulare, aceste întârzieri au semantici ambigue pentru sinteză din cauza modurilor diferite în care ele pot fi specificate.

În plus față de problemele legate de stilul descrierii care afectează calitatea sintezei şi ambiguitatea specificării întârzierilor pentru sinteză, un limbaj bazat pe simulare cum este *VHDL* poate avea construcții de limbaj care nu au implementări hardware echivalente.

Flexibilitatea unui limbaj de descriere permite ca funcționarea să fie descrisă în mai multe moduri. În general, nu este posibilă sinteza optimă pentru fiecare tip de descriere, dar anumite stiluri de descriere asigură o corespondență mai apropiată cu modelul arhitectural. De asemenea, atât calitatea proiectului rezultat prin sinteză, cât şi complexitatea sistemelor de sinteză utilizate sunt influențate de stilul descrierilor utilizate.

Există două *soluții pentru a realiza o apropiere între modelele semantice ale limbajelor şi modelele arhitecturale*. În primul rând, *se pot elabora limbaje specializate* prin extinderea unor limbaje existente, a căror modele semantice sunt adecvate pentru arhitectura destinație sau aplicația respectivă. În al doilea rând, *se poate utiliza un limbaj existent*, cu restricția de a se utiliza stilurile şi principiile de modelare care garantează o sinteză eficientă. În plus, aceste stiluri şi principii de modelare îmbunătățesc documentarea proiectului şi comunicarea între proiectanți, prin producerea unui set mai redus de stiluri de descriere mai bine documentate.

Pentru a se realiza o *sinteză de nivel înalt* eficientă, este necesară formularea problemelor legate de această sinteză şi dezvoltarea unor algoritmi de optimizare pentru diferite arhitecturi, care să utilizeze modele mai realiste de proiectare. Sunt necesare eforturi de proiectare pentru clasificarea arhitecturilor şi elaborarea unor principii de modelare pentru diferitele arhitecturi. De asemenea, există o lipsă a unor seturi de componente standard şi a unor algoritmi pentru adaptarea tehnologică între diferite seturi de componente.

Bibliografie

1. Giovanni De Micheli: *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
2. D. D. Gajski, N. D. Dutt, C. H. Wu, Y. L. Lin: *Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.
3. P. Michel, U. Lauther, P. Duzy: *The Synthesis Approach to Digital System Design*. Kluwer Academic Publishers, 1992.
4. D. D. Gajski, F. Vahid, S. Narayan, J. Gong: *Specification and Design of Embedded Systems*. P T R Prentice Hall, Englewood Cliffs, 1994.
5. D. L. Perry: *VHDL*. McGraw-Hill, 1991.
6. S. Mazor, P. Langstraat: *A Guide to VHDL*. Second Edition. Kluwer Academic Publishers, 1993.
7. N. Wirth: *Lola: An Object-Oriented Logic Description Language*. Technical Report 215, Institute for Computer Systems, ETH Zurich, May 1994.
8. *** *EasyABEL Design Software*. Data I/O Corp., San Jose, CA, 1992.
9. R. Weiss: *Schematics Battle Equations for Design Representation*. EDN, Dec. 19, 1991, pp. 62-68.
10. R. Beachler: *Hardware Description Language and PLDs Generate Waveforms*. EDN, May 23, 1991, pp. 131-140.
11. S. Carlson, E. Girczyc: *Understanding Synthesis Begins with Knowing the Terminology*. EDN, Sept. 3, 1992, pp. 125-131.
12. T. Yu: *HDL Simulation Models*. EDN, May 13, 1993, pp. 133-140.
13. J. C. Napier: *Multilevel ASIC Modeling*. EDN, April 29, 1993, pp. 75-80.
14. D. Patel, M. Schlag, M. Ercegovac: *An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms*. Lecture Notes in Computer Science, Vol. 201: Functional Programming Languages and Computer Architecture, 1985, pp. 238-255.
15. J. P. Banâtre, D. Lavenier, M. Vieillot: *From High Level Programming Model to FPGA Machines*. Rapport de recherche n° 2240, INRIA, 1994.
16. M. Markowitz: *Hands-On VHDL Design Project*, Part 1: *The Adventure Begins*. EDN, January 7, 1993, pp. 58-67.
17. M. Markowitz: *Hands-On VHDL Design Project*, Part 2: *Discovering the Design Foundation*. EDN, January 21, 1993, pp. 74-82.
18. M. Markowitz: *Hands-On VHDL Design Project*, Part 3: *Writing the Code*. EDN, February 4, 1993, pp. 94-103.
19. M. Markowitz: *Hands-On VHDL Design Project*, Part 4: *Simulating the Function*. EDN, February 18, 1993, pp. 112-124.
20. M. Markowitz: *Hands-On VHDL Design Project*, Part 5: *Synthesizing the Gates*. EDN, March 4, 1993, pp. 105-115.

21. M. Markowitz: *Hands-On VHDL Design Project*, Part 6: *Reaching the Conclusion*. EDN, March 18, 1993, pp. 207-212.

22. J. Sztipanovits, D. M. Wilkes, G. Karsai, Cs. Biegl, L. E. Lynd: *The Multipgraph and Structural Adaptivity*. IEEE Transactions on Signal Processing, Vol. 41, No. 8, 1993, pp. 2695-2716.

23. S. Narayan, F. Vahid., D. D. Gajski: *System Specification and Synthesis with the SpecCharts Language*. Proceedings of the International Conference on Computer Aided Design, 1991.

24. S. Dudani, E. Stabler: *Types of Hardware Description*. Proceedings of the IFIP WG 10.2 Sixth International Symposium on Computer Hardware Description Languages and their Applications, pp. 127-136.

25. B. S. Davie, G. J. Milne: *The Role of Behaviour in VLSI Design Languages*. Proceedings of the IFIP WG 10.2 Working Conference on From HDL Descriptions to Guaranteed Correct Circuit Designs, pp. 3-20.

26. P. Eles, K. Kuchinski, Z. Peng, A. Doboli: *Specification of Timing Constraints in VHDL for High-Level Synthesis*. Proceedings of ConTI '94-International Conference on Technical Informatics, vol. 5., pp. 26-36.

27. A. J. Martin: *Tomorrow's Digital Hardware will be Asynchronous and Verified*. Department of Computer Science, California Institute of Technology, Pasadena CA, 1993.