MEDII AVANSATE DE DEZVOLTAREA SOFTWARE-ULUI BAZATE PE METODE FORMALE

mat.Ileana Rabega

Institutul de Cercetări în Informatică

Rezumat:

Lucrarea prezintă modul în care pot fi utilizate metodele formale în medii avansate de dezvoltare a software-ului ; se indică o arhitectură cadru şi se menţionează exemple de astfel de medii.

1. Introducere

Mediile avansate de dezvoltare a software-ului care utilizează metode formale sînt în prezent reprezentate pe de o parte, prin medii experimentale foarte sofisticate - elaborate în institute de cercetare / învăţămînt superior - şi, pe de altă parte, prin sisteme industriale. În continuare sînt prezentate (4) şapte concluzii despre modul cum sînt primite de utilizatori metodele formale încorporate sau nu în medii de dezvoltare tradiţionale şi avansate:

1) metodele formale sînt foarte utile pentru detectarea timpurie a erorilor şi pentru eliminarea anumitor clase de erori;
2) metodele formale impun proiectantului ca, în paşi succesivi, să aibă o concepţie clară (să-şi clarifice concepţia) asupra sistemului pe care doreşte să-l construiască;
3) metodele formale sînt utile pentru orice aplicaţie; pentru sistemele necritice este suficientă o specificaţie formală, la intrare, care se rafinează fără a mai fi însă verificată formal corectitudinea rafinărilor succesive; în schimb, pentru sisteme critice este necesară formalizarea întregului ciclu de viaţă;
4) metodele formale se bazează pe specificaţii matematice, care sînt mai uşor de înţeles decît programele, fiind mai abstracte;
5) specificaţiile formale contribuie la scăderea costurilor de dezvoltare (în special datorită primelor faze ale ciclului de dezvoltare, care sînt supuse la erori în cazul metodelor informale sau semi-formale);
6) specificaţiile formale ajută pe clienţi să înţeleagă ce software cumpără;
7) metodele formale se utilizează cu succes în proiecte la scară industrială.

Aceste concluzii s-au obţinut în urma experimentelor efectuate la Praxis Systems (proiectul CASE), la IBM (proiectul CICS) şi la alte firme, utilizînd proiecte bazate pe metode formale.

2. Metode, limbaje formale şi sisteme software care le implementeaza

Pe plan teoretic s-au întreprins în domeniul metodelor/limbajelor formale, precum şi al sistemelor software care le implementează în ultimii zece ani cercetări în trei direcţii:

1) construirea semanticii formale a limbajelor de specificare/programare;
2) specificarea semanticii procesului de dezvoltare a software-ului;
3) includerea rezultatelor obţinute la 1) şi 2) în medii integrate şi avansate de dezvoltare a software-ului secvenţial şi concurent.

Prima direcţie de cercetare a condus, pe de o parte, la obţinerea de noi metode/limbaje formale de specificare (executabilă) şi, pe de altă parte, la obţinerea de modele formale pentru limbajele de programare/dezvoltare existente (de exemplu, Ada).

O metodă de dezvoltare este formală în măsura în care posedă un aparat matematic riguros, pentru descrierea proprietăţilor sistemului.

Se pot astfel defini precis noţiuni cum ar fi consistenţa şi completitudinea proprietăţilor sistemului, precum şi procesul de dezvoltare.

Descrierea proprietăţilor sistemului real este posibilă cu ajutorul unui limbaj de specificare formală. În timp ce o metodă formală poate sau nu să fie susţinută de instrumente, un limbaj formal este susţinut de instrumente. Cu ajutorul unui limbaj de specificare formală, se poate verifica măsura în care o specificaţie este realizabilă, demonstrînd corectitudinea proprietăţilor sistemului, fără a fi necesară execuţia software-ului corespunzător.

Un limbaj formal se caracterizează printr-o sintaxă şi o **semantică formale**, precum şi printr-o relaţie de **satisfacere** care defineşte mulţimea obiectelor, care satisfac o specificaţie/program.

Sintaxa unui limbaj formal se dă specificînd o sintaxă abstractă, care reprezintă un sistem de funcţii pentru construirea şi descompunerea obiectelor sintactice compuse. Definirea funcţiilor şi a relaţiilor dintre ele se face cu ajutorul unor axiome.

Semantica unui limbaj formal se poate defini în două moduri:

A) **sintetic**: înţelesul unei specificaţii/program este definit prin compunerea înţelesului subspecificaţiilor/subprogramelor sale, utilizînd operatori ataşaţi construcţiilor de limbaj; exemple de semantici sintetice sînt semanticile denotaţionale şi semanticile algebrice;
B) **analitic**: doar sistemul în ansamblul său (reprezentat printr-un set de specificaţii sau de programe) are un înţeles (nu şi fragmentele de specificaţii/programe); exemple de semantici analitice sînt semanticile operaţionale.

Relaţia de satisfacere a unui limbaj formal are două roluri:

- de a scoate în evidenţă diferite aspecte ("views")

ale componentelor/ obiectelor unui sistem;
- de a impune restricţii asupra sistemului.

A doua direcţie de cercetare a condus la obţinerea de modele semantice ale procesului de dezvoltare a software-ului secvenţial (modele care sînt în relativ consens), precum şi la o diversitate de modele semantice ale procesului de dezvoltare a software-ului concurent; în acest din urmă caz, nu există opinii convergente privind modelul semantic optim pentru abordarea procesului de dezvoltare a software-ului concurent.

Un exemplu de model secvenţial (inclus în mediul avansat de dezvoltare PROSPECTRA) este bazat pe abordarea dezvoltării, ca un obiect formal cu dublu rol de documentaţie pentru acţiunile de analiză/proiectare efectuate şi de plan pentru dezvoltări ulterioare.

Exemple de modele concurente sînt sistemele de tranziţii etichetate şi arborii de sincronizare (limbajul CCS), reţelele Petri, modelul trace (limbajul CSP).

A treia direcţie de cercetare a condus la obţinerea de medii integrate şi avansate de dezvoltare a software-ului secvenţial şi concurent, care se bazează pe trei caracteristici esenţiale:

A) ciclul de viaţă operaţional;

B) existenţa unui model formal, care stă la baza tuturor prelucrărilor mediului de dezvoltare;

C) existenţa unei baze de cunoştinţe, care constituie domeniul de expertiză a mediului de dezvoltare.

Pe lîngă direcţiile de cercetare menţionate, metodele/limbajele formale, precum şi mediile care le implementează fac în prezent obiectul aplicaţiilor la scară industrială. Se dau ca exemplu următoarele domenii:

Prelucrarea tranzacţiilor - sistemul CICS (firma IBM) conţine peste 500.000 linii de cod sursă şi este construit de circa 20 de ani. IBM a utilizat metoda de specificare formală Z pentru respecificarea interfeţelor cheie ale sistemului CICS, în scopul îmbunătăţirii mentenabilităţii sale;

Hardware - utilizarea metodei de specificare Z pentru specificarea unei familii de osciloscoape (firma TEKTRONIX);

Compilatoare - utilizarea metodelor formale pentru construirea de compilatoare la scară industrială (DANISH DATAMATIK CENTER);

Medii/instrumente de dezvoltare - proiectul CASE (PRAXIS SYSTEMS);

Controlul reactoarelor - Firma ROLLS ROYCE AND ASSOCIATES a utilizat un limbaj formal propriu, cu interfaţă apropiată de limbajul natural, pentru specificarea software-ului de control al reactorului nuclear.

2. Exemple de clasificări ale metodelor/limbajelor formale

Metodele formale pot fi clasificate după mai multe criterii [7]. După **modul cum este definită comportarea sistemului**, metodele formale se împart în două mari grupe:

A) **metode orientate pe model**, în care sistemul este specificat direct, prin construirea unui model al sistemului, utilizînd structuri matematice ca liste, mulţimi, relaţii, funcţii, şiruri;

B) **metode orientate pe proprietate**, în care sistemul este specificat indirect, cu ajutorul unui set de proprietăţi, de obicei sub forma unor axiome pe care sistemul trebuie să le satisfacă; în acest caz, trebuie introdus numărul minim de restricţii, deci de proprietăţi, care să satisfacă cerinţele sistemului; evident că, numărul implementărilor posibile este cu atît mai mare, cu cît numărul de restricţii este mai mic.

Ca exemple de metode/limbaje formale orientate pe model se menţionează:
- pentru sisteme secvenţiale: metoda VDM (cu limbajul VDL), metoda Z;
- pentru sisteme concurente şi distribuite: reţelele Petri, limbajul CCS (Milner), limbajul CSP (Hoare).

Exemple de metode/limbaje orientate pe proprietate sînt:
- pentru sisteme secvenţiale: familia de limbaje Larch, limbajele OBJ2, OBJ3, P-AnndA-S, Act One, Pluss, metoda Z;
- pentru sisteme concurente şi distribuite: logica temporală, limbajul LOTOS (combinaţie între limbajele Act One şi CCS).

Relativ la semantica limbajelor formale, se poate spune că, în general, limbajele orientate pe model au o semantică denotaţională, iar cele orientate pe proprietate au o semantică algebrică. În plus, ambele tipuri de limbaje formale pot avea şi/sau o semantică operaţională, aceasta din urmă fiind utilă în special la nivelul sistemului construit cu ajutorul unui limbaj formal. De exemplu, limbajul OBJ3 are o semantică algebrică şi una operaţională, care se utilizează atît la specificare cît şi la verificarea formală; pentru limbajul CSP, au fost construite semantici denotaţionale şi operaţionale etc.

Metodele/limbajele formale se împart în două mari grupe, după **forma de prezentare a specificaţiilor**: textuale şi grafice. În general, cele două forme coexistă, forma grafică fiind elaborată ulterior celei textuale şi avînd ca obiectiv să facă interfaţa cu utilizatorul cît mai prietenoasă.

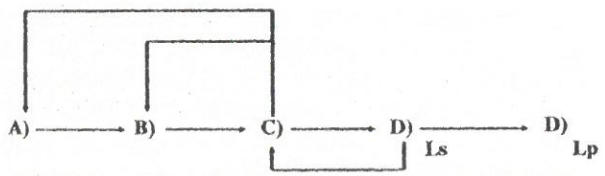
O altă clasificare, în două categorii, a metodelor/limbajelor formale se face după **comportamentul specificaţiilor rezultate**:
- executabile (de exemplu, limbajele OBJ2, OBJ3);
- neexecutabile (de exemplu, metoda VDM cu limbajul VDL, metoda Z).

În sfîrşit, o altă clasificare foarte importantă a metodelor/limbajelor formale este legată de **tipul instrumentarului asociat**:
- instrumente orientate pe metodă/limbaj (de exemplu, editoarele sintactice);
- instrumente generale [de exemplu,

demonstratoare de teoreme - Rewrite Rule Laboratory (RRL), Boyer-Moore Theorem Prover etc].

3. Ciclul de viaţă susţinut de un mediu avansat de dezvoltare a software-ului bazat pe metode formale

Ciclul de viaţă, susţinut de un mediu avansat de dezvoltare a software-ului bazat pe metode formale, se caracterizează prin următoarele etape: [1], [2], [5]:
A) analiza cerinţelor;
B) specificare formală;
C) verificarea formală a specificaţiei;
D) implementarea specificaţiei în limbajul de specificare (executabilă) şi în limbaje de programare.
Etapele A)–D) corespund ciclului de viaţă operaţional [6], în cazul cînd specificaţiile sînt executabile. Ordinea înlănţuirii etapelor este următoarea (fig. 1):



Ls - limbaj de specificare executabilă
Lp - limbaj de programare

Figura 1: Ciclul de viaţă operaţional

Deci ciclul de viaţă operaţional s-ar putea prezenta sub forma a două subcicluri:

- analiza cerinţelor, specificarea formală, verificarea formală precum şi rafinarea succesivă a unei specificaţii iniţiale, executabile (program) descrisă cu ajutorul unui limbaj formal de specificare executabilă; trecerea de la un pas/un grupaj de paşi de dezvoltare la alt pas, presupune demonstrarea corectitudinii specificaţiei în raport cu metodele formale pe care se bazează sistemul; acest ciclu se termină cu specificaţia executabilă finală (prototip);
- codificarea automată/asistată de utilizator a specificaţiei executabile finale în diferite limbaje de programare.

În cele ce urmează se detaliază conţinutul etapelor A)–D) menţionate mai sus.

3.1. Analiza cerinţelor

Analiza cerinţelor este un proces complex, de importanţă vitală pentru viitorul sistem. Pe lîngă culegerea datelor despre sistem, analiza cerinţelor înglobează şi aspectele de organizare a acestor date, astfel încît să poată fi deduse cerinţele relevante sau cele absente.
Datorită naturii sale intuitive, procesul de analiză poate fi cu greu acoperit de o metodă formală, care presupune rigoare, deci cerinţe bine precizate. Într-un mediu avansat pentru dezvoltarea software-ului, se pot cupla metodele formale cu metode euristice/informale sau semi-formale, aşa încît mediul să poată asista etapa de analiză a cerinţelor.
În cazul în care mediul asistă construirea de sisteme secvenţiale se pot introduce tehnici de analiză statică a proprietăţilor structurale (de exemplu, diagrame de flux de date).
În cazul în care mediul asistă şi construirea de sisteme concurente, o soluţie de a putea pune în evidenţă aspectele dinamice constă în construirea de prototipuri de explorare a cerinţelor, care aplică tehnici de animaţie. Mai precis, cu un set minim de cerinţe se construieşte o specificaţie iniţială a sistemului şi se identifică părţi din ea care pot fi animate (executate), în conformitate cu anumite obiective (scenarii); în urma acestui proces, este posibilă identificarea de noi cerinţe pentru viitorul sistem. Această tehnică se practică în principal pentru componentele critice ale unui sistem.
Nu există o departajare netă între analiza şi specificarea cerinţelor. Este cert însă că, pentru a putea specifica formal un sistem, este necesar ca, cerinţele obţinute în urma procesului de analiză să fie clare.

3.2. Specificarea formală

Problema construirii specificaţiilor formale trebuie abordată în mai mulţi paşi:

- specificarea cerinţelor critice în termeni matematici precişi;
- construirea unei specificaţii formale, de nivel înalt (care omite detaliile de implementare, cum ar fi de exemplu, limitele la nivelul resurselor), utilizînd descrieri matematice precise ale noţiunilor, într-un limbaj formal de specificare executabilă;
- rafinarea (implementarea) specificaţiei formale în acelaşi limbaj de specificare executabilă;
- implementarea specificaţiei formale într-un limbaj de programare de nivel înalt;
- verificarea formală a programului obţinut, în raport cu cerinţele critice formale, exprimate iniţial.

Ultimul pas este destul de dificil de demonstrat, de aceea se preferă demonstrarea corectitudinii specificaţiei formale după fiecare pas (a se vedea 3.3).
S-a menţionat deja faptul că, utilizînd metode/limbaje formale, un sistem secvenţial sau concurent se poate specifica în două moduri:

- **direct**, prin construirea unui model matematic al sistemului (**specificaţii orientate pe model**);
- **indirect**, prin construirea unui set de proprietăţi

(sub forma unor axiome), pe care sistemul trebuie să le satisfacă (**specificaţii orientate pe proprietate**).

În cele ce urmează se vor da exemple simple de construire a specificaţiilor formale în limbaje de specificare cunoscute, care să asiste abordări secvenţiale şi concurente.

Cazul secvenţial, specificaţii orientate pe model

Specificarea formală a unei tabele de simboluri în limbajul VDL (Vienna Development Language) se indică în figura 2 [7].

```
ST=map Key to Val

INIT( )
ext wr st: ST
post st' ={ }

INSERT (k:Key, v:Val)
ext wr st: ST
pre k ∉ dom st
post st' = st ∪ {k→v}

LOOKUP (k:Key)v:Val
ext rd st: ST
pre k ∈ dom st
post v' = st (k)

DELETE (k:Key)
ext wr st: ST
pre k ∈ dom st
post st' = st – {k}
```

Fig. 2 - Specificaţia unei tabele de simboluri în limbajul VDL

Starea tabelei este modelată cu ajutorul unei transformări parţiale, de la mulţimea identificatorilor la mulţimea valorilor asociate (ST = map Key to Val).

Observaţie: Limbajul VDL implementează patru modele: şir/listă ("tuple"), mulţime ("set"), transformare ("map") şi arbore ("tree").

Tabela conţine patru operaţii: de iniţializare (INIT), de inserare a unui nou identificator (INSERT), de ştergere a unui identificator (DELETE) şi de producere a valorii unui identificator (LOOKUP). Operaţiile sînt însoţite de predicate, numite pre şi postcondiţii. O precondiţie este un predicat care trebuie să întoarcă valoarea "adevărat", în fiecare stare, la momentul apelului operaţiei; dacă întoarce "fals", atunci comportamentul operaţiei nu este specificat. O postcondiţie reprezintă un predicat care întoarce valoarea "adevărat" în starea consecutivă apelului operaţiei.

Notaţia ', care succede variabilele de stare, semnifică starea tabelei de simboluri după execuţia unei operaţii.

Operatorul "–" exprimă diferenţa a două mulţimi.

Operaţia INIT iniţializează tabela la mulţimea vidă.

Operaţia INSERT introduce un nou indentificator (adresa lui) în tabelă, cu condiţia ca acesta să nu existe deja în tabelă.

Operaţia DELETE modifică tabela ştergînd un indentificator (adresa lui) din ea, cu condiţia ca acest indentificator să aparţină tabelei. Operaţia LOOKUP întoarce valoarea transformării unui identificator (prin map) cu condiţia ca acesta să aparţină tabelei. Precondiţiile şi postcondiţiile asociate semnifică faptul că, operaţia LOOKUP nu modifică tabela de simboluri (deci st'=st) prin utilizarea declaraţiei rd (read-only-access) pentru variabila de stare, în timp ce operaţiile INSERT Şi DELETE modifică tabela de simboluri (utilizează declaraţia wr "write-and-read access").

Cazul secvenţial - specificaţii orientate pe proprietate

Se prezintă o specificaţie orientată pe proprietate a unei tabele de simboluri, utilizînd familia de limbaje Larch. Aici, o specificaţie se descrie la două niveluri (fig. 3):

- un nivel dependent de stare, similar specificaţiei în limbajul VDL, numit specificaţie de interfaţă;
- un nivel independent de stare, conţinînd descrierea prin intermediul algebrelor multisortate a proprietăţilor datelor accesate de programe, numit specificaţie caracteristică ("trait").

În cadrul primului nivel, clauzele **requires** şi **ensures** ţin locul pre şi postcondiţiilor din VDL, iar clauza **modifies** listează obiectele a căror valoare poate fi modificată prin execuţia operaţiei. Deci LOOKUP nu modifică starea tabelei de simboluri, în timp ce INSERT şi DELETE o modifică.

Nivelul independent de stare este o specificaţie algebrică, care conţine:

- o signatură, reprezentată printr-un set de declaraţii ale simbolurilor de funcţii

şi

- axiome, reprezentate printr-un set de ecuaţii care definesc semantica simbolurilor de funcţii prezente în signatură.

Tipul abstract de date SymTab este definit pe mulţimea de date ("sort") S, utilizînd funcţii definite pe alte două mulţimi de date: K (mulţimea identificatorilor) şi V (mulţimea valorilor asociate).

Apar următoarele clauze:

- **generated by:** exprimă faptul că, valorile tabelei de simboluri se pot reprezenta prin termeni formaţi doar cu două simboluri funcţionale: emp şi add;
- **partitioned by:** exprimă faptul că, doi termeni sînt egali, dacă nu pot fi diferenţiaţi prin nici una dintre funcţiile listate în clauză; în exemplul de faţă, se poate utiliza această proprietate pentru a demonstra că, operaţia INSERT este comutativă, deci că ordinea inserării de perechi distincte cheie-valoare nu este importantă;
- **exempting:** exprimă excepţiile de scriere a ecuaţiilor, în exemplul de faţă că rem (emp) şi find (emp) nu au decît membri stîngi.

Se observă că, funcţiile definite în partea din specificaţia Larch, exprimată algebric ("trait") coincid cu cele care apar în specificaţia Larch de interfaţă. Spre

deosebire de VDL, Larch nu are simboluri predefinite, toate simbolurile de operaţii fiind definite de utilizator, ceea ce poate constitui atît un avantaj (nu trebuie să memoreze nimic), cît şi un dezavantaj (trebuie să definească algebric toate simbolurile de funcţii introduse).

```
symbol_table is data type based on S from SymTab
 init = proc( ) returns (s:symbol_table)
 ensures s' = emp ∧ new (s)
 insert = proc (s:symbol_table,k:key,v:val)
 requires ~ isin (s,k)
 modifies (s)
 ensures s'=add (s, k, v)
 lookup = proc (s:symbol_table, k:key) returns (v:val)
 requires isin (s,k)
 ensures v'=find(s,k)
 delete = proc (s:symbol_table, k:key)
 requires isin (s,k)
 modifies (s)
 ensures s'=rem (s,k)
end symbol_table
SymTab: trait
 introduces
   emp:            →S
   add: S, K, V    →S
   rem: S, K       →S
   find: S, K      →V
   isin: S, K      →Bool
 asserts
   S generated by (emp, add)
   S partitioned by (find, isin)
   for all (s:S, k,k1:K, v:V)
rem (add (s,k,v),k1) == if k=k1 then s
          else add (rem (s,k1), k,v)
find (add (s,k,v), k1)==if k=k1 then v
                        else find (s,k1)
isin (emp, k)==false
isin (add (s, k, v),k1)==(k=k1) V isin (s,k1)
 implies
   converts (rem, find, isin) exempting (rem(emp),
                                         find(emp))
end SymTab
```

Figura 3: Specificaţia unei tabele de simboluri în familia de limbaje Larch

Cazul concurent - specificaţii orientate pe model

Limbajul CSP se bazează pe modelul de concurenţă "trace", care presupune că, fiecare proces al viitorului sistem se exprimă printr-o mulţime de trasări (traces), iar o trasare reprezintă un şir finit sau infinit de acţiuni. Tipul de concurenţă susţinut de modelul trace este alternanţa de acţiuni ("interleaving"), ceea ce presupune ca, în cazul proceselor comunicante, o singură acţiune a unui proces să se execute la un moment de timp. Procesele comunică între ele numai transmiţînd mesage prin canale.

În figura 4.[7] se dă specificaţia unui buffer nemărginit în limbaj CSP. BUFFER este un proces definit recursiv astfel:

- prima clauză, $P_{<>}$ =letf?m→$P_{<m>}$ exprimă faptul că, în ipoteza în care BUFFER-ul este vid, dacă apare acţiunea ca mesajul m să se găsească pe canalul stîng (left?m), atunci în BUFFER se citeşte acest mesaj ($P_{<m>}$); notaţia x→P se interpretează astfel: pentru x=acţiune şi P=proces, x→P este procesul care execută mai întîi acţiunea x şi apoi se comportă ca şi procesul P;

-a doua clauză,

$P_{<m>\frown s}$ = (left?n→$P_{<m>\frown s \frown <n>}$ | right ! m→P_s)

se referă la faptul că, în ipoteza cînd BUFFER-ul este nevid, există două posibilităţi:

a) BUFFER-ul primeşte un alt mesaj pe canalul din stînga, pe care îl concatenează la sfîrşit;

b) BUFFER-ul întoarce primul mesaj găsit pe canalul drept.

În CSP notaţia s^t exprimă faptul că şirul s este concatenat cu şirul t. Bara, |, exprimă operatorul de alegere între procese, adică pentru x,y = acţiuni distincte, x→P | y →Q reprezintă procesul care execută iniţial, fie acţiunea x şi apoi continuă sub forma procesului P, fie acţiunea y şi apoi continuă sub forma procesului Q.

Aceste două clauze reprezintă programul CSP prin care se descrie un buffer nemărginit.

Ultima clauză din figura 4 exprimă relaţiile care trebuie să se verifice pentru ca acest program CSP să fie conform cu modelul trace. În cazul de faţă trebuie să fie satisfăcute două condiţii:

1) ordinea de transmitere a mesajelor este de la stînga spre dreapta; mesajele sînt transmise o singură dată şi în aceeaşi ordine în care au fost primite, ceea ce se descrie prin s ≤ t (adică şirul s este un prefix al şirului t);

2) procesul nu se opreşte niciodată: el nu poate refuza comunicaţia, nici pe canalul stîng, nici pe cel drept, ceea ce revine la faptul că mesajele de intrare sînt eventual transmise.

BUFFER = $P_{<>}$

where$P_{<>}$ = left ?m→$P_{<m>}$

and

$P_{<m>\frown s}$ = (left?n→ $P_{<m>\frown s\frown <n>}$ | right !m→P_s)

BUFFER sat (right ≤ left) ∧ (if right = left then left ∉ ref else right ∈ ref

Fig 4. Specificaţia unui buffer nemărginit în limbaj CSP şi condiţia de verificare (demonstrare) a corectitudinii specificaţiei

Cazul concurent - specificaţii orientate pe proprietate

Logica temporală constituie o metodă de specificare a proprietăţilor sistemelor concurente. Operatorii cei mai uzuali pentru logica temporală sînt □, ◇ şi O cu următoarea interpretare (relativă la un sistem inferenţial de logică temporală):

- □P exprimă faptul că, predicatul P are valoarea "adevărat" pentru toate stările viitoare;
- ◊ P exprimă faptul că, există o stare viitoare pentru care predicatul P va avea valoarea "adevărat";
- O P exprimă faptul că, în starea imediat următoare celei curente predicatul P are valoarea "adevărat".

În figura 5. se dă o specificație a unui buffer nemărginit utilizînd notații din logica modală. Formulele sînt interpretate relativ la șiruri de acțiuni și, pentru claritate, se explică în cele ce urmează fiecare formulă în parte. Un buffer are un canal stîng de intrare și un canal drept de ieșire. Expresia <c!m> exprimă acțiunea de a plasa mesajul m pe canalul c.

Primul predicat, $< right\ !\ m > \Rightarrow \Diamond < left\ !\ m >$
exprimă faptul că, orice mesaj transmis către canalul drept (<right ! m>) trebuie să fi fost anterior plasat pe canalul stîng. ($\Diamond$ <left ! m >).

Al doilea predicat,
$(< right\ !\ m >) \wedge O \Diamond (< right\ !\ m' >) \Rightarrow$
$\Rightarrow \Diamond (< left\ !\ m > \wedge O \Diamond < left\ !\ m' >)$
exprimă faptul că mesajele sînt transmise sub forma FIFO. Dacă mesajul m plasat pe canalul de ieșire este precedat de alt mesaj m' plasat, de asemenea, pe canalul de ieșire ($O \Diamond$ <right ! m'>), atunci trebuie să existe o altă acțiune (al doilea $\Diamond$) de plasare a mesajului m pe canalul de intrare (<left!m>) și, chiar mai mult, o acțiune anterioară de plasare a lui m' pe canalul de intrare înaintea lui m ($O \Diamond$ <left ! m'>)

Al treilea predicat
$(< left\ !\ m > \wedge O \Diamond < left\ !\ m' >) \Rightarrow (m \neq m')$
exprimă faptul că toate mesajele sînt unice. Pentru fiecare mesaj m plasat la momentul curent pe canalul de intrare și pentru fiecare mesaj m' plasat anterior lui m tot pe canalul de intrare ($O \Diamond$ <left ! m'>), m și m' nu sînt egale. Această axiomă (exprimată prin al treilea predicat) este esențială pentru validitatea specificației. Fără această axiomă, un buffer care întoarce cîte două copii ale ieșirii, ar putea fi considerat că funcționează corect.

Primele trei predicate indică proprietățile de securitate ("safety") ale sistemului (inclusiv ale mediului său), iar al patrulea predicat,

$(< left\ !\ m\ !>) \Rightarrow \Diamond (< right\ !\ m >)$

indică o proprietate de existență ("liveness") și anume că, orice mesaj de intrare este eventual transmis.

(1) $< right\ !\ m > \Rightarrow \Diamond < left\ !\ m >$

(2) $(< right\ !\ m > \wedge O \Diamond < right\ !\ m' >) \Rightarrow$
$\Rightarrow \Diamond (< left\ !\ m > \wedge O \Diamond < left\ !m' >)$

(3) $(< left\ !\ m > \wedge O \Diamond < left\ !\ m' >) \Rightarrow (m \neq m')$

(4) $(< left\ !\ m >) \Rightarrow \Diamond (< right\ !\ m >)$

Fig. 5: Specificația cu ajutorul logicii temporale a unui buffer nemărginit

3.3. Verificarea formală a specificației

Primul pas al procesului de verificare îl constituie verificarea **informală** a faptului că, cerințele formale critice reflectă cerințele critice ale beneficiarului. Acest pas trebuie să fie informal pentru că, cerințele beneficiarului sînt informale; de asemenea, pasul este neautomatizat.

Apoi se demonstrează faptul că, prima specificație formală obținută este conformă cu cerințele formale critice. Demonstrația depinde de modul de specificare: orientat pe model sau orientat pe proprietate.

În abordarea orientată pe model, se specifică efectul execuției fiecărei operații pe baza anumitor condiții care trebuie satisfăcute, atunci cînd operația este apelată (pre și postcondițiile din VDL, de exemplu). Dacă starea sistemului anterior apelului unei operații satisface condițiile de intrare, atunci starea sistemului, obținută în urma execuției operației, va satisface condițiile de ieșire. Se verifică faptul că starea inițială satisface cerințele critice formale și că fiecare operație păstrează (lasă invariante) cerințele critice.

În abordarea orientată pe proprietate se specifică, la nivelul inițial de dezvoltare software, comportarea esențială a unui tip abstract de date prin axiome, exprimate într-un limbaj asemănător calculului cu predicate de ordinul întîi și se verifică dacă, prin dezvoltările de simboluri funcționale și ecuații specifice lor de la pasul următor de dezvoltare software, aceste axiome sînt în continuare satisfăcute; verificările se referă la construirea morfismelor de algebre dintre cei doi pași de dezvoltare succesivi [2].

Apoi se demonstrează că, următorul nivel de specificație formală este conform cu nivelul curent. Procesul continuă pînă se ajunge la ultimul nivel de specificație formală (prototipul). De asemenea, după obținerea codului (într-un limbaj de nivel înalt), se face verificarea codului, și anume, se demonstrează că programul este conform cu prototipul [8] (fig. 6).

CERINȚE

↓ verificare informală

Cerințe critice exprimate formal

↓ verif. form. specificație

Specificație formală de nivel înalt

↓ verificare formală specificație

↓

Specificație formală de nivel scăzut (prototip)

↓ verif. form. cod

Program în limbaj de nivel înalt

Fig. 6: Ierarhia verificărilor formale

3.4. Implementarea specificației în limbajul de specificare executabilă și în limbaje de programare

Implementarea specificației executabile în limbajul de specificare executabilă a fost deja abordată (par. 3.2, 3.3). Implementarea prototipului (specificația executabilă finală)

în limbaje de programare ţine seama şi de tipul de limbaj ţintă.
Abordările algebrice, datorită nivelului ridicat de abstractizare la care se face specificarea, preferă crearea de limbaje funcţionale proprii, intermediare, între prototip şi un limbaj de programare uzual. Un exemplu este mediul avansat de dezvoltare a software-ului ISDV [2] elaborat în Germania: prototipul este implementat mai întîi în limbajul funcţional de nivel înalt ModPascal, care este orientat pe obiecte şi permite definiţii ierarhizate întocmai ca şi limbajul formal de specificare executabilă de tip algebric, propriu sistemului, numit ASPIK.
Implementarea prototipului ASPIK în ModPascal se poate face asistat de utilizator şi automat, sistemul ISDV fiind dotat cu demonstratoare de teoreme puternice, care execută sinteza programelor (atunci cînd este posibil, asigurînd astfel şi corectitudinea programelor rezultate) şi/sau verifică corectitudinea implementărilor.

4. Arhitectura unui mediu avansat de dezvoltare a software-ului bazat pe metode formale

În tabelul de mai jos sînt prezentate metodele şi instrumentele generice, care ar putea sta la baza unui mediu avansat de dezvoltare a software-ului:

Etapa	Metode	Instrumente generice
analiza cerinţe	verificări de consistenţă	editoare orientate ecran, eventual cu facilit. de animaţie
specificare formală	metode de specificare orientate pe model sau proprietate	- editoare sintactice - biblioteci de specificaţii formale reutilizabile
verificare formală specificaţie	- testare -demonstrarea consistenţei	- interpretoare simbolice - demonstratoare de teoreme pentru demonstrarea consistenţei
implementare specificaţie în limbaj de specificare (executabilă)	rafinare succesivă	- bibl. de specificaţii formale reutilizabile - demonstratoare de teoreme pentru sinteza specificaţiilor exec.
implementare specificaţie în diferite limbaje de programare	- precompilare - compilare	- editor - precompilator - compilator - biblioteci de module - depanator (simbolic)

Un mediu avansat de dezvoltare a software-ului bazat pe metode formale integrează instrumente tehnologice de tipul celor prezentate în tabelul anterior printr-o bază de date a proiectului, care conţine paşi de analiză-specificare şi paşi de demonstrare.
Mai precis, baza de date va conţine reprezentări pentru (a se vedea figura 7):

- obiecte, sub forma specificaţiilor tipurilor de date şi ale proceselor;
- proprietăţi ale obiectelor/model al sistemului (în funcţie de metoda formală de specificare utilizată în sistem);
- paşi de analiză-specificare şi demonstrare.

Fig.7: Arhitectura unui mediu avansat de dezvoltare software.

Se poate deci spune că [3], un mediu avansat de dezvoltare bazat pe metode formale:

- furnizează informaţii despre starea sa

şi

- oferă instrumente pentru două părţi:
  - **partea constructivă** - editare, sinteză (transformare), rafinare a specificaţiilor/ programelor;
  - **partea analitică** -demonstraţii(semi--automate) ale corectitudinii proprietăţilor specificaţiei atît în mod independent, cît şi atunci cînd cele două părţi sînt corelate.

Dacă ar fi dotat în plus cu o bază de cunoştinţe, un mediu avansat de dezvoltare a software-ului bazat pe metode formale, ar deveni un sistem bazat pe cunoştinţe [1], care ar evolua în timp, mărindu-şi capacitatea de expertiză pe măsura elaborării proiectelor software particulare.
Cerinţele pentru o bază de cunoştinţe de ingineria software-ului [6] sînt următoarele:

1) trebuie să existe un model de reprezentare şi de stocare a bazei de cunoştinţe;
2) mecanismul de reprezentare a bazei de cunoştinţe trebuie să aparţină unui mediu de dezvoltare bazat pe cunoştinţe, care să permită accesarea, modificarea şi execuţia cunoştinţelor;
3) baza de cunoştinţe trebuie să aibă toate informaţiile necesare despre versiunea iniţială a modelului operaţional al ciclului de viaţă (etape, faze, activităţi,subactivităţi, metode/tehnici şi instrumente disponibile, criterii de selecţie a metodelor/tehnicilor şi instrumentelor etc.)

Se consideră că, modelul cel mai adecvat de reprezentare a cunoştinţelor (informaţii cuprinzătoare

cu ocuparea eficientă a spaţiului şi acces rapid la informaţii) este oferit de reprezentarea prin frame-uri, similară modelării orientate pe obiecte.
Crearea unei baze de cunoştinţe de ingineria software-ului reprezintă un deziderat, această acţiune fiind de durată; din diferite experimente efectuate pînă în prezent cu porţiuni dintr-o astfel de bază [6], acestea s-au dovedit utile pentru componente necritice.

5. Exemple de medii avansate de dezvoltare software bazate pe metode formale.

În cele ce urmează se dau două exemple de medii de dezvoltare a software-ului secvenţial (PROSPECTRA) şi atît secvenţial cît şi concurent (FOR-ME-TOO).
PROSPECTRA (PROgram development by SPECification and TRAnsformation) a fost realizat ca proiect ESPRIT în cadrul unei echipe mixte din Germania, Spania, Franţa, Marea Britanie, Danemarca. În viziunea PROSPECTRA, o dezvoltare este un obiect formal care are două roluri:

- de documentaţie pentru acţiunile/deciziile de analiză/proiectare efectuate;
- de plan pentru dezvoltări ulterioare.

Obiectul formal "dezvoltare" ridică nivelul unei dezvoltări a unui produs software particular, la o clasă de dezvoltări similare de produse software, pe baza unei strategii: un pas de dezvoltare se obţine aplicînd o regulă de transformare general valabilă, iar o dezvoltare reprezintă o secvenţă de aplicări de reguli de transformare. Sistemul suport pentru dezvoltare din PROSPECTRA ghidează utilizatorul prin rafinări succesive, propunînd un set de reguli, care asigură respectarea corectitudinii implementării treptate a specificaţiei executabile pe parcursul evoluţiei procesului.
FOR-ME-TOO (FORmalisms, MEthods and TOOls) este un alt proiect realizat în cadrul programului ESPRIT de către o echipă mixtă din Germania, Franţa şi Italia. Scopul lui FOR-ME-TOO a fost să definească, să implementeze şi să experimenteze o tehnologie pentru dezvoltarea, verificarea şi validarea sistematică a sistemelor software secvenţiale şi concurente, bazată pe principiul reutilizării componentelor software.
Reutilizarea descrierilor şi analiza aspectelor secvenţiale ale unui sistem software se bazează pe un limbaj de specificare propriu, LPG, dotat cu semantică algebrică. Reutilizarea descrierilor şi analiza aspectelor concurente ale unui sistem software se bazează pe diferite clase de reţele Petri.

Concluzii

1. Metodele/limbajele formale sînt utile pentru detectarea timpurie a erorilor şi pentru eliminarea anumitor clase de erori.
2. Specificaţiile formale contribuie la scăderea costurilor de dezvoltare.
3. Metodele/limbajele formale se utilizează cu succes în proiecte la scară industrială.
4. Utilizînd metode/limbaje formale, un sistem se poate specifica în două moduri:
   - **direct**, prin construirea unui model matematic al sistemului (specificaţii orientate pe model);
   - **indirect**, prin construirea unui set de proprietăţi (sub forma unor axiome), pe care sistemul trebuie să le satisfacă (specificaţii orientate pe proprietate).
5. Mediile avansate de dezvoltare a software-ului susţin ciclul de viaţă operaţional şi oferă suport aplicării metodelor şi limbajelor formale de specificare executabilă pentru construirea software-ului secvenţial şi concurent.

Bibliografie

1. BALZER, R., CHEATHAM JR., TH. E., GREEN, C.: *Software Technology in the 1990's: Using a New Paradigm*.In: Computer, November, 1983.
2. BEIERLE, C., OLTHOFF, W., VOSS, A.: *Towards a Formalization of the Software Development Process*. In: Proceedings of Software Engineering'86, eds. Barnes, D., Braun, P., UK Peter Peregrinus Ltd., London, 1986, pp. 131-144.
3. BROY, M., GESER, A., HUSSMANN, H.: *Towards Advanced Programming Environments Based on Algebraic Concepts*. In: Advanced Programming Environments, LNCS, vol.244, ed. Conradi, R., Editura Springer-Verlag, Berlin, 1986, pp.454-470.
4. HALL, A.: *Seven Myths of Formal Methods*. In: IEEE Software, septembrie, 1990, pp.11-19.
5. JONES, C.B.: *VDM Proof Obligations and Their Justification*. In: VDM'87, ed. Bjorner, D., LNCS, vol.252, Editura Springer-Verlag, Berlin, 1987, pp. 260-286.
6. SYMONDS, A.J.: *Creating a Software Engineering Knowledge Base*. In: Advanced Programming Environments, ed. Conradi, R., LNCS, vol.244, Editura Springer-Verlag, Berlin, pp.494-506.
7. WING, J.M.: *A Specifier's Introduction to Formal Methods*.In: Computer, septembrie, 1990, pp. 8-24
8. KEMMERER, R.A.: *Integrating Formal Methods into the Development Process*. In:IEEE Software, septembrie, 1990, pp. 37-50