MODELAREA *UML* ŞI EXPERIMENTAREA UNOR PROBLEME DE CONCURENŢĂ

Conf.dr.ing. Anca Daniela Ioniţă
ing. Mircea Claudiu Plăişanu

Universitatea „Politehnica" Bucureşti
ancai@mag.pub.ro

Rezumat: Articolul prezintă un mediu integrat, ce permite studierea unor probleme clasice de concurenţă, oferind pentru fiecare descrierea şi modelul orientat pe obiecte în limbajul UML (Unified Modeling Language), precum şi posibilitatea de rulare cu modificarea mai multor parametri, pentru a le observa practic influenţa. Prin urmare, o primă contribuţie a acestui demers este experimentarea gradului în care un limbaj de modelare, care constituie, în prezent, un standard, poate fi utilizat pentru cazul unor aplicaţii bazate pe procese concurente. Scopul programului, realizat în limbajul Java, este de a asigura o mai bună înţelegere a dinamicii firelor de execuţie, a sincronizării între acestea şi a mecanismelor de excludere mutuală. În plus, mediul permite testarea comportamentului metodelor pentru diferite setări legate de funcţionarea în timp real.

Cuvinte cheie: modelarea orientată pe obiecte, UML, procese concurente, procese în timp real.

1. Introducere

Modelarea şi programarea proceselor concurente sunt de importanţă maximă, pentru sisteme de timp real precum cele pentru comanda şi controlul proceselor sau al fabricaţiei, sistemele critice din punct de vedere al securităţii sau al performanţei, sistemele incorporate sau sistemele hibride. Asigurarea unei fiabilităţi crescute constituie un domeniu prioritar, iar calitatea programelor dedicate sistemelor de timp real constituie cheia implementării unei inteligenţe ambientale, bazată pe sisteme complexe, aflate sub controlul societăţii civile sau care controlează mediul acesteia.

Tendinţa actuală este aceea de a defini standarde şi stiluri de modelare specifice pentru diferite domenii de aplicaţie [1], pentru a facilita interoperabilitatea între diferitele instrumente software şi a crea un mediu deschis, bazat pe reutilizarea componentelor software şi extensibilitate. În acest cadru, se plasează şi limbajul UML [2], utilizat în prezentul studiu, care a fost adoptat ca standard de către OMG (Object Management Group), o organizaţie compusă din marile companii din industria software-ului.

Aplicaţia prezentată în acest articol şi-a propus: modelarea UML a unor probleme de concurenţă clasice, implementarea orientată pe obiecte, în limbajul Java, introducerea unor facilităţi de modificare a parametrilor şi de control al rulării firelor de execuţie, testarea dinamică a diferitelor mecanisme de sincronizare şi interblocare cunoscute. Capitolul 2 descrie problemele de concurenţă abordate, cu exemplificări despre modelarea UML şi implementarea în Java a unora dintre ele, însoţite de o ilustrare a mediului integrat dezvoltat. Capitolul 3 evidenţiază facilităţile introduse suplimentar, pentru a se permite efectuarea unor teste specifice fiecărei probleme.

2. Modelarea şi implementarea problemelor de concurenţă

Studiul efectuat a vizat cinci probleme clasice de concurenţă [3], care sunt prezentate succint în continuare.

Problema filosofilor

Se consideră cinci filosofi care stau la o masă rotundă, pe care se află cinci farfurii şi cinci beţişoare chinezeşti. Filosofii pot efectua două acţiuni: să mănânce sau să gândească. Când un filosof gândeşte, acesta nu poate interacţiona cu exteriorul, iar pentru a mânca, el este condiţionat de prezenţa a două resurse, beţişoarele din stânga şi din dreapta lui. Un filosof poate ridica doar un beţişor o dată, şi nu poate lua un beţişor ce se află deja în mâna vecinului. Când termină de mâncat, el pune cele două beţişoare jos şi începe să gândească din nou. Algoritmul care să sincronizeze filosofii trebuie scris în aşa fel încât să prevină interblocarea.

Problema producător/consumator

Producătorul produce informaţii pe care le depune într-un tampon, de unde ele sunt utilizate de consumator. Pentru a permite celor două procese să funcţioneze concurent, este necesar un tampon de elemente, care poate fi umplut de producător şi golit de consumator. Producătorul şi consumatorul trebuie astfel sincronizaţi încât

consumatorul să nu încerce să consume un element ce nu a fost încă produs. Tamponul poate avea o dimensiune fixă sau poate fi nelimitat.

Problema cititori/scriitori

Cititorii citesc anumite informaţii, în timp ce scriitorii scriu într-o resursă comună, spre exemplu un fişier. Algoritmul permite cititorilor accesul concurent la resursă, în timp ce restricţionează scriitorii să aibă acces exclusiv. Trebuie evitată alocarea exclusivă a **priorităţii pentru cititori**, caz în care poate apărea imposibilitatea de a scrie, **sau pentru scriitori**, unde poate apărea imposibilitatea de a citi.

Problema fumătorilor

Se consideră un sistem cu trei procese „*fumător*", care, pentru a rula, au nevoie de trei resurse: tutun, hârtie şi chibrituri. Unul dintre fumători are tutun, altul are hârtie, iar cel de-al treilea are chibrituri. Există un al patrulea proces, *agentul,* care dispune de un stoc infinit de tutun, hârtie şi chibrituri şi oferă, la un moment dat, două din cele trei ingrediente. Fumătorul care are cel de-al treilea ingredient îşi face o ţigaretă şi o fumează, semnalându-i apoi agentului că a terminat.

Problema frizerului somnoros

O frizerie este formată dintr-o cameră de aşteptare, cu *n* scaune, şi camera frizerului, în care se află scaunul pentru tuns. Dacă un client intră în frizerie şi frizerul doarme, atunci îl trezeşte; dacă îl găseşte ocupat şi sunt scaune disponibile în camera de aşteptare, aşteaptă pe scaun, iar dacă toate scaunele sunt ocupate pleacă.

Fiecare dintre aceste probleme a fost modelată cu ajutorul UML, fapt ce poate fi vizualizat chiar din mediul integrat. Un exemplu de diagramă de clase pentru proiectarea problemei filosofilor este dat în figura 1. La nivel global, s-a creat o clasă de interfaţă, denumită Problema, care este implementată de toate clasele care descriu problemele de sincronizare sau exemplele de concurenţă. Această interfaţă cuprinde trei metode:

- Start() – metodă care activează toate firele de execuţie, necesare unei probleme;
- Stop() - metodă care opreşte toate firele de execuţie, necesare unei probleme şi „distruge" obiectele care nu mai sunt necesare;
- Create() – metodă care creează problema respectivă prin instanţierea şi iniţializarea obiectelor necesare problemei. Această metodă întoarce ca parametru un obiect de tip Jpanel, în care se realizează afişarea componentelor grafice ale problemei.

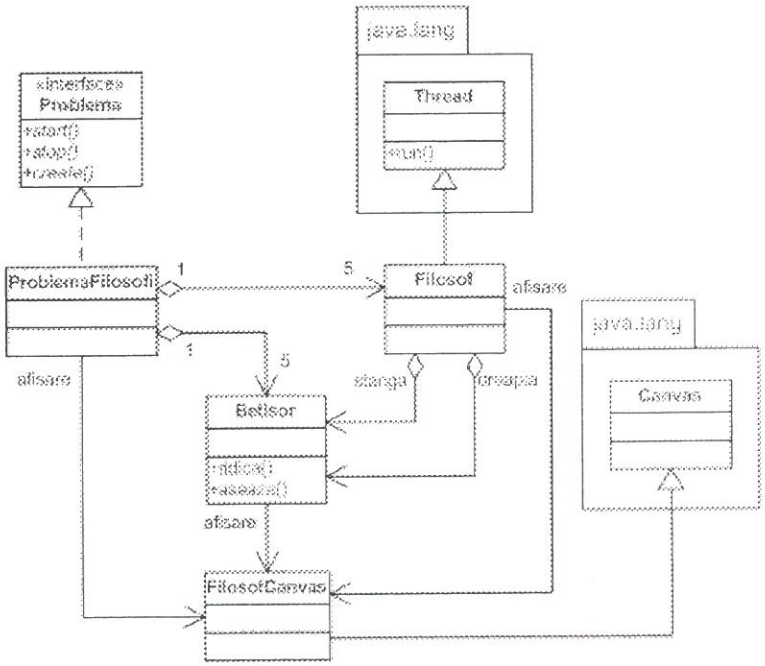

Figura 1. Diagrama de clase pentru problema filosofilor

Mediul integrat al aplicației este format dintr-o fereastră principală, împărțită în două; în partea din stânga, există două butoane radio, care permit selectarea între două categorii de probleme: exemple simple de concurență (precum semafor, cronometru, fire de execuție sincronizate/nesincronizate) și probleme clasice, de tipul celor descrise mai sus. Tot în partea din stânga, există un control *JTree*, care listează pe rând, în funcție de butonul radio selectat, elementele din fiecare categorie. În dreapta, se află un control de tip *JTabbedPane*, care este format din trei *taburi*, ce permit, pe rând, afișarea grafică a problemei selectate, vizualizarea diagramei de clase sau descrierea detaliată cu fragmente de cod al problemei respective. Un exemplu pentru problema fumătorilor este dat în figura 2. Descrierea problemelor se găsește într-un fișier HTML, asociat fiecărei probleme, și se realizează în cadrul unui control *JEditorPane*, căruia i se poate asocia un fișier HTML, prin metoda *setPage*. Implementarea în limbajul Java a integrat și a adaptat algoritmi clasici [4]; un exemplu de cod pentru problema filosofilor este definirea unui monitor asociat resursei bețișor, după cum se vede în continuare:

```
monitor MonitorBetisor {
    betisor: array î0 .. 4ş of Integer;
    OK_mananca: array (0 .. 4) of Condition;
    procedure IaBetisor (i: Integer) {
        if betisor(i) <> 2 then Wait(OK_mananca(i));
        betisor((i + 1)  mod  5) := betisor((i + 1) mod 5) – 1;
        betisor((i - 1) mod 5) := betisor((i - 1) mod 5) – 1;
    }
    procedure LasaBetisor(i: Integer) {
            betisor((i + 1) mod 5) := betisor((i + 1) mod 5) + 1;
            betisor((i - 1) mod 5) := betisor((i - 1) mod 5) + 1;
        if betisor((i + 1) mod 5) = 2 then Signal(OK_mananca((I + 1) mod 5));
        if betisor((i - 1) mod 5) = 2 then Signal(OK_mananca((I - 1) mod 5));
    }
}
```

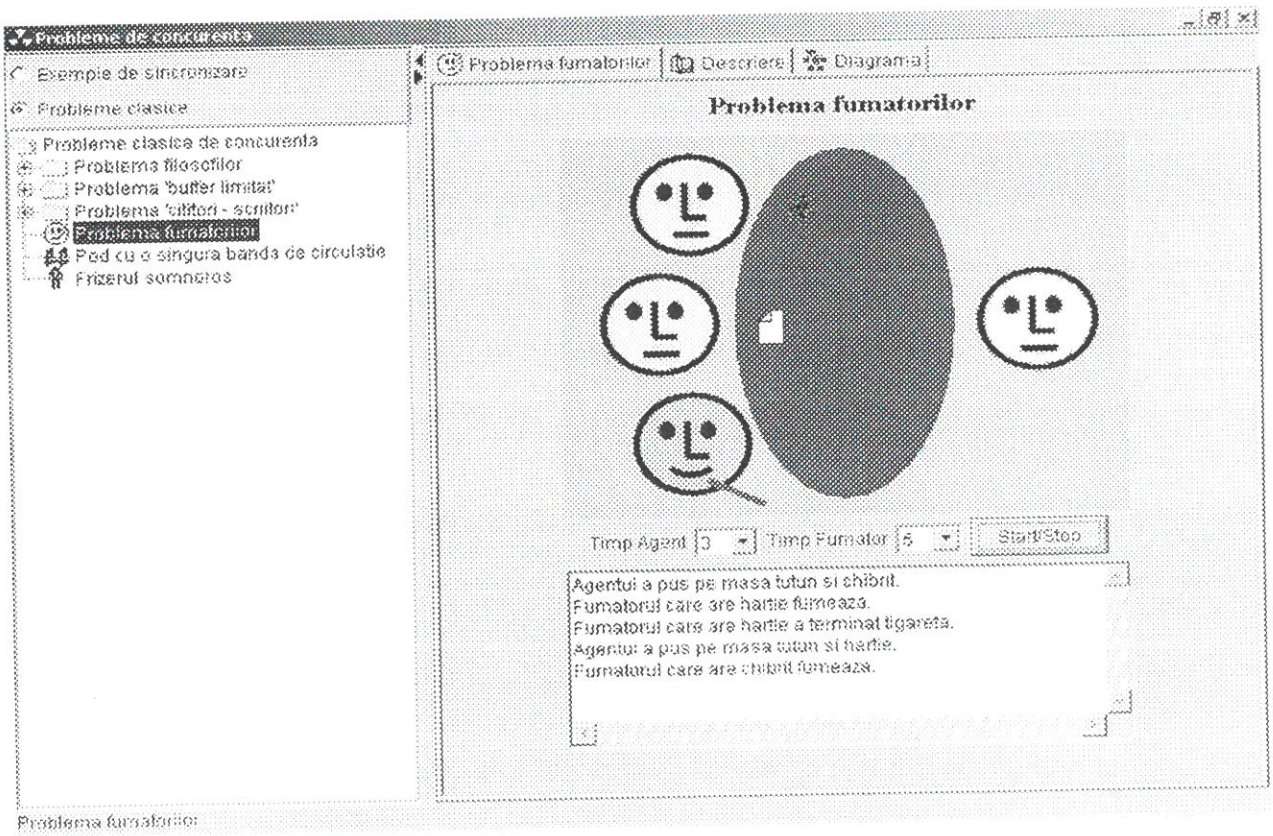

Figura 2. Mediul integrat al aplicației și problema fumătorilor

3. Facilitățile mediului integrat pentru studierea dinamică a concurenței

Programul a integrat, pentru fiecare problemă în parte, facilități pentru modificarea unor parametri ai algoritmilor, pentru a observa influența acestora asupra dinamicii. Acestea vor fi prezentate, în continuare, pentru câteva probleme. Pentru problema filosofilor, există un control al timpului petrecut pentru a mânca, deci, a consuma resursa, respectiv pentru a gândi. El poate fi verificat şi vizual, prin modificarea culorii filosofului. Micşorând intervalul de timp, se poate testa dacă algoritmul este într-adevăr robust la interblocare. În problema cititori/scriitori, se foloseşte un monitor pentru sincronizarea proceselor şi se poate controla intervalul de timp necesar pentru fiecare proces pentru a accesa resursa comună.

Problema fumătorilor foloseşte thread-uri pentru a desemna cele două tipuri de procese: *fumător,* respectiv *agent.* Programul conține un control de tip TextArea, unde este descris programul, şi care îşi modifică textul de fiecare dată când are loc o acțiune. De asemenea, există posibilitatea setării timpului în care agentul plasează cele două ingrediente, precum şi a timpului în care fumătorul care are cel de-al treilea ingredient rulează şi fumează țigareta. În cazul problemei frizerului somnoros, există posibilitatea selectării numărului de clienți, precum şi a setării intervalului de timp, în care apare un client în frizerie, şi a timpului în care este tuns fiecare client.

5. Concluzii

Integrarea mai multor algoritmi de programare concurentă şi posibilitatea urmăririi grafice a diferitelor fire de execuție oferă un instrument educațional avansat, necesar datorită complexității domeniului studiat. În plus, au fost investigate posibilitățile de modelare a problemelor de concurență cu ajutorul UML, creând o deschidere pentru încadrarea aplicațiilor de timp real, într-o abordare standardizată.

Bibliografie

1. **HOLLUNDER, B.**: Next Generation Open Development Environments: Model Driven Architecture. În: Proc. of the Int. Conf. on Information Society Technologies for Broadband Europe, 50-52, Bucharest, Romania, Oct. 2002.
2. **IONIȚĂ, A.D.**: Modelarea UML în ingineria sistemelor de programe, Editura BIC ALL, Bucureşti, 2003.
3. **PETERSON, J. L. A. SILBERSCHATZ**: Operating System Concepts, Second Edition, Addison-Wesley, 1985.
4. **MAGEE, J., J. KRAMER**: Concurrency: State Models & Java Programs, Wiley, 1999.