Specificarea problemelor clasice de concurenţă utilizând Object-Z

Dumitru CIORBĂ
Universitatea Tehnică a Moldovei
dumitru.ciorba@ati.utm.md

***Abstract* — Concurenţa, fiind o proprietate critică de sistem, trebuie luată în considerare încă de la etapele iniţiale de elaborare – arhitecturare [1]. Descrierea proprietăţilor arhitecturale se fac prin intermediul limbajelor de specificare. Din multitudinea limbajelor, datorită avantajelor specifice, anume cele formale au un potenţial ridicat de utilizare în dezvoltarea sistemelor moderne ce sunt inerent concurente. În această lucrare se va examina posibilităţile limbajului Object-Z de specificare a sistemelor concurente analizând două probleme clasice de concurenţă.**

***Index Terms* — Object-Z, probleme clasice de concurenţă, specificare.**

I. LIMBAJUL DE SPECIFICARE Z ŞI OBJECT-Z

Limbajul *Object-Z* este o extensie a limbajului formal de specificare *Z*. Definit de standardul ISO [2], limbajul *Z* permite modelarea structurilor complexe de date şi este potrivit pentru specificarea într-o manieră semi-grafică a funcţionalităţilor de nivel înalt utilizând tehnici *state-based*. O specificaţie *Z* constă din scheme de stări şi operaţii. O schemă de stare grupează totalitatea variabilelor ce definesc starea entităţii (sau a sistemului în general) şi descrie relaţiile dintre acestea (prin predicatele schemei).

De exemplu în figura 1 este prezentat spaţiul de stări al unui sistem de înregistrare a zilelor de naştere: *BirthdayBook*. Schema divizată în două specifică o mulţime de nume *known* şi o funcţie parţială *birthday* definită pe domeniul numelor *NAME*, care returnează zile de naştere (definite de tipul *DATE*) asociate cu acele nume. Partea inferioară liniei de divizare specifică relaţii care sunt adevărate în orice stare a sistemului. În cazul dat, se specifică că după orice operaţie mulţimea *known* corespunde domeniului funcţiei *birthday* şi această relaţie este invariantă pentru sistem. De asemenea din această relaţie putem extrage şi careva informaţii: fiecare persoană poate avea doar o singură zi de naştere (se datorează faptului că *birthday* este o funcţie) şi pot exista persoane cu accesaşi zi de naştere.

BirthdayBook

$known : \mathbb{P}\ NAME$
$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\ birthday$

Figura 1 – Schemă de stare definită în limbajul Z [3]

O schemă de operaţii defineşte relaţii de modificare a variabilelor ce pot aparţine unei sau mai multor scheme de stare. De exemplu în figura 2 avem definite operaţia *AddBirthday* de adăugare a unei noi înregistrări şi operaţia *FindBirthday* de găsire a unei zile de naştere. În prima operaţie declaraţia $\Delta BirthdayBook$ atenţionează că schema modifică starea, pe cînd $\Xi BirthdayBook$ indică că în rezultatul operaţiei *FindBirthday* starea sistemului nu va suferi modificări.

Operaţia *AddBirthday* specifică modificarea prin relaţia dintre *birthday* şi *birthday'*, care reprezintă explicit valorile de până şi după efectuarea operaţiei. Se menţionează că datorită invariantului de stare modificări rezultate din această operaţie va suferi şi variabila de stare *known*.

AddBirthday

$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

FindBirthday

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$
$date! = birthday(name?)$

Figura 2 – Scheme de operaţii definite în limbajul Z [3]

Variabilele *name* şi *date* sunt de intrare şi de aceea prin convenţie numele acestora sunt finisate cu semnul întrebării. Operaţia de adăugare nu poate avea loc în cazul când numele persoanei este deja în sistem. Acest fapt este asigurat de precondiţia $name? \notin known$. Operaţia *FindBirthday*, din contra, pentru efectuarea cu succes cere ca numele să fie în sistem, iar rezultatul îl oferă prin variabila de ieşire *date*, numele cărei este urmat de semnul exclamării.

Multiplele operaţii existente în sistem afectează stările, după cum s-a prezentat mai sus. Dar numărul mare al acestora cere o atenţie sporită la specificare, iar acest lucru în cazul unor sisteme complexe poate deveni impracticabil. Deci, de la sine este clar că se cere îmbunătăţirea specificării bazate pe notaţia *Z* prin adăugarea unor facilităţi de modularizare.

Incluzând o structură specială de clasă, ce încapsulează schema de stare a obiectului şi operaţiile care pot influenţa starea, limbajul *Object-Z* suportă un stil de specificare obiect-orientat. Bazându-se pe ideea de bază a paradigmei date precum că sistemul poate fi considerat o colecţie de componente în interacţiune, în [4,5,6] se trasează o relaţie de asemănare în perspectiva specificării între sistemele concurente şi obiect-orientate. Astfel *Object-Z* este considerat potrivit pentru specificarea sistemelor concurente într-o manieră *obiect-orientată* şi *state-based*.

II. PROBLEMA PRODUCĂTOR-CONSUMATOR

O clasă în *Object-Z* sintactic este reprezentată de o dreptunghi cu nume, în care pot fi definiţii de tipuri şi constante locale, cel mult o schemă de stare şi o schemă de stare iniţială şi multiple scheme de operaţii.

Pentru exemplificare se prezintă problema *producător-consumator* descrisă în [7 p. 58]: „*Producătorul produce articole care sunt aşezate pe o bandă. Consumatorul ia de pe bandă articolele şi le „consumă*". Întru asigurarea coerenţei problemei eunţate se vor defini condiţii auxiliare de sincronizare (care ţin de date) şi de planificare (care ţin de ordonarea activităţilor):

- *producătorul produce articole la orice moment de timp şi le aşează pe bandă doar dacă banda nu este plină;*
- *consumătorul poate prelua un articol doar dacă banda este nevidă;*
- *ordinea în care produsele sunt preluate de pe bandă de către consumator coincide cu ordinea în care sunt aşezate pe bandă de către producător.*

Modelarea benzii pe care sunt plasate articolele se face cu ajutorul unei structuri de date potrivite pentru acest caz: coada (*queue* - eng). Specificarea clasei, ce ar descrie comportamentul unei cozi utilizînd *Object-Z*, este prezentată în figura 3. Clasa *Queue*, fiind descrisă într-o manieră asemănătoare celei din [5], are definită o constantă locală *max* ce denotă lungimea maximală a cozii şi o singură variabilă de stare items, semnificând articolele plasate pe bandă. Ordinea de preluare şi plasare a articolelor este susţinută de un tip special *seq* (secvenţă, sequence – eng.), ce este o mulţime ordonată, definită formal în [3 p. 115] utilizând următoarea funcţie parţială: $\operatorname{seq} X == \{ f : \mathbb{N} \pfun X \mid \operatorname{dom} f = 1 \,..\, \# f \}$.

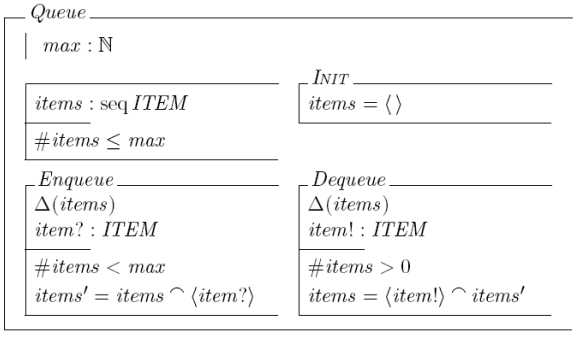

Figura 3 – Clasa Queue definită în limbajul Object-Z

Iniţial coada este vidă şi acest lucru este specificat în schema de stare iniţială, marcată prin cuvântul rezervat INIT. Celelalte scheme sunt de operaţii, în care se adăugă şi se retrag articole de pe bandă într-o manieră FIFO (*first-in/first-out* – eng.) îndeplinindu-se pre-condiţiile respective: operaţia *Enqueue* prin variabila de intrare *item?* adaugă noi valori la variabila de stare *items*; iar operaţia *Dequeue* retrage antetul secvenţei *items* prin variabila de ieşire *item!*. Limbajul *Object-Z* extinde schema de operaţii prin adăugarea Δ-listei (delta-listă), care specifică lista variabilelor de stare care vor fi modificate de operaţie.

Clasele producătorilor (*Producer* – eng.) şi consumatorilor (*Consumer* – eng.) sunt descrise în figura 4. Ambele clase specifică o variabilă de stare *items*, definită ca o mulţime de elemente de tip *ITEM*. Clasele vor utiliza variabila pentru a modela producerea/plasarea şi respectiv preluarea/consumarea articolelor. Operaţiile *Put* şi *Get* au o formă concisă de definire a schemei şi fac abstracţie de detaliile de implementare: cum se produc articolele sau cum se consumă.

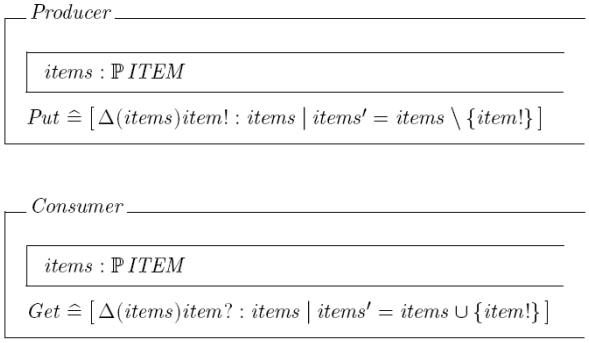

Figura 3.4 – Clasele Producer şi Consumer definite în limbajul Object-Z

Producerea şi consumul articolelor trebuie să deruleze nedeterminist şi reciproc independent. Acest lucru poate fi modelat în limbajul *Object-Z* cu ajutorul operatorului de alegere nedeterministă (*nondeterministic choice* – eng.) – „[]". Mai mult decât atât, acest operator modelează şi situaţia când alegerea nu este influenţată de mediul extern, cum este cazul nedeterminismului din operaţia *ProducerConsumerSimulation* prezentată în figura 5.

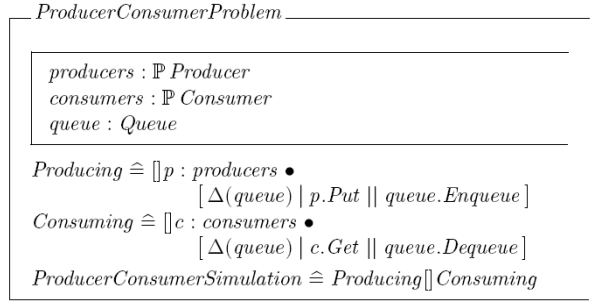

Figura 5 – Clasa principală a problemei Producator-Consumator definită în limbajul Object-Z

Acelaşi operator de alegere nedeterministă poate fi utilizat într-o versiune distribuită în cazul unor colecţii de componente, efectuarea operaţiilor cărora nu trebuie să se influenţeze. Suportul pentru o astfel de distribuire este agregarea (*aggregation* – eng.), care este aplicată la efectuarea unei activităţi asupra unei colecţii de obiecte de acelaşi tip. În operaţiile *Producing* şi *Consuming* (figura 5) operatorul de scop (sau vizibilitate) „•" este util pentru specificarea operaţiilor pe obiecte individuale în agregări: selecţia obiectelor se realizează în partea stângă a declaraţiei, iar activităţile de efectuat sunt indicate în partea dreaptă.

Producerea articolelor de către producător (operaţia *Put* a clasei *Producer*) şi plasarea acestora pe bandă (operaţia *Enqueue* a clasei *Queue*) implică comunicarea operaţiilor respective. Acest lucru este realizat prin intermediul operatorului de compoziţie paralelă – „||", care afectează doar acei operanzi care cuprind intrări şi respectivi ieşiri cu acelaşi nume (în problema de mai sus, este variabila cu numele *item*). Este important de menţionat că şi în cazul

agregărilor pentru a specifica modificarea variabilei de stare este necesar indicarea în Δ-listă a variabilei respective.

III. PROBLEMA CITITORI-SCRIITORI

Pentru evidenţierea posibilităţilor de specificare în limbajul *Object-Z* a sincronizării condiţionate se va analiza problema Cititori-Scriitori. Aceasta este o problemă clasică de concurenţă şi o modificare a problemei de excludere reciprocă. Problema presupune existenţa unor cititori care pot avea acces simultan la o carte (secţiune critică) şi unor scriitori care au acces exclusiv la ea. Diverse politici de sincronizare şi de planificare ale accesului la carte determină existenţa mai multor versiuni ale problemei. Din multitudinea lor se va accepta formularea prezentată în [7 p. 65], care asociază câte un proces (executat concurent) fiecărui cititor şi fiecărui scriitor. Activitatea fiecărui actor al problemei va consta în deschiderea şi închiderea cărţii (făcând distincţie dintre operaţiile efectuate de cititori şi cele efectuate de scriitori). Iar în cazul planificării accesului vom accepta următoarele [7 p. 101]:

- dacă nici un proces nu a intrat în secţiunea critică (nu a deschis cartea), şi există cititori şi scriitori care doresc acest lucru, va fi preferat un scriitor;
- dacă un scriitor a ieşit din secţiunea critică (a închis cartea), şi există cititori şi scriitori care doresc să intre în secţiunea sa critică, vor fi preferaţi cititorii, care s-au expus dorinţa înaintea unui scriitor.

O soluţie *Object-Z* pentru această problemă este propusă în [6]. Specificaţia include clasa *ReadersWriters* care defineşte regulile (clasice şi fără priorităţi) de acces la carte. Utilizarea acestei clase se face în conjuncţie cu o altă clasă specificată Object, care „captează" mecanismul pentru concurenţă oferit de maşina virtuală Java (JVM, *Java Virtual Machine* – eng.), dar care se abate de la semantica stabilită de limbaj.

Astfel de specificări (ce includ detalii de implementare) pot fi utile doar pentru un cadru specific de dezvoltare (sau testare, cum este cazul pentru [6]). În aceste condiţii, dorind facilitarea analizei specificaţiei, respectând în tocmai semantica limbajului şi în contextul regulilor de planificare menţionate, în continuare se prezintă specificaţia problemei Cititori-Scriitori.

Pentru început declarăm *PAGES* – o abreviere de funcţie definită pe domeniul *ADDR* (spaţiul de adrese) şi codomeniul *PAGE* (spaţiul paginilor/datelor). Această abreviere este utilizată în continuare pentru definirea variabilei de stare *pages*, pe lângă alte două ale clasei *Book* (figura 6): *nr* (numărul cititorilor activi) şi *nw* (numărul scriitorilor activi). Accesul la citire (operaţia *Read*) şi scriere în carte (operaţia *Write*) este determinat de strategia specificată prin operaţiile:

- *OpenRead* (deschidere pentru citire), se verifică dacă există scriitori activi şi dacă nu există, se incrementează numărul de cititori activi;
- *CloseRead* (închidere pentru citire), se actualizează numărul cititorilor care au acces la carte;
- *OpenWrite* (deschidere pentru scriere), operaţia va fi blocată în caz dacă există cititori sau cel mult un scriitor ce au acces la carte, în caz contrar va incrementa numărul de scriitori activi (ţinând cont de invariantul *nw* ≤ 1);
- *CloseWrite* (închidere pentru scriere), se actualizează numărul scriitorilor care au acces la carte.

$PAGES == ADDR \rightarrow PAGE$

Book

$pages : PAGES$ $nr : \mathbb{N}$ $nw : \mathbb{N}$ $nw \leq 1$	**INIT** $nr = nw = 0$
OpenRead $\Delta(nr)$ $nw = 0$ $nr' = nr + 1$	**OpenWrite** $\Delta(nw)$ $nw = 0 \land nr = 0$ $nw' = nw + 1$
CloseRead $\Delta(nr)$ $nr' = nr - 1$	**CloseWrite** $\Delta(nw)$ $nw' = nw - 1$
Read $addr? : ADDR$ $page! : PAGE$ $p! = pages(addr?)$	**Write** $\Delta(pages)$ $addr? : ADDR$ $page? : PAGE$ $pages' = pages \oplus \{addr? \mapsto page?\}$

Figura 6 – Clasa Book definită în Object-Z

Planificarea accesului la carte cere luarea în considerare a încă doi parametri: numărul de cititori şi scriitori în aşteptare (care şi-au anunţat intenţia). Aceste valori trebuie incrementate şi decrementate înainte şi, respectiv, după precondiţiile existente ale operaţiilor *OpenRead* şi *OpenWrite* din clasa de bază. Adăugarea noilor reguli ale strategiei de planificare la operaţiile clasei *Book* se poate realiza prin moştenire şi redefinirea operaţiilor necesare (figura 7).

$PREFERED ::= reader \mid writer$

BookWithAccesStrategy

$\upharpoonright(OpenRead, CloseRead, OpenWrite, CloseWrite)$
$Book[OldOpenRead/OpenRead, OldOpenWrite/OpenWrite]$

$prefered : PREFERED$ $ww : \mathbb{N}$ $wr : \mathbb{N}$	**INIT** $prefered = writer$ $wr = ww = 0$
RequestOpenRead $\Delta(wr)$ $wr' = wr + 1$ $ww = 0 \lor$ $prefered = reader$	**RequestOpenWrite** $\Delta(ww)$ $ww' = ww + 1$
OldOpenRead $\Delta(wr)$ $wr' = wr - 1$	**OldOpenWrite** $\Delta(ww)$ $ww' = ww - 1$
CloseRead $\Delta(prefered)$ $nr = 0 \land ww > 0 \Rightarrow$ $prefered' = writer$	**CloseWrite** $\Delta(prefered)$ $wr > 0 \Rightarrow$ $prefered' = reader$

$OpenRead \mathrel{\widehat=} RequestOpenRead \fatsemi OldOpenRead$
$OpenWrite \mathrel{\widehat=} RequestOpenWrite \fatsemi OldOpenWrite$

Figura 3.7 – Adăugarea politicilor de acces prin redefinirea operaţiilor (Object-Z)

Redefinirea operaţiilor în Object-Z este specifică şi

constă doar în adăugarea noilor predicate la cele definite în clasa de bază. Astfel când se doreşte ca operaţia să fie precedată şi urmată în acelaşi timp de noi predicate, procedura este următoarea: a) se redenumeşte operaţia la moştenire (*Book[OldOpenRead/OpenRead, OldOpenWrite/ OpenWrite]*); b) se adăugă post-condiţiile noi la operaţia redenumită; c) se specifică pre-condiţiile noi într-o operaţie nouă (*RequestOpenRead* /*RequestOpenWrite*) d) se defineşte prin compoziţia ambelor operaţii (de exemplu: *RequestOpenRead OldOpenRead*) noua operaţie ce va asigura accesul pentru citire sau scriere (pentru a asigura aceeaşi interfaţă operaţia va lua numele din clasa de bază: *OpenRead*, respectiv OpenWrite).

Operaţiile de închidere specifică politicile preferenţiale definite de condiţiile problemei. De exemplu, când un scriitor închide cartea se va da preferinţă unui cititor la deschidere, prin setarea variabilei prefered în valoarea „*reader*”. Această nouă post-condiţie se va adăuga la predicatele operaţiei *CloseWrite* moştenită de la clasa *Book* după cum este prezentat în figura 3.7. Variabila prefered va fi setată în „*writer*” (indiferent de faptul dacă sunt sau nu alţi cititori în aşteptare) după ce ultimul cititor activ va închide cartea (*nr = 0*) şi dacă există scriitori ce şi-au manifestat dorinţa de a scrie în carte (*ww > 0*). Prin procedeul menţionat şi această nouă post-condiţie se va adăuga la predicatele operaţiei *CloseRead*.

În figura 8 sunt definite clasele *Reader* şi *Writer*, în care se descriu operaţiile de citire şi, respectiv, de scriere efectuate nemijlocit de obiectele cititorilor şi scriitorilor. Aceste operaţii în clasa principală *ReadersWritersProblem* (figura 9) „interacţionează” cu operaţiile clasei *Book* prin operatorul de compoziţie paralelă.

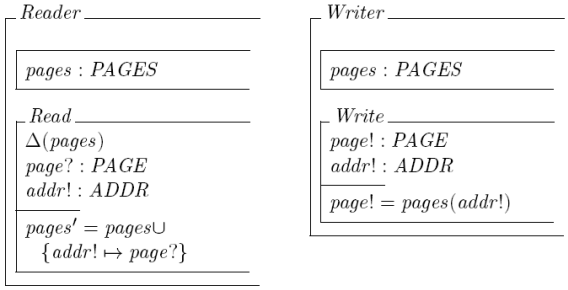

Figura 8 – Clasele Cititori şi Scriitori definite în limbajul Object-Z

Operatorul de alegere nedeterministă permite modelarea nedeterminismului intern la specificarea operaţiilor de citire/scriere. Acelaşi operator în versiunea distribuită permite specificarea independenţei dintre operaţiile obiectelor din agregări (figura 9).

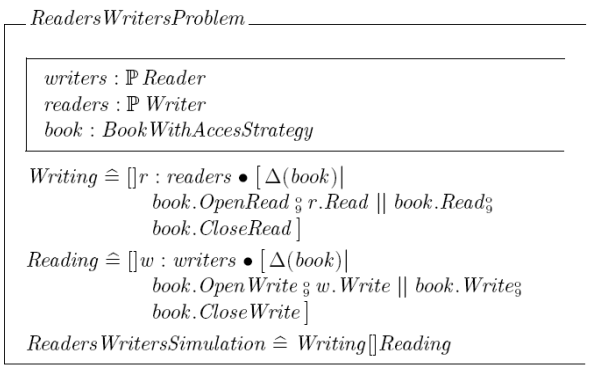

Figura 9 – Clasa principală a problemei Cititori-Scriitori definită în limbajul Object-Z

IV. CONCLUZII

Deşi limbajul prevede operatori ce pot exprima activităţi specifice concurenţei (sincronizare, comunicare, nedeterminism) şi în descrierea limbajului se menţionează că modelul concurenţei acceptat de Object-Z este *concurenţa întreţesută*, semantica concurenţei a limbajului Object-Z nu poate fi considerată definitivată, atît timp cît, în specificarea limbajului nu se vorbeşte de *granularitatea* concurenţei şi posibilitatea de specificare a *atomicităţii* operaţiilor de clasă.

BIBLIOGRAFIE

[1] **D. Ciorbă, V. Beşliu.** *Architecting software concurrency.* Computer Science Journal of Moldova (ISSN 1561-4042), 2011, Vol. 19, 1 (55), pp. 92-108. http://www.math.md/files/csjm/v19-n1/v19-n1-(pp92-108).pdf.

[2] **International Organization for Standardization.** Information technology — Z formal specification notation — Syntax, type system and semantics. 1st edition s.l. : ISO, July 01, 2002. p. 189. http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip. ISO/IEC 13568:2002(E).

[3] **J. Spivey.** *The Z Notation: A Reference Manual.* 2nd Edition. s.l. : Prentice Hall, 2001. http://spivey.oriel.ox.ac.uk/~mike/zrm/.

[4] **G. Smith, R. Duke.** *Specifying Concurrent Systems Using Object-Z.* 15th Australian Computer Science Conference (ACSC-15), Proceedings, 1992, pp. 859-871. http://itee.uq.edu.au/~smith/papers/acsc92.ps.gz.

[5] **G. Smith, J. Derrick.** *Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP.* Formal Methods in System Design, 2001, pp. 249-284. http://www.csee.uq.edu.au/~smith/papers/fmsd2001.pdf.

[6] **L. Wildman, R. Duke, P. Strooper.** Viewpoint-Based Testing of Concurrent Components. [ed.] E. A. Boiten, J. Derrick, G. Smith. *Integrated Formal Methods, 4th International Conference, IFM 2004, Proceedings.* Canterbury : Springer, 2004, pp. 501-520. itee.uq.edu.au/~testcon/_papers/wildman-testconc-viewpoints.pdf.

[7] **H. Georgescu.** *Programare concurentă. Teorie şi aplicaţii.* Bucureşti : Editura Tehnică, 1996. p. 240.