UNIVERSITATEA BABEŞ-BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

Contribuţii la fundamentarea formală a reutilizării softului

Rezumatul tezei de doctorat

Doctorand: **Vladiela Petraşcu**

Conducător ştiinţific: prof. univ. dr. **Militon Frenţiu**

2011

Mulţumiri

Doresc să mulţumesc tuturor persoanelor care, direct sau indirect, au contribuit la finalizarea acestui demers ştiinţific.

Adresez sincere mulţumiri domnului profesor Militon Frenţiu, coordonatorul ştiinţific al acestei teze, atât pentru şansa de a urma un program de doctorat în domeniul Ingineriei Programării, cât şi pentru încrederea şi îndrumarea acordate pe parcursul anilor de studiu.

Mulţumesc, de asemenea, domnului lector doctor Dan Chiorean, pentru oportunităţile de cercetare oferite, cunoştinţele împărtăşite, criticile constructive şi întreaga colaborare ce a contribuit la elaborarea acestei teze.

Nu în ultimul rând, doresc să îmi exprim recunoştinţa familiei, pentru suportul emoţional oferit.

Sprijinul financiar primit este, de asemenea, apreciat.

Rezultatele descrise în Capitolele 4 şi 5, precum şi în Secţiunea 6.2 a acestei teze au fost sprijinite de către CNCSIS–UEFISCSU, prin proiectul PNII–IDEI 2049/2008 “Cadru bazat pe Utilizarea Extensivă a Metamodelării pentru Specificarea, Implementarea şi Validarea Limbajelor şi Aplicaţiilor (CUEM_SIVLA)”.

Cercetările raportate în Secţiunea 6.1 s-au desfăşurat în contextului proiectului ECO-NET Nr. 16293RG/2007 “Behaviour Abstraction from Code: Filling the Gap between Component Specification and Implementation”, sponsorizat de Égide, Franţa.

Cuprins

Capitolul 1

Introducere

Reutilizarea - utilizarea de artefacte existente în dezvoltarea unora noi, este o practică obişnuită în toate disciplinele inginereşti. Printre beneficiile majore ale acesteia se numără creşterea productivităţii, deci reducerea timpului şi a costurilor de dezvoltare, precum şi creşterea calităţii softului. În Ingineria Programării, necesitatea unei abordări mature în acest sens a fost pentru prima data admisă în cadrul conferinţei NATO din 1968, în contextul aşa-numitei "crize soft", reutilizarea fiind propusă ca o soluţie a problemelor care au generat-o. Începând de atunci, tehnicile de reutilizare ale Ingineriei Programării au evoluat cu fiecare nouă paradigmă de dezvoltare apărută. Această evoluţie s-a manifestat atât în termeni de granularitate, cât şi de nivel de abstractizare. Relativ la granularitate, lucrurile au avansat de la reutilizarea procedurilor în programarea structurată, la cea a claselor şi obiectelor în cadrul paradigmei orientate obiect, până la reutilizarea componentelor şi cadrelor de aplicaţie în contextul dezvoltării bazate pe componente (CBSD). În ceea ce priveşte nivelul de abstractizare, s-a constatat o evoluţie majoră de la reutilizarea codului (proceduri, clase, obiecte, componente) la reutilizarea unor artefacte aflate la un nivel mai înalt de abstractizare, precum şabloane de proiectare sau arhitecturi, mergând până la reutilizarea masivă a unor întregi modele şi metamodele, în contextul abordărilor MDE. Acestor două nivele (granularitate şi abstractizare) li se asociază un al treilea, legat de tipul tehnicilor de reutilizare folosite, printre care instanţierea, compunerea sau abordarea generativă.

Reutilizarea îşi poate atinge însă obiectivele doar în prezenţa unui fundament formal adecvat. Acesta ar trebui să acopere atât specificarea formală a artefactelor reutilizabile, cât şi formalizarea procesului lor de reutilizare. *Metodele formale* (incluzând limbaje, tehnici şi instrumente bazate pe concepte matematice, ce permit raţionamente logice) au fost propuse drept o modalitate de creştere a fiabilităţii softului. În sens larg, fiabilitatea se referă la absenţa erorilor, acoperind atât corectitudinea, cât şi robusteţea. În cazul produselor soft reutilizabile, ce urmează a fi refolosite într-o varietate de alte contexte, fiabilitatea reprezintă cu siguranţă un imperativ.

Reutilizarea reprezentând un subiect atât de vast, există o multitudine de probleme deschise legate de formalizarea artefactelor reutilizabile şi a procesului lor de reutilizare. În cadrul acestei teze, am abordat patru subdomenii ale acesteia, legate de şabloane de proiectare, şabloane de constrângeri, metamodelare şi dezvoltare bazată pe componente. În cazul şabloanelor de proiectare, lipsa formalităţii, atât la nivelul definirii soluţiilor acestora, cât şi la nivelul specificării procesului de reutilizare, face imposibilă verificarea consistenţei lor, a corectitudinii instanţierilor, limitează automatizarea şi nu permite aplicarea acestora în dezvoltarea sistemelor critice. Unele dintre şabloanele de constrângeri din modelarea orientată obiect nu beneficiază de soluţii adecvate, în raport cu rolul major al aserţiunilor în verificarea corectitudinii modelelor, în contextul paradigmei MDE. În plus, limbajele de (meta)modelare utilizate în abordările MDE prezintă numeroase deficienţe în specificarea semanticii statice, cu efecte negative asupra activităţilor de verificarea a compilabilităţii

modelelor. În domeniul dezvoltării bazate pe componente, două dintre problemele curente sunt legate de discrepanţa existentă între modelele de componente industriale (orientate pe implementare) şi cele academice (focusate pe specificare), precum şi de lipsa modelelor de componente care să permită o specificare contractuală completă (pe patru nivele, incluzând contractele semantice) a componentelor soft. În cadrul tezei, am urmărit să aducem o contribuţie la soluţionarea fiecăreia dintre problemele anterior menţionate.

Structura tezei. Teza este structurată pe şapte capitole (un capitol introductiv, unul cu noţiuni fundamentale, patru capitole de contribuţii si unul de concluzii), conţine o bibliografie totalizând un număr de 140 de referinţe şi trei anexe.

Capitolul 1 descrie contextul, motivaţia şi scopul tezei, enumeră contribuţiile raportate în cadrul acesteia şi modalitatea de diseminare a lor şi prezintă structura generală a lucrării.

Capitolul 2 oferă o scurtă introducere în domeniul tehnicilor formale şi de reutilizare referite în cadrul tezei. Din categoria metodelor şi limbajelor formale, sunt prezentate metoda B, metodologia Design by Contract (DBC) şi limbajul OCL. În ceea ce priveşte abordările bazate pe reutilizare, se face referire la şabloane de proiectare, paradigma de dezvoltare bazată pe componente (CBSD) şi cea dirijată de modele (MDE).

În *Capitolul 3* este descrisă prima contribuţie raportată în cadrul tezei, situată în domeniul formalizării şabloanelor de proiectare. Aceasta constă într-o formalizare completă a şablonului GoF State în B. Prezentarea acoperă descrierea şablonului, definiţia sa formală în B, formalizarea procesului său de reutilizare, ilustrată printr-un exemplu, validarea abordării şi o analiză detaliată a activităţii de demonstrare efectuate cu instrumentul AtelierB.

Capitolul 4 detaliază contribuţia noastră referitoare la definirea şabloanelor de constrângeri pentru modelele orientate obiect. Prezentarea debutează cu descrierea şabloanelor abordate şi a soluţiilor curente ale acestora în literatură. Ulterior, sunt introduse propunerile noastre, constând într-un număr de noi soluţii aferente şabloanelor *For All* şi *Unique Identifier*, al căror avantaj major rezidă în suportul oferit depanării modelelor. Abordarea propusă este apoi validată folosind instrumentul OCLE şi studii de caz relevante.

În *Capitolul 5* sunt descrise contribuţiile legate de formalizarea semanticii statice a limbajelor de (meta)modelare. Mai intâi, este argumentat caracterul imperativ al cerinţei privind compilabilitatea modelelor în context MDE şi se prezintă starea de fapt în acest domeniu. Pornind de aici, se accentuează ideea necesităţii unui cadru conceptual riguros, care să sprijine specificarea semanticii statice a limbajelor de (meta)modelare, permiţând astfel verificarea eficientă a compilabilităţii modelelor. Ulterior, sunt prezentate principiile recomandate ca bază a unui astfel de cadru, urmate de propuneri de îmbunătăţire a specificării semanticii statice a metamodelelor UML/MOF, Ecore şi XCore, în concordanţă cu aceste principii.

Capitolul 6 prezintă două contribuţii în domeniul specificării componentelor soft. Prima dintre acestea este o contribuţie la o abordare de reverse-engineering menită să extragă abstractizări structurale şi comportamentale din implementări ale sistemelor bazate pe componente. Această contribuţie a fost realizată în cadrul proiectului internaţional ECO-NET [1, 10]. Prezentarea acoperă motivaţia proiectului, abordarea generală propusă, esenţa contribuţiei noastre, precum şi o descriere succintă a activităţii de validare efectuate şi a suportului oferit la nivel de instrumente. Cea de-a doua contribuţie este în direcţia asigurării unui cadru care să suporte o specificare contractuală completă a componentelor soft, cu un accent deosebit pe reprezentarea contractelor semantice. În acest scop, este introdus ContractCML, limbaj de modelare aflat la baza abordării propuse. Esenţa contribuţiei constă în metoda aleasă pentru reprezentarea contractelor semantice la nivelul metamodelului. Utilizarea acestuia este ilustrată printr-un exemplu. În final, este descrisă o metodă de simulare a execuţiei serviciilor componentelor, bazată pe propunerea facută privind specificarea

contractelor semantice.

Fiecare dintre capitolele 3 - 6 debutează cu motivarea abordării domeniului în cauză, include un rezumat al contribuţiilor similare din literatură şi evidenţiază avantajele propunerilor făcute comparativ cu acestea. De asemenea, fiecare astfel de capitol se finalizează cu un sumar al contribuţiilor şi cu indicarea direcţiilor viitoare de cercetare.

Capitolul 7 încheie lucrarea, oferind o imagine de ansamblu asupra tuturor contribuţiilor raportate în cadrul acesteia.

Cuvinte cheie. reutilizare, metode formale, şabloane de proiectare, şabloane de constrângeri, metamodelare, componente soft, MDE, WFR, CBSD, OCL, B

Capitolul 2

Preliminarii

2.1 Metode şi limbaje formale

2.1.1 Consideraţii generale

În sens larg, metodele formale cuprind limbaje, tehnici şi instrumente bazate pe modelarea matematică şi logica formală, utilizate în specificarea, dezvoltarea şi verificarea sistemelor soft. În cadrul acestei secţiuni, am oferit definiţii ale conceptelor de *metodă formală* şi *limbaj formal*, am subliniat importanţa lor în dezvoltarea sistemelor critice şi am abordat problema clasificării acestora. Majoritatea taxonomiilor legate de metode formale disting între *metode formale orientate pe modele* şi *metode formale orientate pe proprietăţi*.

2.1.2 Metoda B

B [6] este o metodă formală orientată pe modele, care suportă întreg ciclul de dezvoltare al unui produs soft. Unitatea fundamentală de structurare a modelelor B este *maşina abstractă*, limbajul folosit fiind limbajul AMN (Abstract Nachine Notation). Dezvoltarea unui sistem B începe cu un model abstract al acestuia, definit în termenii uneia sau mai multor maşini abstracte; acesta este ulterior rafinat sau specializat, până la obţinerea unei implementări. Atât consistenţa modelului iniţial, cât şi corectitdinea paşilor de rafinare sunt demonstrate matematic; metoda beneficiază de un sprijin puternic la nivel de instrumente, unul dintre acestea fiind AtelierB [31].

În cadrul acestei secţiuni, am detaliat notaţia AMN, mecanismele de dezvoltare incrementală a modelelor, precum şi caracteristicile procesului de rafinare.

2.1.3 Metodologia Design by Contract

Design by Contract (*DBC*) [53] reprezintă o metodologie care propune dezvoltarea contractuală a componentelor orientate obiect, bazată pe utilizarea aserţiunilor, scopul final fiind asigurarea fiabilităţii softului. În cadrul acestei secţiuni, am abordat principalele trei tipuri de aserţiuni (precondiţii, postcondiţii, invarianţi), am oferit definiţia corectitudinii unei clase în raport cu acestea şi am discutat despre rolul utilizării aserţiunilor.

2.1.4 Limbajul OCL

OCL (*Object Constraint Language* [60, 86]) reprezintă un limbaj formal utilizat în scopul definirii expresiilor în modelele UML (Unified Modeling Language [61, 62]). În cadul acestei secţiuni, am abordat elemente de limbaj OCL, rolul OCL în metamodelare şi dialecte OCL

(XOCL [30]), precum şi suportul oferit limbajului la nivel de instrumente. În contextul metamodelării, am definit conceptele de *regulă de bună formare* (eng. *Well-Formedness Rule*, *WFR*) şi *operaţie adiţională* (eng. *Additional Operation*, *AO*). Relativ la instrumente, am prezentat facilităţile oferite de OCLE (OCL Environment [48]) în specificarea şi evaluarea aserţiunilor.

2.2 Reutilizarea softului

2.2.1 Consideraţii generale

În sens larg, reutilizarea softului vizează folosirea unor artefacte soft existente în definirea unora noi [47]. În această secţiune, am definit conceptele de *reutilizare*, *produs soft reutilizabil* şi *reutilizabilitate*, am enumerat avantajele adoptării unui proces de dezvoltare bazat pe reutilizare şi am prezentat evoluţia tehnicilor de reutilizare în Ingineria Programării.

2.2.2 Şabloane de proiectare

În dezvoltarea softului, şabloanele de proiectare oferă soluţii generale, recomandate, unor probleme de proiectare recurente. În cadrul acestei secţiuni, ne-am referit la originea şi evoluţia conceptului de *şablon* în Ingineria Programării, am prezentat o taxonomie a şabloanelor de proiectare funcţie de nivelul lor de granularitate şi am discutat modalitatea de descriere a şabloanelor de proiectare în cadrul catalogului GoF [42], accentuând asupra caracterului informal al acesteia.

2.2.3 Dezvoltarea softului bazată pe componente

Această secţiune prezintă, pe scurt, paradigma de dezvoltare a softului bazată pe componente (*Component-Based Software Development*, *CBSD*, [72]). În cadrul acesteia, am oferit definiţia unei componente soft, am discutat despre modele idustriale (COM (Component Object Model [19]), .NET [87], Web Services [54], CCM (CORBA Component Model [55]), JavaBeans [71], EJB (Enterprise JavaBeans [34])) şi academice (Fractal [20], SOFA (SOFtware Appliances [21]), KobrA [13], Kmelia [9]) de componente şi am abordat problema specificării acestora. Relativ la acest ultim aspect, am rezumat metoda de specificare a componentelor *UML Components*, introdusă în [23].

2.2.4 Ingineria softului dirijată de modele

Această secţiune oferă o scurtă introducere în cea mai recentă paradigmă de dezvoltare a softului, paradigma dirijată de modele. Am definit conceptele de bază cu care aceasta operează (*model*, *metamodel*, *meta-metamodel*, *limbaj de modelare specific unui domeniu* (*Domain Specific Modeling Language*, *DSML*)). Am prezentat, pe scurt, diferitele abordări ale acesteia, Model Driven Architecture (MDA [56]), Model Driven Engineering (MDE [16, 17, 69]) şi Language Driven Development (LDD, [30]). Abordarea standard MDA se bazează în exclusivitate pe standarde OMG (Object Management Group), meta-metamodelul ei fiind reprezentat de MOF (Meta Object Facility [59]). MDE este numele generic atribuit abordărilor care includ formalisme şi tehnologii independente de standardele OMG. Cel mai cunoscut framework MDE este EMF (Eclipse Modeling Framework [35]), acesta bazându-se pe meta-metamodelul Ecore şi integrând un suport puternic la nivel de instrumente, printre care MDT OCL (Model Development Tools OCL [37]) şi oAW (openArchitectureWare [5]). LDD se bazează pe meta-metamodelul XCore, abordarea fiind implementată în cadrul instrumentului XMF Mosaic [3].

Capitolul 3

Formalizarea şabloanelor de proiectare

Şabloanele de proiectare sunt astăzi recunoscute drept una dintre cele mai populare abordări în domeniul reutilizării softului. Acest capitol rezumă cercetările noastre referitoare la modalităţile de formalizare a şabloanelor de proiectare, în general, precum şi de formalizare a acestora folosind metoda formală B, în particular. Principala contribuţie raportată aici constă într-o formalizare completă a şablonului GoF State în B. Aceasta ne-a permis explorarea limitelor unei abordări existente privind formalizarea şabloanelor de proiectare cu ajutorul metodei B ([44]) şi propunerea unei modalităţi de îmbunătăţire a acesteia.

3.1 Motivaţie şi abordări în domeniu

Cercetările întreprinse în acest domeniu sunt motivate de caracterul informal al descrierii iniţiale a şabloanelor de proiectare în catalogul GoF. Această lipsă de formalitate se manifestă sub două aspecte: pe de o parte, şabloanele insele sunt descrise folosind un amalgam de limbaj natural, diagrame UML/OMT şi secvenţe de cod, pe de altă parte, procesul de reutilizare a acestora este, de asemenea, neformalizat. Trebuie recunoscut însă faptul ca ambele aspecte au avut o contribuţie majoră la succesul de care se bucură astăzi şabloanele de proiectare. Astfel, descrierea detaliată, în limbaj natural, conferă un grad ridicat de inteligibilitate, permiţând identificarea facilă a soluţiilor aferente unei probleme concrete de proiectare, în timp ce lipsa de formalitate la nivelul procesului de reutilizarea oferă o anumită flexibilitatea în adaptarea şabloanelor la particularităţile problemei concrete [18]. Cu toate acestea, caracterul informal face imposibilă atât certificarea consistenţei şabloanelor însele, cât şi verificarea corectitudinii diferitelor instanţieri ale acestora, limitând astfel aplicabilitatea lor în dezvoltarea sistemelor critice [51]. În plus, procesele de selecţie şi reutilizare sunt astfel dificil de automatizat.

În acest context, în literatură au fost semnalate diferite abordări în direcţia formalizării şabloanelor si a procesului lor de reutilizare. Printre cele mai relevante, menţionăm crearea unui limbaj specializat de formalizare numit LePUS (Language for Patterns Uniform Specification [38, 39]), o abordarea folosind limbajul RSL (RAISE Specification Language) [22, 40], specificarea formală a framework-urilor în Catalysis [49], precum şi câteva propuneri de formalizare cu ajutorul metodei B [18, 44, 50, 51].

3.2 O abordare privind formalizarea şablonului State în B

După cum am precizat anterior, contribuţia noastră în acest domeniu constă într-o formalizare completă în B a şablonului de proiectare State, acoperind atât definiţia formală a şablonului, cât şi formalizarea procesului său de reutilizare.

3.2.1 Şablonul State

State [42] este un şablon GoF comportamental, care oferă o soluţie problemei de proiectare a claselor (`Context`) ale căror obiecte au un comportament complex, dependent de stare. Structura soluţiei este dată de diagrama de clase ilustrată în Figura 3.1. Ideea de bază constă în proiectarea unei clase abstracte/interfeţe `State` pentru încapsularea comportamentul dependent de stare, comportament ce urmează a fi implementat în clase concrete derivate din `State`. Fiecare instanţă `Context` reţine o referinţă către starea curentă şi delegă metodelor $\texttt{Handle}_i$ ale acesteia (parte din) rezolvarea fiecărei cerinţe $\texttt{Request}_i$ primite.

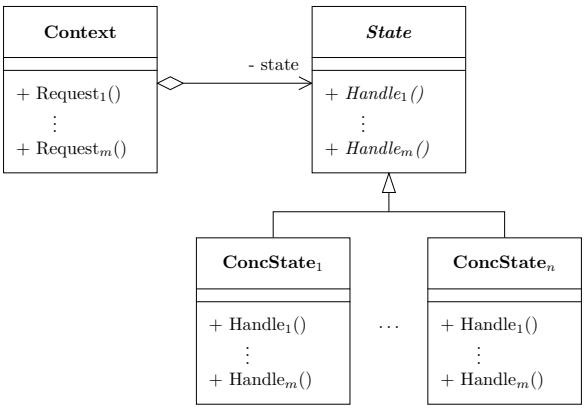

Figura 3.1: Şablonul State

3.2.2 Definiţia B a şablonului State

Definiţia B a şablonului a fost creată manual, pe baza descrierii informale a acestuia prezentate în catalogul GoF şi a unor reguli de traslatare UML-B disponibile în literatură [70]. Un aspect cheie aici se referă la faptul că formalizarea comportamentului polimorfic si a ideii de delegare nu s-au efectuat la acest nivel, fiind amânate până la prima etapă a procesului de reutilizare (instanţiere cu redenumire). Acest fapt ne-a permis evitarea unei limitări severe privind numărul de stări concrete posibile[1]. În plus, experimentele ne-au arătat că această propunere nu afectează numărul de demonstraţii reutilizate în procesul de instanţiere a şablonului în B.

Rezultatul acestei etape este reprezentat de maşina abstractă `StatePattern`. Consistenţa acesteia a fost certificată cu ajutorul intrumentul AtelierB [31]. Toate demonstraţiile necesare au fost realizate automat.

3.2.3 Reutilizarea şablonului State în B

Procesul de reutilizare a şablonului State în B a fost ilustrat prin intermediul unui studiu de caz de complexitate medie, referitor la modelarea formală a comportamentului unui ceas electronic (LCD Wallet Travelling Clock [73, 81]). În formalizarea procesului de reutilizare, am urmat cadrul general introdus în [44]. În consecinţă, formalizarea propusă s-a realizat în doi paşi: o instanţiere, urmată de o extindere. Primul pas se bazează pe mecanismul B de includere a maşinilor abstracte (clauza de structurare INCLUDES), în timp ce ultimul face uz de mecanismul de rafinare (clauza REFINES). În afara modularităţii, o astfel de separare a responsabilităţilor prezintă avantajul obţinerii maşinii abstracte de instanţiere cu efort zero de demonstrare.

Pasul 1, de includere. Acesta este un pas intermediar în procesul de reutilizare, concretizat prin maşina abstractă `RenamedStatePattern`. La acest nivel s-a efectuat specializarea şablonului, fiind introduse redenumirile necesare, adaugându-se informaţii referitoare la stările concrete, precum şi noi operaţii. În cadrul acestei maşini s-a realizat formalizarea comportamentului polimorfic şi a ideii de delegare. Construcţia maşinii a vizat

[1] Această limitare se referă la restricţionarea numărului de stări concrete. Este una dintre principalele limitări ale formalizării propuse în [44] pentru şablonul Composite.

evitarea introducerii unor proprietăţi suplimentare de demonstrat (eng. *proof obligations*, *POs*). Ca urmare, demonstrarea consistenţei acesteia s-a realizat în mod trivial.

În cadrul tezei, am arătat că acest prim pas de reutilizare poate fi în întregime automatizat, pe baza unor informaţii oferite interactiv privind: numele clasei context, numărul şi numele stărilor concrete, starea iniţială a unui obiect context, numărul şi numele cererilor adresate contextului, o reprezentare textuală a diagramei de tranziţie a stărilor implicând toate stările concrete şi cererile posibile, precum şi numele operaţiilor suplimentare din cadrul modelului. Ca urmare, am arătat că ideea propusă, de amânare a formalizării comportamentului polimorfic, nu atrage efort manual suplimentar în etapa de reutilizare.

Pasul 2, de rafinare. Rezultatul acestui pas este reprezentat de maşina de rafinare `Clock` , ce formalizează integral comportamentul sistemului considerat.

Figura 3.2 oferă o vedere de ansamblu a tuturor componentelor B implicate în modelul formal realizat si a relaţiilor stabilite între acestea.

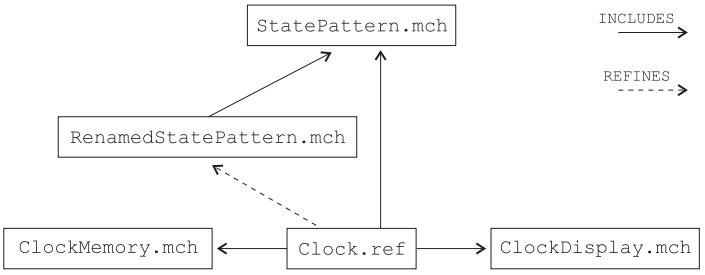

Figura 3.2: Structura modelului B aferent formalizării propuse

3.2.4 Validarea abordării propuse. Analiza activităţii de demonstrare

Întregul proces de dezvoltare în B a fost sprijinit de instrumentul AtelierB. Tabelul 3.1 oferă un sumar al activităţii de demonstrare efectuate.

Machine	TC	POG	nPOs	nAut	nInt	%Pr
`StatePattern`	OK	OK	7	7	0	100%
`RenamedStatePattern`	OK	OK	0	0	0	100%
`ClockMemory`	OK	OK	72	70	2	100%
`ClockDisplay`	OK	OK	29	29	0	100%
`Clock`	OK	OK	564	412	152	100%

Tabelul 3.1: Sumar al activităţii de demonstrare

Atât analiza sintactică, cât şi generarea proprietăţilor de demonstrat (POs) s-au finalizat cu success, fapt consemnat în cea de-a doua şi cea de-a treia coloană a tabelului. Coloanele rămase indică, de la stânga spre dreapta, numărul de proprietăţi netriviale generate (nPOs), volumul de demonstraţii realizate automat (nAut) şi interactiv (nInt) de către AtelierB, precum şi rata de succes a activităţii de demonstrare (%Pr).

Am realizat o analiză detaliată, per-operaţie, a proprietăţilor generate pentru maşinile `StatePattern` şi `Clock`. Pentru fiecare dintre operaţii am indicat numărul total de proprietăţi generate, numărul de astfel de proprietăţi pe categorii (păstrarea invarianţilor vs. satisfacerea precondiţiilor), precum şi volumul de demonstraţii realizate automat, respectiv interactiv din fiecare categorie. Această analiză a evidenţiat faptul că, dintre cele 564

de proprietăţi generate în cazul maşinii `Clock`, doar trei sunt legate direct de instanţierea şablonului, demonstrarea acestora realizându-se automat[2]. Celelalte 561 de demonstraţii se datorează particularităţilor sistemului formalizat. Deşi numărul de demonstraţii interactive e relativ mare, strategiile aplicate pentru soluţionarea lor au fost similare. Acestea includ decizii umane legate de efectuarea unor demonstraţii pe cazuri sau prin contradicţie, instanţierea unor predicate conţinând cuantificator universal, adăugarea unor ipoteze suplimentare sau aplicarea regulii de inferenţă *modus ponens*.

După cum se evidenţiază în [18], unul dintre principalele avantaje ale reutilizării şabloanelor în dezvoltarea sistemelor în B constă în reutilizarea nu numai a şablonului însuşi (fapt ce determină o mai bună modularitate şi o creştere a gradului de siguranţă privind corectitudinea), ci şi a demonstraţiilor formale asociate acestuia. În scopul estimării volumului de demonstraţii reutilizate ca urmare a aplicării propunerii noastre de formalizare, am definit maşina abstractă `ClockNoPattern`, ce modelează acelaşi sistem ca şi `Clock`, însă fară a (re)utiliza în mod explicit şablonul State. O analiză comparativă a activităţii de demonstrare la nivelul celor două maşini ne-a permis să concluzionăm că numărul de demonstraţii reutilizate prin aplicarea formalizării propuse pentru şablonul State la nivelul unui proiect B este de $4 + m * n$, n reprezentând numărul de stări concrete, iar m numărul de cereri adresate obiectului context.

3.3 Sumar

În cadrul acestui capitol am propus o formalizare completă a şablonului de proiectare State în B. Contribuţia cuprinde:

- o definiţie a şablonului State în B;
- o formalizare a procesului său de reutilizare, cu un potenţial ridicat de automatizare;
- o ilustrare a reutilizării şablonului în B, într-un studiu de caz de complexitate medie;
- demonstrarea completă (folosind instrumentul AtelierB) a corectitudinii maşinilor B incluse în studiul de caz, un număr relativ mare de demonstraţii fiind realizate interactiv;
- o analiză detaliată a activităţii de demonstrare efectuate, finalizată cu o estimare privind numărul de demonstraţii reutilizate prin aplicarea formalizării propuse într-un proiect B.

În formalizarea procesului de reutilizare, am urmat cadrul general introdus în [44]. Raportat la exemplul prezentat de lucrarea în cauză (formalizarea în B a şablonului de proiectare Composite), abordarea noastră prezintă următoarele avantaje:

- formalizează ideea de delegare - aspectul comportamental central al şabonului;
- nu impune nici o restricţie asupra numărului de clase concrete (stări în acest caz); acest lucru se datorează propunerii noastre de amânare a formalizării comportamentului polimorfic până la primul pas al procesului de reutilizare.

Dintre direcţiile viitoare de cercetare în acest domeniu, menţionăm automatizarea abordării propuse în B, precum şi investigarea propunerii privind amânarea formalizării comportamentului polimorfic în raport cu alte şabloane de proiectare GoF.

Contribuţia descrisă în cadrul acestui capitol a fost publicată în [80]. Aceasta se bazează pe rezultate anterioare cu privire la demonstrarea consistenţei modelelor orientate obiect prin utilizarea metodei formale B, raportate în [81].

[2]Luând în considerare şi faptul că maşina `StatePattern` a fost demonstrată automat în întregime iar `RenamedStatePattern` nu generează proprietăţi suplimentare, rezultă că întreaga activitate de demonstrare legată de definirea şi instanţierea în B a şablonului State a fost realizată automat.

Capitolul 4

Şabloane de constrângeri în modelarea orientată obiect

În capitolul anterior, am abordat domeniul şabloanelor de proiectare, discutând despre formalizarea soluţiilor aferente. În cadrul acestui capitol, entităţile reutilizabile considerate sunt şabloanele de constrângeri din modelarea orientată obiect. Principala contribuţie prezentată aici constă într-o nouă abordare privind definirea acestor şabloane, abordare impusă de cerinţele paradigmei MDE referitoare la utilizarea aserţiunilor.

4.1 Motivaţie

Paradigma MDE a impus necesitatea şi a creat premisele verificării automate a corectitudinii modelelor şi aplicaţiilor generate pe baza acestora. Astfel de verificări au la bază utilizarea aserţiunilor. Aserţiunile (pre/post-condiţii şi invarianţi) sunt necesare pentru a compensa deficitul de expresivitate existent la nivelul limbajelor de modelare diagramatice. În dezvoltarea tradiţională a softului, care utiliza modelele preponderent în scop de documentare, corectitudinea şi claritatea erau singurele cerinţe de calitate impuse aserţiunilor. În contextul MDE însă, în care verificarea automată a corectitudinii modelelor constituie un imperativ, aserţiunile trebuie proiectate astfel încât să ofere sprijin eficient activităţii de diagnosticare a erorilor.

Utilizarea masivă a aserţiunilor în cadrul MDE a condus în mod natural la identificarea diferitor şabloane de constrângeri, în timp ce necesitatea reducerii timpului necesar proiectării lor şi a evitării erorilor sintactice a motivat câteva abordări menite să formalizeze şi să automatizeze aplicarea acestora. Cu toate acestea, unele dintre soluţiile propuse pentru şabloanele de constrângeri existente nu oferă sprijin adecvat depanării modelelor.

4.2 Abordări în domeniu

Cele mai relevante abordări în domeniul definirii şabloanelor de constrângeri, abordări de la care am pornit şi în raport cu care am evaluat avantajele propunerilor făcute, sunt cuprinse în lucrările [7, 8], respectiv [83–85].

4.3 O nouă abordare: Şabloane de specificare OCL dirijate de MDE

Abordarea propusă în cadrul acestui capitol (pe care o referim cu sintagma *dirijată de MDE*) este fundamentată pe următoarele două principii:

1. Dat fiind rolul aserţiunilor în stabilirea corectitudinii modelelor în cadrul MDE, soluţiile şabloanelor de constrângeri trebuie proiectate în ideea asigurării unui suport adecvat depanării modelelor;
2. În concordanţă cu finalitatea utilizării modelelor şi a aserţiunilor într-un proces MDE de dezvoltare (translatare în cod), cât şi cu principiile metodologiei Design by Contract, soluţiile şabloanelor de constrângeri trebuie formulate nu doar ca şi invarianţi (situaţia actuală), ci şi în termenii precondiţiilor aferente.

4.3.1 Şabloane abordate. Soluţii existente

Reprezentând o temă relativ nouă de cercetare comparativ cu echivalentele lor din domeniul proiectării, şabloanele de constrângeri nu beneficiază la ora actuală de o denumire sau o definiţie unanim acceptate şi unitar utilizate în literatură[1]. În consecinţă, în cadrul acestui capitol, am propus şi uzat de următoarele definiţii:

Definiţia 4.1. Un *şablon de constrângeri* referă o restricţie recurentă impusă la nivelul diagramelor de clase, împreună cu o soluţie generală de specificare a acesteia.

Definiţia 4.2. Un *şablon de specificare OCL* denotă soluţia unui şablon de constrângeri, exprimată în sintaxă OCL.

Contribuţiile prezentate în acest capitol referă trei şabloane de constrângeri identificate în literatură - *Attribute Value Restriction*, *Unique Identifier* şi *For All*.

Attribute Value Restriction [83] (denumit şi *Invariant for Attribute Value* în [8]) este un şablon atomic care abstractizează diferite constrângeri elementare impuse asupra valorilor unui atribut al unei clase. Şablonul de specificare OCL corespunzător[2] propus în [83] este următorul:

```
pattern AttributeValueRestriction(property:Property, operator,value:OclExpression)=
 self.property operator value
```

Unique Identifier [83] (referit cu numele de *Semantic Key* în [8]) surprinde situaţia în care un atribut (grup de atribute) al unei clase joacă rolul unui identificator unic pentru clasa în cauză (instanţele clasei diferă prin valorile lor pentru acel atribut sau grup). Mai jos, ilustrăm template-urile OCL aferente acestuia propuse în [83], respectiv [7].

```
pattern UniqueIdentifier(property:Tuple(Property))=
 self.allInstances()->isUnique(property)
```

```
pattern SemanticKey(class:Class, property:Property)=
 class.allInstances()->forAll(i1, i2 | i1 <> i2 implies i1.property <> i2.property)
```

În sfârşit, şablonul *For All* [83] impune satisfacerea anumitor restricţii de către fiecare dintre elementele unei colecţii date. O aproximare a descrierii soluţiei sale în termenii unui template OCL este următoarea:

```
pattern ForAll(collection:OclExpression, properties:Set(OclExpression))=
 collection->forAll(y | oclAND(properties, y)).
```

4.3.2 Soluţii propuse

4.3.2.1 Şablonul *For All*

Abordarea propusă are la bază două soluţii oferite pentru şablonul de constrângeri *For All*. Şabloanele de specificare OCL asociate acestora sunt următoarele:

[1] Sintagmele utilizate, uneori interschimbabil, sunt cele de *şabloane de constrângeri* sau *şabloane de specificare OCL*.

[2] Toate soluţiile şabloanelor sunt date în termeni de *template-uri OCL parametrizate* [83].

```
pattern ForAll_Reject(collection:OclExpression, properties:Set(OclExpression))=
 collection->reject(y | oclAND(properties, y))->isEmpty()

pattern ForAll_Select(collection:OclExpression, properties:Set(OclExpression))=
 collection->select(y | not oclAND(properties, y))->isEmpty().
```

Există două meta-constrângeri care trebuie îndeplinite pentru ca instanţierea şabloanelor anterioare să furnizeze specificaţii OCL corecte din punct de vedere sintactic şi anume:
(FA1) *collection* trebuie să reprezinte o expresie OCL validă, a cărui tip să fie unul dintre tipurile OCL colecţie;
(FA2) fiecare dintre elementele mulţimii *properties* trebuie să reprezinte o expresie OCL booleană validă.

Avantajele soluţiilor noastre (sub aspectul suportului oferit pentru diagnosticarea erorilor la nivelul modelelor) în raport cu cele existente deja în literatură au fost argumentate prin intermediul unui exemplu de modelare relevant. În plus, am demonstrat echivalenţa semantică a propunerilor făcute cu soluţia în uz, prin translatarea acestora la nivelul unei maşini abstracte B, a cărei consistenţă a fost demonstrată cu ajutorul instrumentului AtelierB.

4.3.2.2 Şablonul *Unique Identifier* - soluţii de tip invariant

În cazul şablonului de constrângeri *Unique Identifier*, am diferenţiat (pentru prima dată în literatură) între două contexte în care pot fi impuse astfel de restricţii de unicitate, oferind soluţii de tip invariant adecvate fiecărei situaţii. Primul context vizează o aşa-numită "unicitate la nivel global" (toate instanţele unei clase diferă prin valoarea unui anumit atribut). Cel de-al doilea surprinde unicitatea "la nivelul unui container" (toate instanţele unei clase din cadrul unui container sunt unic identificabile la nivelul containerului prin valoarea unui anumit atribut). Am evidenţiat faptul că, în ciuda utilizării lor frecvente în literatură în situaţii corespunzând cazului unicităţii "la nivelul unui container", soluţiile existente pentru şablonul *Unique Identifier* se referă exclusiv la cazul unicităţii "globale". Mai mult, am punctat două probleme ale acestor soluţii, prima referitoare la nerespectarea semanticii invarianţilor, iar cea de-a doua la absenţa suportului pentru diagnosticarea erorilor.

Cazul unicităţii "la nivel global" (GUID). Pentru cazul unicităţii "globale", am propus următorul şablon de specificare OCL, menit a fi instanţiat ca şi invariant în contextul lui *class*.

```
pattern inv_GloballyUniqueIdentifier(class:Class,attribute:Property)=
 class.allInstances()->select(i | i.attribute = self.attribute)->size() = 1
```

Condiţiile necesare pentru asigurarea corectitudinii sintactice a expresiilor OCL rezultate prin instanţierea acestuia sunt următoarele:
(invGUID1) instanţierea şablonului trebuie realizată în contextul *class*;
(invGUID2) *attribute* este un atribut valid al lui *class*.

Cazul unicităţii "la nivelul unui container" (CUID). În acest caz, am propus următoarea soluţie.

```
pattern inv_ContainerRelativeUniqueIdentifier(container,contained:Class,
                    navigationToContained:Property, attribute:Property)=
 let bag:Bag(OclAny) = self.navigationToContained.attribute in
 ForAll_Reject(self.navigationToContained, Set{UniqueOccurrenceInBag(bag,attribute)})
```

Meta-constrângerile aferente acesteia sunt următoarele:
(invCUID1) instanţierea şablonului trebuie realizată ca şi un invariant în contextul *container*;
(invCUID2) *navigationToContained* este o referinţă în *container* având tipul *contained*;
(invCUID3) *attribute* este un atribut valid al lui *contained*.

Şablonul de specificare OCL anterior prezentat utilizează un şablon de constrângeri nou identificat - *Unique Occurrence in Bag* (*UOB*). Acesta din urmă cere ca "valoarea unui atribut precizat al unei clase să fie unică în cadrul unei colecţii de valori de acelaşi tip". Pentru UOB, am propus următorul şablon de specificare OCL.

```
pattern UniqueOccurrenceInBag(bag:OclExpression, class:Class, attribute:Property)=
 bag->count(self.attribute) = 1
```

Meta-constrângerile asociate acestuia impun ca:
(invUOB1) instanţierea şablonului să se realizeze ca şi un invariant în contextul *class*;
(invUOB2) *attribute* să fie un atribut valid al lui *class*;
(invUOB3) *bag* să fie o expresie OCL validă al cărei tip la evaluare să fie Bag;
(invUOB4) *attribute* şi elementele din *bag* să aibă acelaşi tip.

Şablonul `inv_ContainerRelativeUniqueIdentifier` poate fi generalizat, astfel încât constrângerea de unicitate să nu se aplice tuturor elementelor conţinute, ci doar unui subset, filtrat convenabil, al acestora. Mai jos, oferim şablonul de specificare OCL propus în aces caz.

```
pattern inv_GenContainerRelativeUniqueIdentifier(container,contained:Class, attribute:Property,
                                                 navigation:Feature, properties:Set(OclExpression)) =
 let subset:Set(contained) = self.navigation->select(e | oclAND(properties,e)) in
 let bag:Bag(OclAny) = subset.attribute in
 ForAll_Reject(subset,Set{UniqueOccurrenceInBag(bag,attribute)})
```

Instanţierea şablonului este constrânsă de următoarele condiţii necesare:
(invGCUID1) instanţierea trebuie efectuată ca şi un invariant in contextul *container*;
(invGCUID2) *navigation* este o proprietate în *container* având tipul *contained*;
(invGCUID3) *attribute* este un atribut valid al lui *contained*;
(invGCUID4) fiecare expresie din mulţimea *properties* este o expresie booleană validă.

4.3.2.3 Şablonul *Unique Identifier* - soluţii de tip precondiţie

Până acum, soluţiile oferite în literatură pentru şabloane de constrângeri precum *Unique Identifier* au fost formulate exclusiv in termeni de invarianţi. În cadrul tezei, am justificat însă faptul că, în contextul MDE şi în acord cu principiile Design by Contract, astfel de soluţii ar trebui furnizate şi în termenii precondiţiilor impuse asupra operaţiilor care pot viola respectiva constrângere. Ca urmare, am propus şabloane de specificare OCL corespunzătoare precondiţiilor operaţiilor a căror execuţie ar putea viola restricţia privind identificarea unică, considerând atât cazul unicităţii "la nivel global", cât cel ai unicităţii "la nivelul unui container".

Cazul unicităţii "la nivel global" (GUID). În caz "global", am propus următorul şablon de tip precondiţie pentru operaţia *class*`::set`*attribute*`()`. Precondiţia generată conservă unicitatea valorilor lui *attribute* între toate instanţele *class*.

```
pattern preSet_GloballyUniqueIdentifier(class:Class, attribute:Property, parameter:Parameter)=
 ForAll_Reject(class.allInstances(), Set{AttributeValueRestriction(attribute,<>,parameter)})
```

Următoarele constrângeri constituie condiţii necesare pentru a asigura validitatea sintactică a expresiilor OCL rezultate prin instanţiere:
(preGUID1) instanţierea trebuie realizată în contextul *class*`::set`*attribute*`()`;
(preGUID2) *attribute* este un atribut valid al lui *class*;
(preGUID3) *parameter* este unicul parametru al metodei `set`*attribute*, având acelaşi tip cu *attribute*.

Cazul unicităţii "la nivelul unui container" (CUID). În acest caz, am propus următoarele şabloane:

```
pattern preAdd_ContainerRelativeUniqueIdentifier(container,contained:Class,
       navigationToContained:Property, attribute:Property, parameter:Parameter) =
 ForAll_Reject(navigationToContained,
       Set{AttributeValueRestriction(attribute,<>,parameter.attribute)})

pattern preSet_ContainerRelativeUniqueIdentifier(container,contained:Class,
       navigationToContainer,navigationToContained:Property,
       attribute:Property, parameter:Parameter) =
 ForAll_Reject(navigationToContainer.navigationToContained,
       Set{AttributeValueRestriction(attribute,<>,parameter)}).
```

Instanţierea primului dintre şabloanele de mai sus generează o precondiţie pentru operaţia `add`*contained*`()` din *container*, menită să păstreze unicitatea valorilor lui *attribute* între toate instanţele *contained* conţinute în *container*. Aceste instanţe sunt accesibile prin intermediul referinţei *navigationToContained* din *container*. Pentru a asigura validitatea expresiilor OCL rezultate prin instanţiere, trebuie îndeplinite următoarele meta-constrângeri:
(preAddCUID1) contextul de instanţiere al şablonului trebuie să fie *container*`::add`*contained*`()`;
(preAddCUID2) *navigationToContained* este o referinţă în *container* de tip *contained*;
(preAddCUID3) *attribute* este un atribut al lui *contained*;
(preAddCUID4) *parameter* este unicul parametru al lui `add`*contained*, având tipul *contained*;

Cel de-al doilea şablon poate fi instanţiat în scopul generării unei precondiţii pentru operaţia `set`*attribute* a clasei *contained*. Meta-constrângerile aferente parametrilor acestui şablon sunt următoarele:
(preSetCUID1) contextul de instanţiere a şablonului este *contained*`::set`*attribute*`()`;
(preSetCUID2) *navigationToContained* este o referinţă în *container* de tip *contained*;
(preSetCUID3) *navigationToContainer* este o referinţă în *contained* de tip *container*, cu multplicitate 1;
(preSetCUID4) *navigationToContained* este referinţa opusă lui *navigationToContainer*;
(preSetCUID5) *attribute* este un atribut al lui *contained*;
(preSetCUID6) *parameter* este unicul parametru al metodei `set`*attribute*`()`, având acelaşi tip cu *attribute*.

4.3.3 Validarea abordării propuse

Am realizat validarea abordării propuse, demonstrând utilitatea acesteia în stabilirea corectitudinii modelelor cu ajutorul intrumentului OCLE. Pentru a accentua obligativitatea acestei cerinţe, problema corectitudinii modelelor a fost abordată prin analogie cu cea a corectitudinii programelor. În consecinţă, stabilirea corectitudinii modelelor necesită atât verificarea compilabilităţii acestora, cât şi testarea lor.

4.3.3.1 Verificarea compilabilităţii modelelor

Verificarea compilabilităţii modelelor necesită evaluarea pe respectivele modele a regulilor de bună formare (WFRs) de la nivelul metamodelului; eşuarea oricăreia dintre aceste verificări indică o eroare la nivelul modelului în cauză. Scrierea regulilor de bună formare în ideea oferirii suportului necesar depanării modelelor (abordare promovată de şabloanele propuse în cadrul acestui capitol) facilitează considerabil această etapă, eficientizând procesul de dezvoltare. Am demonstrat fezabilitatea acestei idei prin intermediul unui studiu de caz realizat pe baza unui model UML pentru componente şi a unei reguli de bună formare incluse în specificaţia metamodelului UML 1.5 [57], referitoare la unicitatea numelor în cadrul spaţiilor de nume. Am ilustrat avantajele formulării regulii în cauză ca şi o instanţiere

a şablonului de specificare `inv_GenContainerRelativeUniqueIdentifier`, în contrast cu exprimarea acestuia prin instanţierea şablonului clasic `ForAll` (situaţie existentă la nivelul documentului de specificare [57]).

4.3.3.2 Testarea modelelor

Activitatea de testare a modelelor presupune evaluarea invarianţilor definiţi la nivelul acestora (eng. Business Constraint Rules, BCRs) pe snapshot-uri (instanţieri ale modelelor). Detectarea oricăror fals-pozitive (snapshot-uri greşite pentru care evaluarea se finalizează cu succes) sau fals-negative (snapshot-uri corecte pentru care evaluarea cel puţin a unui invariant eşuează) indică erori logice la nivelul constrângerilor. Proiectarea acestor reguli in ideea asigurării suportului necesar diagnosticării erorilor uşurează considerabil procesul de depanare[3]. În cadrul acestei secţiuni, am ilustrat avantajele utilizării şablonului `ForAll_Reject` comparativ cu şablonul `ForAll` clasic în formularea unei constrângeri BCR la nivelul unui model.

4.4 Sumar

În acest capitol, am propus o nouă abordare privind definirea şabloanelor de constrângeri din modelarea orientată obiect. Contribuţiile aduse în acest domeniu sunt următoarele:

- o propunere de clarificare a terminologiei, diferenţiind între conceptul de *şablon de constrângeri* şi cel de *şablon de specificare OCL*;
- o pereche de soluţii pentru şablonul de constrângeri *For All* (formulate ca şi şabloane de specificare OCL), oferind suport adecvat diagnosticării erorilor;
- demonstrarea echivalenţei semantice a soluţiilor propuse pentru şablonul *For All* cu cea existentă în literatură, realizată folosind metoda formală B şi instrumentul AtelierB;
- o nouă abordare privind definirea şablonului de constrângeri *Unique Identifier*, distingând între unicitatea globală şi unicitatea la nivelul unui container;
- soluţii pentru şablonul *Unique Identifier* corespunzătoare fiecăruia dintre cele două cazuri, formulate în termeni de invarianţi;
- soluţii pentru şablonul *Unique Identifier* la nivel de precondiţii, în fiecare dintre cele două cazuri;
- un nou şablon de constrângeri atomic, *Unique Occurrence in Bag*;
- argumentarea validităţii şi a utilităţii soluţiilor propuse în domeniul stabilirii corectitudinii modelelor (cu ajutorul instrumentului OCLE), acoperind atât verificarea compilabilităţii modelelor, cât şi testarea acestora.

Singurul inconvenient al acestei abordării vine din faptul că soluţiile propuse generează expresii OCL mai complexe decât şablonele existente deja în literatură. Cu toate acestea, abordarea noastră are un potenţial ridicat de automatizare, ca urmare aspectul complexităţii nu reprezintă o limitare. Aplicarea şabloanelor propuse poate fi efectuată fie de la zero, fie cu scop de refactorizare a unor expresii OCL deja existente.

O direcţie viitoare de lucru vizează automatizarea abordării propuse în OCLE. De asemenea, ne focusăm în continuare pe identificarea unor noi şabloane de constrângeri, precum şi pe îmbunătăţirea soluţiilor celor existente.

Diseminarea rezultatelor raportate în acest capitol s-a realizat prin lucrările [26] şi [27].

[3] O astfel de cerinţă este un imperativ în cazul testării metamodelelor, întrucât, dat fiind potenţialul lor reutilizabil, metamodelele necesită o testare amplă pe modele de mari dimensiuni.

Capitolul 5

Semantica statică a limbajelor de (meta)modelare

Contribuţia descrisă în capitolul anterior a fost justificată ca reprezentând o modalitate eficientă de diagnosticare a erorilor, unul dintre scopuri fiind asigurarea compilabilităţii modelelor. În cadrul acestui capitol, detaliem problema compilabilităţii modelelor în context MDE, accentuând asupra caracterului imperativ al acestei cerinţe, prezentând starea de fapt în domeniu şi cauzele acesteia şi propunând soluţii de îmbunătăţire a ei.

Contribuţia noastră la acest nivel vizează stabilirea unui cadru destinat sprijinirii specificării corecte a semanticii statice a limbajelor de (meta)modelare, condiţie necesară asigurării compilabilităţii modelelor. Această contribuţie are un caracter dual. În primul rând, am propus un set de principii referitoare la specificarea semanticii statice a limbajelor de (meta)modelare. În al doilea rând, am făcut propuneri de îmbunătăţire a specificării semanticii statice a metamodelului UML şi a meta-metamodelelor MOF, Ecore şi XCore.

5.1 Motivaţie

5.1.1 Problema compilabilităţii modelelor

Paradigma MDE a cauzat o schimbare majoră de interes în ingineria programării, acesta deplasându-se de la programe şi limbaje de programare, spre modele şi limbaje de modelare. În calitatea lor de artefacte care dirijează un proces de dezvoltare puternic automatizat, o cerinţă obligatorie impusă modelelor în contextul MDE vizează compilabilitatea acestora. Prin analogie cu compilabilitatea programelor, am definit *compilabilitatea modelelor* ca şi conformanţa acestora la sintaxa abstractă şi semantica statică a limbajului lor de modelare. Sintaxa abstractă este reprezentată prin metamodelul limbajului, iar semantica statică prin regulile de bună formare (WFRs) asociate acestuia.

Starea de fapt actuală arată însă că, în ciuda caracterului imperativ al acestei cerinţe în context MDE, compilabilitatea modelelor este astăzi mai mult un deziderat decât o realitate. Aceastei situaţii i se atibuie atât cauze umane, cât şi tehnologice.

5.1.2 Analiza stării de fapt privind compilabilitatea modelelor

Susţinem că, dincolo de factorul tehnologic acuzat de majoritatea autorilor, sursa problemelor actuale din acest domeniu este dată de specificarea inadecvată şi validarea deficitară a semanticii statice a limbajelor de (meta)modelare. Afirmaţia anterioară este motivată de o analiză detaliată pe care am realizat-o, cu privire la specificarea şi utilizarea aserţiunilor

la nivelul metamodelului UML şi a trei dintre cele mai populare meta-metamodele: MOF, Ecore şi XCore.

Apreciem că soluţia acestor probleme constă în definirea şi adoptarea unui cadru conceptual riguros privind specificarea semanticii statice a limbajelor de (meta)modelare. Un astfel de cadru ar trebui să se bazeze pe un set coerent de principii referitoare la specificarea unei semantici statice.

5.2 Un cadru conceptual adecvat verificării compilabilităţii modelelor - Principii privind specificarea semanticii statice

Am identificat un număr de cerinţe necesar a fi îndeplinite de către orice set de reguli de bună formare, în scopul asigurării unei verificări eficiente a compilabilităţii modelelor.

O primă astfel de cerinţă se referă la *completitudine*a setului de reguli, acestea trebuind să acopere în totalitate semantica statică a limbajului. Acest lucru presupune o înţelegere perfectă a tuturor conceptelor metamodelului, precum şi a modalităţilor corecte de interrelaţionare a lor.

A doua cerinţă se referă la *existenţa unei formalizări în OCL sau un dialect al acestuia a întregului set de reguli.* Unele dintre abordările curente realizează implementarea explicită a regulilor direct în limbaj de programare (cazul Ecore) sau încearcă abordarea unei strategii de conservare a respectării lor la orice pas, printr-o implementare corespunzătoare a modificatorilor (cazul XCore).

În plus, fiecare WFR trebuie să îndeplinească un număr de criterii de calitate. Următoarele trei sunt printre cele mai importante, primele două fiind şi printre cele mai puţin abordate în literatură.

1. *Specificare informală detaliată, dirijată de teste.* Precedarea expresiei formale a unei reguli de o specificare detaliată şi riguroasă a variantei sale informale reprezintă cerinţa fundamentală pentru a asigura înţelegerea corectă a acesteia. La rândul său, specificarea informală trebuie însoţită de snapshot-uri relevante de test, necesare în etapa de validare a regulii (atât pozitive, cât şi negative). Prin analogie cu abordarea cunoscută sub numele de *dezvoltare dirijată de teste* (eng. Test-Driven Development), această *specificare dirijată de teste* permite o înţelegere mai bună a regulilor, având efect pozitiv asupra corectitudinii expresiilor formale ale acestora.
2. *Specificare formală orientată spre testare.* Expresiile OCL aferente regulilor trebuie proiectate astfel încât sa faciliteze depanarea modelelor. În acest sens, se recomandă utilizarea şabloanelor de specificare de tipul celor descrise în capitolul precedent.
3. *Specificare formală corectă şi eficientă.* Corectitudinea unei WFR OCL presupune două aspecte distincte: corectitudine în raport cu echivalentul său informal şi corectitudine în raport cu specificarea limbajului OCL. Primul cere o conformanţă totală a specificării OCL cu cea în limbaj natural, celălalt impune compilabilitatea, deci conformanţa cu standardul OCL.

Un alt aspect care trebuie avut în vedere la specificarea regulilor de bună formare se referă la *alegerea contextului*, fapt care necesită înţelegerea diferenţei dintre o WFR şi un invariant în sensul clasic, aşa cum a fost el introdus de tehnicile de programare orientate obiect.

Pornind de la aceste principii, am evaluat starea de fapt cu privire la specificarea semanticii statice a metamodelelor UML/MOF, Ecore şi XCore şi am facut propuneri concrete de îmbunătăţire a acesteia.

5.3 Contribuţii la specificarea semanticii statice a limbajelor de (meta)modelare

5.3.1 Semantica statică a metamodelului UML şi a meta-metamodelului MOF

5.3.1.1 Starea de fapt. Abordări în domeniu

UML este astăzi recunoscut drept limbajul *de facto* în modelarea orientată obiect, versiunea 1.4.2 [58] a acestuia fiind adoptată ca şi standard ISO. La rândul său, MOF este meta-metamodelul aflat la baza celei mai cunoscute abordări MDE şi anume OMG MDA.

În ultima decadă, au apărut diverse lucrări ([68], [41], [24]) care semnalează probleme existente la nivelul definirii WFR-urilor în documentele OMG, majoritatea concentrându-se pe aspectul necompilabilităţii regulilor în raport cu specificaţia limbajului OCL. O analiză mai atentă a specificaţiilor standard (atât a regulilor de bună formare, cât şi a operaţiilor adiţionale) relevă însă diferite alte neajunsuri ale acestor specificaţii, printre care incompletitudine, inconsistenţe, erori logice şi lipsa suportului necesar depanării modelelor.

5.3.1.2 Noi propuneri

Propunerile de îmbunătăţire a specificării semanticii statice pentru UML/MOF raportate în teză vizează două aspecte fundamentale în metamodelare, primul referitor la semantica relaţiei de compunere, iar cel de-al doilea la unicitatea numelor în cadrul spaţiilor de nume.

Asupra relaţiei de compunere UML/MOF
În conformitate cu informaţia inclusă în specificaţiile OMG ([58], [61]), am identificat patru constrângeri elementare care definesc semantica relaţiei de compunere, ca o formă particulară de asociere:

[C1] *Doar asocierile binare pot fi compuneri*;
[C2] *Cel mult un capăt al unei asocieri poate să specifice compunere* (*un container nu poate fi conţinut de o parte a sa*);
[C3] *Multiplicitatea capătului aferent compunerii poate fi cel mult 1* (*o parte nu poate fi inclusă în mai mult de un container la un moment dat*);
[C4] *Obiectele conţinute trebuie să poată fi accesate pornind de la container* (*navigarea de la container la părţi trebuie să fie permisă*).

Am investigat acoperirea acestei semantici prin WFR-uri OCL, atât în versiunile 1.x, cât şi 2.x ale metamodelelor UML/MOF. Studiul a relevat incompletitudinea setului de reguli aferent compunerii la nivelul ambelor versiuni şi a evidenţiat prezenţa unor inconsistenţe între variantele formală şi informală de specificare a regulilor în versiunea 2.x. Soluţiile propuse provin dintr-o analiză bazată pe utilizarea principiului *specificării dirijate de teste*. De asemenea, am exemplificat posibilitatea de a exprima o aceeaşi regulă în diferite contexte şi sub diferite forme, discutând şi criteriile implicate în alegerea celei mai potrivite specificări.

UML 1.x. Fragmentul din metamodelul UML 1.x destinat specificării asocierilor este ilustrat în Figura 5.1. Dintre cele patru reguli menţionate anterior, specificaţia standard le acoperă doar pe primele trei. Expresiile OCL aferente regulilor [C1] şi [C2] sunt formulate în contextul clasei `Association`, în timp ce cea corespunzătoare lui [C3] este dată în contextul `AssociationEnd`.

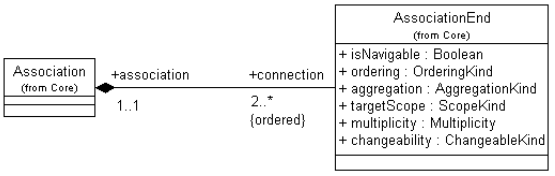

Figura 5.1: Relaţia de asociere în UML 1.4

Alternativele propuse de noi privind formalizarea regulii lipsă, [C4], sunt ilustrate în Listing-ul 5.1. Alegerea variantei optime reprezintă o decizie dependentă atât de semantica limbajului, cât şi de facilităţile existente la nivelul instrumentelor utilizate.

```
context AssociationEnd inv validCompositionNavigability1:
 self.aggregation = #composite implies
 self.association.connection->any(ae | ae <> self).isNavigable

context AssociationEnd inv validCompositionNavigability2:
 self.association.connection->exists(ae | ae <> self and
  ae.aggregation = #composite) implies self.isNavigable

context Association inv validCompositionNavigability3:
 self.connection->exists(ae | ae.aggregation = #composite) implies
 self.connection->any(ae | ae.aggregation <> #composite).isNavigable
```

Listing 5.1: Propunere de formalizare a regulii [C4] în MOF şi UML 1.x

Dintre expresiile OCL anterioare, ultima (formulată în contextul `Association`) este singura pe deplin conformă cu semantica relaţiei de compunere în UML 1.x. Din acest considerent, am reformulat şi regula [C3] din specificaţia standard în contextul `Association`, după cum urmează.

```
context Association inv validCompositionUpperBound:
 self.connection->exists(ae | ae.aggregation = #composite) implies
 self.connection->any(ae | ae.aggregation = #composite).multiplicity.max = 1
```

Listing 5.2: Propunere de formalizare a regulii [C3] în MOF şi UML 1.x

Regula propusă în Listing-ul 5.2 şi ultima din Listing-ul 5.1 pot fi combinate la nivelul unei singure expresii OCL. Aceasta însă prezintă ca şi dezavantaj necesitatea efectuării unor evaluări parţiale în cazul violării ei, în scopul identificării subexpresiei evaluate negativ.

```
context Association inv validCompositionUpperAndNavigability:
 self.connection->exists(ae | ae.aggregation = #composite) implies
 (self.connection->any(ae | ae.aggregation = #composite).multiplicity.max = 1 and
  self.connection->any(ae | ae.aggregation <> #composite).isNavigable)
```

Listing 5.3: Propunere de formalizare a regulilor [C3] şi [C4] în MOF şi UML 1.x

UML/MOF 2.x. Documentul UML 2.x Infrastructure introduce o serie de modificări în definirea relaţiei de asociere (surprinse de diagrama din Figura 5.2), modificări propagate şi la nivelul specificaţiei MOF. În ceea ce priveşte formalizarea semanticii relaţiei de compunere, evoluţia specificaţiei este una negativă. Dintre cele patru constrângeri formulate la începutul secţiunii, doar [C1] beneficiază de o formalizare corectă la nivelul documentului de specificare; expresia OCL aferentă regulii [C4] lipseşte, cea corespunzătoare regulii [C3] prezintă inconsistenţă cu echivalentul său informal, iar [C2] se regăseşte doar la nivelul documentului de specificare MOF 2.0, fiind formulată ca şi precondiţie informală a unei operaţii. În consecinţă, am propus formalizări OCL adecvate fiecareia dintre constrângerile [C2]-[C4].

Contextul natural pentru formalizarea regulii [C2] este reprezentat de metaclasa `Association`. Mai jos, prezentăm invariantul OCL aferent acesteia.

```
context Association inv atMostOneCompositeEnd:
 self.memberEnd->select(p | p.isComposite)->size() <= 1
```

Listing 5.4: Propunere de formalizare a regulii [C2] în MOF şi UML 2.x

Regulile [C3] şi [C4] pot fi formulate atât în contextul `Association`, cât şi `Property`, după cum se ilustrează în Listing-urile 5.5 şi 6.1.

```
context Association inv validCompositionMultiplicity1:
 self.memberEnd->exists(p | p.isComposite) implies
 self.memberEnd->any(p | not p.isComposite).upper = 1

context Property inv validCompositionMultiplicity2:
```

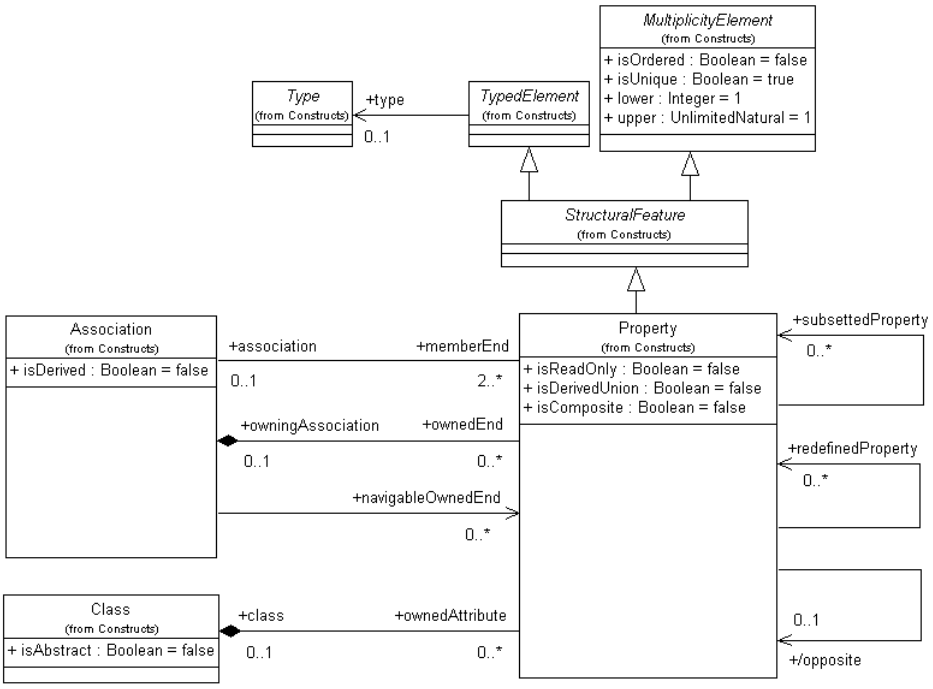

Figura 5.2: Relaţia de asociere în MOF 2.0 şi UML 2.3

```
self.isComposite and self.association->notEmpty() implies
self.association.memberEnd->any(p | p <> self).upper = 1
```

Listing 5.5: Propunere de formalizare a regulii [C3] în MOF şi UML 2.x

```
context Association inv validCompositionNavigability1:
 self.memberEnd->exists(p | p.isComposite) implies
 self.memberEnd->any(p | p.isComposite).isNavigable()

context Property def: isNavigable() : Boolean =
(self.class->notEmpty()) xor (self.owningAssociation->notEmpty() and
 self.owningAssociation.navigableOwnedEnd->includes(self))

context Property inv validCompositionNavigability2:
  self.isComposite and self.owningAssociation->notEmpty() implies
  self.owningAssociation.navigableOwnedEnd->includes(self)
```

Listing 5.6: Propunere de formalizare a regulii [C4] în MOF şi UML 2.x

Asupra constrângerii privind evitarea conflictelor în spaţii de nume
Am evidenţiat trei tipuri de probleme întâlnite la nivelul regulii de bună formare şi a operaţiilor adiţionale referitoare la evitarea conflictelor în spaţii de nume, incluse în specificaţia standard: erori sintactice, erori logice şi lipsa suportului oferit diagnosticării erorilor. Soluţia propusă în ultimul caz a necesitat utilizarea unui şablon de specificare OCL adecvat.

5.3.2 Semantica statică a meta-metamodelului Ecore

5.3.2.1 Starea de fapt

Ecore este meta-metamodelul framework-ului EMF, reprezentând cea mai cunoscută implementare EMOF (Essential MOF). Cele două meta-metamodele nu sunt însă identice. Pe de o parte, abordarea Ecore e mai pragmatică şi orientată spre implementare, pe de altă parte, începând cu versiunea EMF 2.3, Ecore include suport pentru genericitate [52], suport inexistent în EMOF.

Repository-ul Ecore include un set de reguli de bună formare implementate în Java, în cadrul clasei EcoreValidator. Însă, deşi EMF integrează un plugin OCL (MDT-OCL [37]) şi există o abordare funcţională permiţând translatarea automată a aserţiunilor OCL în cod Java [33], nu este oferită nici o specificare OCL explicită a constrângerilor implementate în EcoreValidator. Mai mult, există o singură lucrare în literatură care tratează problema formalizării în OCL a WFR-urilor Ecore. Lucrarea în cauză [43] abordează însă doar câteva reguli legate de genericitate.

5.3.2.2 Noi propuneri

În condiţiile stării de fapt rezumate anterior şi în acord cu unul dintre principiile expuse în Secţiunea 5.2, referitor la necesitatea existenţei unei formalizări (de tip) OCL a semanticii statice a limbajelor de (meta)modelare, am definit, testat şi validat în OCLE un set cuprinzător de WFR-uri OCL pentru Ecore. Setul de reguli este disponibil la adresa [4]. În cadrul acestei secţiuni, am abordat câteva reguli legate de definirea genericităţii în Ecore. Alegerea acestor reguli pentru exemplificare a fost motivată atât de nivelul lor de complexitate (fiind vorba de reguli netriviale), cât şi de faptul că permit un studiu comparativ al soluţiilor propuse cu cele descrise în [43].

Genericitate în Ecore. Fragmentul din meta-metamodelul Ecore care asigură suport pentru genericitate este ilustrat în Figura 5.3. Similar limbajului Java (al cărui model de genericitate l-a inspirat pe cel din Ecore), Ecore permite definirea de operaţii şi tipuri generice, precum şi instanţierea acestora din urmă, cu obţinerea de tipuri parametrizate. Metaconceptele suport sunt reprezentate de `ETypeParameter` şi `EGenericType`. Presupunând o mai mare familiaritate a cititorului cu Java decât Ecore, am detaliat ambele concepte, pe baza unor exemple Java relevante. O instanţă `ETypeParameter` denotă un parametru utilizat de o declaraţie a unui clasificator generic sau a unei operaţii generice. O instanţă `EGenericType` poate reprezenta o referinţă a unui parametru de tip, o instanţiere a unui tip generic, sau un wildcard. Instanţele `EGenericType` pot juca diferite roluri în cadrul unui model Ecore, fiecare rol fiind constrâns de reguli specifice. O astfel de instanţă poate reprezenta: un supertip generic al unei clase, tipul unui element (atribut, referinţă, operaţie, parametru), o limită a unui parametru de tip, unul dintre argumentele unui tip parametrizat, limita superioară sau inferioară a unui wildcard, sau un tip excepţie.

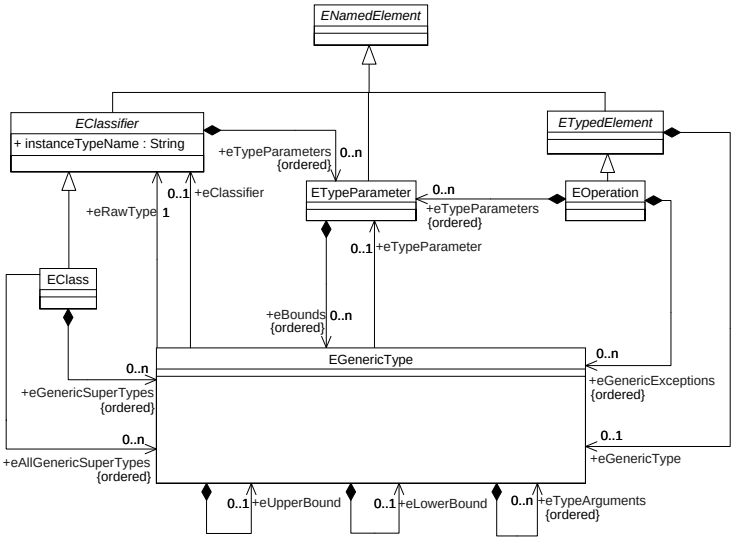

Figura 5.3: Fragment Ecore oferind suport pentru genericitate

Asupra unei reguli privind genericele Ecore

În cadrul acestei secţiuni, am oferit o formalizare OCL pentru următoarea constrângere: "*Presupunând că o instanţă EGenericType reprezintă o referinţă a unui parametru de tip, parametrul referit trebuie să fie în domeniul de vizibilitate şi să nu fie referit anterior definirii*

sale (referire de tip forward). Parametrul de tip este considerat a fi în domeniu în cazul în care containerul acestuia este un strămoş al instanţei EGenericType în cauză, în raport cu relaţia de compunere din Ecore". Invarianţii OCL propuşi pentru formalizarea acestei reguli de bună formare sunt ilustraţi în Listing-ul 5.7, în timp ce operaţiile adiţionale necesare sunt oferite în Listing-urile 5.8 şi 5.9.

```
context EGenericType
 -- The referenced type parameter must be in scope, i.e.,
 -- its container must be an ancestor of this generic type ...
 inv InScopeTypeParameter:
  self.isTypeParameterReference() implies
  self.ancestors()->includes(self.eTypeParameter.eContainer())

context EGenericType
 -- ... and must not be a forward reference.
 inv NotForwardReference:
  (self.isTypeParameterReference() and self.isUsedInATypeParameterBound())
  implies
  (let refParameter : ETypeParameter = self.eTypeParameter
   let boundedParameter : ETypeParameter = self.boundedTypeParameter()
   let paramSeq:Sequence(ETypeParameter)=
     (if refParameter.eContainer().oclIsKindOf(EClassifier)
      then refParameter.eContainer().oclAsType(EClassifier).eTypeParameters
      else refParameter.eContainer().oclAsType(EOperation).eTypeParameters
      endif)
   let posRefParameter : Integer = paramSeq->indexOf(refParameter)
   let posBoundedParameter : Integer =
     (if paramSeq->includes(boundedParameter)
      then paramSeq->indexOf(boundedParameter)
      else -1
      endif)
   in
   ((posBoundedParameter <> -1) implies
    ((posRefParameter < posBoundedParameter) or
     ((posRefParameter = posBoundedParameter) and (not boundedParameter.eBounds->includes(self)))
    )
   )
  )
```

Listing 5.7: WFR-uri OCL propuse pentru `EGenericType` restricţionând referirea incorectă a parametrilor de tip

```
context EGenericType def: isTypeParameterReference() : Boolean =
 not self.eTypeParameter.isUndefined()

context EObject def: ancestors() : Set(EObject) =
 let empty : Set(EObject) = Set{} in
 if self.eContainer().isUndefined() then empty
 else Set{self.eContainer()}->union(self.eContainer().ancestors())
 endif

context EObject def: eContainer() : EObject = oclUndefined(EObject)

context EParameter def: eContainer() : EObject = self.eOperation
--analogous definitions of eContainer() for EPackage, EClassifier, EStructuralFeature, EOperation

context ETypeParameter def: eContainer() : EObject =
 let classifier = EClassifier.allInstances()->any(c | c.eTypeParameters->includes(self))
 in (if not classifier.isUndefined() then classifier
     else EOperation.allInstances()->any(o | o.eTypeParameters->includes(self))
     endif)
--analogous definition of eContainer() for EGenericType
```

Listing 5.8: Operaţii adiţionale utilizate de invariantul `InScopeTypeParameter`

```
context EGenericType def: isUsedInATypeParameterBound() : Boolean =
 self.ancestors()->exists(o | o.oclIsTypeOf(ETypeParameter))

context EGenericType def: boundedTypeParameter() : ETypeParameter =
 self.ancestors()->any (o | o.oclIsTypeOf(ETypeParameter)).oclAsType(ETypeParameter)
```

Listing 5.9: Operaţii adiţionale utilizate de invariantul `NotForwardReference`

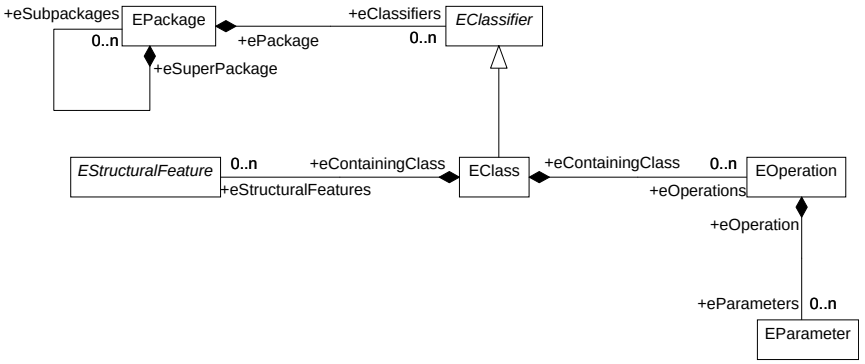

Figura 5.4: Relaţii de compunere în Ecore

5.3.2.3 Abordări în domeniu

Conform celor menţionate anterior, singurele abordări similare disponibile au fost reprezentate de implementarea existentă a clasei EMF EcoreValidator şi de lucrarea [43].

EMF EcoreValidator. Susţinem că existenţa unei formalizări OCL explicite a regulilor de bună formare Ecore este mai avantajoasă comparativ cu implementarea lor directă în Java, din cel puţin două raţiuni. Pe de o parte, OCL reprezintă limbajul standard pentru formularea unor astfel de reguli, aserţiunile OCL fiind, prin natura lor, mai compacte şi mai inteligibile raportat la echivalente lor într-un limbaj de programare. Pe de altă parte, în contextul MDE, există instrumente care permit translatarea automată a expresiilor OCL într-un limbaj de programare (OCLE şi abordarea propusă în [33] reprezintă exemple concludente în aces sens).

Conform afirmaţiilor autorilor acestuia, modelul de genericitate din Ecore îl urmează îndeaproape pe cel din Java. Cu toate acestea, testele pe care le-am efectuat au pus în evidenţă anumite inadvertenţe între specificarea genericelor în Java şi regulile implementate de validatorul Ecore. Spre exemplu, următoarea regulă privind declararea corectă a tipurilor şi metodelor generice este conţinută în specificarea limbajului Java [45] (pp. 50), însă nu se regăseşte în implementarea EMF: "*Variabilele de tip pot avea o limită opţională, T & I_1 ... I_n. Aceasta reprezintă fie o variabilă de tip, fie un tip clasă sau interfaţă (T), urmat eventual de alte tipuri interfaţă I_1, ..., I_n. ... Dacă oricare dintre I_1 ... I_n denotă o variabilă de tip sau un tip clasă, aceasta reprezintă o eroare de compilare . Ordinea tipurilor este relevantă doar în sensul în care ... un tip clasă sau o variabilă de tip poate apărea exclusiv pe prima poziţie.*". Pentru această regulă, am propus următoarea formalizare OCL.

```
context ETypeParameter
 inv ValidBounds:
  -- If a type parameter has bounds and the first bound is a
  -- type parameter reference, then there are no other bounds.
  (self.eBounds->notEmpty() and self.eBounds->first().isTypeParameterReference() implies
   self.eBounds->size() = 1)
  and
  -- If there are at least two bounds, then all
  -- except (maybe) the first one should refer to interface types.
  (self.eBounds->size() >= 2 implies Sequence{2..self.eBounds->size()}->reject(i |
                                     self.eBounds->at(i).hasInterfaceReference())->isEmpty())
```

Listing 5.10: WFR OCL propus pentru restricţionarea limitelor unui parametru de tip

Constrângerea anterioară face uz de următoarele operaţii adiţionale:

```
context EGenericType
 def: hasClassifierReference() : Boolean = not self.eClassifier.isUndefined()
```

```
def: hasClassReference() : Boolean =
 self.hasClassifierReference() and self.eClassifier.oclIsTypeOf(EClass)

def: hasInterfaceReference() : Boolean =
 self.hasClassReference() and self.eClassifier.oclAsType(EClass).interface

def: isTypeParameterReference() : Boolean = not self.eTypeParameter.isUndefined
```

Listing 5.11: Operaţii adiţionale utilizate de către invariantul `ValidBounds`

Abordarea propusă în [43]. Lucrarea în cauză propune un număr de reguli de bună formare OCL menite să testeze validitatea declaraţiilor de tipuri generice şi a tipurilor parametrizate. Am efectuat o analiză a acestor reguli, atât în raport cu obiectivul declarat al acestora, cît şi în raport cu obiectivul nostru de definire a unui set complet de WFR-uri OCL pentru Ecore. Ca urmare, am concluzionat că, deşi reprezintă un bun punct de pornire şi o bază de comparaţie, respectivele reguli prezintă diverse deficienţe datorate incompletitudinii, redundaţei şi a utilizării şablonului clasic de specificare `forAll`. Ca şi exemplu, regulile introduse pentru verificarea validităţii declaraţiilor de tipuri generice constrâng limitele unui parametru de tip la a referi doar parametri din cadrul respectivelor declaraţii, fără a interzice referinţele de tip forward (spre deosebire de propunerea noastră ilustrată în Listing-ul 5.7). Mai mult, regulile sunt axate doar pe definirea şi instanţierea corectă a tipurilor generice; operaţiile generice nu sunt luate în calcul şi nici diferitele utilizări posibile ale unui tip generic.

5.3.3 Semantica statică a meta-metamodelului XCore

5.3.3.1 Starea de fapt

XCore reprezintă nucleul abordării XMF, o facilitate de metamodelare similară MOF, axată pe surprinderea tuturor aspectelor legate de definirea unui limbaj - sintaxă abstractă, sintaxă concretă şi semantică. Spre deosebire de MOF însă, XMF este complet autodefinit şi oferă suport independent de platformă pentru executabilitate, prin intermediul unui dialect OCL executabil, numit XOCL.

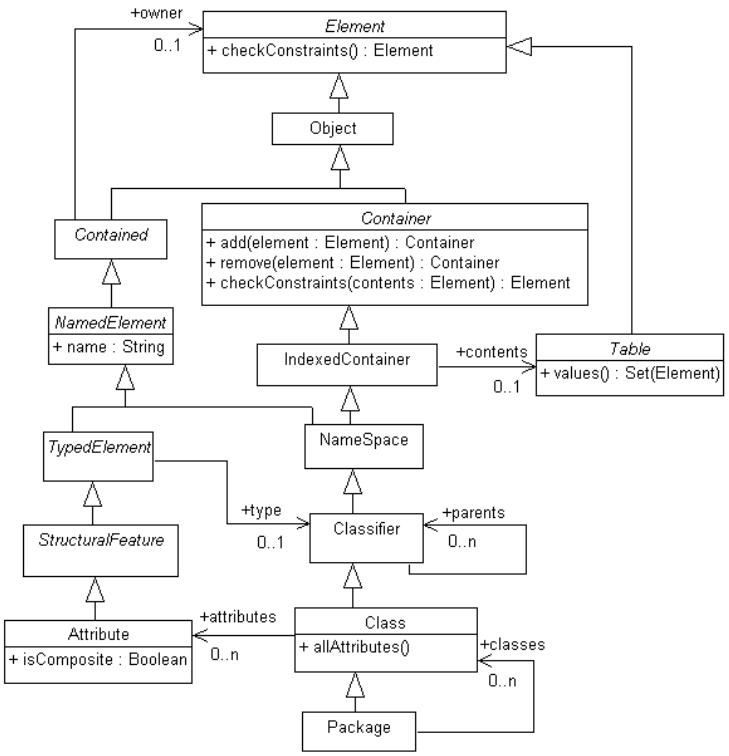

Figura 5.5: Un fragment din meta-metamodelul XCore

Referinţa standard XMF [30] accentuează relevanţa regulilor de bună formare şi promovează utilizarea acestora în definirea semanticii statice a limbajelor de modelare. Cu toate acestea, documentul nu descrie (nici la nivel formal şi nici informal) nici o astfel de regulă pentru meta-metamodelul XCore. În ceea ce priveşte implementarea XMF, aceasta include doar două WFR-uri XOCL explicite. În afara acestora, există un număr de alte reguli a căror satisfacere continuă a fost forţată printr-o implementarea corespunzătoare a modificatorilor. Am

identificat diverse puncte slabe ale acestei din urmă abordări. În plus, chiar şi cu abordarea propusă, implementarea XMF nu acoperă unele dintre regulile elementare din modelarea orientată obiect, printre care evitarea conflictelor de nume între atribute/proprietăţi ale aceleiaşi clase sau gestiunea adecvată a dependeţelor container-obiect conţinut.

5.3.3.2 Noi propuneri

În scopul soluţionării problemelor semnalate anterior, am propus un set de reguli de bună formare XOCL pentru meta-metamodelul XCore, care au fost validate folosind exemple relevante de test. Întregul set de reguli propus, împreună cu testele aferente, pot fi consultate la adresa [4]. În cadrul acestei secţiuni, am discutat două exemple relevante, referitoare la restricţia de unicitate a numelor şi relaţia de compunere.

Asupra unei reguli privind evitarea conflictelor de nume
Conform celor menţionate anterior, una dintre regulile de bună formare neacoperite de implementarea XMF priveşte conflictele de nume dintre atributele proprii ale unei clase şi atributele moştenite din ascendenţi. În scopul identificării modelelor invalide în raport cu această constrângere, am propus următoarea regulă de bună-formare: *La nivelul unei clase, sunt interzise conflictele de nume între atributele proprii si cele moştenite.* Listing-ul 5.12 oferă formalizarea XOCL a acestei reguli. Fragmentul referit din metamodelul XCore este ilustrat în Figura 5.5.

```
context Attribute @Constraint uniqueName
 let allAtts = self.owner.allAttributes() then
     sameNameAtts = allAtts->excluding(self)->select(att |
                    att.name.asSymbol() = self.name.asSymbol())
 in sameNameAtts->isEmpty()
 end

fail
  let sameNameAtts = self.owner.allAttributes()->excluding(self)->
                     select(att | att.name.asSymbol() = self.name.asSymbol()) then
      msg = "Attribute name duplication! Inherited/owned attributes of " +
            self.owner.toString() + " with the same name: "
  in @While not sameNameAtts->isEmpty() do
       let att = sameNameAtts->sel
       in msg := msg + att.owner.toString() + "::" + att.toString() + "; ";
          sameNameAtts := sameNameAtts->excluding(att)
       end
     end;
     msg
  end
end
```

Listing 5.12: WFR XOCL propus interzicând conflictele de nume între atributele proprii şi cele moştenite ale unei clase

Asupra relaţiei de compunere XCore
Spre deosebire de UML/MOF şi Ecore, XCore reprezintă compunerile în mod explicit, prin intermediul metaclaselor abstracte `Contained` şi `Container`. În concordanţă cu semantica relaţiei de compunere, susţinem că există două reguli fundamentale care trebuie îndeplinite de către orice model XCore. Aceste reguli corespund constrângerilor [C3] şi [C2] formulate în Secţiunea 5.3.1.2, în contextul UML/MOF.

[C1'] *O parte este inclusă în cel mult un container la un moment dat.*

[C2'] *Un container nu poate fi conţinut de către una dintre părţile sale.*

În cazul primei reguli, am propus constrângerea XOCL ilustrată în Listing-ul 5.13, cu următorul echivalent informal: "*Toate instanţele* `Contained` *aparţinând unui* `IndexedContainer` *trebuie să aibă acel container ca şi* `owner`.". Constrângerea propusă surprinde o

categorie mai largă de anomalii (nu doar părţi conţinute simultan în containere diferite), spre exemplu părţi aparţinând unui container şi neavând nici un `owner` curent.

```
context IndexedContainer @Constraint validOwnerForContents
  self.contents.values()->select(v | v.oclIsKindOf(Contained) and
        v <> null)->reject(v | v.owner = self)->isEmpty()

  fail "The elements from " + self.contents.values()->select(v | v.oclIsKindOf(Contained)
        and v <> null)->select(v | v.owner <> self).toString() +
      " should have " + self.toString() + " as the owner!"
end
```

Listing 5.13: Constrângere XOCL propusă pentru regula [C1']

În cazul celei de-a doua reguli, am propus constrângerea XOCL listată mai jos, ce se aplică tuturor containerelor indexate, mai puţin spaţiului de nume `Root` (în XMF, `Root` este spaţiul de nume global, fiind autoconţinut). Echivalentul său informal este următorul "*Nici o instanţă* `IndexedContainer` *diferită de spaţiul de nume* `Root` *nu poate fi conţinută de către una dintre părţile sale.*".

```
context IndexedContainer @Constraint notOwnedByPart
  (self <> Root and self.oclIsKindOf(Contained)) implies
   self.contents.values()->select(v | self.owner = v)->isEmpty()

   fail "This container is owned by each of its parts from " +
        self.contents.values()->select(v | self.owner = v).toString()
end
```

Listing 5.14: Constrângere XOCL propusă pentru regula [C2']

5.4 Sumar

Prin propunerile făcute în acest capitol, am contribuit la stabilirea unui cadru conceptual riguros, menit să sprijine specificarea corectă a semanticii statice a limbajelor de (meta)modelare şi să permită verificări eficiente ale compilabilităţii modelelor. Abordarea prezentată vizează îmbunătăţirea situaţiei actuale în domeniu prin următoarele:

- o analiză detaliată a problemei compilabilităţii modelelor, prin analogie cu compilabilitatea programelor;
- un set de principii referitoare la specificarea unei semantici statice;
- un număr de îmbunătăţiri în specificarea semanticii statice a metamodelului UML şi a meta-metamodelului MOF;
- un set de reguli de bună formare (WFRs) şi operaţii adiţionale (AOs) specificate în OCL, menite să îmbunătăţească specificarea semanticii statice a meta-metamodelului EMF Ecore;
- un set de WFRs şi AOs specificate în XOCL, menite să îmbunătăţească specificarea semanticii statice a meta-metamodelului XMF XCore;

Cercetările viitoare vizează, în primul rând, extinderea setului de reguli corectate/propuse, pentru a acoperi în întregime versiunile 2.x ale metamodelelor UML şi MOF. Un aspect complementar este legat de investigarea problemelor legate de consistenţă şi redundanţă într-un set dat de constrângeri (WFRs, în particular). Translatarea regulilor în B şi folosirea instrumentelor de demonstrare asociate metodei poate fi utilă în acest sens. O altă direcţie de cercetare ar putea fi reprezentată de identificarea şi formalizarea unui nucleu de şabloane de constrângeri întâlnite la nivelul meta-metamodelelor.

Principiile, împreună cu propunerile făcute în cazul UML/MOF au fost publicate în [25], cele referitoare la Ecore în [79], iar cele pentru XCore în [75] şi [74]. Contribuţiile prezentate în acest capitol sunt legate şi de rezultatele raportate în [28].

Capitolul 6

Specificarea componentelor soft

În capitolele precedente, ne-am axat pe artefacte reutilizabile la un nivel înalt de abstractizare (şabloane de proiectare, şabloane de constrângeri şi metamodele), oferind soluţii pentru fundamentarea formală a reutilizării acestora. În acest capitol, descriem două contribuţii în domeniul componentelor soft. Prima reprezintă o contribuţie în cadrul unei abordări de inginerie inversă, având ca şi scop extragerea unor specificări structurale şi comportamentale din codul sistemelor bazate pe componente. Această abordare a făcut tema proiectului internaţional ECO-NET [1, 10]. A doua contribuţie vizează stabilirea unui cadru care să permită o specificare contractuală completă a componentelor soft, insistând pe definirea contractelor semantice.

6.1 Ingineria inversă a specificării componentelor din cod

6.1.1 Motivaţie şi abordări în domeniu

Motivaţia cercetărilor raportate în cadrul acestei secţiuni este dată de discrepanţa existentă între abordările bazate pe componente academice şi industriale [10]. Mai precis, cele academice (Fractal, SOFA, Kmelia, KADL) sunt axate pe specificare; acestea definesc modele de componente abstracte, ierarhice, dotate cu formalisme comportamentale şi modalităţi de verificare a diferitor proprietăţi. Unele dintre ele acoperă şi rafinarea şi generarea de cod, însă majoritatea nu abordează probleme legate de implementare. În schimb, abordările industriale (CCM, EJB, OSGI, .NET) sunt focusate pe implementare, oferă infrastructuri de execuţie puternice şi mature, însă nu dispun de mecanismele de verificare necesare unei reutilizări sigure a componentelor.

O astfel de discrepanţă determină lipsa unei trasabilităţi între specificările şi implementările componentelor. La rândul său, aceasta face imposibilă asigurarea, la nivelul implementării, a proprietăţilor demonstrate pentru modelul abstract şi determină probleme de întreţinere. În cadrul MDE, astfel de probleme pot fi soluţionate atât prin abordări de inginerie directă (eng. *direct engineering*, [64, 66]) cât şi de inginerie inversă (eng. *reverse engineering*, [14, 63]). În acest context, obiectivul proiectului ECO-NET [1][1] a fost acela de a contribui la domeniul ingineriei inverse, prin dezvoltarea de tehnici şi instrumente pentru extragerea unor abstractizări structurale şi comportamentale din codul componentelor [10].

[1] un proiect internaţional la care am participat ca şi membru

6.1.2 Cadrul general

Scopul proiectului a fost acela de a stabili o legătură între codul componentelor (*modelul concret* sau *sursă*) şi specificarea acestora (*modelul abstract* sau *ţintă*). Pentru a gestiona complexitatea unui astfel de proces, modelul concret a fost limitat la cod Java, iar cel abstract la înstanţieri ale unor modele de componente incluzând atât aspecte structurale, cât şi comportamentale (SOFA [21], Kmelia [9], KADL [64], sau Fractal [20]).

Principala contribuţie raportată în cadrul proiectului constă într-o abordare privind ingineria inversă a componentelor, oferind următoarele facilităţi:

- *Metamodelul CCMM* (*Common Component MetaModel*) pentru componente, adresând atât problema gestionării unitare a diferitor modele concrete de componente (SOFA, Kmelia, KADL, Fractal), cât şi pe cea a păstrării legăturilor care asigură trasabilitatea între modelele sursă şi ţintă. Metamodelul a fost utilizat pentru generarea API-ului (Application Programming Interface) partajat de cele două procese de abstractizare;
- Un *proces de abstractizare structurală* (procesul *SA*) şi un instrument asociat folosit pentru extragerea informaţiilor structurale din modelul sursă;
- Un *proces de abstractizare comportamentală* (procesul *BA*) şi un prototip de instrument, care utilizează modelul sursă şi ieşirea procesului SA, în scopul extragerii unei specificări comportamentale a componentelor.

Legăturile care asigură trasabilitatea sunt memorate atât în modelul ţintă (prin intermediul unor atribute specializate definite în clasele CCMM), cât şi în cel sursă (prin intermediul unor adnotări Java dedicate).

6.1.3 Metamodelul CCMM

Contribuţia noastră în cadrul proiectului s-a concretizat prin participarea le definirea metamodelului CCMM (în special a semanticii statice a acestuia), validarea lui şi generarea API-ului aferent. Asupra metamodelului s-au impus un număr de trei constrângeri fundamentale. Prima a vizat genericitatea; metamodelul a trebuit să abstractizeze peste diferite modele concrete de componente (modelele partenerilor, SOFA, Kmelia, KADL, Fractal), reunind un set comun de concepte ale acestora. Cea de-a doua constrângere a vizat definirea unei modalităţi de păstrare a legăturilor dintre modelul sursă şi cel ţintă. În sfârşit, cea de-a treia cerinţă s-a referit la specificarea completă a metamodelului, cu includerea tuturor regulilor de bună formare şi operaţiilor adiţionale necesare. În plus, s-a impus necesitatea asigurării unui suport la nivel de instrumente pentru testarea metamodelului şi generarea repository-ului asociat.

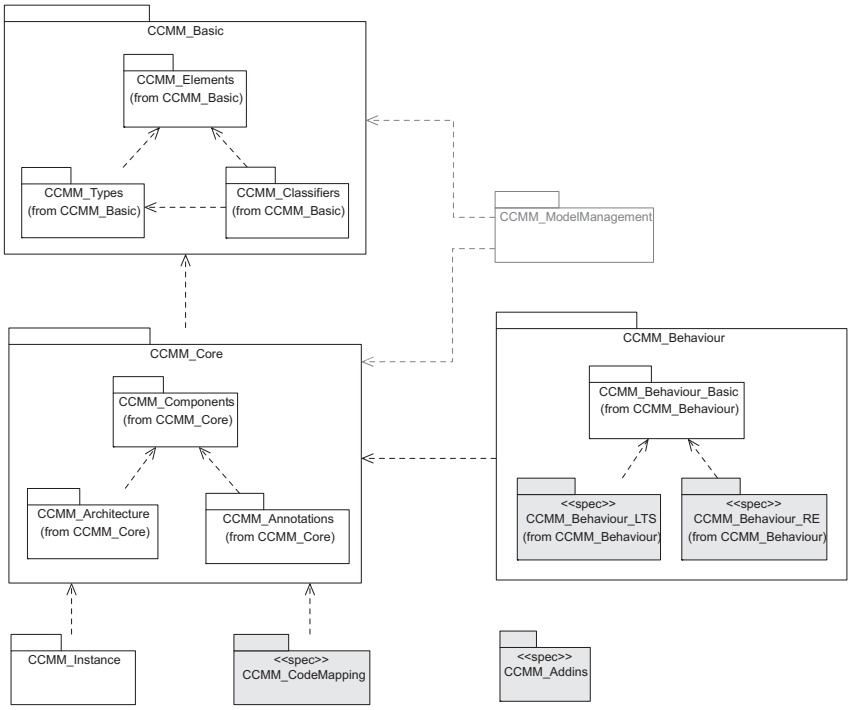

Figura 6.1: CCMM v1.1

Aceste cerinţe au condus la definirea metamodelului CCMM, a cărui arhitectură generală este ilustrată în Figura 6.1. Teza cuprinde o descriere detaliată a tuturor pachetelor utilizate pentru generarea API-ului, incluzând concepte, relaţii, operaţii adiţionale şi reguli de bună formare.

Pentru exemplificare, Figura 6.2 ilustrează conţinutul pachetului `CCMM_Components`, conţinând concepte care oferă o descriere de tip black-box a componentelor. Conform di-

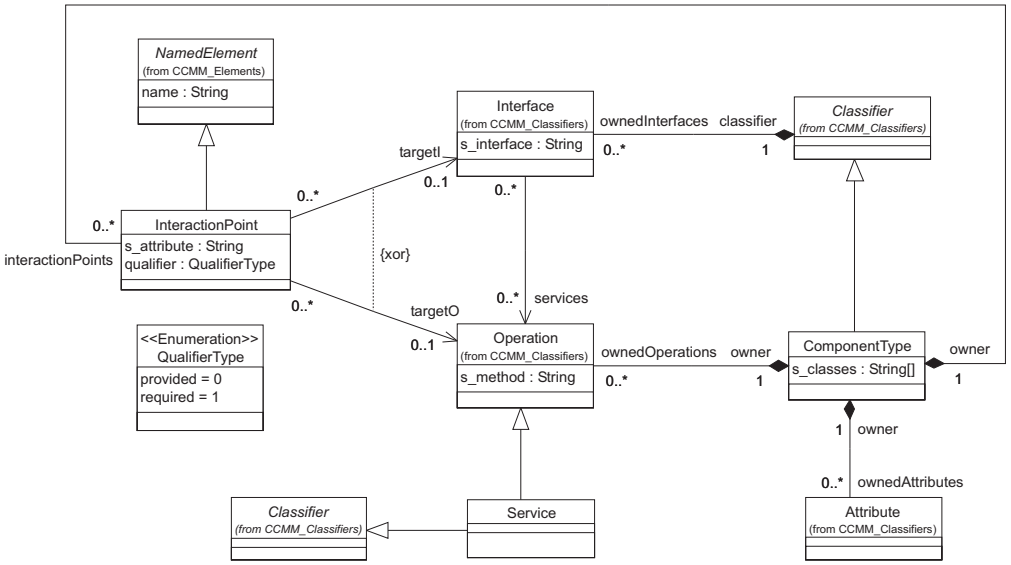

Figura 6.2: `Pachetul CCMM_Components`

agramei, un `ComponentType` denotă o entitate black-box, definită ca o specializare a lui `Classifier`. Orice `ComponentType` interacţionează cu mediul prin intermediul unui număr de `InteractionPoints`. Fiecare dintre acestea exprimă fie o funcţionalitate furnizată, fie una solicitată, putând referi o interfaţă sau o operaţie, funcţie de modelul concret de componente considerat (interfaţă în cazul SOFA şi operaţie în cazul modelului Kmelia). Toate punctele de interacţiune conţinute de un anumit tip de componentă trebuie să aibă acelaşi tip, fapt exprimat prin intermediul invariantului OCL propus în cele ce urmează.

```
context ComponentType
 inv consistentInteractionPoints:
  -- if at least one interaction point targets an interface,
  self.interactionPoints->exists(ip:InteractionPoint | ip.targetsInterface()) implies
  -- all interaction points should target interfaces
  self.interactionPoints->reject(ip:InteractionPoint | ip.targetsInterface())->isEmpty()

context InteractionPoint::targetsInterface():Boolean
 body: self.targetO.oclIsUndefined()
```

6.1.4 Validare şi instrumente

CCMM a fost implementat ca un metamodel Ecore, pentru a beneficia de suportul oferit de către platforma EMF. Pe baza acestuia, am generat codul repository-ul asociat (incluzând implementarea completă a regulilor de bună formare şi a operaţiilor adiţionale), precum şi un editor de modele arborescent. Atât repository-ul, cât şi editorul au fost furnizate ca si pluginuri Eclipse. Validarea întregii abordări s-a efectuat pe modelul netrivial CoCoME [67]. Aceasta a inclus testarea şi validarea metamodelului CCMM, cu toate operaţiile adiţionale şi regulile de bună formare. Detalii referitoare la procesele, instrumentele şi experimentele realizate pot fi găsite în SVN-ul ECONET [2].

6.2 ContractCML - un limbaj contractual de specificare a componentelor

6.2.1 Motivaţie şi abordări în domeniu

Reutilizarea sigură, în regim black-box, a unei componente soft necesită o specificare contractuală detaliată a tuturor serviciilor furnizate şi solicitate de către respectiva componentă. Patru nivele contractuale au fost identificate în contextul specificării componentelor soft [15] şi anume: *nivelul elementar*, *nivelul comportamental*, *nivelul de sincronizare* şi *nivelul nefuncţional*. Nivelul contractual elementar presupune *specificarea sintactică* a componentelor soft [32], incluzând doar signaturile serviciilor oferite şi solicitate. Nivelul doi, cel comportamental, vizează *specificarea semantică* a serviciilor folosind pre şi post-condiţii. Contractele de pe acest nivel oferă o descriere comportamentală a componentelor în termenii serviciilor individuale, privite ca şi operaţii atomice care se execută într-un context secvenţial. Spre deosebire de acestea, contractele de pe nivelul următor descriu comportamentul global al componentelor. Acesta include dependenţe între serviciile unei componente, precum secvenţiere sau paralelism, într-un mediu concurent, distribuit. Contractele de pe nivelul patru acoperă proprietăţile nefuncţionale ale componentelor.

În ciuda unui acord unanim în literatură privind obligativitatea existenţei unei specificări semantice a componentelor soft (nivelul doi contractual), singura formă de specificare utilizată de către modelele industriale dedicate de componente (EJB, COM, sau CCM) rămâne cea sintactică [32]. Modelele academice de componente aduc unele îmbunătăţiri ale acestei stări de fapt, prin includerea unor formalisme de descriere comportamentală. Fractal şi SOFA, spre exemplu, utilizează protocoale comportamentale în acest scop [65]. Acest tip de specificare corespunde însă nivelului trei contractual, fiind omise informaţiile legate de specificarea semantică a serviciilor.

În acest context, obiectivul nostru general a fost acela de a pune bazele unui cadru care să sprijine o specificare contractuală completă, pe patru nivele, a componentelor soft, permiţând verificarea interoperabilităţii componentelor. Un astfel de cadru este imaginat ca având la bază un limbaj specific de descriere a componentelor (DSML). Motivaţia acestui fapt are un caracter dual. Pe de o parte, limbajele DSML fiind dedicate anumitor domenii de aplicaţie, modelele realizate cu ajutorul lor sunt mai uşor de înţeles şi de gestionat, comparativ cu cele care utilizează un limbaj general de modelare (UML, spre exemplu). În al doilea rând, optarea pentru un DSML permite exploatarea suportului generos existent la nivel de instrumente. Astfel EMF [35], GMF [36], oAW [5] şi XMF-Mosaic [3] sunt meta-instrumente MDE care oferă posibilitatea specificării, testării şi validării DSML-urilor la toate nivelele (sintaxă abstractă, concretă şi semantică), precum şi a dezvoltării de instrumente specifice limbajelor în cauză.

6.2.2 Metamodelul ContractCML

În această secţiune, am introdus limbajul ContractCML (Contract Component Modeling Language), un DSML pentru componente pe care l-am propus ca şi bază a cadrului imaginat. ContractCML este un limbaj de modelare a componentelor ierarhic, axat pe specificarea contractelor. Momentan, metamodelul acestuia acoperă primele două nivele contractuale (ne-am axat pe reprezentarea contractelor la nivel semantic, absente din modelele actuale de componente), însă arhitectura sa extensibilă facilitează integrarea facilă a celorlalte două nivele.

6.2.2.1 Arhitectura metamodelului

Metamodelul ContractCML a fost proiectat în manieră modulară şi incrementală, pornind de la concepte sintactice de bază, cărora li s-au adăugat aspecte semantice şi arhitecturale. Arhitectura generală a metamodelului, ilustrând pachetele conţinute şi dependenţele acestora e dată în Figura 6.3. Pachetul `Basic` conţine concepte de modelare elementare, de interes general. Depinzând de `Basic`, pachetul `InterfaceSpec` reuneşte conceptele utilizate în specificarea contractelor sintactice şi semantice ale componentelor. `BlackBoxComponent` include metaclase care oferă o descriere black-box a componentelor. Din această perspectivă, fiecare componentă are un tip de componentă asociat, acesta din urmă fiind imaginat ca şi colecţia tuturor porturilor furnizate şi solicitate de către componenta în cauză; fiecare port defineşte o interacţiune cu mediul şi este tipizat de o interfaţă. Pachetul `Architecture` conţine concepte utilizate pentru descrierea arhitecturilor de componente; o astfel de arhitectură conţine o mulţime de instanţe ale componentelor şi un set de leagături (eng. *assembly bindings*) între acestea. În final, având conceptele legate de definirea black-box a componentelor şi cele arhitecturale definite, pachetul `WhiteBoxComponent` depăşeşte perspectiva client, oferind o descriere arhitecturală a componentelor. Acestea sunt clasificate în primitive şi compuse, cele din urmă fiind definite printr-o arhitectură şi un număr de legături de delegare (eng. *delegation bindings*).

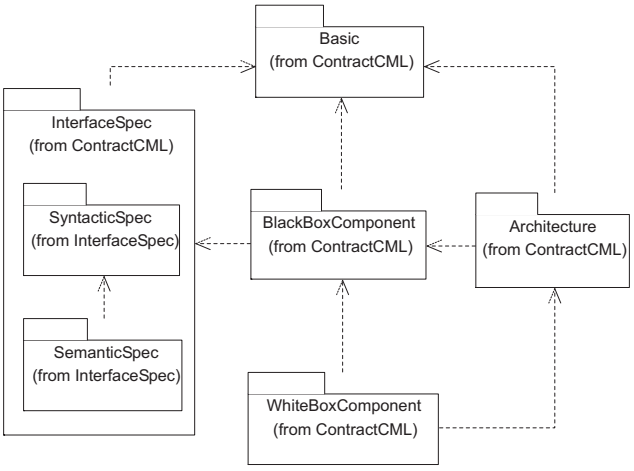

Figura 6.3: Arhitectura metamodelului ContractCML

Această secţiune detaliază semantica metaconceptelor conţinute în pachetele menţionate, incluzând regulile de bună formare şi operaţiile adiţionale asociate. Mai interesante sunt regulile care definesc semantica celor două tipuri de legături între componente.

6.2.2.2 Contracte pe nivelul 1. Specificarea sintactică a interfeţelor

Elementele pachetului `SyntacticSpec`, unul dintre cele două subpachete din `InterfaceSpec`, descriu interfeţele componentelor din perspectivă sintactică. O interfaţă e o entitate cu nume constând dintr-o colecţie de operaţii, fiecare operaţie având, la rândul său, un nume, o listă ordonată de parametri şi, posibil, un tip returnat. Un parametru este definit prin nume, tip şi categorie, ultima specificând direcţia fluxului de date (intrare, ieşire, mixt sau nespecificat). Prin intermediul metaclaselor definite în acest pachet (`Interface`, `Operation`, `Parameter`, `ParameterSort`), ContractCML permite exprimarea contractelor la nivelul întâi. Astfel de contracte stau la baza verificării interoperabilităţii componentelor. Au fost definite operaţii adiţionale necesare verificării compatibilităţii sintactice a operaţiilor şi interfeţelor (în termeni de potrivire exactă, eng. *exact match*).

Existenţa unei specificări sintactice a tuturor interfeţelor furnizate şi solicitate de către o componentă este obligatorie pentru a permite utilizarea acesteia. O utilizare corectă necesită însă şi prezenţa unei specificări comportamentale. Informaţiile de natură comportamentală pot fi specifice unei interfeţe (surprinzând aspecte legate de funcţionalitatea unei interfeţe independent de alte interfeţe ale aceleiaşi componente) sau globale (implicând mai multe interfeţe). În afara conceptelor prezentate anterior, am inclus în cadrul acestui pachet metaclasa `BehaviorSpec`, cu rol de abstractizare a comportamentului specific unei interfeţe;

fiecare instanţă `BehaviorSpec` este conţinută de obiectul `Interface` aferent. Specificările comportamentale concrete (la nivelul doi - exprimând semantica serviciilor folosind pre şi post-condiţii sau la nivelul trei - restricţionând ordinea apelurilor), ar trebui derivate din `BehaviorSpec` şi descrise în pachete specializate. Această abordare asigură o uşoară gestiune şi extindere a metamodelului.

6.2.2.3 Contracte pe nivelul 2. Specificarea semantică a interfeţelor

Dependent de `SyntacticSpec`, pachetul `SemanticSpec` adaugă suport contractual pe nivelul doi limbajului ContractCML. Pentru a asigura o specificare semantică a interfeţelor, am urmat o abordare Design by Contract. Metaclasele reprezentate în Figura 6.4 şi relaţiile dintre acestea au fost inspirate de conceptele de specificare introduse în [23].

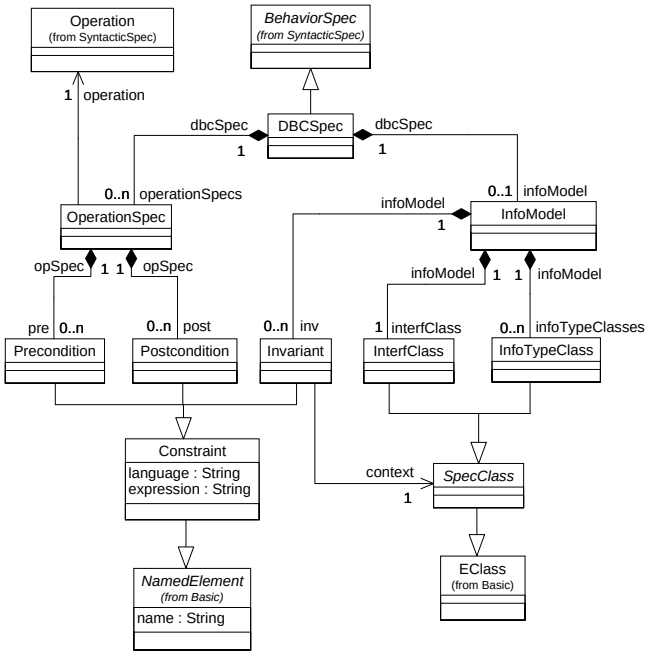

Figura 6.4: Specificarea semantică a interfeţelor

O instanţă `DBCSpec` denotă o astfel de specificare semantică ataşată unei interfeţe. Aceasta constă dintr-o mulţime de specificări de operaţii (câte una pentru fiecare serviciu expus de interfaţa în cauză), împreună cu un model informaţional. Modelul informaţional ataşat unei interfeţe reprezintă o abstractizare a acelei părţi a stării unei componente, care afectează sau poate fi afectată de execuţia operaţiilor din interfaţă [23]. Acesta nu expune detalii de implementare, fiind doar o abstractizare care permite definirea comportamentului operaţiilor.

Metaclasa `OperationSpec` permite specificarea comportamentului instanţei `Operation` asociate, sub forma unor perechi pre/post-condiţie. O precondiţie e un predicat definit în termenii parametrilor de intrare şi a modelului informaţional; o postcondiţie este un predicat ce poate referi parametrii de intrare şi ieşire, starea imediat anterioară apelului şi cea imediat următoare.

Am descris provocările apărute la transpunerea în metamodelul ContractCML a conceptelor introduse în [23] pentru specificarea semantică a interfeţelor. Pentru a putea defini modelele informaţionale, există necesitatea integrării în metamodelul ContractCML a conceptelor utilizate în reprezentarea modelelor de clase.

6.2.2.4 O strategie de tip *model weaving* pentru reprezentarea modelului informaţional

Pentru soluţionarea problemei menţionate anterior, am abordat o strategie de tip *model weaving*. Model weaving-ul reprezintă o tehnică de transformare, ce permite compunerea unor modele diferite, dar conexe, într-un întreg coerent [46]. Se disting două tipuri de weaving, simetric şi asimetric; cel asimetric lucrează cu un model de bază şi unul sau mai multe modele aspect pe care le integrează în bază, spre deosebire de cel simetric, în care nu există un model de bază declarat explicit [46] .

Am utilizat XWeave [46], un instrument de weaving asimetric bazat pe meta-metamodelul EMF Ecore, considerând ca şi bază o reprezentare Ecore a metamodelul ContractCML, iar ca şi aspect o copie Ecore. Compunerea celor două metamodele s-a realizat pe baza unor

potriviri de nume[2] între conceptele evidenţiate în Figura 6.5. În cadrul tezei, am inclus workflow-ul oAW utilizat pentru a realiza weaving-ul dorit.

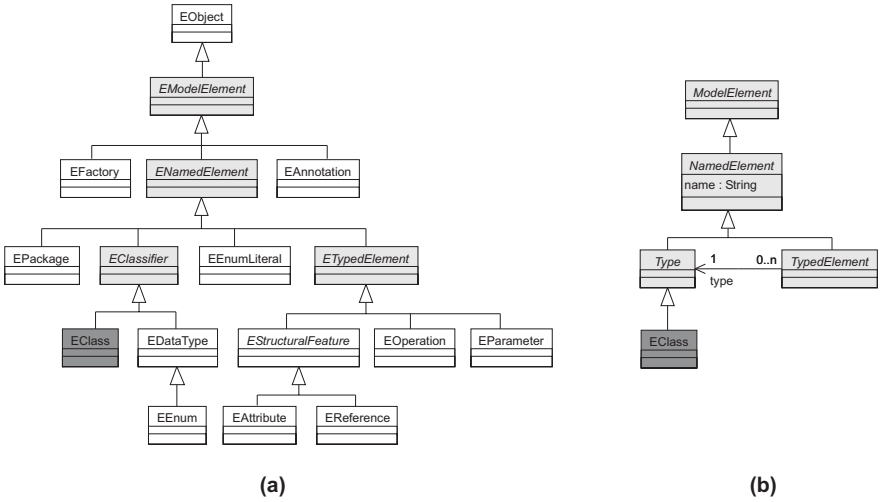

Figura 6.5: `Concepte analoage Ecore` (a) şi `ContractCML::Basic` (b)

6.2.3 Exemplu de modelare folosind ContractCML

Pentru a oferi un exemplu de modelare utilizând ContractCML, am considerat ca şi studiu de caz o variantă simplificată a sistemului de rezervări descris în [23] şi [29]. Acesta a permis ilustrarea unor aspecte legate de sintaxa concretă a limbajului propus şi a oferit condiţiile evidenţierii avantajelor unui limbaj de modelare a componentelor incluzând facilităţi de specificare semantică. Astfel, ulterior construirii modelului, am raţionat asupra compatibilităţii de tip plug-in a două interfeţe, pe baza specificării lor semantice; au fost luate în considerare atât specificaţiile operaţiilor, cât şi modelul informaţional.

6.2.4 Simularea execuţiei componentelor folosind ContractCML

În cadrul acestei secţiuni, am propus o metodă de simulare a execuţiei serviciilor componentelor ContractCML, având rol în testarea interoperabilităţii acestor componente. Metoda se bazează pe propunerea anterioară privind reprezentarea modelelor informaţionale ale interfeţelor. Simularea are loc în cadrul framework-ului XMF Mosaic, fiind susţinută de o reprezentare XCore a metamodelului ContractCML şi de utilizarea limbajului XOCL.

6.2.4.1 Metoda de simulare propusă

În secţiunea 6.2.2.3, am evidenţiat rolul modelului informaţional al unei interfeţe în specificarea semantică a serviciilor respectivei interfeţe. În cadrul acestei secţiuni, am ilustrat utilitatea modelului informaţional în simularea execuţiei acestor servicii, în contextul folosirii unui dialect OCL executabil (XOCL) şi a unui cadru de execuţie corespunzător (XMF Mosaic). Metoda de simulare propusă a necesitate extinderea metamodelului ContractCML, prin integrarea suportului adecvat reprezentării contractelor de realizare[3] asociate tipurilor de componente [23]. Un astfel de contract detaliază modalitatea de proiectare a serviciilor oferite de către o componentă în termenii serviciilor solicitate. Restricţiile referitoare la definirea acestui tip de contracte au fost formalizate ca şi constrângeri XOCL.

[2] În acest scop, unele concepte Ecore au fost în prealabil redenumite.

[3] Contractele discutate anterior sunt numite generic *contracte de utilizare*.

Logica simulării este implementată în XOCL, la nivelul metaclasei `ConstractCML::Simulator`. Metoda `simulate()` a acestuia permite simularea execuţiei unui serviciu al unei componente, atât serviciul, cât şi argumentele de apel fiind transmise ca şi parametri. Se presupune că acea componentă e parte a unei arhitecturi, în care serviciile solicitate sunt furnizate de către alte componente. Aspectul central al strategiei propuse este reprezentat de modalitatea de configurare a obiectului care asigură infrastructura de simulare.

6.2.4.2 Validarea metodei propuse

Validarea metodei de simulare propuse s-a realizat folosind o variantă extinsă a studiului de caz utilizat în 6.2.3, ilustrând utilitatea acesteia în testarea interoperabilităţii componentelor.

6.3 Sumar

În cadrul acestui capitol, am prezentat două contribuţii în domeniul specificării componentelor soft.

Prima contribuţie este parte a unei abordări de inginerie inversă a componentelor, abordare propusă în contextul unei colaborări internaţionale. Sarcinile noastre în cadrul proiectului au vizat:

- participarea la definirea unui metamodel pentru componente (CCMM), care să abstractizeze peste modelele de componente ale partenerilor şi să fie utilizat ca şi ţintă a procesului de inginerie inversă. Metamodelul a fost testat şi validat pe un studiu de caz complex;
- generarea repository-ului asociat şi a unui editor de modele. Ambele au fost oferite ca şi plugin-uri Eclipse, primul integrând funţionalitatea necesară verificării compilabilităţii modelelor.

Specificarea completă a metamodelului (incluzând toate WFRs şi AOs definite) se regăseşte în documentul [11]. Înstrumentele dezvoltate în cadrul proiectului au fost prezentate în [12].

A doua contribuţie vizează o abordare unitară privind tratarea contractelor şi compunerii componentelor. Propunerea noastră oferă beneficii în domeniul verificării interoperabilităţii componentelor, prin următoarele:

- ContractCML - un limbaj de modelare a componentelor (DSML) focusat pe reprezentarea contractelor. Momentan, ContractCML oferă suport pentru primele două nivele contractuale (sintactic şi semantic), însă arhitectura sa flexibilă permite adăugarea facilă a celorlalte două (nivelul de sincronizare şi cel nefuncţional). Principalul avantaj al ContractCML raportat la modelele de componente existente constă în capacitatea acestuia de reprezentare a contractelor la nivel semantic;
- o metodă de simulare a serviciilor componentelor ContractCML pe platforma XMF. Aceasta se bazează pe propunerea anterioară privind reprezentarea contractelor semantice şi permite raţionamente referitoare la interoperabilitatea semantică a componentelor.

Aceste propuneri au fost diseminate prin intermediul lucrărilor [76], respectiv [77, 78], fiind bazate şi pe cercetarea raportată în [82].

Ca şi direcţii viitoare de lucru menţionăm definitivarea cadrului imaginat, prin integrarea nivelelor contractuale rămase în metamodel, definirea unei sintaxe concrete vizuale pentru ContractCML şi dezvoltarea instrumentelor necesare. Acest cadru ar trebui să permită crearea unor arhitecturi de componente şi testarea interoperabilităţii acestora. În acest sens, intenţionăm să investigăm posibilitatea de a demonstra interoperabilitatea (sau absenţa ei) prin folosirea unei metode formale si a unui instrument asociat (B şi AtelierB, cel mai probabil).

Capitolul 7

Concluzii

Atingerea unui nivel înalt de reutilizare a artefactelor şi proceselor sale constituie o dovadă de maturizare a domeniului dezvoltării softului ca şi disciplină inginerească. Nici o tehnică bazată pe reutilizare nu-şi poate atinge însă obiectivele în absenţa unui cadru formal adecvat. Această teză reuneşte un număr de contribuţii în direcţia asigurării unei fundamentări formale a reutilizării softului.

Prima contribuţie raportată în cadrul tezei se situează în domeniul formalizării şabloanelor de proiectare. Propunerea noastră constă într-o formalizare completă a şablonului de proictare GoF State, folosind metoda formală B. Contribuţia acoperă atât definiţia B a şablonului, cât şi formalizarea procesului său de reutilizare. Corectitudinea propunerilor făcute a fost demonstrată formal, cu ajutorul instrumentului AtelierB.

Cea de-a doua contribuţie descrisă în teză se referă la o nouă abordare privind soluţionarea şabloanelor de constrângeri din modelarea orientată obiect. Soluţiile oferite (constând într-un număr de şabloane de specificare OCL ce oferă suport adecvat depanării modelelor) sunt în acord cu rolul aserţiunilor în contextul MDE, referitor la asigurarea corectitudinii modelelor. Abordarea a fost validată folosind instrumentul OCLE, ilustrându-se beneficiile ei atât în domeniul verificării compilabilităţii modelelor, cât şi în domeniu testării acestora.

Contribuţiile în direcţia asigurării unui cadru conceptual riguros verificării compilabilităţii modelelor au un caracter dual. În primul rând, am propus un set de principii privind specificarea semanticii statice a limbajelor de (meta)modelare. În al doilea rând, am făcut o serie de propuneri privind îmbunătăţirea semanticii statice a metamodelului UML şi a metametamodelelor MOF, Ecore şi XCore, în conformitate cu aceste principii. Toate regulile de bună formare şi operaţiile adiţionale definite au fost testate şi validate corespunzător folosind OCLE, EMF, respectiv XMF Mosaic.

În ultima parte a tezei am raportat două contribuţii referitoare la specificarea componentelor soft. Prima dintre acestea reprezintă o parte a unei abordări mai complexe de inginerie inversă privitoare la extragerea de abstractizări structurale şi comportamentale din codul Java al componentelor, în scopul asigurării trasabilităţii între specificarea şi implementarea sistemelor bazate pe componente. Cea de-a doua se referă la stabilirea bazelor unui cadru menit să asigure o specificare contractuală adecvată a componentelor soft, cu rol în verificarea interoperabilităţii acestora. În acest scop, am propus ContractCML, un limbaj de modelare a componentelor axat pe reprezentarea contractelor. În plus, am introdus o tehnică de simulare a execuţiei serviciilor componentelor ContractCML pe platforma XMF Mosaic.

Toate propunerile făcute au fost motivate corespunzător şi comparate cu abordări similare din literatură, în scopul certificării relevanţei acestora.

Bibliografie

[1] ECO-NET Project "Behavior Abstraction from Code: Filling the Gap between Component Specification and Implementation". http://www.lina.sciences.univ-nantes.fr/coloss/wiki/doku.php?id=econet:start.

[2] ECO-NET SVN Repository. svn://aiya.ms.mff.cuni.cz/econet.

[3] eXecutable Metamodeling Facility (XMF) Home Page, Ceteva Ltd. 2007. http://itcentre.tvu.ac.uk/~clark/xmf.html.

[4] Frame Based on the Extensive Use of Metamodeling for the Specification, Implementation and Validation of Languages and Applications (EMF_SIVLA). http://www.cs.ubbcluj.ro/~chiorean/CUEM_SIVLA.

[5] openArchitectureWare (oAW). http://www.openarchitectureware.org/.

[6] Abrial, J.-R. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[7] Ackermann, J. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. In *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*, T. Baar, Ed., Technical Report LGL-REPORT-2005-001. EPFL, 2005, pp. 15–29.

[8] Ackermann, J. Frequently Occurring Patterns in Behavioral Specification of Software Components. In *COEA* (2005), pp. 41–56.

[9] André, P., Ardourel, G., and Attiogbé, C. Defining Component Protocols with Service Composition: Illustration withe Kmelia Model. In *6th International Symposium on Software Composition, SC'07* (2007), vol. 4829 of *LNCS*, Springer.

[10] André, P., Chiorean, D., Plasil, F., and Royer, J.-C. Behavior Abstraction from Code: Filling the Gap between Component Specification and Implementation, 2006. ECO-NET 16293RG/2007 Project Proposal.

[11] André, P., and **Petraşcu, Vladiela**. ECONET Project - CCMM Specification v.1.1, 2008. http://www.cs.ubbcluj.ro/~vladi/ThesisReferences/ECONET/ECONET_CCMM_v1_1.pdf.

[12] Anquetil, N., Royer, J.-C., André, P., Ardourel, G., Hnetynka, P., Poch, T., Petraşcu, D., and **Petraşcu, Vladiela**. JavaCompExt: Extracting Architectural Elements from Java Source Code. In *Proceedings of 16th Working Conference on Reverse Engineering - WCRE'09* (2009), IEEE Computer Society, pp. 317–318. Tool Demo [DBLP, IEEE Xplore, IEEE CSDL].

[13] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., and Zettel, J. *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2001.

[14] Barros, T., Henrio, L., and Madelaine, E. Model-Checking Distributed Components: The Vercors Platform. In *Proceedings of Formal Aspects of Component Software (FACS'06)* (2006), ENTCS.

[15] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. Making Components Contract Aware. *Computer 32*, 7 (1999), 38–45.

[16] Bezivin, J. On the Unification Power of Models. *Software and System Modeling (SoSyM) 4*, 2 (2005), 171–188. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OnTheUnificationPowerOfModels.pdf.

[17] Bezivin, J. Introduction to Model Engineering, 2006. http://www.modelware-ist.org/index.php?option=com_remository&Itemid=74&func=fileinfo&id=72.

[18] Blazy, S., Gervais, F., and Laleau, R. Reuse of Specification Patterns with the B Method. In *ZB 2003: Formal specification and development in Z and B* (2003), D. Bert, J. Bowen, S. King, and M. Waldén, Eds., vol. 2651 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 40–57.

[19] Box, D. *Essential COM*. Addison-Wesley, 1998.

[20] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. The Fractal Component Model and Its Support in Java. *Software Practice and Experience 36*, 11-12 (2006).

[21] Bures, T., Hnetynka, P., and Plasil, F. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications* (2006), IEEE CS, pp. 40–48.

[22] Cechich, A., and Moore, R. A formal specification of GoF design patters. Technical Report 151, UNU/IIST, P.O. Box 3058, Macau, 1999.

[23] Cheesman, J., and Daniels, J. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[24] Chiorean, D., Corutiu, D., Bortes, M., and Chiorean, I. Good Practices for Creating Correct, Clear and Efficient OCL Specifications. In *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language (NWUML'2004)* (2004), K. Koskimies, L. Kuzniarz, J. Lilius, and I. Porres, Eds., no. 35 in TUCS General Publications, Turku Center for Computer Science (TUCS), Finland, pp. 127–142.

[25] Chiorean, D., and **Petraşcu, Vladiela**. Towards a Conceptual Framework Supporting Model Compilability. In *Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010) at ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems - MoDELS'10* (2010), vol. 36 of *Electronic Communications of the EASST*, European Association of Software Science and Technology (EASST). 14 pages, http://modeling-languages.com/events/OCLWorkshop2010/submissions/ocl10_submission_10.pdf [DBLP].

[26] Chiorean, D., **Petraşcu, Vladiela**, and Ober, I. Testing-Oriented Improvements of OCL Specification Patterns. In *Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics - AQTR* (2010), vol. II, IEEE Computer Society, pp. 143–148. [ISI Proc., IEEE Xplore, IEEE CSDL].

[27] Chiorean, D., **Petraşcu, Vladiela**, and Ober, I. MDE-Driven OCL Specification Patterns. *Control Engineering and Applied Informatics (CEAI) n/a* (n/a), n/a. [ISI Journal].

[28] Chiorean, D., **Petraşcu, Vladiela**, and Petraşcu, D. How My Favorite Tool Supporting OCL Must Look Like. In *Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS* (2008), vol. 15 of *Electronic Communications of the EASST*, European Association of Software Science and Technology (EASST). 17 pages, http://journal.ub.tu-berlin.de/index.php/eceasst/article/viewFile/180/177 [DBLP].

[29] Chouali, S., Heiser, M., and Souquières, J. Proving Component Interoperability with B Refinement. *Electronic Notes in Theoretical Computer Science 160* (2006), 157–172.

[30] Clark, T., Sammut, P., and Willans, J. *Applied Metamodeling: A Foundation for Language Driven Development (Second Edition)*. Ceteva, 2008. http://itcentre.tvu.ac.uk/~clark/docs/Applied%20Metamodelling%20%28Second%20Edition%29.pdf.

[31] ClearSy System Engineering. Atelier B. http://www.atelierb.eu/index-en.php.

[32] Crnkovic, I., and Larsson, M., Eds. *Building Reliable Component-Based Software Systems*. Artech House, Inc., 2002.

[33] Damus, C. W. Implementing Model Integrity in EMF with MDT OCL. Eclipse Corner Articles, Eclipse Foundation. http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html.

[34] DeMichiel, L., Yalcinalp, L., and Krishnan, S. Enterprise JavaBeans Specification Version 2.0, 2001.

[35] Eclipse Foundation. Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf.

[36] Eclipse Foundation. Graphical Modeling Project (GMP). http://www.eclipse.org/modeling/gmp/.

[37] Eclipse Foundation. Model Development Tools (MDT) OCL. http://www.eclipse.org/modeling/mdt/?project=ocl.

[38] Eden, A. H. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, Department of Computer Science, Tel Aviv University, 2000.

[39] Eden, A. H., Hirshfeld, Y., and Yehudai, A. LePUS - A Declarative Pattern Specification Language. Tech. rep. 326/98, Department of Computer Science, Tel Aviv University, 1998.

[40] Flores, A., Moore, R., and Reynoso, L. A Formal Model of Object-Oriented Design and GoF Design Patterns. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity* (2001), FME '01, Springer-Verlag, pp. 223–241.

[41] Fuentes, J. M., Quintana, V., Llorens, J., Génova, G., and Prieto-Díaz, R. Errors in the UML metamodel? *ACM SIGSOFT Software Engineering Notes 28*, 6 (2003), 3–3.

[42] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[43] Garcia, M. Rules for Type-checking of Parametric Polymorphism in EMF Generics. In *Software Engineering (Workshops)* (2007), W.-G. Bleek, H. Schwentner, and H. Züllighoven, Eds., vol. 106 of *Lecture Notes in Informatics (LNI)*, GI, pp. 261–270.

[44] Gervais, F. Réutilisation de composants de spécification en B. Master's thesis, DEA IIE(CNAM) - University of Évry-INT, Évry, France, 2002. http://cedric.cnam.fr/PUBLIS/RC394.ps.gz.

[45] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification (Third Edition)*. Addison-Wesley Longman, 2005.

[46] Groher, I., and Voelter, M. XWeave: Models and Aspects in Concert. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling* (2007), ACM Press, pp. 35–40.

[47] Krueger, C. W. Software Reuse. *ACM Computer Surveys 24*, 2 (1992), 131–183.

[48] Laboratorul de Cercetare în Informatică (LCI). Object Constraint Language Environment (OCLE). http://lci.cs.ubbcluj.ro/ocle/.

[49] Lau, K.-K., and Ornaghi, M. OOD frameworks in component-based software development in computational logic. In *Proceedings of LOPSTR98* (1999), vol. 1559 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 101–123.

[50] Marcano, R., Meyer, E., Levy, N., and Souquières, J. Utilisation de patterns dans la construction de spécifications en UML et B. In *Proceedings of AFADL 2000: Approches formelles dans l'assistance au développement de logiciels* (2000). Tech. rep. A00-R-009, LSR Laboratory, Grenoble, France, 15 pages.

[51] Marcano-Kamenoff, R., Lévy, N., and Losavio, F. Spécification et spécialisation de patterns en UML et B. In *Proceedings of LMO'2000: Langages et modèles à objets* (2000), C. Dony and H. A. Sahraoui, Eds., Hermès Science Publications, Mont Saint-Hilaire, Québec, Canada, pp. 245–260.

[52] Merks, E., and Paternostro, M. Modeling Generics with Ecore. In *EclipseCon 2007* (2007). http://www.eclipsecon.org/2007/index.php?page=sub/&id=3845.

[53] Meyer, B. Applying "Design by Contract". *Computer 25*, 10 (1992), 40–51.

[54] Newcomer, E. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley, 2002.

[55] Object Management Group (OMG). CORBA Component Model, V3.0, 2002. http://www.omg.org/technology/documents/formal/components.htm.

[56] Object Management Group (OMG). Model Driven Architecture (MDA) Guide, Version 1.0.1, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf.

[57] Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 1.5, 2003. http://www.omg.org/spec/UML/1.5/PDF/.

[58] Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 1.4.2, 2005. http://www.omg.org/spec/UML/ISO/19501/PDF/.

[59] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification, Version 2.0, 2006. http://www.omg.org/spec/MOF/2.0/PDF.

[60] Object Management Group (OMG). Object Constraint Language (OCL), Version 2.2, 2010. http://www.omg.org/spec/OCL/2.2/PDF/.

[61] Object Management Group (OMG). Unified Modeling Language (UML), Infrastructure, Version 2.3, 2010. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/.

[62] Object Management Group (OMG). Unified Modeling Language (UML), Superstructure, Version 2.3, 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[63] Parizek, P., and Plasil, F. Modeling Environment for Component Model Checking from Hierarchical Architecture. In *Proceedings of Formal Aspects of Component Software (FACS'06)* (2006), ENTCS.

[64] Pavel, S., Noyé, J., Poizat, P., and Royer, J.-C. A Java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)* (2005), Springer-Verlag, Ed., vol. 3628 of *Lecture Notes in Computer Science*, pp. 115–125.

[65] Plasil, F., and Visnovsky, S. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering 28*, 11 (2002), 1056–1076.

[66] Pospisil, R., and Plasil, F. Describing the Functionality of EJB using the Behavior Protocols. In *In Week of Doctoral Students (WDS 99)* (1999).

[67] Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., Eds. *The Common Component Modeling Example: Comparing Software Component Models*, vol. 5153 of *LNCS*. Springer, 2008.

[68] Richters, M., and Gogolla, M. Validating UML models and OCL constraints. In *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference Proceedings* (2000), A. Evans, S. Kent, and B. Selic, Eds., vol. 1939 of *Lecture Notes in Computer Science*, Springer, pp. 265–277.

[69] Schmidt, D. C. Model-Driven Engineering. *Computer 39*, 2 (2006), 25–31.

[70] Snook, C., and Butler, M. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM) 15*, 1 (2006), 92–122.

[71] Sun Microsystems. JavaBeans Specification, 1997. http://java.sun.com/products/javabeans/docs/spec.html.

[72] Szyperski, C., Gruntz, D., and Murer, S. *Component Software: Beyond Object-Oriented Programming (Second Edition)*. Addison-Wesley, 2002.

[73] **Ciobotariu-Boer, Vladiela**, and Petraşcu, D. X-Machines Modeling. A Case Study. In *Proceedings of the Symposium "Colocviul Academic Clujean de Informatică"* (2005), M. Frenţiu, Ed., Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania, pp. 75–80.

[74] **Petraşcu, Vladiela**, and Chiorean, D. Towards Improving the Static Semantics of XCore. *Studia Informatica LV*, 3 (2010), 61–70. http://www.cs.ubbcluj.ro/~studia-i/2010-3/06-PetrascuChiorean.pdf [MathSciNet, Zentralblatt].

[75] **Petraşcu, Vladiela**, and Chiorean, D. XCore Static Semantics - from Requirements to Implementation. In *Proceedings of the National Symposium Zilele Academice Clujene (ZAC)* (2010), M. Frenţiu, Ed., Presa Universitară Clujeană, pp. 73–78.

[76] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. ContractCML - a Contract Aware Component Modeling Language. In *Proceedings of 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (2008), IEEE Computer Society, pp. 273–276. [ISI Proc., DBLP, IEEE CSDL, ACM DL].

[77] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. Component Models' Simulation in ContractCML. In *Proceedings of KEPT 2009: Knowledge Engineering Principles and Techniques - Extended Abstracts* (2009), M. Frenţiu and H. F. Pop, Eds., vol. II of *Studia Informatica, Special Issue*, Babeş-Bolyai University of Cluj-Napoca, pp. 198–201. http://cs.ubbcluj.ro/~studia-i/2009-kept/Studia-2009-Kept-3-KSE.pdf [MathSciNet, Zentralblatt].

[78] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. Component Models' Simulation in ContractCML. In *KEPT 2009: Knowledge Engineering Principles and Techniques - Selected Papers* (2009), M. Frenţiu and H. F. Pop, Eds., Presa Universitară Clujeană, pp. 231–238. (extended version of [77]) [ISI Proc.].

[79] **Petraşcu, Vladiela**, Chiorean, D., and Petraşcu, D. Proposal of a Set of OCL WFRs for the Ecore Meta-Metamodel. *Studia Informatica LIV*, 2 (2009), 89–108. http://www.cs.ubbcluj.ro/~studia-i/2009-2/09-PetrascuChiorean.pdf [MathSciNet, Zentralblatt].

[80] **Petraşcu, Vladiela**, and Petraşcu, D. Formalizing the State Pattern in B. *Pure Mathematics and Applications (PU.M.A) 17*, 3-4 (2006), 397–411. http://www.bke.hu/puma/17_3/PetrascuPetrascu.pdf [MathSciNet].

[81] **Petraşcu, Vladiela**, and Petraşcu, D. Proving the Soundness of an OO Model using the B Method. In *Proceedings of the Symposium "Zilele Academice Clujene"* (2006), Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania, pp. 107–112.

[82] **Petraşcu, Vladiela**, and Petraşcu, D. Architecting and Specifying a Software Component Using UML. In *Proceedings of KEPT 2007 - Knowledge Engineering:*

Principles and Technologies (2007), M. Frenţiu and H. F. Pop, Eds., vol. I of *Studia Informatica, Special Issue*, Babeş-Bolyai University of Cluj-Napoca, pp. 332–340. http://cs.ubbcluj.ro/~studia-i/2007-kept/415-Petrascu.pdf [MathSciNet, Zentralblatt].

[83] Wahler, M. *Using Patterns to Develop Consistent Design Constraints*. PhD thesis, ETH Zurich, Switzerland, 2008. http://e-collection.ethbib.ethz.ch/eserv/eth:30499/eth-30499-02.pdf.

[84] Wahler, M., Basin, D., Brucker, A. D., and Koehler, J. Efficient Analysis of Pattern-Based Constraint Specifications. *Software and Systems Modeling 9*, 2 (2010), 225–255.

[85] Wahler, M., Koehler, J., and Brucker, A. D. Model-Driven Constraint Engineering. *Electronic Communications of the EASST 5* (2006).

[86] Warmer, J., and Kleppe, A. *Object Constraint Language: Precise Modeling with UML*, first ed. Addison-Wesley, 1999.

[87] Wigley, A., Sutton, M., MacLeod, R., Burbidge, R., and Wheelwright, S. *Microsoft .NET Compact Framework (Core Reference)*. Microsoft Press, 2003.

Abrevieri

AMN	**A**bstract **M**achine **N**otation
AO(s)	**A**dditional **O**peration(s)
BCR	**B**usiness **C**onstraint **R**ule
CBSD	**C**omponent **B**ased **S**oftware **D**evelopment
CBSE	**C**omponent **B**ased **S**oftware **E**ngineering
CCMM	**C**ommon **C**omponent **M**eta**M**odel
CIM	**C**omputation **I**ndependent **M**odel
CMOF	**C**omplete **MOF**
DBC	**D**esign **b**y **C**ontract
DSML	**D**omain **S**pecific **M**odeling **L**anguage
EMF	**E**clipse **M**odeling **F**ramework
EMOF	**E**ssential **MOF**
GMF	**G**raphical **M**odeling **F**ramework
GoF	**G**ang **o**f **F**our
GSL	**G**eneralized **S**ubstitution **L**anguage
HOL	**H**igher **O**rder **L**ogic
HOML	**H**igher **O**rder **M**onadic **L**ogic
LDD	**L**anguage **D**riven **D**evelopment
MDA	**M**odel **D**riven **A**rchitecture
MDD	**M**odel **D**riven **D**evelopment
MDE	**M**odel **D**riven **E**ngineering
MOF	**M**eta **O**bject **F**acility
oAW	**o**pen**A**rchitecture**W**are
OCL	**O**bject **C**onstraint **L**anguage
OCLE	**OCL E**nvironment
OMG	**O**bject **M**anagement **G**roup
OMT	**O**bject **M**odeling **T**echnique
PIM	**P**latform **I**ndependent **M**odel
PO(s)	**P**roof **O**bligation(s)
PSM	**P**latform **S**pecific **M**odel
UML	**U**nified **M**odeling **L**anguage
WFR(s)	**W**ell **F**ormedness **R**ule(s)
XMF	E**X**ecutable **M**etamodeling **F**acility