

Univerzitet u Beogradu
Elektrotehnički fakultet

Dragan Milićev

**Automatska transformacija modela
u softverskim alatima za modelovanje**

Doktorska disertacija

Beograd, 2001.

Zahvalnice

Zahvaljujem članovima Komisije za pregled i ocenu ove disertacije, prof. dr Dušanu Velaševiću, doc. dr Igoru Tartalji, prof. dr Miroslavu Bojoviću i prof. dr Sanji Vraneš, na korisnim sugestijama koje su doprinele kvalitetu ovog rada. Posebnu zahvalnost dugujem mentoru, prof. dr Zoranu Jovanoviću na izuzetnoj podršci tokom izrade ove teze kao i tokom celog mog stručnog rada.

Ovaj rad ne bi bio uspešno izveden do kraja da nije bilo dragocene pomoći saradnika koji su učestvovali na izradi različitih verzija prototipskih alata za metamodelovanje koji podržava predloženu metodu. Mr Dejan Marjanović je u okviru svoje magistarske teze konstruisao prvi prototipski alat za metamodelovanje koji je kasnije poslužio kao odlična polazna osnova za dalja istraživanja. Kolege Pavle Nikolić, Milorad Ljeskovac i Miloš Aćimović su, u okviru svojih diplomskih radova, učestvovali na izradi korisničkog interfejsa za ovaj alat. Kolege Ljubica Lazarević, Marko Zarić i Ivan Đorđević su, u okviru svojih diplomskih radova, prvi posle autora uspešno primenili predloženu metodu na nekoliko praktičnih i složenih primera (konačni automati, ROOM i logičko projektovanje hardvera, respektivno). Svima njima najsrdačnije zahvaljujem na saradnji.

Ideje izložene u ovom radu potekle su iz projekata u praksi. Početna istraživanja nastala su u okviru projekta softvera telefonske centrale u preduzećima Iritel i Pupin Telecom DKTS. Kolegama iz ovih preduzeća dugujem zahvalnost na saradnji. Dalja istraživanja u oblasti modelovanja u specifičnom domenu i metamodelovanja vršena su u okviru projekta informaciono-upravljačkog sistema NIS Rafinerije nafte Novi Sad. Kolegama iz Rafinerije, a posebno mr Vladi Kruniću, koji je u okviru svog magistarskog rada realizovao pomenuti sistem, najiskrenije zahvaljujem na izuzetno prijatnoj saradnji. Metoda za modelovanje Web aplikacija, u okviru koje je takođe primenjeno predloženo rešenje, razvijena je u okviru projekta Socratenon koji je rađen u saradnji sa Univerzitetom u Salernu, Italija, a pod rukovodstvom prof. dr Veljka Milutinovića. Zahvaljujem kolegama mr Dejanu Marjanoviću, Nenadu Nikoliću, Milanu Milićeviću, Milanu Trajkoviću i Nebojši Piroćancu koji su učestvovali na tom projektu. Istraživanja u oblasti metamodelovanja vršena su takođe i u okviru razvoja sistema Midis, u saradnji sa preduzećem Rail Consult, Nemačka. Svim kolegama koji su učestvovali na tom projektu takođe zahvaljujem na pomoći.

Veoma korisne komentare na rane verzije teksta ove teze i radova koji su iz nje nastali dali su Brane Selić iz korporacije Rational, dr Igor Tartalja, mr Dragan Bojić, dr Juha-Pekka Tolvanen iz kompanije MetaCASE Consult, Finska, kao i anonimni recenzenti, na čemu im dugujem zahvalnost.

Zahvaljujem bratu, roditeljima, tašti i tastu, rođacima, prijateljima i kolegama koji su me podrškom i čestim pitanjima o toku izrade ovog rada podsticali da ga završim.

Na kraju, teško mi je da pronađem reči kojima bih izrazio zahvalnost na neizmernoj podršci i beskonačnom strpljenju koje je ispoljila moja supruga Snežana tokom svih godina mog rada na ovoj temi. Zbog toga njoj i posvećujem ovaj rad.

U Beogradu, marta 2001.

Dragan Milićev

Rezime

Jedna od najvažnijih funkcija softverskih alata za modelovanje jeste generisanje različitih izlaznih formi. Izlazne forme mogu biti dokumentacija, izvorni kôd na nekom programskom jeziku, specifikacija hardverske mreže, ili bilo koji model sistema koji se konstruiše različit od onoga koga je formirao korisnik. Proces generisanja izlazne forme može se posmatrati kao automatsko stvaranje modela u ciljnom domenu, polazeći od modela u izvornom domenu (ili kao transformacija izvornog u odredišni model). Proces specifikacije ove transformacije i njena implementacija mogu da budu izuzetno složeni. U radu se najpre analiziraju uzroci ove složenosti i postojeće metode definisanja transformacija modela. Predlaže se klasifikacija tih metoda i uočavaju njihove prednosti i nedostaci, a zatim predlaže i novo rešenje.

Rešenje polazi od zapažanja da se potrebna transformacija ne mora izvesti u jednom koraku, već da se može formirati više međumodela. Ti međumodeli mogu da predstavljaju različite poglede na sistem koji se konstruiše i koji ga preciznije opisuju, ili koji spuštaju nivo apstrakcije korisnički definisanog modela, kako bi se sukcesivne transformacije lakše definisale, i kako bi se do ciljnog modela došlo postupno. Ukoliko se metamodeli definišu pomoću objektno orijentisanih strukturnih koncepata, modeli predstavljaju skupove instanci apstrakcija iz domena i veza između njih.

U radu se predlaže nova tehnika vizuelne specifikacije transformacije modela. Tehnika koristi proširene UML objektno dijagrame koji prikazuju instance apstrakcija iz ciljnog domena i veze između njih koje treba automatski generisati. Standardni UML objektni dijagrami prošireni su konceptima uslovnog, repetitivnog, parametrizovanog, polimorfnog i rekurzivnog kreiranja. Ovi koncepti mogu se implementirati standardnim mehanizmima proširivosti jezika UML.

Pored toga, u radu se predlaže generalizacija celog pristupa prema kojoj se transformacije modela mogu vršiti lančano, uz korišćenje različitih raspoloživih gotovih i ponovno upotrebljivih metamodela domena i njihovih preslikavanja. Ovo zapravo predstavlja novi pristup visoko apstraktnom programiranju uz automatsko generisanje aplikacija. Visoka apstraktnost se ogleda u tome što se kao polazni modeli, koje definiše korisnik, mogu koristiti modeli iz specifičnih domena čiji su metamodeli prilagođeni datoj oblasti primene. Konkretno implementacije dobijaju se izborom željenih transformacija, u zavisnosti od ciljnog okruženja za izvršavanje aplikacije.

U radu je dat niz primera upotrebe predložene metode iz konkretnih realizovanih projekata u praksi. Analiza ovih primera upotrebe ukazuje na potencijale predložene metode i na njene pogodnosti: specifikacije transformacija su jasne i pregledne, jednostavne za modifikaciju i proširivanje, i obezbeđuju značajno kraće vreme konstrukcije sistema.

Ključne reči

Objektno orijentisano modelovanje (engl. *object-oriented modeling*), Unified Modeling Language (UML), objektni dijagram (engl. *object diagram*), metamodelovanje (engl. *metamodeling*), modelovanje u specifičnom domenu (engl. *domain-specific modeling*), meta CASE alat (engl. *meta CASE tool*), automatsko generisanje koda (engl. *automatic code generation*), transformacija modela (engl. *model transformation*)

Kratak sadržaj

I UVOD	1
II DEFINICIJA PROBLEMA	7
III PREGLED POSTOJEĆIH REŠENJA	25
IV SUŠTINA PREDLOŽENOG REŠENJA	53
V DETALJI PREDLOŽENOG REŠENJA	79
VI PRIMERI UPOTREBE	107
VII ANALIZA PREDLOŽENOG REŠENJA	133
VIII ZAKLJUČAK	143
IX LITERATURA	149
X PRILOZI	165

Detaljan sadržaj

Zahvalnice	i
Rezime	iii
Ključne reči	iii
Kratak sadržaj	v
Detaljan sadržaj	vii
 I UVOD	 1
Predmet i cilj rada	3
Struktura i sadržaj rada	4
 II DEFINICIJA PROBLEMA	 7
Problem generisanja izlaznih formi i motivacija	9
Definicije pojmova i pretpostavke	13
Polazni pojmovi: sistem, apstrakcija, relacija i jezik	13
Sistem koji se konstruiše	13
Apstrakcija, instanca, svojstvo i relacija	13
Jezik i specifikacija jezika	14
Implicitna semantika i konsekvence	15
Izvedeni pojmovi: model, domen i metamodel	15
Model i modelovanje	15
Domen modelovanja	16
Metamodel i metamodelovanje	17
Implicitna semantika i konsekvence	17
Pojmovi vezani za transformacije modela	18
Transformacija modela	18
Preslikavanje domena	18
Četvoroslojna arhitektura metamodelovanja	19
Pretpostavke i ograničenja	20
Leksičko-sekvencijalni domeni	21
Gramatičko-hijerarhijski domeni	22
Objektno-grafovski domeni	22
 III PREGLED POSTOJEĆIH REŠENJA	 25
Softverski alati za metamodelovanje	27
Motivi za konstruisanje alata za metamodelovanje	27
Bitni aspekti alata za metamodelovanje	28
Meta-metamodeli u alatima za metamodelovanje	29
Generisanje izlaznih formi u alatima za modelovanje	33
Generisanje izlaznih formi programski (pomoću skriptova)	34
Generisanje izlaznih formi pomoću šablona	35
Problem obilaska strukture i projektni obrazac <i>Visitor</i>	37

Generisanje izlaznih formi pomoću apstraktnih specifikacija	39
Ad-hoc pristupi	40
Ostali relevantni pristupi	41
Transformatori bazirani na gramatikama	41
Tradicionalni programski prevodioci	41
Transformatori zasnovani na pravilima	42
Intencionalno programiranje	44
Vizuelni jezici	45
UML kao vizuelni jezik	46
Analiza i klasifikacija postojećih rešenja	47
Tipovi domena prema strukturi modela	47
Načini obilaska izvornih struktura	48
Načini specifikacije transformacija i njihova organizacija	50
IV SUŠTINA PREDLOŽENOG REŠENJA	53
Geneza i osnovni elementi ideje	55
Preslikavanje udaljenih domena i uvođenje međudomena	55
Ideja i namena metode preslikavanja domena	57
Preslikavanje domena	61
Instance, atributi i veze	61
Repeticije	62
Uslovi	64
Sekvence	65
Parametrizovane podstrukture i rekurzija	65
Polimorfizam	68
Organizacija specifikacija	70
Korišćenje specifične notacije domena	71
Sukcesivne transformacije modela	73
Lančane transformacije modela	73
Višestruka upotreba preslikavanja	75
Manuelne modifikacije modela	76
V DETALJI PREDLOŽENOG REŠENJA	79
Semantika preslikavanja domena	81
Osnovne definicije	82
Sekvencijalne zavisnosti	83
Pravila konzistentnosti preslikavanja domena	83
Semantika generisanja modela	84
Paket bez stereotipa	84
Paket <<ForEach>>	84
Paketi <<Substruct>> i <<Ref>>	84
Instance	85
Veza	85
Opis implementacije	86
Metamodel preslikavanja domena	86
Generisanje transformatora	87
Paket bez stereotipa	88
Paket <<ForEach>>	88
Instanca bez stereotipa	89
Veza	89
Organizacija koda transformatora	91

Paket <<Substruct>> bez vrednosti ElemType	93
Paket <<Ref>> bez vrednosti Elem	95
Paketi <<Substruct>> i <<Ref>> vezani za tip	97
Konstrukcija alata za metamodelovanje	100
Arhitektura alata	101
Postupak konstrukcije alata	103
Glavne mogućnosti i pogodnosti alata	104
VI PRIMERI UPOTREBE	107
Konstrukcija alata za metamodelovanje	109
Metamodel domena OOPL	109
Preslikavanje iz UMLCore u OOPL	111
Preslikavanje iz DomainMapping u OOPL	112
Transformacija objektnog u relacioni model	113
Metoda ROOM	115
Osnovni koncepti domena ROOM	115
Generatori koda i preslikavanja	117
Logičko projektovanje hardvera	119
Metamodel domena logičkog projektovanja hardvera	119
Preslikavanje u domen OOPL	121
Modelovanje Web aplikacija	122
Predložena tehnika modelovanja Web aplikacija	122
Ideja predloženog rešenja	123
Tehnika modelovanja	126
Proces razvoja aplikacije	127
Implementacija	128
Metamodeli i preslikavanja	128
Generisanje strukturnih slučajeva upotrebe	131
VII ANALIZA PREDLOŽENOG REŠENJA	133
Zaključci iz primera primene i predlog metodologije	135
Zaključci iz primera primene	135
Predlog metodologije	139
Odnos predloženog i postojećih rešenja	140
VIII ZAKLJUČAK	143
Osnovni doprinosi rada	145
Potencijali korišćenja rezultata rada	146
Pravci daljeg razvoja	147
IX LITERATURA	149
Spisak referenci	151
Klasifikovana bibliografija	157
A Softversko inženjerstvo i razno	157
B Objektno orijentisano modelovanje i UML	157

C	Metamodelovanje	158
D	Podesivi CASE i metaCASE alati	159
E	Vizuelni jezici	160
F	Transformacije modela	160
G	Projektovanje Web aplikacija	161
H	Alati za projektovanje Web aplikacija	161
I	Sistemi za rad u realnom vremenu	162
J	Modelovanje industrijskih procesa	162
K	Radovi autora vezani za temu teze	163
L	Radovi saradnika vezani za temu teze	163
X PRILOZI		165
Prilog A Algoritmi za generisanje koda transformatora		167
Prilog B Specifikacije preslikavanja u alatu za metamodelovanje		170
Prilog C Specifikacije preslikavanja za primer metode ROOM		174
Prilog D Specifikacije preslikavanja za primer logičkog projektovanja hardvera		177

I Uvod

Predmet i cilj rada

Modelovanje predstavlja centralnu aktivnost koja vodi izradi dobrog softvera, kao i bilo kog drugog inženjerskog sistema. Za mnoge inženjerske domene postoje softverski alati koji podržavaju proces modelovanja. Takav alat predstavlja okruženje za primenu metode i notacije koje se koriste u datom domenu i pruža podršku za automatsku proveru konzistentnosti modela, bolju preglednost i navigaciju kroz model, kao i vizuelizaciju i dokumentaciju sistema koji se konstruiše. Ovakva podrška značajno smanjuje broj grešaka i celokupno vreme projektovanja.

Svaki specifični domen modelovanja počiva na opštijem konceptualnom modelu koji definiše sledeće elemente: ključne apstrakcije domena, njihova svojstva, relacije, semantiku, ponašanje u specifičnom domenu, kao i vizuelnu pojavu (notaciju) i ponašanje u softverskom alatu koji podržava modelovanje u tom specifičnom domenu modelovanja. Ovaj drugi, opštiji konceptualni model naziva se *metamodelom* posmatranog domena, a proces njegove konstrukcije - *metamodelovanjem*. Za model koji počiva na metamodelu nekog domena kaže se da pripada datom domenu. Ovaj rad bavi se metamodelovanjem specifičnih domena softverskog i drugog inženjerstva, kao i problemima konstrukcije softverskih alata koji podržavaju pravljenje modela koji pripadaju specifičnim domenima.

Osim svojih važnih uloga u procesu specifikacije, dokumentovanja i vizuelizacije sistema, alati za modelovanje imaju primarnu ulogu u procesu *konstruisanja* inženjerskih sistema. U kontekstu softverskih alata za modelovanje, konstruisanje predstavlja transformaciju izvornog modela koji pripada specifičnom domenu u neku izlaznu formu. Ta forma se može interpretirati od strane nekog eksternog subjekta (izvršnog okruženja, programskog prevodioca, programa, korisnika itd.) da bi se dobilo željeno ponašanje sistema. Neki primeri izlaznih formi su dokumentacija, izvorni kôd na nekom ciljnom programskom jeziku, šema baze podataka, šema hardvera, ili bilo koja druga formalno definisana struktura sa preciznom semantikom u nekom ciljnom domenu. Ovaj rad posmatra izlaznu formu kao model koji pripada specifičnom ciljnom domenu koji je različit od izvornog domena modelovanja (zasnovan je na različitom metamodelu), a produkciju izlazne forme kao transformaciju modela koji pripada izvornom domenu u model koji pripada ciljnom domenu.

Širi predmet ovog rada su opšti problemi metamodelovanja i konstrukcije softverskih alata za modelovanje u specifičnim domenima (pretežno, ali ne isključivo softverskim). Uži predmet ovog rada su načini definisanja transformacije modela (odnosno generisanja izlaznih formi) u softverskim okruženjima za modelovanje, kao i konstrukcija alata za automatsko sprovođenje opisane transformacije.

Postojeće metode definisanja načina generisanja izlaznih formi u alatima za metamodelovanje i modelovanje poseduju izvesne slabosti i ograničenja. Cilj ovog rada je da najpre izvrši analizu i klasifikaciju postojećih rešenja, sa naglaskom na njihove pogodnosti i nedostatke, a zatim predloži i definiše metodu za formalnu specifikaciju načina generisanja izlaznih formi. U radu će zatim ta metoda biti generalizovana u opštu metodu za transformaciju modela između različitih domena. U okviru rada će biti opisan realizovani prototipski alat koji podržava predloženu metodu. Najzad, primena i prednosti predložene metode biće demonstrirane na nizu konkretnih praktičnih problema i projekata iz prakse.

Struktura i sadržaj rada

Rad je podeljen na deset glava. Nakon ove uvodne glave, sledi Definicija problema. U toj glavi je na jednom konkretnom primeru opisan kontekst ovog rada – problem specifikacije načina generisanja izlaznih formi u alatima za modelovanje. Na tom primeru su samo okvirno i intuitivno navedeni problemi koji se ovde pojavljuju i mane rešenja koje se najčešće sreće u postojećim alatima za modelovanje. U nastavku su date precizne definicije pojmova i pretpostavke koje su bitne za ostatak ovog rada, sa značenjima koja su donekle prilagođena ovom specifičnom kontekstu.

U trećoj glavi ukratko su opisana neka najznačajnija postojeća rešenja datog problema koja su bila dostupna iz otvorene literature ili se koriste u raspoloživim komercijalnim i akademskim alatima za modelovanje. Takođe su opisane i komentarisane oblasti koje su u bližoj vezi sa temom ovog rada: transformatori bazirani na gramatikama, intencionalno programiranje i vizuelni jezici. Na kraju je data analiza prednosti i nedostataka navedenih rešenja i izvršena njihova klasifikacija u zavisnosti od nekoliko uočenih bitnih parametara.

Četvrta glava ukratko opisuje osnovne ideje predloženog originalnog rešenja. Najpre je prikazana geneza ideje: zapažanje da se potrebna transformacija ne mora izvesti u jednom koraku, već da proces specifikacije transformacije postaje jednostavniji ukoliko se uvedu međudomeni. Zatim su opisani osnovni koncepti predložene metode za vizuelnu specifikaciju transformacija pomoću proširenih UML objektnih dijagrama koji prikazuju instance apstrakcija ciljnog domena, kao i veze između njih, koje treba automatski kreirati polazeći od izvornog modela. Najzad, ukratko je opisana generalizacija predložene metode koja se odnosi na povezivanje transformacija u lanac, čime se dobija mogućnost apstraktnog modelovanja uz fleksibilno generisanje ciljnog modela.

Peta glava iznosi detalje predloženog rešenja. U njoj je formalno i kompletno opisana semantika svih predloženih koncepata koji se mogu koristiti u specifikacijama transformacija: instance, atributi, veze, uslovi, repeticije, sekvence, podstrukture i polimorfizam. Takođe je ukratko opisana implementacija prototipskog alata koji podržava predloženu metodu.

Šesta glava sadrži opise nekoliko primera upotrebe predložene metode. Većina ovih primera potiče iz konkretnih realizovanih projekata u praksi. Navedeni su, sa više ili manje detalja, primeri samog alata za metamodelovanje, transformacije objektnog u relacioni model, generisanja Web aplikacija, postojeće metode ROOM i logičkog projektovanja hardvera. Na ovaj način je primena predložene metode potvrđena na nekoliko veoma različitih praktičnih domena.

Sedma glava sadrži analizu predložene metode. Analizirani su najpre rezultati primera upotrebe i predložen postupak sprovođenja metode. Pokazano je mesto predložene metode u klasifikacijama metoda za specifikaciju transformacija modela koje su predložene u drugoj glavi. Komentarisane su prednosti i uočena ograničenja predložene metode.

Osma glava donosi zaključak. Rekapitulirani su osnovni doprinosi rada i ukazano na pravce mogućeg daljeg razvoja.

Deveta glava sadrži dva različito uređena spiska korišćene literature. Prvi spisak sadrži sve reference poredane po abecednom redu oznaka. Drugi spisak je klasifikovana bibliografija: reference su grupisane po tematici. Svaka grupa označena je jednim slovom abecede. U tekstu rada literatura se referiše, radi lakšeg pronalaženja u oba spiska, navođenjem oznake grupe u klasifikovanoj bibliografiji i oznake reference. Na primer,

referenca [K-Mil00a] se može naći kao [Mil00a] u prvom spisku literature po abecednom redu, kao i u K grupi klasifikovane bibliografije.

U prilogima su dati algoritmi generisanja koda koji izvršava transformacije definisane pomoću predložene tehnike, kao i detaljnije specifikacije preslikavanja za nekoliko odabranih primera.

II Definicija problema

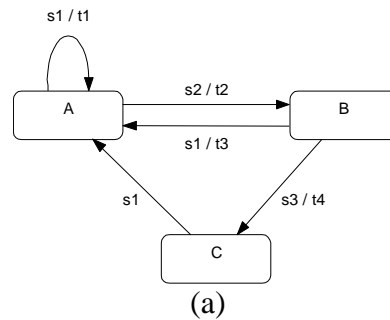
Problem generisanja izlaznih formi i motivacija

Problem, kao i ideja predloženog rešenja, biće demonstrirani pomoću jednog jednostavnog primera modelovanja u oblasti telekomunikacionog softvera. Posmatra se jedno jednostavno okruženje za modelovanje ponašanja sistema pomoću konačnih automata (engl. *state machines*) [A,I-Har87][I-Sel94][B-Boo99]. To okruženje treba da omogući korisniku da definiše konačne automate, poput primera na slici 2.1a. U cilju konstrukcije programskog izvršnog sistema čije je ponašanje opisano pomoću konačnih automata, dato okruženje treba da generiše kod na nekom programskom jeziku polazeći od definisanog modela. Ovde će u te svrhe kao primer biti korišćen jezik C++ [B-Mil95].

Postoji mnogo načina da se definiše semantika izvršavanja sistema modelovanih pomoću konačnih automata [A,E,I-Har87][B,I-Har96a][I-Har96b][I-Sel94][B-Boo99][K-Mil96][K-Laz99a][H-Laz99b]. Izbor semantike zavisi od više parametara, kao što su: da li je potrebno konkurentno i distribuirano izvršavanje, da li je potrebno ugnežđivanje stanja i pamćenje istorije stanja, da li je slanje događaja automatu sinhrono ili asinhrono, itd. Implementacioni parametri ciljne platforme, kao što su distribuiranost sistema, komunikacioni mehanizmi, operativni sistem, programski jezik, optimizacija performansi, itd., takođe utiču na izbor načina generisanja koda. Dakle, u zavisnosti od izabrane semantike kao i od implementacionih parametara potrebno je definisati i način generisanja koda.

Prema tome, način generisanja izlazne forme (ili kratko "izlaza"), u ovom slučaju izvornog koda, u alatu za modelovanje može zavistiti od više faktora. Dobar alat bi trebalo da u ovom pogledu obezbedi maksimalnu fleksibilnost: korisniku bi trebalo omogućiti ili da iz većeg skupa načina izabere željeni, ili da sam definiše način na koji će se generisati izlaz. U svakom slučaju, postoji problem raspoloživosti jednostavne i efikasne metode za specifikaciju i implementaciju mehanizma generisanja izlaza u alatu za modelovanje.

Primer koji demonstrira problem prikazan je na slici 2.1. Radi jednostavnosti objašnjenja, pretpostavlja se da je izabran način generisanja koda kod koga se konačni automati implementiraju pomoću projektnog obrasca (engl. *design pattern*) *State* [B-Gam95], kao što je prikazano na slici 2.1b. Za primer automata sa slike 2.1a i za dati projektni obrazac *State* potrebno je generisati nekoliko klasa u izlaznom C++ kodu. Prva klasa nazvana je ovde *FSM* i predstavlja klasu čije instance imaju ponašanje definisano konačnim automatom sa slike 2.1a. Ona sadrži operacije koje odgovaraju događajima na koje automat reaguje (operacije s_1 do s_3), kao i operacije koje realizuju akcije za tranzicije (t_1 do t_3). Druga klasa je nazvana *FSMState*. Ona sadrži po jednu polimorfnu operaciju za svaki događaj. Dalje, potrebno je generisati po jednu klasu izvedenu iz klase *FSMState* za svako stanje automata. Svaka takva klasa redefiniše operacije koje odgovaraju događajima na koje dato stanje reaguje. Ove operacije izvršavaju odgovarajuće prelazne akcije i vraćaju pokazivač na određeno stanje automata. Klasa *FSM* sadrži po jedan podatak član za svako stanje automata, kao i pokazivač na *FSMState* koji ukazuje na tekuće stanje u kome se automat nalazi. Sve operacije s_1 do s_3 klase *FSMState* implementirane su na isti način: one pozivaju odgovarajuću operaciju tekućeg stanja koja, posredstvom virtuelnog mehanizma, izvršava odgovarajuću tranziciju i vraća novo tekuće stanje. Na ovaj način se dobija efekat da objekat klase *FSM* reaguje na događaje u zavisnosti od svog tekućeg stanja [B-Gam95].



```

class FSM {
public:
    FSM ();

    void s1 ();
    void s2 ();
    void s3 ();

protected:
    friend class FSMStateA;
    friend class FSMStateB;
    friend class FSMStateC;

    void t1() {...}
    void t2() {...}
    void t3() {...}

    FSMStateA stateA;
    FSMStateB stateB;
    FSMStateC stateC;

    FSMState* curSt;
};

FSM::FSM () :
    stateA(this), stateB(this), stateC(this),
    curSt(&stateA) { curSt->entry(); }

void FSM::s1 () {
    curSt->exit();
    curSt = curSt->s1();
    curSt->entry();
}

...

```

```

class FSMState {
public:
    FSMState (FSM* fsm) : myFSM(fsm) {}

    virtual FSMState* s1 () {return this;}
    virtual FSMState* s2 () {return this;}
    virtual FSMState* s3 () {return this;}

    virtual void entry () {}
    virtual void exit  () {}

protected:
    FSM* fsm () const { return myFSM; }
private:
    FSM* myFSM;
};

class FSMStateA : public FSMState {
public:

    FSMStateA (FSM* fsm) : FSMState(fsm) {}

    virtual FSMState* s1 ();
    virtual FSMState* s2 ();

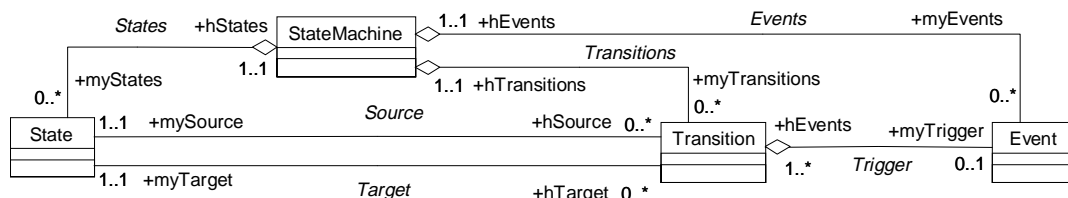
    virtual void entry () { ... }
    virtual void exit  () { ... }
};

FSMState* FSMStateA::s1 () {
    fsm()->t1();
    return &(fsm()->stateA);
}

FSMState* FSMStateA::s2 () {
    fsm()->t2();
    return &(fsm()->stateB);
}

```

(b)



(c)

Slika 2.1: Demonstrativni primer: generisanje koda konačnih automata. (a) Primer konačnog automata. (b) Isečak iz generisanog koda. (c) Metamodel domena konačnih automata.

Metamodel domena modelovanja konačnih automata prikazan je na slici 2.1c. Metamodel je definisan korišćenjem osnovnih strukturnih koncepata jezika UML (*The Unified Modeling Language*) [B-OMG99]. Pretpostavlja se dalje da korisnik (ili projektant alata za modelovanje) želi da definiše način generisanja koda koji će se primeniti za svaku instancu apstrakcije `StateMachine` definisanu u modelu. Problem je kako ovo izvesti.

Jedan način (koji se i najčešće sreće u postojećim alatima) je da korisnik direktno koduje način generisanja izlaznog koda, na jeziku koji je u alatu dostupan za ove svrhe. Tipično je to neki tzv. jezik za skriptovanje (engl. *scripting*). Ovaj kod (ili "skript", engl. *script*) treba da obilazi definisani model, tj. instance apstrakcija datog domena (stanja, tranzicije, događaje itd. u definisanom modelu konačnog automata), čita vrednosti njihovih atributa i proizvodi tekstualni izlaz poštujući C++ sintaksu i semantiku. Za potrebe ovog kodovanja, a radi izbegavanja vezivanja za specifičnosti nekog konkretnog jezika za skriptovanje, ovde će biti korišćen takođe jezik C++. Jedan izuzetno mali isečak iz ovakvog koda koji generiše samo početak deklaracije klase `FSMState` može da izgleda ovako (ovaj kôd izvršava se za instancu apstrakcije `StateMachine` koja se referiše preko pokazivača `this`):

```
// Generate base state class:
output<<"class "<<(this->name+"State")<<"{\n";
output<<"public:\n";
output<<"    "<<(this->name+"State")<<"(" ;
output<<(this->name)<<"* fsm) : myFSM(fsm){}\n";
//...
```

Nedostaci navedenog pristupa su očigledni:

- (1) Proces specifikacije načina generisanja koda izuzetno je složen, zamoran, dugo traje i veoma je podložan greškama.
- (2) Korisnik mora da se bavi svim detaljima i specifičnostima ciljnog domena kome pripada izlazna forma (u ovom primeru C++ sintaksa i semantika), što može biti izuzetno složeno.
- (3) Korisnik mora da se bavi svim zamornim tehničkim detaljima, kao što su otvaranje i manipulacija izlaznom datotekom u koju se upisuje kôd, pravljenjem više takvih datoteka (za C++ mora se napraviti i .h i .cpp datoteka), obradom grešaka prilikom upisa u datoteku, formatom generisanog koda, itd.
- (4) Bilo koja modifikacija ili proširenje ovako definisanog generatora koda veoma je teška zbog toga što je specifikacija nepregledna i teško razumljiva.
- (5) Može se pretpostaviti da već postoje raspoloživi (možda u alat već ugrađeni) generatori koda za neke druge opšte domene, kao što je generator C++ koda iz UML modela. U navedenom slučaju ovi raspoloživi generatori koda uopšte se ne koriste, što bi bilo poželjno ukoliko je moguće.
- (6) Specifikacija zbog toga nije lako ponovno upotrebljiva (engl. *reusable*).

Kao zaključak, problem koji se postavlja u ovom radu je iznalaženje jednostavne i efikasne metode za specifikaciju i implementaciju mehanizma za generisanje izlaznih formi u okviru alata za modelovanje. Ta metoda treba da podrži brzu i laku izradu specifikacija generisanja izlaznih formi koje su pregledne i jednostavne za modifikaciju i proširivanje. Najзад, ove specifikacije treba da omoguće automatsku implementaciju mehanizma za generisanje izlaza unutar alata za modelovanje. Ovakva metoda mogla bi da se koristi u različitim kontekstima:

a) Prilikom konstrukcije alata za modelovanje kod kojih je domen modelovanja, kao i način generisanja izlaza (ili više raspoloživih načina, od kojih korisnik bira željeni) fiksiran u vreme razvoja alata. Predložena metoda koristila bi se za brži i lakši razvoj samih alata za modelovanje.

b) U prilagodljivim (engl. *customizable*) alatima za modelovanje, u kojima je domen modelovanja fiksiran konstrukcijom samog alata, ali je korisniku omogućeno da sam definiše, u celini ili delimično, način na koji se generiše izlaz. U ovom slučaju, cilj je da korisnik ima na raspolaganju jasnu i jednostavnu metodu za brzo definisanje i implementaciju željenog mehanizma generisanja izlaza.

c) U alatima za metamodelovanje, u kojima je korisnik u mogućnosti da sam definiše i metamodel specifičnog domena za modelovanje, kao i način na koji se iz modela u tom domenu generiše željeni izlaz. I ovde je cilj da korisnik ima na raspolaganju efikasan način definisanja i implementacije mehanizma generisanja izlaza.

Definicije pojmova i pretpostavke

U cilju preciznijeg izražavanja i boljeg razumevanja izloženog, u ovom poglavlju navedene su definicije osnovnih pojmova koji će biti korišćeni u radu i koji su u bliskoj vezi sa njegovom temom. U literaturi se, nažalost, mnogi od ovde navedenih pojmova ponekad koriste i bez preciznih definicija, ili sa donekle različitim i nekonzistentnim značenjima. Zbog toga postoji izvesna konfuzija u razumevanju među autorima u ovoj oblasti [C-Nor98b]. Ipak, kako vremenom oblast modelovanja i metamodelovanja sazreva, različita shvatanja osnovnih koncepata polako konvergiraju, pa je utisak da ipak većina autora ima slično ili potpuno isto razumevanje ključnih pojmova. Ovde se težilo usvajanju upravo takvih opšteprihvaćenih značenja, uz eventualna mala prilagođenja konkretnom kontekstu ovoga rada. Za svaki pojam data je usvojena definicija, spisak sinonima koji se koriste u literaturi ili u ovom radu, kratko objašnjenje, kao i objašnjenje na demonstrativnom primeru iz prethodnog poglavlja i eventualno na još nekim primerima. U ovom poglavlju su takođe date i osnovne pretpostavke na kojima se bazira dalja analiza problema.

Polazni pojmovi: sistem, apstrakcija, relacija i jezik

Ovde su najpre pobrojani osnovni pojmovi od kojih se polazi u ostalim definicijama i koji su od interesa za kontekst ovoga rada. Ovi pojmovi su definisani i objašnjeni intuitivno, uz komentare i primere.

Sistem koji se konstruiše

Definicija. *Sistem koji se konstruiše* je proizvod koji je cilj procesa konstrukcije pomoću softverskog alata za modelovanje.

Objašnjenje. Kontekst ovoga rada predstavljaju softverski alati za modelovanje koji, između ostalog, služe i za konstrukciju inženjerskih sistema. Sistem koji se konstruiše je, dakle, proizvod tog procesa konstrukcije. U daljem tekstu, u odgovarajućem kontekstu, sistem koji se konstruiše biće nazivan kratko *sistemom*, ukoliko takva upotreba ne unosi dvosmislenost sa opštim značenjem pojma *sistem*, ili sa nekim konkretnim sistemom o kome je reč u datom kontekstu.

U primerima u ovom radu sistemi koji se konstruišu biće najčešće, ali ne isključivo, računarski bazirani sistemi (engl. *computer-based systems*, CBS). Ukoliko se radi o softverskom sistemu, takav sistem biće nazivan još i (*ciljnom*) *aplikacijom*.

Primeri. Za demonstrativni primer konačnih automata, sistem koji se konstruiše predstavlja neki konkretni softverski sistem (aplikacija) čije se ponašanje može definisati pomoću koncepta konačnog automata. Na primer: softver za digitalni telefonski aparat, automat za prodaju pića, bankomat (ATM), i slično. Sistem koji se konstruiše ne mora biti računarski sistem. Na primer: projekat građevine ili konstrukcija aviona.

Apstrakcija, instanca, svojstvo i relacija

Definicija. *Apstrakcija* (engl. *abstraction*) je uprošćenje nekog pojma u oblasti od interesa, koje se fokusira na elemente tog pojma koji su bitni u određenom kontekstu, a zanemaruje detalje koji nisu bitni za taj kontekst [B-Boo94]; apstrakcijom se definiše skup konkretnih

primeraka koji poseduju zajedničke osobine koje su bitne za dati kontekst. *Instanca* (engl. *instance*) apstrakcije je konkretan primerak date apstrakcije. *Svojstvo* (engl. *property*) apstrakcije je obeležje (osobina) koje poseduju sve instance date apstrakcije. *Relacija* (engl. *relationship*) je konceptualna veza koja postoji između apstrakcija.

Objašnjenja. Navedeni pojmovi su jako rašireni, doduše pod različitim nazivima, u različitim oblastima softverskog inženjerstva (ovi različiti nazivi ovde su navedeni kao sinonimi). Zbog toga su njihova značenja opšte prihvaćena i prilično razumljiva i bez formalnih definicija, pa ovde navedene definicije treba prihvatiti krajnje uslovno. Prema tim značenjima, apstrakcija je tip koji predstavlja skup instanci koje poseduju zajednička svojstva, pri čemu svaka instanca može imati svoju vrednost datog svojstva. Osnovni implicitni odnos koji se ističe kao izuzetno bitan je dihotomija *tip-instanca* [B-OMG99]. Konkretna značenja koja su ovde prihvaćena su već sasvim jasno definisana i usvojena u oblasti objektno orijentisanog softverskog inženjerstva [B-Boo94][B-Boo99][B-OMG99]. Ovaj rad se suštinski oslanja na ta značenja.

Sinonimi. Apstrakcija: koncept, pojam, entitet (engl. *entity*), klasa (engl. *class*), tip (engl. *type*). Instanca: primerak, objekat (engl. *object*). Svojstvo: atribut (engl. *attribute*), obeležje.

Jezik i specifikacija jezika

Definicija. *Jezik* (engl. *language*) je sredstvo izražavanja koje definiše (najčešće beskonačni) skup iskaza koji imaju odgovarajuće značenje (semantiku) u oblasti od interesa, i koje se može formalno definisati pomoću konačnih specifikacija.

Objašnjenja. Pojam jezika je takođe veoma raširen i opšteprihvaćen, pa ni ova definicija nije sasvim neophodna, ali je navedena zbog kompletnosti. Treba obratiti pažnju da se pod jezikom posmatra apstraktno sredstvo izražavanja u oblasti od interesa, i nikako se ne odnosi isključivo na tekstualne (tradicionalne) jezike. Dakle, pod ovim pojmom su obuhvaćena sva sredstva izražavanja koja se mogu formalno definisati. Važno je uočiti i sledeće bitno ograničenje: posmatraju se samo skupovi iskaza koji se mogu definisati pomoću konačnih specifikacija. Naime, jezik može biti konačan ili, češće, beskonačan skup mogućih iskaza, ali je bitno da se taj skup može opisati pomoću konačnih specifikacija. Načini na koje je moguće formalno specifikovati jezik su jedan od najširih oblasti interesovanja softverskog inženjerstva. Samo jedan od najstarijih i najvažnijih sredstava su gramatike koje opisuju sintakse tekstualnih [A-Aho86] i vizuelnih jezika [E-Cos97].

Sinonimi. Za specifikaciju jezika biće ravnopravno korišćen i termin *definicija jezika*, pri čemu se pod tim podrazumeva definicija sintakse i semantike jezika.

Primeri. Za demonstrativni primer konačnih automata, jezikom se može smatrati (inače beskonačan) skup svih konačnih automata koji se mogu definisati pomoću koncepta stanja, prelaza, događaja, itd. Sa druge strane, isti ti konačni automati mogu biti definisani i na drugom jeziku, npr. na jeziku C++, kao što je prikazano u prethodnom poglavlju. Prema tome, treba uočiti da se isti sistem može prikazati na različitim jezicima kao sredstvima izražavanja, u zavisnosti od ugla posmatranja i namene prikaza. Slika 2.1c prikazuje formalnu specifikaciju jezika konačnih automata za dati primer. Za neračunarske primere, jezik može biti sredstvo za definisanje projekta građevine (ovaj jezik poseduje pojmove kao što su npr. "zid", "greda", "prozor" itd.), ili aviona (sa pojmovima npr. "nosač konstrukcije", "oplata", "krilce" itd.).

Implicitna semantika i konsekvence

U ovom radu, a i generalno, može se smatrati da se jezik definiše pomoću skupa pojmova (apstrakcija), njihovog značenja, svojstava i dozvoljenih relacija (načina povezivanja). Prema tome, formalna specifikacija nekog jezika sadrži definicije apstrakcija, njihovih svojstava i relacija. Iskazi datog jezika predstavljaju (uređene) skupove instanci apstrakcija tog jezika.

Za demonstrativni primer konačnih automata, specifikacija jezika na slici 2.1c definiše apstrakcije kao što su: "StateMachine" (konačni automat), "State" (stanje), "Transition" (tranzicija) i "Event" (događaj), kao i njihove relacije sa odgovarajućim značenjem: "States" ("konačni automat sadrži stanja"), "Source" (tranzicija ima svoje izvorišno stanje) ili "Target" (tranzicija ima svoje odredišno stanje). Ovom specifikacijom definisan je skup svih iskaza datog jezika, tj. (beskonačan) skup svih konačnih automata koji se mogu definisati, od kojih je samo jedan prikazan šematski na slici 2.1a.

Sa druge strane, sama formalna specifikacija jezika je opet skup iskaza na nekom drugom (ili čak istom!) jeziku. Za isti primer konačnih automata, specifikacija jezika na slici 2.1c je zapravo iskaz na jeziku UML. Taj iskaz ima svoje tačno definisano značenje, prema definiciji semantike jezika UML [B-Boo99][B-OMG99]. Dakle, sama formalna specifikacija jezika je iskaz na nekom jeziku, pa je ovaj odnos "specifikacija-iskaz" zapravo (potencijalno beskonačan) lanac relacija "tip-istanca". Na primer, stanje A na slici 2.1a je instanca apstrakcije "State" iz specifikacije jezika na slici 2.1c, dok je "State" sa slike 2.1c zapravo instanca apstrakcije "Class" iz specifikacije jezika UML [B-OMG99]. (Kao kuriozitet, "Class" u specifikaciji jezika UML je instanca iste te apstrakcije istog tog jezika, jer je UML definisan pomoću sebe samog [B-OMG99].)

Izvedeni pojmovi: model, domen i metamodel

U ovom odeljku definisani su i objašnjeni najvažniji pojmovi u okviru teme ovog rada. Ovi pojmovi su u tesnoj vezi sa ranije definisanim polaznim pojmovima, pa se mogu smatrati izvedenim.

Model i modelovanje

Definicija. *Model* (engl. *model*) je opis sistema koji se konstruiše pomoću nekog jezika. Ovaj jezik naziva se *jezikom modelovanja* (engl. *modeling language*). *Modelovanje* (engl. *modeling*) je proces pravljenja modela.

Objašnjenja. Model je pojednostavljenje realnog sistema koji se konstruiše [B-Boo99]. Model predstavlja jedan pogled na sistem. Taj pogled koristi određeni jezik da bi iskazao odgovarajuća zapažanja o sistemu i specifikovao njegove pojedinosti koje se žele istaći tim pogledom, pri čemu se eventualno neke druge pojedinosti zanemaruju. Prema tome, za jedan isti sistem koji se konstruiše, može se definisati više različitih modela, potencijalno na različitim jezicima, sa dva bitna cilja:

- Da se pomoću različitih pogleda sistem bolje opiše i prikaže iz različitih uglova. Za pojedine poglede mogu se koristiti različiti jezici, tako da se za prikaz sistema iz određenog ugla koristi jezik koji je najpogodniji za izražavanje.
- Da se pomoću različitih modela prikaže isti sistem, ali na različitim nivoima apstrakcije. Sistem se može prikazati na visokom nivou apstrakcije, pomoću jezika koji nudi sredstva izražavanja koja su najpogodnija konstruktoru jer su najbliža njegovom načinu razmišljanja. Sa druge strane, takav apstraktan model može biti nepogodan ili potpuno neupotrebljiv nekom drugom subjektu koji treba da interpretira specifikaciju sistema da bi ga realizovao. Taj subjekat može da bude čovek, okruženje za izvršavanje aplikacije,

programski prevodilac i slično. Zbog toga je potrebno formirati model istog sistema, ali na potencijalno nižem nivou apstrakcije, koji je moguće interpretirati na opisani način.

Modelovanje je proces pravljenja modela. Modelovanjem se postižu sledeći ciljevi [B-Boo99]:

- Model pomaže da se sistem vizuelno prikaže onakav kakav jeste ili kakav se želi postići (*vizuelizacija*).
- Modelom se specifikuje struktura i ponašanje sistema (*specifikacija*).
- Model predstavlja uzor koji pokazuje kako sistem treba konstruisati (*konstrukcija*).
- Model predstavlja dokumentaciju projektnih odluka (*dokumentacija*).

Više detalja i diskusije o principima modelovanja može se naći u [B-Boo99].

Sinonimi. Model predstavlja opis, specifikaciju, odnosno definiciju sistema koji se konstruiše.

Korelisani pojmovi. Model je opis sistema na nekom jeziku, dakle model predstavlja skup iskaza na nekom jeziku. Ti iskazi sadrže instance apstrakcija i relacija definisanih u datom jeziku.

Primeri. Za demonstrativni primer konačnih automata, jedan primer modela šematski je prikazan na slici 2.1a. To je jedan konkretan konačni automat, definisan pomoću pojmova definisanih u datom jeziku na slici 2.1c. Drugi model istog sistema je izvorni C++ kod na slici 2.1b. Ovaj model je iskaz na drugom jeziku (C++). Ovaj model je model nižeg nivoa apstrakcije i formiran je sa ciljem da bude interpretiran od strane subjekta – programskog prevodioca za C++, koji je zadužen da taj model dalje transformiše u konačnu izlaznu formu – mašinski izvršni kôd. Ova forma se interpretira od strane računarskog hardvera koji obezbeđuje zahtevano ponašanje sistema koji je konstruisan (npr. digitalni telefonski aparat). Prema tome, korisnik (konstruktor–softverski inženjer), definisao je apstraktni model na slici 2.1a, korišćenjem apstraktnog jezika konačnih automata, jer je on pogodan za brzu i jednostavnu specifikaciju ponašanja sistema koji se konstruiše. Sa druge strane, taj model je transformisan u druge oblike koji se interpretiraju od strane drugih subjekata u cilju dobijanja željenog sistema.

Za primer građevinskog projektovanja može se zamisliti slična situacija: projektant koristi neki svoj apstraktni jezik koji je njemu najpogodniji za pravljenje modela građevine koja se konstruiše, jer se tim jezikom najbolje rešavaju problemi kompleksnosti. Sa druge strane, iz tog modela treba formirati konkretne modele nižih nivoa apstrakcije (koji koriste možda neke sasvim konkretne pojmove, kao što su građevinski materijali, konkretni pojedinačni nacrti elemenata građevine i slično), koji će biti interpretirani od drugih subjekata – građevinskih radnika na gradilištu a koji treba da implementiraju željeni sistem. Ova dva subjekta (projektant i građevinski radnik) verovatno koriste različite jezike za izražavanje svojih pogleda na isti sistem – građevinu koja se gradi.

Domen modelovanja

Definicija. *Domen modelovanja* (engl. *modeling domain*) je konceptualni prostor u kojem se vrši modelovanje.

Objašnjenja. Domen je konceptualni prostor, dakle prostor u kome postoje odgovarajuće apstrakcije, u kome one imaju odgovarajuće značenje (semantiku), svojstva i relacije. Zbog toga je pojam domena blisko povezan sa pojmom jezika, kao što je opisano u nastavku. Pojam domena se ovde uvodi i koristi jer je široko zastupljen u literaturi, iako je njegova definicija ovde donekle neodređena. Zato se ovaj pojam uzima kao intuitivan i blisko povezan sa pojmom jezika i metamodela koji je definisan kasnije.

Korelisani pojmovi. Domen predstavlja prostor modelovanja koje se vrši korišćenjem nekog jezika. Specifikacija jezika sadrži definicije apstrakcija, svojstava i relacija koje postoje u datom domenu, kao i njihove semantike, pa je zato specifikacija jezika zapravo formalni opis domena.

Primeri. Za demonstrativni primer konačnih automata, jedan domen modelovanja je konceptualni prostor modelovanja konačnih automata pomoću apstrakcija i njihovih relacija prikazanih na slici 2.1c (konačni automat, stanje, prelaz itd.). Drugi domen je domen programa pisanih na jeziku C++. Ovaj drugi domen oslanja se na sintaksu i semantiku jezika C++. Treba primetiti da se u prvom domenu (konačnih automata) mogu modelovati samo sistemi čije se ponašanje može opisati konceptom konačnog automata uz usvojenu semantiku. Sa druge strane, u drugom domenu programa na jeziku C++, mogu se modelovati i mnogi drugi sistemi. Dakle, u ovom primeru ovaj drugi domen je iskorišćen za modelovanje samo jedne od mnogo mogućih vrsta sistema.

Metamodel i metamodelovanje

Definicija. *Metamodel* (engl. *metamodel*) je formalna specifikacija jezika modelovanja [C-Nor99]. *Metamodelovanje* (engl. *metamodeling*) je proces definisanja metamodela.

Objašnjenja. Metamodel je formalna specifikacija jezika modelovanja, odnosno specifikacija domena modelovanja. Zbog toga se još kaže da je i metamodel zapravo "(konceptualni) model domena " ili "metamodel za domen modelovanja".

Kako je i sam metamodel formalna specifikacija, odnosno iskaz na nekom jeziku, i sam metamodel se može smatrati modelom, ali na nekom drugom jeziku višeg nivoa hijerarhije. Zbog toga ne postoji apsolutno značenje pojma "meta", već je "meta" relativna referenca: "metamodel" je "model domena modelovanja". Drugim rečima, termin "model" se najčešće upotrebljava u značenju opisa sistema koji se konstruiše; kada je sam sistem koji se konstruiše zapravo sredstvo (okruženje, alat) za modelovanje drugih sistema, onda se njegov model naziva metamodelom. Ovaj lanac "meta" relacija je (teorijski) neograničen, ali će ovde on biti ograničen kao što će biti opisano u nastavku.

Sinonimi. Metamodel je model domena modelovanja. Ako se metamodel odnosi na dati domen, biće upotrebljavan i izraz "metamodel domena" ili "metamodel za domen"; eventualno postojanje ili nepostojanje prefiksa "meta" u prvom izrazu neće izazivati zabunu, jer su ovde ovi pojmovi precizno definisani.

Primeri. Za demonstrativni primer konačnih automata, metamodel za domen konačnih automata prikazan je na slici 2.1c. On sadrži definicije apstrakcija iz datog domena i njihovih relacija. Za domen programa na jeziku C++, metamodel nije prikazan, ali se on implicitno podrazumeva: to je definicija sintakse i semantike jezika C++.

Implicitna semantika i konsekvence

Ukoliko je neki model formiran korišćenjem metamodela za neki domen, za taj model će se govoriti da "pripada tom domenu", ili da je "model iz tog domena", ili da je "model u tom domenu".

Modelovanje u specifičnom domenu (engl. *domain-specific modeling*) je termin koji se često koristi u literaturi. Njegovo značenje direktno sledi iz navedenih definicija: označava proces pravljenja modela u datom domenu.

Pojmovi vezani za transformacije modela

U ovom odeljku definisani su i objašnjeni pojmovi vezani za uži predmet ovog rada – transformacije modela u okruženjima za modelovanje.

Transformacija modela

Definicija. Proces dobijanja jednog (odredišnog) modela iz drugog (izvorišnog) modela datog sistema koji se konstruiše naziva se *transformacija modela* (engl. *model transformation*).

Objašnjenja. Kao što je već objašnjeno, za dati sistem koji se konstruiše veoma je često potrebno formirati više modela koji predstavljaju različite opise istog sistema, ali iz različitih uglova posmatranja ili na različitim nivoima apstrakcije. Iako se to ne zahteva po navedenoj definiciji, u praksi su najčešće ti modeli iz različitih domena, odnosno oslanjaju se na različite metamodele. Iz istog razloga je data definicija ograničena na različite modele istog sistema koji se konstruiše, mada ni to u opštem slučaju ne mora da važi. Formalna zapažanja i rezultati u ovom radu uglavnom neće biti ograničeni na različite modele istog sistema, ali je to, čini se, jedini slučaj od značaja u praksi.

Proces dobijanja odredišnog modela iz izvorišnog modela datog sistema naziva se transformacija modela, što je zapravo i glavni predmet ovog rada. Navedena definicija ne nameće način na koji se data transformacija vrši. Naime, ovu transformaciju može vršiti sam korisnik (konstruktor) ručno. Naravno, znatno je pogodnije transformaciju vršiti automatski, uz pomoć softverskog alata, jer se time značajno štedi na vremenu i smanjuje mogućnost unošenja greške. Osnovni preduslov da se transformacija može automatizovati je mogućnost formalne definicije načina transformisanja modela. Metode za specifikaciju načina transformisanja modela i automatsko generisanje alata koji vrše transformaciju su osnovna tema ovoga rada.

Sinonimi. Proces dobijanja odredišnog modela iz izvorišnog modela je zapravo proces *kreiranja*, odnosno *generisanja* modela, pa bi ovi nazivi možda bili primereniji. Međutim, u literaturi se za ovaj pojam koristi skoro isključivo termin *transformacija*, a ponekad i *translacija* ili *prevođenje* (engl. *translation*, *compilation*), pa će on i ovde biti uglavnom upotrebljavan, dok će ostali navedeni termini biti smatrani sinonimima.

Primer. Za demonstrativni primer konačnih automata, izvorišni model od interesa je model jednog automata prikazan na slici 2.1a, a odgovarajući odredišni model na slici 2.1b. Problem transformacije je detaljno diskutovan u prethodnom poglavlju, kao problem od interesa u ovom radu.

Preslikavanje domena

Definicija. Formalna specifikacija načina vršenja transformacije modela iz različitih domena naziva se *preslikavanjem domena* (engl. *domain mapping*).

Objašnjenja. Ukoliko se posmatrana transformacija modela može formalno opisati, onda se njen opis (specifikacija, definicija) naziva preslikavanjem izvorišnog domena (domena izvorišnog modela) u odredišni domen (domen odredišnog modela). Ovo preslikavanje zapravo definiše kako se od nekog (bilo kog) modela iz izvorišnog domena može dobiti odgovarajući model iz odredišnog domena, sa ciljem da se taj proces automatizuje uz pomoć date formalne specifikacije.

Ovaj pojam pod ovim nazivom nije pronađen u literaturi, već je prvi put predložen u okviru ovog rada.

Sinonimi. Kako je metamodel zapravo formalni opis domena, preslikavanje (ili *mapiranje*) domena će ovde biti ponekad nazivano i *preslikavanjem metamodela*.

Primer. Za demonstrativni primer konačnih automata jedan način definisanja transformacije modela diskutovan je u prethodnom poglavlju. Tu se radilo o specifikaciji procesa pomoću koga se iz nekog modela iz domena konačnih automata može dobiti odredišni model – program na jeziku C++. Te specifikacije predstavljaju preslikavanja domena.

Četvoroslojna arhitektura metamodelovanja

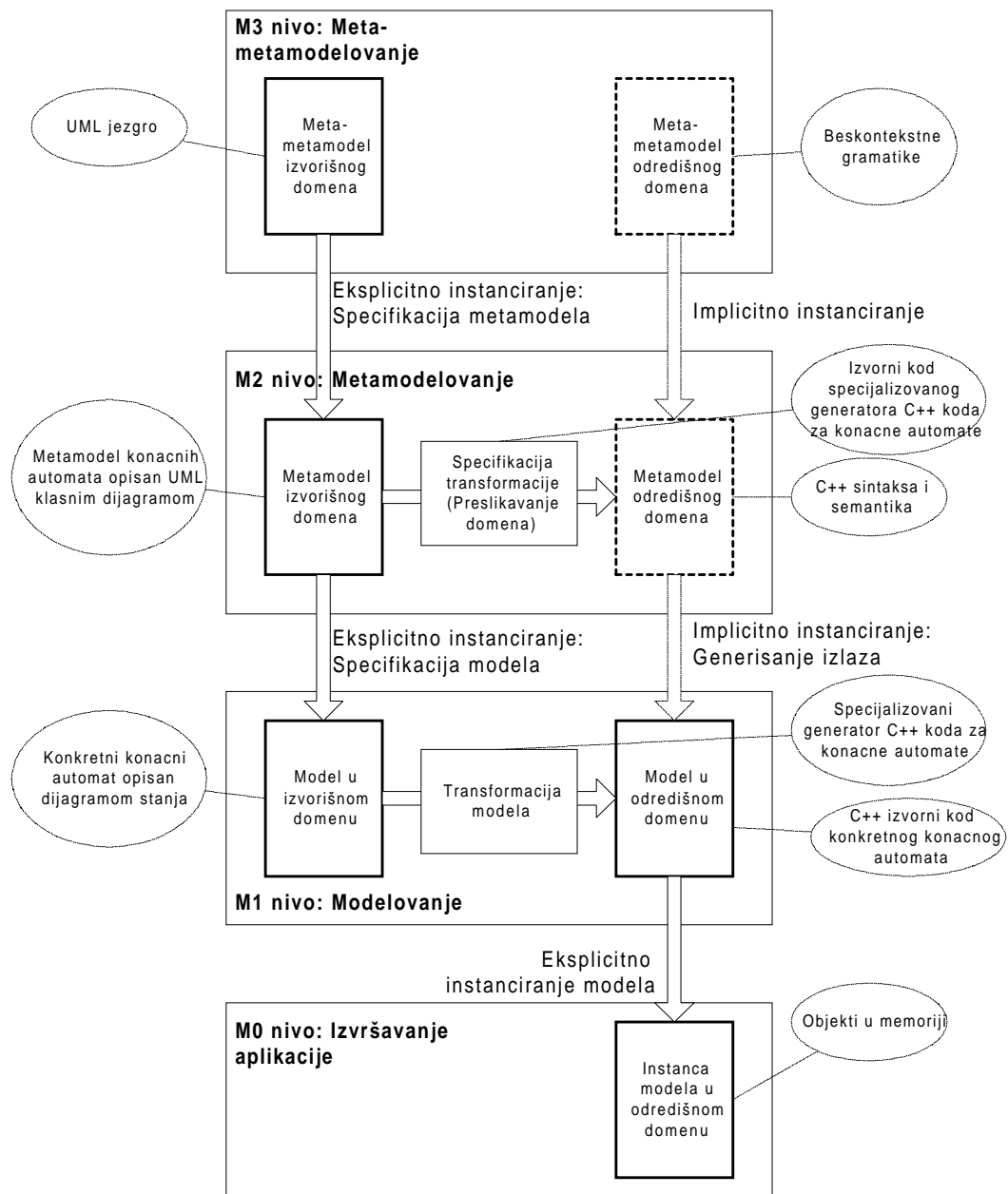
Iako lanac relacija "meta" u opisanoj dihotomiji "tip-instanca" modela i njihovih metamodela može teoretski biti neograničen, za praksu je značajan samo lanac male ograničene dužine. Zbog toga je u ovoj oblasti definisana tzv. *četvoroslojna arhitektura metamodelovanja* (engl. *four-level metamodeling architecture*) [C-MetW][C-Nor98a][C-Nor99] prikazana na slici 2.2.

Prema toj arhitekturi, najniži, tzv. M0 nivo predstavlja nivo podataka aplikacije, odnosno skup instanci unutar konstruisane aplikacije. U demonstrativnom primeru konačnih automata to su objekti C++ klasa koji postoje u memoriji računara prilikom izvršavanja prevedenog C++ programa.

Model same aplikacije definisan je na M1 nivou. Kao što je već opisano, na ovom nivou u navedenom primeru formirana su dva modela. Prvi, izvorišni model formirao je sam konstruktor, eksplicitno, korišćenjem apstraktnih pojmova konačnog automata, stanja, prelaza, događaja itd. Drugi, odredišni model predstavlja izvorni C++ kôd programa koji implementira zadato ponašanje. On je dobijen implicitno, odgovarajućom transformacijom. Tu transformaciju vrši specijalizovani generator C++ koda, za rešenje diskutovano u prethodnom poglavlju.

Modeli na M1 nivou su zapravo instance metamodela definisanih na M2 nivou. Izvorišni metamodel je prikazan na slici 2.1c. Odredišni metamodel je ovde implicitan; to je gramatika jezika C++. Specifikacija koja definiše transformacije modela je preslikavanje domena, realizovano pomoću programa specijalizovanog generatora koda opisanog u prethodnom poglavlju.

Konačno, metamodeli na nivou M2 su instance svojih meta-metamodela sa nivoa M3. Nivo M3 sadrži specifikacije jezika izražavanja pomoću kojih su definisani metamodeli na nivou M2. Za izvorišni domen to je takođe jezik UML, tačnije samo njegovo jezgro koje definiše osnovne strukturne koncepte (klasa, atribut, asocijacija i nasleđivanje). Za odredišni domen to su beskontekstne gramatike tekstualnih jezika [A-Aho86].



Slika 2.2: Četvoroslojna arhitektura metamodelovanja u kontekstu demonstrativnog primera konačnih automata. Isprekidanim linijama označeni su elementi koji se implicitno pojavljuju u navedenom primeru.

Pretpostavke i ograničenja

Iako u opštem slučaju metamodeli domena mogu biti definisani na različite načine i za izvorišne i za odredišne domene u preslikavanjima, ovaj rad će se ograničiti samo na tzv. *leksičko-sekvencijalne*, tzv. *gramatičko-hijerarhijske* i tzv. *objektno-grafovske* domene (ova klasifikacija i termini uvedeni su samo za potrebe ovog rada). Ovakvo ograničenje je učinjeno kao suženje opsega rada u cilju iznalaženja rešenja. Ono nimalo ne umanjuje upotrebljivost rezultata ovoga rada jer, kao što će i kasnije biti diskutovano i na primerima pokazano, većina praktičnih domena od interesa spada u jednu od ove tri kategorije.

		Tipovi domena		
		Leksičko-sekvencijalni	Gramatičko-hijerarhijski	Objektno-grafovski
Aspekti	Meta-metamodel	Leksička pravila	Beskontekstne gramatike	Objektni koncepti (jezgro UML)
	Struktura modela	Sekvenca	Tipizirano stablo	Tipizirani graf
	Pojavna forma	Tekst	Nema	Grafički (simbolički, vizuelni) prikaz

Tabela 2.1: Klasifikacija domena i aspekti: vrsta meta-metamodela, struktura kojom se predstavljaju modeli i odgovarajuće pojavne forme.

Podela je izvršena na osnovu vrste meta-metamodela datog domena, tj. na osnovu apstrakcija i relacija koje postoje u meta-metamodelima. Kao što će biti objašnjeno u nastavku, ta priroda uslovljava i način na koji se sam model inherentno predstavlja, odnosno vrstu strukture pomoću koje se model u datom domenu izgrađuje.

Treba naglasiti da se ova podela nikako ne odnosi na prirodu *modela* u datom domenu, već isključivo na prirodu *meta-metamodela* i *strukture modela*. Naime, reč "objektno" u nazivu treće vrste domena ne odnosi se na objektno orijentisane modele, kao što su npr. objektni model nekog softverskog sistema definisan na jeziku UML ili objektno orijentisani program na jeziku C++. Drugim rečima, objektni model nekog sistema može biti formiran i u objektno-grafovskom domenu (kao što je model na jeziku UML), ali i u leksičko-sekvencijalnom domenu (kao što je program na jeziku C++). Ovakvo značenje pojma "objektni" koji se odnosi na model nije od užeg interesa u ovom radu. Sa druge strane, ukoliko je sam metamodel domena objektni, što znači da je definisan pomoću instanci objektno orijentisanih apstrakcija, onda se dati domen naziva objektno-grafovskim.

Slično, ova podela nije primarno uslovljena pojavnim oblikom modela (notacijom), odnosno formom u kojoj se neki model datog domena prikazuje korisniku. Iako su određeni pojavni oblici prirodno prilagođeni odgovarajućim vrstama domena, ovaj aspekt nije od primarnog značaja u ovom radu. Navedeni aspekti (vrsta meta-metamodela, struktura kojom se predstavlja model i pojavna forma) sumirani su u Tabeli 2.1, a detaljnija diskusija sledi uz opise pojedinih vrsta domena. Bez obzira na malu nedoslednost u načinu skraćivanja naziva, zbog pogodnosti naziva, leksičko-sekvencijalni domeni biće u ovom radu kraće nazivani i samo sekvencijalnim, gramatičko-hijerarhijski samo gramatičkim, a objektno-grafovski samo objektnim domenima.

Leksičko-sekvencijalni domeni

Ovo su domeni u kojima su metamodeli iskazani pomoću leksičkih pravila, a modeli su jednostavne linearne (sekvencijalne) strukture. Takve strukture sastoje se iz nizova (sekvenci) elemenata (instanci apstrakcija). Drugim rečima, modeli se sastoje od instanci apstrakcija između kojih postoji samo jedna relacija – "sledi" (odnosno "prethodi"). Apstrakcije mogu biti, u zavisnosti od nivoa posmatranja i konteksta, znakovi (engl. *character*), tokeni (engl. *token*) koji imaju svoj tip i vrednost kao atribut, ili neke druge apstrakcije specifične za dati domen. Najčešći pojavni oblik modela iz ovih domena je tekst, pa će zbog toga ovi domeni u ovom radu biti nazivani jednostavno i *tekstualnim* domenima. Iako se ovaj termin strogo odnosi samo na jednu vrstu pojavne forme, on će se koristiti jer je kratak i asocira na jednostavnu sekvencijalnu strukturu modela, a osim toga predstavlja i najčešće korišćenu formu koja je od značaja za ovaj rad.

Ovakvi domeni se u praksi uglavnom koriste ili kao krajnje ulazni, ili kao krajnje izlazni domeni. U slučaju kada je sekvencijalna forma ulazni model, ona se najčešće transformiše u neki drugi model, tzv. *internu formu* pomoću tradicionalnih tehnika parsiranja [A-Aho86]. Ova interna forma predstavlja model u nekom gramatičko-hijerarhijskom domenu.

Sa druge strane, tekstualni (sekvencijalni) domeni se veoma često pojavljuju i kao ciljni domeni u procesu transformisanja modela. Primera ima mnogo, počev od dokumentacije, sve do izvornog programskog koda (koji je zapravo opet samo ulazni model za programski prevodilac), i uopšte specifikacija na bilo kom tekstualnom jeziku. Ovaj slučaj je, kao što je već objašnjeno, posebno značajan za razmatranje u okviru ovog rada. U tom kontekstu, tekstualni domeni biće posmatrani u ovom radu samo kao ciljni domeni u koje se preslikavaju polazni domeni koji su bitni za korisnika.

Gramatičko-hijerarhijski domeni

Kao što je navedeno, modeli na tekstualnim jezicima se u cilju dalje analize i transformacije najpre prevode u interne forme tradicionalnim tehnikama programskih prevodilaca [A-Aho86]. Ove tehnike imaju za cilj da prepoznaju da li ulazna sekvencijalna forma zadovoljava sintaksu datog tekstualnog jezika i da izgrade internu formu koja je pogodna za kasnije prevođenje u druge oblike (npr. opet u sekvencijalnu formu, kao što je mašinski program). Interne forme su u tradicionalnim tehnikama po pravilu strukture koje predstavljaju tzv. *stablo izvođenja* (engl. *derivation tree*), u kome su čvorovi tipizirani: zna se "tip" svakog čvora, odnosno čvor je instanca tačno određene apstrakcije, terminalnog ili neterminalnog simbola. Zbog svog značaja za klasifikaciju transformacija modela u ovom radu, ovakvi modeli posmatraju se kao modeli u posebnoj vrsti tzv. *gramatičko-hijerarhijskih* domena.

Metamodeli ovih domena izraženi su pomoću pravila beskontekstnih gramatika, a struktura modela iz ovih domena je, kao što je navedeno, tipizirano stablo. Kako se modeli iz ovih domena praktično isključivo koriste kao interne forme, oni nemaju svoje pojavne forme bitne za korisnika. Štaviše, u većini gramatika u upotrebi nije ni potrebno eksplicitno graditi strukturu stabla, već se to stablo implicitno prilikom transformacija obilazi pomoću odgovarajućih koncepata automata.

Iako su tipizirana stabla samo specijalni slučaj tipiziranih grafova koji predstavljaju strukturu za modele iz objektno-grafovskih domena, ova vrsta domena izdvojena je zbog postojanja značajne klase transformatora koji su prilagođeni samo tim domenima. Prema tome, problem transformisanja leksičko-sekvencijalnih izvorišnih modela u odredišne gramatičko-hijerarhijske modele, tj. interne forme, jeste veoma stari i dobro proučen problem iz oblasti tradicionalnih programskih prevodilaca [A-Aho86], pa nije predmet užeg interesovanja ovog rada.

Uopštenje sekvencijalnih jezika jesu grafički (simbolički, vizuelni) jezici i njihovo parsiranje pomoću metoda sličnih onima za sekvencijalne jezike [E-Cos97]. Ovaj slučaj biće kratko komentaran u narednoj glavi, ali ni on nije predmet ovoga rada.

Objektno-grafovski domeni

Pod *objektno-grafovskim domenom* ovde se podrazumeva domen čiji je metamodel definisan pomoću osnovnih objektno orijentisanih koncepata. Skup osnovnih objektno orijentisanih koncepata i njihova semantika uzeti su iz jezgra strukturnog dela jezika UML. To je onaj skup koncepata pomoću kojih je definisan sam metamodel UML [B-OMG99]. Drugim rečima, objektni domen je onaj domen čiji je meta-metamodel (model na nivou M3) jezgro jezika UML.

Pošto je tema ovog rada transformacija modela koji su definisani strukturom sa određenom semantikom, od svih koncepata jezika UML i uopšte objektno orijentisanih koncepata bitni su samo oni koji se odnose na definiciju strukture sistema. Osnovni koncepti koji čine jezgro strukturnog dela jezika UML (dalje kratko samo "jezgro UML") i koji su ovde uzeti kao osnova su sledeći [B-Boo99][B-OMG99]:

- *Klasa* (engl. *class*) je opis skupa objekata (instanci) koji dele iste attribute, operacije, relacije i semantiku.
- *Atribut* (engl. *attribute*) je imenovano svojstvo klase koje opisuje opseg vrednosti koje instance tog svojstva mogu imati. Svaki objekat (instanca klase) poseduje konkretnu vrednost svakog atributa svoje klase.
- *Asocijacija* (engl. *association*) je strukturna relacija između klasa koja opisuje skup veza koje mogu da postoje između objekata tih klasa. *Veza* (engl. *link*) je instanca asocijacije koja predstavlja strukturnu povezanost objekata klasa koje su povezane asocijacijom. Ovaj rad ograničava se na binarne asocijacije (između samo dve klase), jer su asocijacije između više klasa retko potrebne u praksi i mogu se uvek transformisati u klasu koja predstavlja tu asocijaciju, povezanu novim binarnim asocicijama sa klasama koje učestvuju u polaznoj asocijaciji.
- *Generalizacija/specijalizacija* (engl. *generalization/specialization*, ili *nasleđivanje*, engl. *inheritance*) je relacija između klasa koja ima dve važne semantičke manifestacije:
 - (a) značenje *nasleđivanja*: specijalizovana (izvedena) klasa implicitno poseduje (nasleđuje) sve attribute, operacije i asocijacije osnovne klase (važi i tranzitivnost);
 - (b) značenje *supstitucije* (engl. *substitution*): objekat (instanca) izvedene klase može se naći svugde gde se očekuje objekat osnovne klase (važi i tranzitivnost); za strukturni aspekt sistema ovo pravilo ima sledeću bitnu manifestaciju: ako u nekoj asocijaciji učestvuje osnovna klasa, onda u nekoj vezi kao instanci te asocijacije mogu učestvovati objekti svih klasa izvedenih iz te klase.

Svi modeli iz ovako definisanih objektnih domena imaju jedno bitno zajedničko svojstvo: njihova struktura se može posmatrati kao *tipizirani graf*. Značenje ovog pojma je sledeće. Model iz objektnog domena sastoji se od objekata (instanci klasa) povezanih vezama (instancama asocijacija). Dakle, objekti predstavljaju čvorove, a veze grane jednog grafa. Pri tom, objekti kao čvorovi grafa imaju svoje tipove (to su klase čiji su to instance), kao i veze koje su instance odgovarajućih asocijacija. Na stranama svake veze koja je instanca neke asocijacije nalaze se instance onih klasa koje povezuje ta asocijacija, ili klasa izvedenih iz njih (uključujući i tranzitivnost nasleđivanja).

Objektni domen najčešće su praćeni odgovarajućim grafičkim (vizuelnim) notacijama koje predstavljaju najpogodniju pojavnu formu za ove domene.

Kao što je već rečeno, stabla izvođenja u tradicionalnim prevodiocima predstavljaju specijalne slučajeve tipiziranih grafova, pa se i ove strukture mogu uslovno smatrati modelima iz objektnih domena.

Ovaj rad bavi se isključivo ovako definisanim objektnim domenima, kao i transformacijama modela iz takvih domena, uključujući tu i transformacije modela iz objektnih domena u modele iz sekvencijalnih domena. Kao što je već rečeno, problem transformacije modela iz sekvencijalnih domena u modele iz drugih domena je problem tradicionalnih programskih prevodilaca i nije predmet ovog rada.

III Pregled postojećih rešenja

Softverski alati za metamodelovanje

U kontekstu ovoga rada, softverski alati za modelovanje se, prema stepenu svoje prilagodljivosti potrebama korisnika, mogu podeliti na:

- a) Fiksne, nepodesive alate za modelovanje (engl. *non-customizable modeling tool*). Pod tim se ovde podrazumevaju alati kod kojih su metamodeli i izvorišnog i odredišnog domena, kao i njihovo preslikavanje fiksirani konstrukcijom samog alata. Kod ovih alata korisniku su na raspolaganju samo domen modelovanja koji su neposredno podržani konstrukcijom samog alata, kao i ugrađena preslikavanja. Ukoliko je domen modelovanja vezan za softversko inženjerstvo, alat se naziva CASE alatom (od engl. *Computer Aided Software Engineering*).
- b) Podesive alate za modelovanje (engl. *customizable modeling tool*). Ovde se pod tim podrazumevaju alati kod kojih su metamodeli izvorišnog ili oba domena fiksirani konstrukcijom samog alata, ali korisnik može sam da definiše njihovo preslikavanje. Primer su mnogi CASE alati kod kojih postoje ugrađeni generatori koda za ciljne programske jezike, ali u kojima korisnik može i sam definisati sopstvene generatore koda [D-RSCW][D-MGSW]. Ovakvih alata je najviše na tržištu.
- c) Alati za metamodelovanje (engl. *metamodeling tool*), koji se zbog toga što im je često osnovna namena konstrukcija alata za modelovanje softverskih sistema nazivaju i meta-CASE alatima. U ovim alatima korisnik može sam definisati metamodel za domen od interesa, kao i način generisanja željene izlazne forme.

Iako se, kao što je rečeno u prethodnoj glavi, rezultati ovoga rada mogu primeniti na alate iz sve tri kategorije, od posebnog interesa su alati za metamodelovanje, jer su istraživanja u toj oblasti donela mnogo rezultata od važnosti za ovaj rad. Zbog toga će rezultati iz ove oblasti biti ukratko analizirani u ovom poglavlju. Najpre će biti istaknuti motivi za konstruisanje ovakvih alata i razlozi njihovog postojanja, a zatim i neki od njihovih najznačajnijih aspekata. U posebnom odeljku je analiziran aspekt meta-metamodela na koji se oslanjaju alati za metamodelovanje i njihova podrška kontroli strukture modela. Jedan od tih elemenata – način generisanja izlaznih formi, koji je i uži predmet ovoga rada – analiziran je u sledećem poglavlju.

Motivi za konstruisanje alata za metamodelovanje

Značaj modelovanja za specifikaciju, konstrukciju i dokumentaciju kompleksnih sistema odavno je opšte priznat. Sa rastom kompleksnosti računarski baziranih sistema i razvojem softverskog inženjerstva, intenzivno su predlagane i brzo napredovale mnoge metode modelovanja u specifičnim domenima. Naročito bogatstvo metoda modelovanja odlikovalo je u praksi najviše eksploatisane oblasti softverskog inženjerstva: informacionih sistema i baza podataka, industrijskih sistema, sistema za kontrolu tokova poslova (engl. *workflow management*) i druge. Krajem 80-ih i početkom 90-godina nastala je nagla ekspanzija metoda za objektno orijentisano modelovanje. Bez obzira na domen, stepen prihvaćenosti u praksi, kompleksnost ili originalnost metoda, sve one imaju mnogo zajedničkih elemenata.

Prvo, sve one su predložene sa ciljem da olakšaju modelovanje sistema u određenom specifičnom domenu. Tu svoju namenu metode ispunjavaju, više ili manje uspešno, počivajući na odgovarajućem metamodelu koji poseduje apstrakcije koje su pogodne za dati domen. Drugo, praktična upotrebljivost date metode je veoma mala ukoliko ona nije podržana odgovarajućim softverskim alatom za modelovanje. Takav alat predstavlja okruženje za

primenu metode i notacije koje se koriste u datom domenu i pruža podršku za automatsku proveru konzistentnosti modela, bolju preglednost i navigaciju kroz model, kao i vizuelizaciju i dokumentaciju sistema koji se konstruiše. Ovakva podrška značajno smanjuje pojavu grešaka i celokupno vreme modelovanja.

Međutim, izrada specijalizovanog alata koji podržava određenu metodu nije nimalo jednostavan posao. Mnoge metode ostale su potpuno neprimenjene upravo zbog nedostatka alata koji ih podržavaju. Dakle, mogućnost brzog i lakog pravljenja alata za modelovanje predstavlja jedan od ključnih faktora u procesu razvoja neke metode. Sa druge strane, praktično nijedna metoda nije odmah po nastajanju sasvim prilagođena korisniku i potpuno zaokružena. Za uspeh jedne metode potrebno je mnogo eksperimentisanja sa njom, odnosno njena primena na različitim eksperimentalnim i realnim sistemima u praksi. Tokom tih eksperimenata veoma je verovatno da će metoda doživeti izmene i proširenja. Takve izmene i proširenja zahtevaju brzu proveru u praksi, koja je opet moguća samo ukoliko je alat za modelovanje lako izmenjiv i proširiv.

Sve su ovo razlozi koji su doveli do izučavanja mogućnosti izgradnje generičkih, tzv. alata za metamodelovanje (ili meta-alata), pomoću kojih bi se proces izgradnje, izmene i proširivanja alata za modelovanje u specifičnom domenu olakšao i ubrzao. To je, naravno, moguće samo ukoliko se pokaže da sve metode za modelovanje imaju nešto zajedničko. Upravo na tome se zasniva i cela oblast istraživanja o kojoj je ovde reč, kao i motivacija za konstrukciju mnogih komercijalnih i akademskih alata za metamodelovanje koji danas postoje na tržištu [D].

Mnogi autori i izvori u ovoj oblasti takođe diskutuju motive za intenzivno proučavanje principa metamodelovanja i konstruisanje softverskih alata koji ga podržavaju [C-Hah96][C-Lau98][C-MetW][C-Mip98][C-Nor99][C-Pla97][C-Szt97].

Bitni aspekti alata za metamodelovanje

Iako je motiv i cilj svih pristupa konstruisanju alata za metamodelovanje isti, ostali bitni aspekti im se značajno razlikuju. Ovde su istaknuti sledeći aspekti alata za metamodelovanje:

- Meta-metamodel na kome počiva i podrška definisanju strukture modela. Alat za metamodelovanje je zapravo alat za modelovanje različitih domena i konstrukciju (generisanje) alata za modelovanje u tim domenima. Isto kao što izražajnost metode modelovanja i njena prilagođenost domenu zavisi od njenog metamodela, tako i spektar primenjivosti alata za metamodelovanje značajno zavisi od njegovog meta-metamodela. Ovde se pod ovom podrškom podrazumeva i podrška za kontrolu konzistentnosti modela, kao i podrška interaktivnoj i programskoj navigaciji kroz strukturu modela u generisanom alatu. Ovo je jedan od najvažnijih aspekata i zato mu je u istraživanjima u ovoj oblasti posvećeno čak i nesrazmerno mnogo pažnje, na račun drugih, takođe bitnih aspekata.
- Stepenn podrške i otvorenosti za definisanje ponašanja specifičnog za domen. Naime, većina alata akcenat stavlja na strukturu modela. Sa druge strane, bitan aspekt je takođe i ponašanje modela koje može biti krajnje specifično za dati domen. Pod stepenom otvorenosti se zato ovde podrazumeva mogućnost proizvoljnog programskog proširivanja generisanog alata za modelovanje i ugrađivanje ponašanja u programske elemente koji implementiraju apstrakcije iz domena u generisanom alatu za modelovanje. Ovaj problem je znatno manje rešavan i većina alata nema nikakvu ili ima veoma slabu podršku ovom aspektu.
- Stepenn podrške definisanju vizuelne notacije za metodu. Po pravilu, metode su praćene odgovarajućim notacijama koje definišu simbole i njihove grafičke relacije, kao i semantiku tih simbola i relacija, odnosno njihov odnos sa metamodelom date metode.

Raspoloživi alati se prilično razlikuju u pristupu ovom aspektu i nude različita rešenja, od veoma skromnih do krajnje složenih i fleksibilnih.

- Način podrške generisanju izlaznih formi i transformacije modela.

Diskusija koja je ovde iznesena zasniva se na analizi devet raspoloživih akademskih i komercijalnih alata za modelovanje ili metamodelovanje, navedenih u grupi D klasifikovane bibliografije.

Prvi aspekt, meta-metamodel i podrška strukturi modela je u bliskoj vezi sa problemom generisanja izlaznih formi (koja se oslanja na strukturu modela), pa će zato detaljnije biti komentarisano u narednom odeljku.

Drugi aspekt, podrška ponašanju i stepen programske proširivosti generisanih alata za modelovanje nije predmet užeg interesovanja ove teze, pa ovde neće biti detaljnije diskutovan. U svakom slučaju, kod svih postojećih analiziranih alata uočava se veoma slaba podrška ovom aspektu. Ona se uglavnom svodi na proširivanje alata za modelovanje posebnim programskim modulima pisanim u specijalizovanim programskim jezicima za skriptovanje koji su zavisni od proizvođača. Prototipski alat za metamodelovanje koji je realizovan u okviru ovog rada generiše alat za modelovanje na standardnom jeziku C++ koji je potpuno proizvoljno proširiv korisničkim kodom. Taj korisnički kod ima puni pristup do strukture modela preko klasa u C++ kodu koje generiše alat za metamodelovanje. Drugim rečima, realizovani alat za metamodelovanje generiše izvorni C++ kôd alata koji se prevodi i povezuje sa korisničkim kodom, pa je fleksibilnost generisanog alata za modelovanje potpuna.

Treći aspekt, podrška definisanju specifične notacije, takođe nije tema ovog rada, pa ni on neće biti diskutovan. Realizovani prototipski alat još uvek ne podržava ovaj aspekt, mada je ta podrška u razvoju.

Četvrti aspekt, podrška generisanju izlaznih formi i transformaciji modela je najuža tema ovoga rada i biće diskutovana detaljno u narednim poglavljima. Ovde se ističe samo to da je u svim analiziranim alatima ta podrška veoma jednostavna, uglavnom svedena na generisanje tekstualnih izlaznih formi, i da ni jedan pristup nije sličan onome koji je predložen u ovom radu, tj. ne promoviše apstraktni pristup transformaciji modela kao što je ovde predloženo.

Meta-metamodeli u alatima za metamodelovanje

Od vremena nastanka alata za metamodelovanje uglavnom zavisi i njihov meta-metamodel. Svi analizirani alati u svom meta-metamodelu poseduju koncepte apstrakcije, odnosno klase, atributa i asocijacije, doduše pod različitim nazivima. Najveće razlike su u tome da li u potpunosti podržavaju i objektno orijentisane koncepte, odnosno nasleđivanje. Koncepti koje poseduju meta-metamodeli analiziranih alata sumirani su u tabeli 3.1.

Alati starije generacije zasnivaju se uglavnom na nekoj varijanti modela entitet-relacija (engl. *entity-relationship*). Kod ovih pristupa postoje koncepti klase (uglavnom pod nazivom *entitet*), atributa (pod različitim nazivima) i asocijacije (uglavnom pod nazivom *relacija*). Koncept nasleđivanja ili nije podržan uopšte [D-ASTW][D-MCCW], ili je sasvim ograničen [D-UAIW]. Alati iz ove grupe razlikuju se po dodatnim konceptima koje pružaju (tabela 3.1).

Alat	Proizvođač	Tip alata	Koncepti u meta-metamodelu
<i>Graphical Designer</i> [D-ASTW]	Advanced Software Technologies, Inc.	Alat za metamodelovanje	<i>Object</i> (klasa), <i>Relationship</i> (asocijacija), <i>Attribute</i>
<i>IPSYS ToolBuilder</i> [D-LSLW]	Lincoln Software, Ltd.	Alat za metamodelovanje	<i>Entity</i> (klasa), <i>Relationship</i> (asocijacija) sa podvrstama <i>Composite</i> (odgovara pojmu agregacije u jeziku UML, engl. <i>aggregation</i>) i <i>Reference</i> (asocijacija koja nije agregacija), <i>Attribute</i> , <i>Generalization</i> , <i>Derived Relationship</i> (ne sa značenjem kao u jeziku UML, nego kao agregatne ili rekurzivne asocijacije), <i>Derived Attribute</i>
<i>MetaEdit+ Method Workbench</i> [D-MCCW]	MetaCase Consulting	Alat za metamodelovanje	<i>Object</i> (klasa), <i>Property</i> (atribut), <i>Relationship</i> (asocijacija), <i>Role</i> (odgovara pojmu asocijacione uloge u jeziku UML, engl. <i>association role</i>)
<i>WithClass Scripting Tool</i> [D-MGSW]	MicroGold Software, Inc.	Podesivi alat za modelovanje	Osnovni OO koncepti, ali ne prema definiciji jezika UML
<i>Alfabet</i> [D-MipW]	mip GmbH	Alat za metamodelovanje	<i>Class</i> , <i>Generalization</i> (<i>is-a</i> taksonomija), <i>Property</i> , koncepti vezani za relacioni model (<i>View</i> , <i>Index</i>)
<i>Rational Rose</i> [D-RSCW]	Rational Software Corporation	Podesivi alat za OO modelovanje	Osnovni OO koncepti, ali ne prema definiciji jezika UML
<i>ConceptBase</i> [C-Hah96]	Technische Hochschule (RWTH) Aachen, Germany	Objektna baza i dinamički OO jezik	<i>Class</i> , <i>Attribute</i> , <i>Generalization</i> i asocijacija indirektno preko <i>Reference Attribute</i>
<i>MetaView</i> [D-UAIW]	University of Alberta	Alat za metamodelovanje	<i>Entity</i> (klasa), <i>Aggregate</i> (odgovara pojmu agregacije u jeziku UML koja je poseban slučaj asocijacije), <i>Relationship</i> (asocijacija), <i>Attribute</i> , <i>Generalization</i> (samo nasleđivanje atributa)
<i>Multigraph Architecture</i> [D-VUnW]	Vanderbilt University	Alat za metamodelovanje	UML jezgro

Tabela 3.1: Koncepti koje sadrže meta-metamodeli analiziranih alata

Alati srednje generacije podržavaju i koncept nasleđivanja, pa se zato mogu smatrati objektno orijentisanim. Međutim, kako su oni razvijani uglavnom u vreme ekspanzije najrazličitijih objektnih metoda, i njihovi meta-metamodeli se razlikuju u mnogim detaljima. Ovakvih alata je najviše, jer su oni i nastajali kao odgovor na ekspanziju različitih metoda.

Alati iz ove grupe su *IPSYS ToolBuilder* [D-LSLW], *WithClass Scripting Tool* [D-MGSW], *Alfabet* [D-MipW], *Rational Rose* [D-RSCW], i *ConceptBase* [C-Hah96]. I ovi alati se razlikuju po varijantama svojih meta-metamodela, odnosno po dodatnim konceptima koje nude (tabela 3.1).

Najzad, tek kada je u drugoj polovini devedesetih godina zaustavljena "poplava" objektnih metoda modelovanja definicijom standardnog jezika UML za objektno modelovanje, počeli su da nastaju i alati koji za svoj meta-metamodel uzimaju definiciju jezika UML [B-OMG99]. Za sada je, koliko je autoru poznato, konstruisan samo jedan alat koji tvrdi da za svoj meta-metamodel koristi UML. To je alat *Multigraph Architecture* razvijen na Vanderbilt Univerzitetu, SAD [D-VUnW]. Pored njega, i prototipski alat realizovan u okviru ovog rada za svoj meta-metamodel uzima definiciju jezika UML (tj. UML metamodel).

Jezik UML, kao novodefinisani standard [B-OMG99] promovise drugačiji koncept implicitnog, ograničenog i kontrolisanog metamodelovanja preko tzv. *mehanizama proširivosti* (engl. *extensibility mechanisms*). Za svaki element modela na jeziku UML može se definisati [B-OMG99][B-Boo99]:

- *Stereotip* (engl. *stereotype*) je proširenje rečnika jezika UML; stereotip dozvoljava da se u jezik uvedu novi pojmovi odnosno apstrakcije i relacije. Stereotip koji se vezuje za elemente modela datog tipa predstavlja zapravo novu apstrakciju u metamodelu. Ako se u modelu za instancu neke apstrakcije T iz jezika UML veže stereotip S, onda je to ekvivalentno uvođenju nove (virtuelne, nepostojeće) apstrakcije S koja specijalizuje apstrakciju T u metamodelu.
- *Označena vrednost* (engl. *tagged value*) je proširenje svojstava elemenata jezika UML. Ona dozvoljava da se svakom elementu modela pridruži proizvoljna informacija. Dodavanje označene vrednosti nekoj instanci apstrakcije T je zapravo implicitno dodavanje (virtuelnog, nepostojećeg) atributa apstrakciji T u metamodelu.
- *Ograničenje* (engl. *constraint*) je proširenje semantike jezika UML koje dozvoljava uvođenje novih ili izmenu postojećih pravila u metamodelu.

Samo postojanje ovih elemenata proširivanja jezika u UML standardu potvrđuje značaj metamodelovanja. Naime, već je sasvim jasna i opšteprihvaćena činjenica da ni jedan jezik, odnosno metamodel, koliko god bio složen, ne može pokriti sve specifične domene modelovanja. Potrebno je zato obezbediti da sam korisnik definiše svoje elemente metamodela za specifičan domen od interesa.

Autori jezika UML ističu značaj i opravdavaju ovaj implicitni pristup proširivanju jezika UML time što je skup koncepata mali, ali dovoljno generalan, pa je zato lako razumljiv. Sa druge strane, način proširivanja metamodela je kontrolisan, pa se teže može desiti da različiti korisnici uvođe proizvoljno svoje koncepte sa rizikom da drugi ne razumeju njihovu semantiku i da složenost metamodela izmakne kontroli.

Međutim, postoje i veoma oštre kritike ovakvog implicitnog pristupa metamodelovanju [C-Pla97]. Nedostaci ovog pristupa, kod koga su zapravo i metamodelovanje i modelovanje pomešani na M1 nivou četvoroslojne arhitekture metamodelovanja u odnosu na eksplicitno metamodelovanje su sledeći [C-Pla97]:

- semantika novih elemenata metamodela definisanih implicitno nije sasvim precizno definisana;
- ovaj pristup značajno otežava konstrukciju okruženja za modelovanje; u prilog ovoj tvrdnji možda ide i činjenica da još uvek nije široko zastupljen nijedan alat za modelovanje koji u potpunosti podržava navedene mehanizme proširivosti jezika UML, dok su alati za (eksplicitno) metamodelovanje već odavno rašireni i stabilni;

- korisnici su prinuđeni da uče nove apstraktne i prilično fluidne koncepte kao što su stereotipovi, označene vrednosti i ograničenja, dok su im često mnogo bliži i jasniji koncepti koje sami eksplicitno definišu u metamodelu, ukoliko im je to omogućeno;
- pošto nije dozvoljena eksplicitna izmena metamodela, stvara se utisak da je taj metamodel fiksiran, što zapravo nije slučaj; eksplicitnost često smanjuje konfuziju;
- ako je UML metamodel zapisan u nekom perzistentnom medijumu, npr. u repozitorijumu, onda ovi mehanizmi proširivosti nemaju formalnu predstavu u tom repozitorijumu; drugim rečima, mehanizmi proširivosti nisu eksplicitan nego implicitan način za proširenje metamodela, pa se ne mogu iskoristiti za definisanje standarda interoperabilnosti alata za modelovanje koji se oslanjaju na različite metamodele.

Generisanje izlaznih formi u alatima za modelovanje

Praktično svi alati za modelovanje i metamodelovanje omogućavaju korisniku da definiše način generisanja izlaznih formi. Međutim, način na koji to čine razlikuje se donekle od jednog do drugog alata. Većina alata nudi mogućnost pisanja programa (skriptova) na specifičnom programskom jeziku koji generišu izlazne forme. Samo mali broj alata podržava i drugačije pristupe, kao što su korišćenje šablona ili apstraktnih specifikacija. Pregled podrške generisanju izlaznih formi u analiziranim alatima dat je u Tabeli 3.2

Alat	Proizvođač	Tip alata	Podrška generisanju izlaza
<i>Graphical Designer</i> [D-ASTW]	Advanced Software Technologies, Inc.	Alat za metamodelovanje	Samo za tekstualni izlaz. Jezik za skriptovanje nalik na C.
<i>IPSYS ToolBuilder</i> [D-LSLW]	Lincoln Software, Ltd.	Alat za metamodelovanje	Nema informacija.
<i>MetaEdit+ Method Workbench</i> [D-MCCW]	MetaCase Consulting	Alat za metamodelovanje	Specifični jezik za skriptovanje. Ugrađeni generatori koda za Smalltalk, C++, Java, Delphi (Object Pascal), SQL, i CORBA IDL.
<i>WithClass Scripting Tool</i> [D-MGSW]	MicroGold Software, Inc.	Podesivi alat za modelovanje	Tekstualni izlaz baziran na šablonima.
<i>Alfabet</i> [D-MipW]	mip GmbH	Alat za metamodelovanje	Nema podrške.
<i>Rational Rose</i> [D-RSCW]	Rational Software Corporation	Podesivi alat za OO modelovanje	Skripting jezik baziran na VBA. Ugrađeni generatori koda za C++, Java, SQL, Visual Basic i CORBA IDL
<i>ConceptBase</i> [C-Hah96]	Technische Hochschule (RWTH) Aachen, Germany	Objektna baza i dinamički OO jezik	Skripting jezik za definiciju i manipulaciju objektima (TELOS).
<i>MetaView</i> [D-UAIW]	University of Alberta	Alat za metamodelovanje	Podrška višestrukim pogledima (modelima iz različitih domena) preko PCTE standarda (prošireni ER metamodel).
<i>Multigraph Architecture</i> [D-VUnW]	Vanderbilt University	Alat za metamodelovanje	Apstraktne specifikacije bazirane na vizitorima.

Tabela 3.2: Podrška generisanju izlaznih formi u analiziranim alatima

Generisanje izlaznih formi programski (pomoću skriptova)

Najveći broj alata za modelovanje i metamodelovanje omogućava pristup koji je opisan u glavi "Definicija problema". Ovaj pristup sastoji se u pisanju programa na nekom specifičnom programskom jeziku za skriptovanje. Ti programi obilaze strukturu modela koji je definisan u alatu i proizvode izlaznu formu. Naravno, strukturu izvornog modela čine instance apstrakcija iz metamodela, pa je za pisanje ovakvog programa potrebno poznavati taj metamodel.

Primer dela skripta pisanog na jeziku za skriptovanje alata *Rational Rose* [D-RSCW] koji generiše kod za operaciju klase na jeziku Visual Basic izgleda ovako:

```
Private Function generateOperation (anOper As Operation) As Boolean
    Dim status As Boolean
    status = True

    Dim visibility As String
    visibility = getVBVisibilityForOperation(anOper.ExportControl)
    Dim type As String
    type = anOper.ReturnType
    Dim isSub As Boolean
    isSub = False
    If Trim(type) = "" Then isSub = True

    OutputCode visibility
    If isSub Then
        OutputCode " Sub "
    Else
        OutputCode " Function "
    End If

    OutputCode anOper.Name + " ("
    tot% = anOper.Parameters.Count
    For i% = 1 To tot%
        status = status And generateParameter(anOper.Parameters.GetAt(i%))
        If Not i%=tot% Then OutputCode ", "
    Next i%
    OutputCode ")"

    If Not isSub Then OutputCode " As " + type
    OutputCode NewLine

    status = status And generateOperationBody(anOper)

    If isSub Then
        OutputCode "End Sub"
    Else
        OutputCode "End Function"
    End If
    OutputCode NewLine + NewLine

    generateOperation = status
End Function
```

Programski jezik koji se koristi za skriptovanje je najčešće specifičan za proizvođača. U poslednje vreme to je često neka verzija jezika Basic, najčešće Microsoft Visual Basic for Applications (MS VBA), jer se VBA nameće kao nezvanični standard. Međutim, mnogi alati za metamodelovanje koriste sasvim specifične jezike jer su konstruisani znatno davnije od pojave VBA. Neke najbitnije karakteristike i ujedno i nedostaci ovih jezika jesu:

- Nijedan od njih nije objektno orijentisan. Prema tome, korisniku nisu dostupni moderni koncepti apstrakcije, enkapsulacije, nasleđivanja i polimorfizma kao u objektnim jezicima. Ovo je ozbiljno ograničenje u pisanju složenijih generatora izlaza.
- Na raspolaganju je isključivo prosta dekompozicija samo na procedure. Po pravilu nema većih organizacionih jedinica, čak ni modula, a kamoli hijerarhijske dekompozicije. Ovo značajno smanjuje preglednost generatora izlaza većeg obima.

Obzirom da je generator izlazne forme program na datom programskom jeziku, sama konstrukcija alata ne nameće ograničenja u pogledu kog je tipa odredišni domen. Drugim rečima, moguće je u tom programu formirati model iz sekvencijalnog domena, ali i model iz nekog objektnog domena, tj. instance i veze iz odredišnog domena pod uslovom da je njegov metamodel takođe dostupan. Ipak, većina pristupa generisanju izlaznih formi prilagođena je prvenstveno generisanju tekstualnog izlaza. Zbog toga odgovarajući programski jezici poseduju solidnu podršku za kontrolu formata tekstualnog izlaza, rad sa datotekama i slično. Bez obzira na sve to, mnogi nedostaci ovog pristupa koji su navedeni u glavi "Definicija problema" znatno smanjuju njegovu upotrebnost vrednost.

Generisanje izlaznih formi pomoću šablona

Jedan od nedostataka prethodnog pristupa u slučaju generisanja tekstualnog izlaza je zamorno i nepregledno definisanje formata izlaznog teksta. Ukoliko je taj format složen, ili je mnogo bitniji od same sadržine izlazne forme, onda je pristup pravljenja te forme pomoću programa zamoran i podložan greškama. U tom slučaju može biti mnogo pogodniji pristup gde se sama izlazna forma definiše kao parametrizovani i formatizovani tekstualni šablon u kome se pojavljuju:

- konstantni delovi teksta u odgovarajućem formatu,
- parametri koji referišu delove izvornog modela, a koji će u procesu generisanja izlaza biti zamenjeni konkretnim vrednostima očitanim iz izvorišnog modela i
- kontrolne sekvence koje kontrolišu generisanje pojedinih delova šablona.

Primer alata koji podržava ovaj pristup je alat *WithClass Scripting Tool* [D-MGSW]. Sličan pristup pojavljuje se i pod nazivom *arhitipa* (engl. *archetype*) [F-Sh197]. Primer šablona za generisanje deklaracije klase na jeziku C++ u alatu *WithClass* je sledeći:

```
class CLASS[NO_RETURN NO_REPEAT:
    NO_REPEAT public BASE_CLASS,DELETE_LAST_SYMBOL]
    CLASS_LIBRARY_BASE_CLASS
{ [ATTRIBUTE_TYPE ATTRIBUTE_NAME$; ]
  [ASSOCIATION_ONE_CLASS$* ASSOCIATION_ONE_NAME$; ]
  [AGGREGATION_ONE_CLASS AGGREGATION_ONE_NAME$; ]
  [ASSOCIATION_MANY_CLASS$* ASSOCIATION_MANY_NAME LITERAL_SYMBOL[ 6
LITERAL_SYMBOL]; ]
  [AGGREGATION_MANY_CLASS AGGREGATION_MANY_NAME LITERAL_SYMBOL[ 6
LITERAL_SYMBOL]; ]

public:

[  CPP_OPERATION_VIRTUAL CPP_OPERATION_STATIC OPERATION_RETURN_TYPE
OPERATION_NAME (CPP_OPERATION_PARAMETERS) CPP_OPERATION_CONSTANT
CPP_OPERATION_PURE_VIRTUAL;
]

};
```

U ovom primeru uočavaju se delovi koji predstavljaju konstantan tekstualni sadržaj (tekst pisan malim slovima bez specijalnih oznaka), opcioni delovi (unutar uglastih zagrada), kao i reference na sadržaj izvorišnog modela koje će biti zamenjene konkretnim vrednostima (reči pisane velikim slovima). Te reference su ključne reči u datom jeziku i odnose se na tačno definisane elemente modela (npr. `CLASS` ili `ATTRIBUTE_NAME` u prethodnom primeru).

Važno je primetiti da kod ovog pristupa treba obezbediti elementarne kontrolne strukture za generisanje delova izlaznog teksta – uslove i petlje. Takođe treba razlikovati te kontrolne strukture u šablonu od konstantnih delova šablona. Sve ovo mora se predvideti u sintaksi šablona, dok generator izlaza mora da prepozna i interpretira te elemente.

Prema tome, šabloni predstavljaju zapravo pojednostavljenje pisanja programskih generatora izlaznog teksta u kome su kontrolne programske strukture sekundarni a formatizovani šablon teksta primarni element. To je upravo suprotan pristup od prethodnog, kod koga se o formatu izlaza mora brinuti programski. Kao ilustracija ove razlike može da posluži sledeći primer dela koji generiše zaglavlje potprograma na jeziku Visual Basic. Za programski pristup, kôd generatora izgleda približno ovako:

```
Dim type As String
type = anOper.ReturnType
Dim isSub As Boolean
isSub = False
If Trim(type) = "" Then isSub = True

If isSub Then
    OutputCode " Sub "
Else
    OutputCode " Function "
End If

OutputCode anOper.Name + " ("
tot% = anOper.Parameters.Count
For i% = 1 To tot%
    status = status And generateParameter(anOper.Parameters.GetAt(i%))
    If Not i%=tot% Then OutputCode ", "
Next i%
OutputCode ")"
```

Za pristup preko šablona u nekoj zamišljenoj sintaksi isti primer može da izgleda ovako (kontrolne strukture su unutar uglastih zagrada, a reference između znakova \$ kada su izvan kontrolnih struktura):

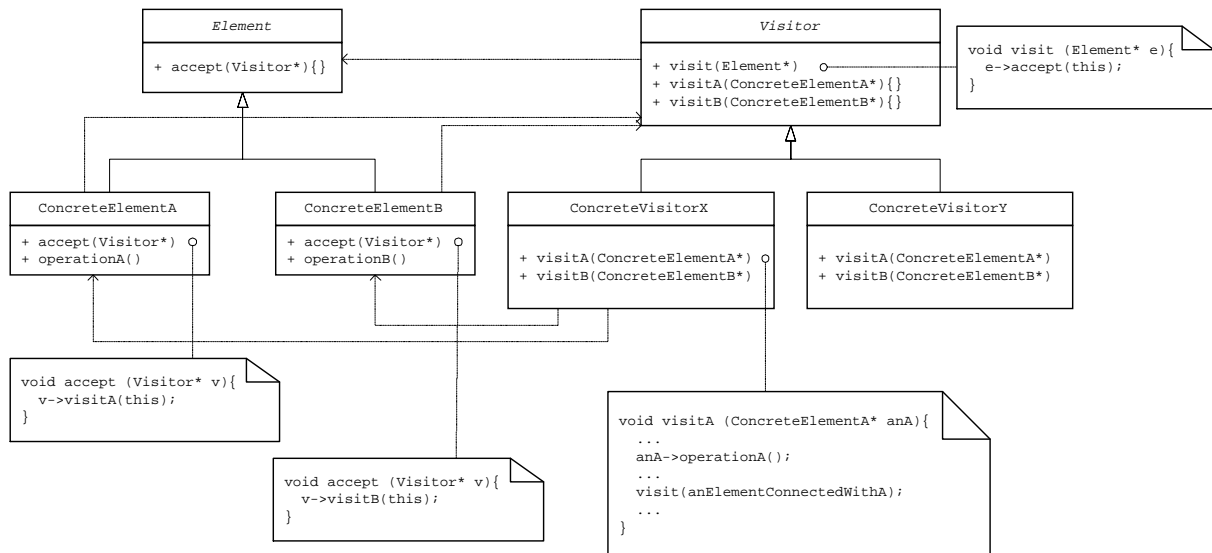
```
[ IF Trim(anOper.ReturnType)="" ]Sub[ ELSE ]Function[ ENDIF] $anOper.Name$ ( \
[ FOREACH param IN anOper.Parameters SEP " , " ] \
$param.Name$ As $param.Type$ [ ENDFOR ] \
) [ IF Not Trim(anOper.ReturnType)="" ]As $anOper.ReturnType$[ ENDIF]
```

Za demonstrativni primer konačnih automata, šablon deklaracije apstraktne klase State može da izgleda ovako:

```
class $fsm.name$State {
public:
    $fsm.name$State ($fsm.name$* fsm) : myFSM(fsm) {}

    [ FOREACH ev:Event IN fsm.myEvents ]
    virtual $fsm.name$* $ev.name$ () {return this;}
    [ ENDFOR ]

    virtual void entry () {}
}
```



Slika 3.1: Projektni obrazac *Visitor* obezbeđuje fleksibilno rešenje za problem definisanja strategije obilaska strukture modela.

```
virtual void exit  () {}

protected:
    $fsm.name$* fsm () const { return myFSM; }
private:
    $fsm.name$* myFSM;
};
```

Ukoliko se želi još bogatija podrška generisanju izlaza, pored kontrolnih struktura mogu se uvesti i koncepti lokalnih promenljivih šablona ili podstrukture (analogne pozivima potprograma u programskom pristupu). Tada se zapravo ovaj pristup značajno usložnjava, a šabloni mogu da postanu i nepregledniji od njima ekvivalentnih programa.

Konačno, iz međusobnog odnosa ova dva pristupa sledi i zaključak o pogodnosti upotrebe jednog ili drugog. Ukoliko je bitan ili složen format izlaznog teksta, dok je sam način izvlačenja informacija iz izvorišnog modela jednostavan (nema složene kontrole), onda je pogodniji pristup pomoću šablona. Ukoliko je pak način obilaska izvorišnog modela složen i postoji potreba za složenijim kontrolnim strukturama i dekompozicijom postupka generisanja izlaza, onda je programski pristup u prednosti.

Problem obilaska strukture i projektni obrazac *Visitor*

Problem generisanja izlaza može se posmatrati na sledeći način [F-Kar99]. Pod uvedenom pretpostavkom da se radi o objektnom domenu, izvorišni model je graf koji se sastoji od čvorova različitih tipova. Taj graf treba obilaziti odgovarajućom strategijom i za svaki čvor na koji se naiđe izvršiti odgovarajuću operaciju (u ovom slučaju generisanje izlaza). U opštem slučaju, operacije koje se vrše nad čvorom mogu biti različite: generisanje različitih vrsta izlaza (dokumentacija, kôd), kontrola konzistentnosti modela u tom čvoru itd. Isto tako i strategije obilaska mogu da se menjaju u zavisnosti od potrebnih operacija. Svi ovi parametri (tip čvora, strategija obilaska i vrsta operacije) trebalo bi da budu međusobno nezavisni.

Fleksibilno rešenje za ovaj često sretan problem nudi se u vidu projektnog obrasca (engl. *design pattern*) pod nazivom *Visitor* [B-Gam95] koji je prikazan na slici 3.1. Kako bi se obezbedila fleksibilnost izbora strategije obilaska i vrste operacije u zavisnosti od tipa čvora, ovi parametri nisu pomešani sa samom strukturom, nego izdvojeni u posebne delove koda (hijerarhije klasa). Na taj način definicija strukture ostaje sasvim nezavisna od parametara obilaska, pa se nove strategije obilazaka i operacije prilikom obilaska mogu definisati i proširivati nezavisno od strukture. Takođe se i struktura može proširivati bez ugrožavanja već postojećih strategija obilazaka.

Za obrazac *Visitor* prikazan na slici 3.1, struktura je predstavljena hijerarhijom klasa na levoj strani. Klasa `Element` predstavlja neku osnovnu (najčešće apstraktnu) klasu koja je koren te hijerarhije. Konkretno klase predstavljene su kao `ConcreteElementA` i `ConcreteElementB`. U ovim klasama postoje operacije koje su specifične za semantiku neke obrade, poput `operationA()` i `operationB()`.

Sa druge strane, strategije obilazaka izdvojene su u posebnu hijerarhiju klasa koja u osnovi ima apstraktnu klasu `Visitor`. U ovoj klasi definisane su polimorfne operacije za svaki potrební konkretni tip u strukturi; u primeru su to operacije `visitA()` i `visitB()`. Konkretno strategije obilazaka definišu se unutar konkretnih klasa izvedenih iz klase `Visitor` (u primeru klase `ConcreteVisitorX` i `ConcreteVisitorY`). Unutar redefinisanih operacija ovih klasa vrše se odgovarajuće obrade za čvor strukture koji se obilazi, kao i dalji obilazak čvorova. Operacija `visit()` apstraktne klase `Visitor` služi samo da razreši dvostruki polimorfizam: izvršavanje konkretne operacije u zavisnosti od konkretne klase iz strukture i konkretne strategije obilaska.

Na ovaj način struktura se može lakše proširivati. Kada se doda nova konkretna klasa `ConcreteElementC` u strukturi, potrebno je u njoj samo definisati operaciju `accept()` na potpuno isti generički način, kao što je prikazano na slici (što se može uraditi i automatski), i u klasu `Visitor` dodati odgovarajuću operaciju `visitC()`. Sa druge strane, ukoliko se želi nova strategija obilaska `z`, nije potrebno uopšte menjati strukturne klase, već samo definisati konkretnu izvedenu klasu `ConcreteVisitorZ` i u njoj redefinisati odgovarajuće operacije. Treba primetiti da ukoliko operacije klase `Element` i `Visitor` nisu apstraktne nego imaju (bar prazno) podrazumevano ponašanje kao na slici, onda izvedene klase ne moraju da redefinišu sve operacije, već samo one koje su im bitne.

Ovaj projektni obrazac veoma je značajan za kontekst ovoga rada jer su strategija obilaska strukture kao i fleksibilnost izbora te strategije i operacija koje se vrše prilikom obilaska elementa strukture veoma značajni faktori u procesu generisanja izlazne forme. Ovaj obrazac zato predstavlja jedan od najpraktičnijih i najznačajnijih elemenata u rešavanju datog problema. Njegova primarna snaga je u dobroj dekompoziciji za slučajeve kada je programski pristup generisanju izlaza najpogodniji, što je najčešće slučaj kod generisanja tekstualnog izlaza.

Na žalost, kako nijedan programski jezik za podršku generisanju izlaza u analiziranim alatima nije objektno orijentisan, ovi alati ne omogućavaju direktnu implementaciju projektnog obrasca *Visitor*. Jedini način da se on iskoristi je da se generisanje izlaza konceptualno osmisli i dekomponuje korišćenjem ovog obrasca, a zatim implementira konceptima proceduralnog programiranja. To podrazumeva pravljenje odgovarajućih procedura koje su ekvivalent navedenim polimorfnim operacijama u ovom obrascu. Ovakav pristup znatno otežava implementaciju i smanjuje preglednost generatora izlaza.

Generisanje izlaznih formi pomoću apstraktnih specifikacija

Obzirom na svoju pravilnost, implementacija obrasca *Visitor* se može dobiti i automatski iz konciznih apstraktnih specifikacija koje definišu samo njegove suštinske elemente:

- strukturu izvorišnog modela; ovaj aspekt se zapravo odnosi na sam metamodel i način njegove definicije, pa nije uži predmet posmatranja u ovom kontekstu;
- način obilaska; ovaj element definiše koje čvorove grafa (objekte) treba sledeće obići za dati čvor (objekat) tipa *X*;
- operaciju koju treba izvršiti prilikom posete datog čvora (objekta) tipa *X*.

Razvojni tim sa Vanderbilt Univerziteta (SAD) je tek nedavno u okviru svog alata za metamodelovanje *Multigraph Architecture* [D-VUnW] predložio način da se ovi elementi definišu pomoću konciznih, apstraktnih, ali tekstualnih specifikacija [F-Kar99], a zatim i da se generator izlaza implementira automatski pomoću obrasca *Visitor*.

Kod ovog pristupa, pored metamodela, potrebno je zadati i tzv. *specifikacije obilaska* (engl. *traversal specifications*) i tzv. *specifikacije posete* (engl. *visitor specifications*). Specifikacija obilaska definiše se za neki tip *X* i njome se ukazuje na skup objekata koje treba dalje obići prilikom posete objekta tipa *X*. Taj skup objekata mora biti dostupan iz objekta tipa *X* i zadaje se nekim iskazom koji određuje datu navigaciju kroz model. Na primer, specifikacija obilaska:

```
from Compounds to {locals, blocks, flows}
```

definiše da se iz objekta tipa *Compounds* prelazi na objekte koji su dostupni iz tog objekta preko veza asocijacija *locals*, *blocks* i *flows*.

Specifikacija posete definiše šta treba uraditi prilikom posete objekta tipa *X*. Tu su na raspolaganju dve mogućnosti: obavljanje neke specifične korisnički definisane obrade ili dalji obilazak objekata kako je to definisano specifikacijom obilaska za dati tip. Bilo koji od ovih delova može da se izostavi, a i njihov redosled je proizvoljan, što može biti bitno. Na primer, specifikacija posete:

```
at Compound <<User-Code-1>> traverse <<User-Code-2>>
```

definiše da se prilikom posete objekta tipa *Compound* najpre izvršava korisnički definisani kôd *User-Code-1*, zatim vrši obilazak ostalih objekata (ključna reč *traverse*) kako je to definisano odgovarajućom specifikacijom *from Compound*, i najzad korisnički kôd *User-Code-2*. Korisnički kôd je bilo koji C++ kôd. Iz ovakvih specifikacija generišu se C++ klase koje u potpunosti implementiraju generator izlaza pomoću obrasca *Visitor*.

Ovakve specifikacije su veoma koncizne i praktične. Njihova pogodnost se zasniva na pogodnosti samog obrasca *Visitor*, s tim što one zahtevaju mnogo manje truda jer se njima definišu samo suštinski elementi, dok se implementacija dobija automatski. Međutim, ovaj pristup se ne bavi time šta se zapravo pravi kao odredišni model. U slučaju da je to tekstualni izlaz, one su sasvim pogodne. Ukoliko je pak odredišni domen objektni, one imaju iste nepogodnosti kao i ostali pristupi. Ove nepogodnosti biće diskutovane kasnije.

Još jedan pristup apstraktnim specifikacijama strategije obilaska strukture predstavlja tzv. *adaptivno programiranje* (engl. *adaptive programming*) [B-Lie96]. Iako se ovaj pristup ne bavi problemom transformacije modela ili generisanja izlazne forme, on nudi rešenje za problem obilaska izvorišne objektno strukture koji je jedan od elemenata ukupnog problema transformacije modela. Ovaj pristup zasniva se na opservaciji da su objektno orijentisani programi strukturirani na različit način od tradicionalnih, proceduralnih. Naime, uočeno je da

se funkcionalnost u objektno orijentisanim programima postiže *kolaboracijama* između objekata koji međusobno razmenjuju poruke da bi ispunili zahtevano ponašanje. Složeno ponašanje se implementira pomoću jednostavnijih odgovornosti koje su raspoređene po klasama kojima objekti koji učestvuju u kolaboracijama pripadaju. U adaptivnom programiranju, struktura sistema se predstavlja pomoću uobičajenih klasnih dijagrama, dok se ponašanje definiše načinima propagacije operacija kroz strukturu. Iako je ovaj pristup zastupljen u praktično svim objektnim metodologijama pomoću neke vrste dijagrama interakcije [B-Boo99], ovde se propagacija operacija definiše pomoću tzv. *dijagrama strategije obilaska* strukture (engl. *traversal strategy graph*). Ovaj dijagram je zapravo specifikacija propagacije na veoma visokom nivou apstrakcije u kome su implementacioni detalji izostavljeni. Ovaj dijagram predstavlja proširenje dijagrama klasa u kome se definiše kako se neka operacija za objekte neke klase propagira na objekte drugih klasa sa kojima je ova u asocijaciji. Na primer, ukoliko je klasa A u asocijaciji sa klasom B a ova u asocijaciji sa klasom C, onda se može reći prosto da se od objekta klase A neka operacija propagira na objekte klase C, preko veza asocijacija A-B i B-C. Pored toga, može se i eksplicitno definisati kôd koji se izvršava prilikom obilaska objekta date klase. Iz ovakvih apstraktnih specifikacija može se automatski dobiti izvršni kôd koji implementira zahtevano ponašanje.

Ad-hoc pristupi

Kako je modelovanje u specifičnom domenu izuzetno široko rasprostranjeno i podržano softverskim alatima, funkcija generisanja izlaznih formi je takođe u većini tih alata implementirana na neki sasvim specifičan način. Drugim rečima, u praksi su najraširenije ad-hoc metode generisanja izlaza, što znači da je učinjen mali ili nikakav napor da se taj postupak sistematizuje.

Ipak, u nekim specifičnim domenima postoje pokušaji sistematizacije rešavanja ovog problema, ali usko vezano samo za te domene. Kao primer ovde će biti naveden samo postupak transformacije domena objektno orijentisanog modelovanja softvera u relacioni model baza podataka, i to u oba smera, opisan u [F-Pap95]. (Ovaj slučaj biće prikazan kasnije u ovom radu samo kao jedan od primera upotrebe metode koja je ovde predložena.) Ovaj postupak koristi poseban međumodel u koji se dati izvorišni model (objektno orijentisani ili relacioni) transformiše. Iz ovog međumodela generiše se potom odredišni model. Ovaj međudomen (tj. njegov metamodel) opisan je precizno u [F-Pap95], ali je on sasvim specifičan za ovaj slučaj.

Ovaj pristup je veoma interesantan zbog toga što sadrži neke rudimentarne elemente pristupa koji je ovde predložen. Prvo, on predstavlja sistematizovani pristup transformaciji modela. Drugo, on koristi međumodel koji znatno olakšava definisanje transformacija.

Na žalost, ovaj pristup je posebno namenjen navedenom preslikavanju, pa je daleko od generalnog. Međudomen je ovde definisan fiksnim metamodelom. Dalje, taj metamodel je definisan pomoću usmerenog acikličnog grafa nasleđivanja klasa, a ne koristi ostale objektno orijentisane koncepte. Drugim rečima, njegov meta-metamodel je daleko od standardnih objektnih konceptata, a naročito jezika UML. Transformacije se sprovode primenom fiksnog skupa pravila koji se ne može prilagoditi posebnim potrebama korisnika. Najzad, ta pravila i sam algoritam transformacije veoma su složeni.

Ostali relevantni pristupi

Postoje i značajna istraživanja vezana za modelovanje u specifičnim domenima koja se ne oslanjaju na pretpostavke o objektnim metamodelima, uglavnom zbog toga što su rađena mnogo pre uvođenja objektnih koncepata. Naime, potreba za specifičnim jezicima prilagođenim određenim domenima pojavila se odavno, pa su takvi jezici nastajali zajedno sa ostalim programskim jezicima opšte namene. Isto tako su zajedno sa tekstualnim specifičnim jezicima nastajali i prevodioci za te jezike, kao i generički transformatori između različitih jezika. Slično, istraživanja u oblasti vizuelnih jezika takođe se odavno bave generičkim konstruisanjem prevodilaca za te jezike. Najzad, interesantna ideja tzv. *intencionalnog programiranja* (engl. *intentional programming*) koju promoviše Microsoft Research takođe je u indirektnoj vezi sa temom ovog rada. U ovom poglavlju su ovi relevantni pristupi ukratko komentarisani.

Transformatori bazirani na gramatikama

Prva i daleko najraširenija primena obilaska grafa (kao vrste strukture izvorišnog modela) i izvršavanja operacija za njegove čvorove je proces generisanja koda u tradicionalnim programskim prevodiocima (engl. *compilers*) [A-Aho86]. Literatura iz ove oblasti predstavlja ogroman izvor efikasnih algoritama obilazaka i transformacija tih struktura. Ovde su samo neki osnovni aspekti i relevantna rešenja ukratko komentarisani.

Tradicionalni programski prevodioci

Prvi zadatak programskog prevodioca je da analizira ulaznu tekstualnu sekvencu i detektuje da li ona zadovoljava pravila sintakse datog jezika. Tokom te analize, prevodilac izgrađuje internu formu – sintaksno stablo izvođenja čiji su čvorovi tipizirani. Svaki čvor predstavlja instancu tačno jednog neterminalnog ili terminalnog simbola gramatike. Ova interna struktura služi da se nad njom vrše različite analize i transformacije (provera semantike, optimizacija koda i slično). Prema tome, ovakav programski prevodilac vrši transformaciju izvornog, tekstualnog modela u internu predstavu pomoću tipiziranog stabla koje se može smatrati posebnim slučajem modela iz objektnog domena.

Drugi zadatak programskog prevodioca je da iz ove interne forme generiše kôd koji se opet može smatrati izlaznom formom. Ovaj zadatak se svodi na već opisani problem obilaska izvorne strukture (u ovom slučaju stabla) i generisanja odgovarajućeg sekvencijalnog ili nekog drugog izlaza. Preciznije rečeno, prevodioci uglavnom ne grade eksplicitno strukturu stabla izvođenja tokom analize ulaznog teksta. Postoje dva osnovna razloga za to. Prvo, takvo stablo bi za iole veće ulazne tekstove bilo izuzetno veliko i nepraktično za smeštanje u memoriju i manipulisanje. Drugo i važnije, to najčešće nije ni neophodno. Naime, različite kategorije gramatika dozvoljavaju da se tokom analize ulaznog teksta implicitno grade čvorovi datog stabla i preduzimaju odmah potrebne operacije, posle čega se dati čvor napušta, oslobađa iz memorije i prelazi se na sledeći. Ovakav pristup moguć je uz podršku različitih formalnih automata (konačnih automata, stek-mašina itd.) koji inherentno svojim unutrašnjim stanjem predstavljaju jedan manji deo stabla izvođenja koji se trenutno gradi i posećuje. Značajan deo teorije prevodilaca upravo se bavi definisanjem vrsta gramatika i njima odgovarajućih formalnih automata [A-Hop69][A-Aho86].

Oblast automatskog generisanja prevodilaca nudi interesantna rešenja strukturiranih specifikacija obilaska grafova. Jedan dobro poznat i široko rasprostranjen pristup su atributivne gramatike (engl. *attributed grammars*) [A-Aho86] koje je predložio Knuth. Atributivna gramatika zapravo povezuje semantičke specifikacije sa sintaksnim pravilima jezika. Ako se pretpostavi da je sintaksa jezika definisana u obliku beskontekstne gramatike, pomoću pravila izvođenja, terminalnih, neterminalnih i startnog simbola, onda stablo izvođenja prikazuje koja su pravila izvođenja primenjena na startni simbol da bi se dobila zadata ulazna sekvenca terminalnih simbola, naravno pod uslovom da je ta ulazna sekvenca sintakсно korektna. Svaki list ovog stabla predstavlja jednu instancu terminalnog simbola kojoj odgovara jedan simbol iz ulazne sekvence, dok čvor koji nije list predstavlja instancu neterminalnog simbola koji je razvijen u podstablo primenom odgovarajućeg pravila. Dalje, svakom simbolu gramatike može se pridružiti niz atributa, tako da svaki čvor u stablu može imati niz vrednosti svojih atributa. Pravila izvođenja definišu način na koji se vrednosti tih atributa izračunavaju. U tim izračunavanjima atributi mogu zavisiti od vrednosti drugih atributa na različite načine. Atribut može biti *nasleđeni* (engl. *inherited*), što znači da se njegova vrednost izračunava pomoću atributa roditeljskog i bratskih čvorova, ili *sintetizovani* (engl. *synthesized*), što znači da se njegova vrednost izračunava pomoću atributa čvorova potomaka. Treba primetiti da ovakva svojstva atributa (nasleđeni ili sintetizovani) implicitno definišu zavisnosti između vrednosti atributa, a da ta zavisnost opet implicitno opisuje način obilaska stabla. Prema tome, specifikacije atributa određuju na koji način stablo treba obilaziti i šta u svakom čvoru treba uraditi (izračunati attribute). Eventualne kružne zavisnosti vode ka neodređenostima, ali su one posledica nekorektnih specifikacija. Pored toga, specifikacija atributa može da sadrži i proizvoljan kôd koji se izvršava prilikom izračunavanja tog atributa odnosno obilaska čvora, pa taj kôd može da proizvodi i odgovarajući izlaz.

Kao zaključak, atributivne gramatike predstavljaju formalizam visokog nivoa apstrakcije kojim se definiše način obilaska izvorne strukture (stabla) i operacije koje se pri obilasku vrše. Međutim, one ipak poseduju mnoge praktične nedostatke [F-Kar99]. Za neku određenu strategiju obilaska nije nimalo trivijalno definisati attribute i njihovo izračunavanje koje će implicitnim zavisnostima obezbediti tu strategiju. Ponekad je potrebno uvoditi i posebne attribute samo da bi se obezbedio odgovarajući poredak pri obilasku. Dalje, referenciranje vrednosti atributa koji se nalaze dalje u stablu izvođenja je problematično. Dakle, implicitnost u ovom pristupu unosi praktična ograničenja. Kada je izvorni domen objektni, mnogo je jednostavnije eksplicitno definisati strategije obilaska, kao što je to opisano u prethodnom poglavlju (obrazac *Visitor* i apstraktne specifikacije). Najzad, kao i do sada, ovaj pristup se isključivo bavi obilaskom izvorišne strukture, dok je struktura odredišnog modela potpuno van interesa.

Transformatori zasnovani na pravilima

Isti problem transformacije modela koji je tema ovog rada, ali u kontekstu tekstualnih jezika, rešava se i pomoću tehnika transformatora baziranih na gramatikama (engl. *grammar-based transformers*) vođenih pravilima (engl. *rule-based techniques*) [F-Gar86][F-Gar92][F-Hab86]. Cilj ovih tehnika je da ulazni model, koji je definisao korisnik na nekom specifičnom jeziku u nekom specifičnom ali tekstualnom domenu, transformiše u model na nekom drugom jeziku. Pri tome, i izvorišni i odredišni domen su tekstualni, jezici su definisani pomoću gramatika, dok se transformacije zasnivaju na definisanim pravilima. Motiv ovakvog pristupa je da se omogući automatsko prevođenje modela na nekom specifičnom jeziku od interesa u model na nekom opštem programskom jeziku za koji već postoji prevodilac u dalje forme, pri čemu je cilj da se ta automatska transformacija što lakše implementira, po mogućstvu uz pomoć alata.

Postupak kod ovih tehnika zasniva se na sledećem [F-Gar92]. Korisnik najpre definiše gramatiku specifičnog jezika za domen od interesa (u terminologiji ovoga rada, definiše zapravo metamodel izvorišnog domena). Dalje, pretpostavlja se da je gramatika odredišnog jezika već definisana. Zatim korisnik, u alatu za podršku, primenjuje niz definisanih operacija nad produkcionim pravilima izvorišne gramatike da bi od nje dobio odredišnu gramatiku. Ključ ovog pristupa je upravo u skupu tih operacija. Kao primer, ovde su navedene samo neke:

- Sužavanje i preimenovanje, koje uključuje: brisanje produkcionog pravila iz gramatike, brisanje nekog pravila iz skupa opcionih pravila zamene nekog neterminalnog simbola, preimenovanje pravila tako da reflektuje potrebe odredišnog domena, i kopiranje pravila kao priprema za kasniju specijalizaciju.
- Restruktuiranje, koje uključuje: brisanje simbola iz produkcionog pravila, promena mesta simbola u pravilu, promena neterminala u terminal i obratno, i još neke složenije operacije.
- Krupne izmene: dodavanje pravila u gramatiku, dodavanje simbola u pravilo i dodavanje pravila u skup opcionih pravila zamene nekog neterminalnog simbola.

Posle niza ovakvih operacija nad izvorišnom gramatikom koje alat beleži, korisnik dolazi do odredišne gramatike. Alat zatim, pomoću definisanog složenog algoritma, generiše transformator koji je u stanju da automatski prevodi tekst na izvorišnom jeziku u tekst na odredišnom jeziku. Taj transformator kao svoj ulaz ima stablo izvođenja dobijeno parsiranjem teksta na izvorišnom specifičnom jeziku, na koje onda primenjuje niz strukturnih transformacija kako bi proizveo stablo izvođenja koje odgovara odredišnoj gramatici.

Nažalost, ovaj algoritam generisanja transformatora ne može da reši sve složene slučajeve transformacije gramatike, pa u mnogima zahteva intervenciju korisnika. Zbog toga ovaj postupak nije potpuno automatizovan i zahteva značajan napor od korisnika.

Iako je cilj ovog pristupa isti kao i cilj pristupa predloženog u ovom radu (to je transformacija modela), postoje brojne razlike. Prvo, ovaj pristup se bavi isključivo tekstualnim modelima. To znači da su metamodeli i izvorišnog i odredišnog domena ovde gramatike. Kod ovog pristupa, najpre se izvorišni sekvencijalni model prevodi u internu strukturu stabla (pomoću tradicionalnih prevodilaca), zatim se ta izvorišna struktura stabla prevodi u drugu, odredišnu strukturu stabla pomoću transformatora, a ona opet u izlazni sekvencijalni model koji će ponovo biti analiziran od strane prevodioca ali za odredišni jezik. Dakle, interni model (ili međumodel) je ovde ponovo stablo, za razliku od opštijeg grafa u objektnim domenima.

Drugo, ovaj pristup zahteva intervenciju korisnika u mnogim slučajevima, dok pristup koji je predložen u ovom radu omogućava preslikavanje domena uz potpuno automatsko generisanje transformatora. Treće, definisanje gramatika za domene, a naročito definisanje preslikavanja gramatika različitih domena može biti značajno teže od definisanja objektnih metamodela i njihovog preslikavanja u mnogim slučajevima. U slučaju podesivih alata za modelovanje gde korisnik ne mora sam da definiše metamodel domena, može se očekivati da je korisniku lakše da za složene domene razume metamodel definisan pomoću objektnih koncepata nego pomoću gramatika. To uglavnom važi za sve domene sa vizuelnim notacijama modelovanja. Zbog toga se rešenje predloženo u ovom radu može smatrati kao komplementarno transformatorima baziranim na gramatikama. Za domene koji se lako opisuju gramatikama primereni su i takvi transformatori; za domene koji se lakše opisuju objektnim metamodelima primereniji je pristup predložen u ovom radu.

Intencionalno programiranje

Interesantan pristup koji je posredno u vezi sa temom ovog rada je nova paradigma programiranja koju predlaže Microsoft Research pod nazivom *intencionalno programiranje* (engl. *intentional programming*) [F-Sim96][F-Ait97]. Ključna ideja ovog pristupa je da se (tekstualni) program piše kao skup *intencija*, koje se kasnije automatski transformišu u konkretnu implementaciju. Intencija predstavlja ono što programer želi da "kaže" u određenom specifičnom kontekstu, na način koji ne zavisi od konkretnog programskog jezika. Drugim rečima, intencija predstavlja apstrakciju u specifičnom domenu od interesa, čije se preslikavanje u konkretnu implementaciju na ciljnom programskom jeziku definiše formalno. Intencija predstavlja zapravo proširenje određenog tekstualnog programskog jezika apstrakcijom specifičnog domena.

Na primer, jedno proširenje jezika C može da se definiše sledećom intencijom. Ako je x promenljiva tipa `int`, onda se može definisati da izraz:

```
(x=x+3) min 5
```

treba da se transformiše u sledeći izraz na standardnom jeziku C (tačnije, ovo nije ispravan izraz na standardnom jeziku C, ali ovo može da se realizuje pomoću nešto više složenijih intencija):

```
(
  int left = (x=x+3),
  int right = 5,
  (left<right) ? left : right
)
```

Kada se izvorni tekstualni program na jeziku proširenom intencijama sintaksno analizira i predstavi internom formom pomoću stabla izvođenja, onda se intencije (tačnije njihove instance) pojavljuju kao čvorovi u ovom stablu. Posle toga, ovakvo početno stablo se transformiše nizom koraka, pri čemu se svaki korak sastoji u izvršavanju operacije transformacije (nazvane *xmethod*) koja je definisana za datu intenciju. Ta operacija primenjuje skup elementarnih transformacija stabla koja su podržana u datom okruženju. Ovaj pristup definiše takođe i pravila uređenja ovih operacija kao i druge relevantne detalje. Posle primene ovih operacija dobija se stablo izvođenja koje odgovara ciljnom programskom jeziku i iz koga se posle može generisati mašinski kôd.

Prema tome, intencionalno programiranje je takođe jedan pristup transformaciji modela i način da se definiše preslikavanje domena. Međutim, kontekst je sasvim drugačiji od konteksta ovoga rada. Ovde se, naime, radi samo o tekstualnim programskim jezicima baziranim na gramatikama, čijim se parsiranjem dobija stablo izvođenja. Preslikavanje domena se definiše pomoću operacija transformacija tog stabla. Zbog toga se ovaj pristup može svrstati u oblast transformatora baziranih na gramatikama, pa svi komentari koji su do sada izneseni za njih važe i ovde. Detaljnija diskusija o odnosu intencionalnog programiranja i ostalih tehnika transformacija jezika baziranih na gramatikama može se naći u [F-Ait97]. Neka opšta razmatranja prednosti i nedostataka apstraktnog programiranja baziranog na transformacijama modela, pa samim tim i intencionalnog programiranja ali i pristupa definisanog u ovom radu mogu se naći u [F-Sim96].

Vizuelni jezici

Oblast istraživanja koja je takođe u vezi sa temom ovog rada je oblast vizuelnih jezika (engl. *visual language*, VL). Do sada su definisani mnogi vizuelni jezici za modelovanje u specifičnim domenima, ne samo softverskim [E-Bur95][E-Cha95][E-Har87][G-Dia95][G-Sch95][I-Sel94][I-Ste97][J-Liu98]. Potreba za brzim razvojem vizuelnog jezika za specifičan domen dovela je takođe i do razvoja metoda i alata za generički razvoj okruženja za vizuelno programiranje i modelovanje [E-Anl98][E-Cos95][E-Cos97][E-Cox98][E-Min98][E-Rep95][E-Zha98].

Ove metode bitno se međusobno razlikuju u pristupu koji koriste. Jedan od značajnijih pristupa [E-Cos95][E-Cos97] zasniva se na postojanju posebne vrste tzv. *pozicionih gramatika* (engl. *positional grammars*) koje predstavljaju jedno moguće uopštenje tradicionalnih beskontekstnih gramatika na vizuelne jezike. Za takve gramatike definisan je postupak automatskog generisanja parsera iz definicije gramatike [E-Cos97]. Ovaj postupak je takođe proširenje postupaka za tradicionalne gramatike sekvencijalnih jezika [A-Aho86].

Kod ovog pristupa korisnik formira dijagrame (vizuelne rečenice) koji se sastoje od vizuelnih terminalnih simbola (sličica). Okruženje za vizuelno programiranje (vizuelni editor) nudi korisniku raspoloživi skup vizuelnih simbola i raspoloživi skup pozicija na dijagramu na koje je dozvoljeno postaviti te simbole, u skladu sa definisanom gramatikom vizuelnog jezika. Kada se završi izgradnja vizuelne rečenice, aktivira se sintakсни analizator (parser). Ovaj parser analizira ulaznu vizuelnu rečenicu i izgrađuje stablo izvođenja analogno stablu izvođenja kod prevodilaca za sekvencijalne jezike. To stablo je interna strukturna predstava postupka izvođenja vizuelne rečenice pomoću produkcionih pravila vizuelne gramatike.

U opštem slučaju, prilikom interpretacije ulazne rečenice i izgradnje stabla izvođenja može se naići na dvosmislenosti. Postoje tri vrste parsera vizuelnih jezika koji na različite načine rešavaju te dvosmislenosti [E-Cos95]:

- Generalni parseri "metodom iscrpljivanja" izgrađuju sva moguća stabla izvođenja za ulaznu vizuelnu rečenicu. Ovakvi parseri su spori i primereni samo malim gramatikama i jednostavnim rečenicama.
- Fazi (engl. *fuzzy*) parseri izgrađuju samo jedno stablo izvođenja za ulaznu rečenicu, i to ono koje ima najviši stepen sigurnosti prema pravilima fazi logike. Fazi funkcije se pridružuju vizuelnim terminalnim simbolima i produkcionim pravilima, a koriste se za rešavanje dvosmislenosti.
- Parseri za pozicione gramatike rešavaju dvosmislenosti na sledeći način. Svaki put kada parser traži novi ulazni simbol za analizu, on takođe zadaje i očekivanu poziciju tog simbola prema pravilima pozicione gramatike. Ova informacija o poziciji sledećeg simbola u vizuelnoj rečenici smanjuje broj mogućih stabala izvođenja i rešava dvosmislenosti. Ovaj pristup je najefikasniji, ali je moguć samo za pozicione gramatike. On je zapravo veoma sličan pristupu u tzv. "slevo-udesno" gramatikama sekvencijalnih jezika (engl. *left-to-right grammar*), kod kojih se naredni simbol ulazne rečenice nedvosmisleno određuje implicitno, zbog toga što je rečenica sekvenca simbola koji se uzimaju slevo udesno (odavde potiče i naziv). Glavna razlika je što se ovde pravila pozicione evaluacije moraju eksplicitno zadati u gramatici tako da se naredni ulazni simbol može nedvosmisleno odrediti, što je zapravo generalizacija "slevo-udesno" gramatika. Upravo zbog ovako definisane determinisanosti interpretacije ulazne rečenice, pozicione gramatike i omogućuju generalizaciju tradicionalnih metoda parsiranja na vizuelne jezike, kao i analogno proširenje metoda automatskog generisanja parsera iz definicije gramatike [E-Cos97].

Iz date analize slede i zaključci o odnosu vizuelnih jezika sa pristupom na kome počiva ovaj rad. Naime, osnova vizuelnih jezika su, kao što je objašnjeno, vizuelne gramatike

analogne tradicionalnim gramatikama, pa komentari izneseni u odeljku o tradicionalnim programskim prevodiocima u potpunosti važe i ovde. Metode generisanja vizuelnih parsera koncentrišu se na te gramatike koje definišu sintaksu vizuelnih jezika pomoću hijerarhije vizuelnih elemenata (definisane produkcionim pravilima), gde je rekurzija glavni otežavajući faktor. U automatski generisanom okruženju za vizuelno programiranje korisnik unosi simbole na dijagram prilično proizvoljno. Kada je korisnik završio sa unosom simbola, on eksplicitno poziva operaciju interpretacije dijagrama kao ulazne vizuelne rečenice. Uloga okruženja je da tada sintaksno analizira dijagram, proveriti njegovu ispravnost i izgradi internu predstavu pomoću stabla izvođenja. Potom okruženje može da, prilikom obilaska te strukture, pozove korisnički definisane operacije koje proizvode izlaznu formu. Prema tome, jedina suštinska razlika od sekvencijalnih jezika je u otežanom postupku izgradnje interne predstave pomoću stabla. Ona se u ovom slučaju ne dobija iz sekvencijalne, nego implicitno grafolike strukture koja se sastoji iz vizuelnih simbola povezanih pozicionim relacijama. Transformacija te interne strukture u neku drugu formu ponovo nije težište ni kod ovih pristupa.

Sa druge strane, u okruženjima za modelovanje koja se zasnivaju na objektnim metamodelima i koja su predmet interesovanja u ovom radu, korisnik najčešće eksplicitno pravi elemente modela i njihove veze, iako se za unos modela u novije vreme takođe pretežno koriste vizuelni jezici. Drugim rečima, samim tim što korisnik na dijagram postavi neki simbol, okruženje odmah kreira instancu neke apstrakcije i njene veze sa ostalim instancama. Na taj način okruženje u svakom trenutku poseduje internu grafoliku predstavu modela, dok su dijagrami sa vizuelnim simbolima i njihovim pozicionim relacijama samo pogled na tu strukturu. Stoga nema potrebe za posebnom fazom analize i interpretacije vizuelnih rečenica i izgradnje interne predstave.

UML kao vizuelni jezik

Nedavno standardizovani vizuelni jezik za modelovanje pretežno ali ne isključivo softverskih sistema je jezik UML (engl. *Unified Modeling Language*)[B-Boo99]. On podržava formiranje više različitih modela istog sistema, kao i njihovu hijerarhijsku organizaciju na sledeći način. Elementi modela su hijerarhijski organizovani u tzv. *pakete* (engl. *package*) koji ih sadrže. Različiti pogledi na sistem predstavljaju se pomoću dijagrama koji prikazuju samo neke elemente modela koji su od interesa za određeni specifični pogled. Najzad, modeli na različitim nivoima apstrakcije, ili čak u različitim domenima, mogu se formirati u različitim paketima. Međutim, u jeziku UML ne postoji strategija formalne specifikacije automatske transformacije ovih modela.

Predlog tehnike preslikavanja domena u ovom radu oslanjaće se na jezik UML i njegove koncepte modelovanja. UML nudi mnoge koncepte, ali će ovde od najvećeg značaja biti dijagrami objekata (engl. *object diagram*). Dijagram objekata prikazuje skup objekata (kao instanci klasa) i njihove veze koji postoje u sistemu u nekom trenutku vremena. Kako standardni objektni dijagrami nisu dovoljni za potrebe preslikavanja domena, ovde će biti korišćeni standardni mehanizmi proširivosti jezika UML, tačnije stereotipovi i označene vrednosti, koji su objašnjeni ranije.

Analiza i klasifikacija postojećih rešenja

Iako su oblasti u kojima se sreću opisana rešenja veoma raznorodne, pažljivom analizom elemenata tih rešenja mogu se uočiti neki njihovi osnovni parametri pomoću kojih je moguće izvršiti njihovu klasifikaciju. Ti parametri su sledeći:

- Tipovi struktura modela između kojih se vrše transformacije. Naime, iako sve navedene tehnike vrše transformacije između nekih modela iz različitih domena, struktura tih modela je veoma različita. Preciznije rečeno, meta-metamodeli domena koji se preslikavaju su veoma različiti, pa se po njima može izvršiti odgovarajuća klasifikacija.
- Način obilaska izvornih struktura i specifikacija tog načina. Jedan od bitnih elemenata procesa transformacije je strategija obilaska strukture izvorišnog modela. U zavisnosti od tih načina, kao i podrške za apstraktnu specifikaciju načina obilaska, moguće je klasifikovati postojeće metode.
- Način specifikacije transformacija i njihova dekompozicija. Naime, transformacije koje se vrše između modela po pravilu su složene, pa je potrebna dobra podrška za njihovu jednostavnu specifikaciju. Sa druge strane, te specifikacije mogu da budu veoma obimne, pa postoji problem njihove dekompozicije i dobre organizacije.

Ovi najvažniji uočeni parametri metoda biće posebno diskutovani u narednim poglavljima.

Tipovi domena prema strukturi modela

Kao što je u glavi "Definicija problema" već istaknuto, prema tipovima struktura pomoću kojih se interno predstavljaju modeli, domeni se mogu razvrstati u tri kategorije:

- Leksičko-sekvencijalni (tekstualni). Meta-meta model je implicitan i veoma jednostavan, i sadrži samo pojam (tekstualnog) simbola i relaciju sekvence. Modeli su jednostavne sekvence simbola. Ovaj tip domena obuhvata zapravo različite tradicionalne, tekstualne programske jezike i tekstualne jezike za specifične domene.
- Gramatičko-hijerarhijski. Meta-metamodel je sam pojam beskontekstne gramatike. Struktura modela je tipizirano stablo izvođenja.
- Objektno-grafovski. Meta-metamodel sadrži osnovne strukturne objektno orijentisane koncepte. Struktura modela je tipizirani graf.

Prema ovakvoj podeli moguće je izvršiti i analizu i klasifikaciju postojećih metoda. U tabeli 3.3 prikazana je ta klasifikacija za sve parove tipova izvorišnog i odredišnog domena. U tabeli nisu navođene ad-hoc metode koje naravno mogu da postoje za svaki tip, ali samo za posebne domene, jer takav nesistematski pristup nije od opšteg interesa. Osim toga, u opštem slučaju se programski (skriptovanjem) može generisati ne samo sekvencijalni, nego i model sa strukturom grafa, ali je ovaj pristup izostavljen iz odgovarajućih polja tabele iz dva razloga. Prvo, takav pristup nije sistematizovan i predstavlja zapravo ad-hoc programiranje neke transformacije, pa nije od opšteg interesa. Drugo, ni u jednom poznatom okruženju se ne promoviše preslikavanje u druge modele, nego se programski pristup koristi praktično isključivo za generisanje sekvencijalnog izlaza. Kao što će biti pokazano u ovom radu, taj pristup i nije pogodan za takve složenije transformacije u strukturu grafa, nego je i najpogodniji za generisanje samo sekvencijalnih modela.

Odredišni model				
Izvorišni model		Sekvenca	Stablo	Graf
	Sekvenca	(Posredno: sekvenca-stablo-sekvenca)	<ul style="list-style-type: none"> • Tradicionalni prevodioci (parsiranje) 	-
	Stablo	<ul style="list-style-type: none"> • Programski (skriptovanje) • Tekstualni šabloni • Apstraktne specifikacije • Tradicionalni prevodioci (generisanje koda) 	<ul style="list-style-type: none"> • Transformatori bazirani na gramatikama • Intencionalno programiranje 	-
	Graf	<ul style="list-style-type: none"> • Programski (skriptovanje) • Tekstualni šabloni • Apstraktne specifikacije 	<ul style="list-style-type: none"> • Prevodioci za vizuelne jezike 	-

Tabela 3.3: Klasifikacija transformacija prema tipu strukture izvorišnog i odredišnog modela.

U tabeli se takođe može uočiti da za preslikavanje iz sekvence u sekvencu ne postoji metoda za direktnu transformaciju, već se ona u praksi vrši posredno preko transformacija u i iz internih modela strukture stabla. Takođe je bitno uočiti da za neke parove, tačnije za slučaj kada je odredišna struktura graf, uopšte ne postoji podrška. Preciznije, za složenija preslikavanja između grafovskih struktura postoji slabija ponuda metoda, pri čemu su još i te metode prilično složene za razumevanje i implementaciju. Cilj ovog rada je da upravo ove nedostatke ublaži.

Načini obilaska izvornih struktura

Jedan od bitnih elemenata procesa transformacije je strategija obilaska strukture izvorišnog modela i način njene specifikacije. Neki od važnijih uočenih parametara ovog aspekta su sledeći:

- Da li je specifikacija načina obilaska eksplicitna ili implicitna. Na primer, kod atributivnih gramatika, način obilaska je određen zavisnostima između atributa simbola, dok je kod ostalih pristupa on najčešće eksplicitno definisan.
- Da li strategija obilaska dozvoljava rekurziju ili ne. Rekurzivnost je veoma bitan faktor u nekoliko elemenata celog postupka transformacije. Prvo, metamodel (bilo izvorišnog bilo odredišnog domena) može biti definisan rekurzivno. Ovo znači da struktura modela može sadržati sebi izomorfne podstrukture, i to rekurzivno.

Primeri su mnogo. Iole složenije gramatike tekstualnih jezika su najčešće rekurzivne (posle primene nekoliko zamena nekog neterminalnog simbola dobija se iskaz koji sadrži taj isti simbol). Struktura stabla kojom se predstavlja izvođenje u nekoj gramatici je zapravo po svojoj prirodi rekurzivna. Dalje, i metamodel definisan pomoću objektno orijentisanih koncepata takođe može dozvoljavati rekurzivne strukture modela. Primer je projektni obrazac (engl. *design pattern*) *Composite* [B-Gam95]. Strukture ili samo neke podstrukture modela sa takvim metamodelima su opet tipizirana stabla. Na primer, struktura direktorijuma i datoteka u nekom sistemu, ili struktura paketa i elemenata modela na jeziku UML, grupa simbola i pojedinačni grafički simbol u vizuelnim editorima dijagrama i uopšte slične strukture hijerarhijskog pakovanja ili grupisanja elemenata.

Iz ovakvog posmatranja može se zaključiti da se neka beskontekstna gramatika zapravo može prikazati ekvivalentnim objektnim metamodelom koji ima rekurzivnu definiciju (korišćenjem asocijacija i nasleđivanja). Analiza ove tvrdnje prevazilazi obim ovog rada, ali je ona interesantna zbog toga što se zapravo stabla izvođenja mogu smatrati specijalnim slučajem objektnih modela. Naravno, to ni u kom slučaju ne umanjuje izuzetan značaj osnovne namene prevodilaca – provere sintaksne ispravnosti i transformacije sekvencijalnog modela u model sa strukturom stabla koji je pogodniji za dalju manipulaciju.

Ukoliko izvorišni metamodel poseduje ovakvu rekurzivnu definiciju, onda je jako bitno da strategija obilaska takođe dozvoljava rekurziju. Drugim rečima, bitno je da metoda dozvoljava da se za neki tip elementa izvorišnog modela može definisati postupak obilaska elemenata strukture prilikom posete tog elementa, i da se taj isti postupak može obavljati rekurzivno, za svaki element tog tipa u rekurzivnoj strukturi izvorišnog modela.

- Da li strategija obilaska dozvoljava polimorfizam ili ne. Pod tim se ovde podrazumeva da se za neki tip (apstrakciju) u izvorišnom metamodelu može definisati način obilaska, a da se taj način može (ali i ne mora) redefinisati za tipove koji predstavljaju specijalizaciju datog tipa. Naravno, ovo može biti slučaj samo kod meta-metamodela koji podržavaju generalizaciju/specijalizaciju.

Metoda	EksPLICIT- nost	Rekur- zija	Polimor- fizam	Način specifi- kacije	Organizacija i dekompozicija
Programski (skriptovanje)	EksPLICITNO	Zavisno od jezika, uglavnom da	Zavisno od jezika, uglavnom ne	Programski	Zavisno od jezika, uglavnom proceduralna
Tekstualni šabloni	EksPLICITNO	Ne	Ne	Tekstualni	Nema direktne podrške
Obrazac <i>Visitor</i>	EksPLICITNO	Da	Da	Programski	Objektna
Apstraktne specifikacije	EksPLICITNO	Da	Da	Apstraktne specifi- kacije	Objektna
Adaptivno programiranje	EksPLICITNO	Da	Da	Vizuelno	Objektna
Tradicionalni prevodioci	Implicitno	Da	Ne	Preko atributa	Nema direktne podrške
Transformatori bazirani na gramatikama (generisanje koda)	Implicitno	Da	Ne	Ugrađeno u algoritam transfor- macije	Nema direktne podrške
Intencionalno programiranje	EksPLICITNO	Da	Ne	Programski unutar operacija transfor- macija	Nema direktne podrške
Prevodioci za vizuelne jezike (parsiranje)	EksPLICITNO	Da	Ne	Pozicionom gramatikom	Nema direktne podrške

Tabela 3.4: Pregled karakteristika načina obilazaka strukture izvorišnog modela i njihovih specifikacija za analizirane metode

- Način na koji se definiše strategija obilaska za dati tip elementa (programski, apstraktnim specifikacijama, vizuelno i slično).
- Način organizacije i dekompozicije specifikacija obilaska, što je bitan faktor ukoliko su one složene. U ovo je uključena i mogućnost lakog definisanja različitih strategija obilaska za isti metamodel ali različite željene transformacije.

Navedeni aspekti analiziranih metoda sumirani su u tabeli 3.4. Potrebno je uočiti sledeće važne zajedničke karakteristike. Rekurzija je podržana kod svih metoda. Sa druge strane, polimorfizam je podržan samo kod onih metoda koje koriste objektno orijentisane koncepte, odnosno nasleđivanje. Čak ni kod programskog pristupa polimorfizam uglavnom nije podržan, jer jezici za skriptovanje u postojećim okruženjima uglavnom nisu objektno orijentisani. Najzad, podrška dekompoziciji i organizaciji složenih specifikacija je uglavnom slaba, osim kod metoda koje se oslanjaju direktno ili posredno na objektno orijentisani obrazac *Visitor*. Ove nedostatke metoda predložena u ovom radu takođe treba da ispravi.

Načini specifikacije transformacija i njihova organizacija

Treći važan aspekt procesa transformacije modela je način specifikacije samih transformacija i njihova dekompozicija i organizacija. Ovaj aspekt je u većini metoda zapravo blisko povezan i ne može se razdvojiti od prethodno analiziranog aspekta. Naime, metode se uglavnom oslanjaju na pristup da se prilikom posete nekog elementa izvorišnog modela tokom obilaska njegove strukture ujedno vrši i transformacija tog dela modela ili generisanje dela odredišnog modela. Iako je ovo sasvim prirodan pristup, treba ipak uočiti suptilnu razliku između ova dva elementa procesa transformacije, koja će biti razjašnjena u nastavku.

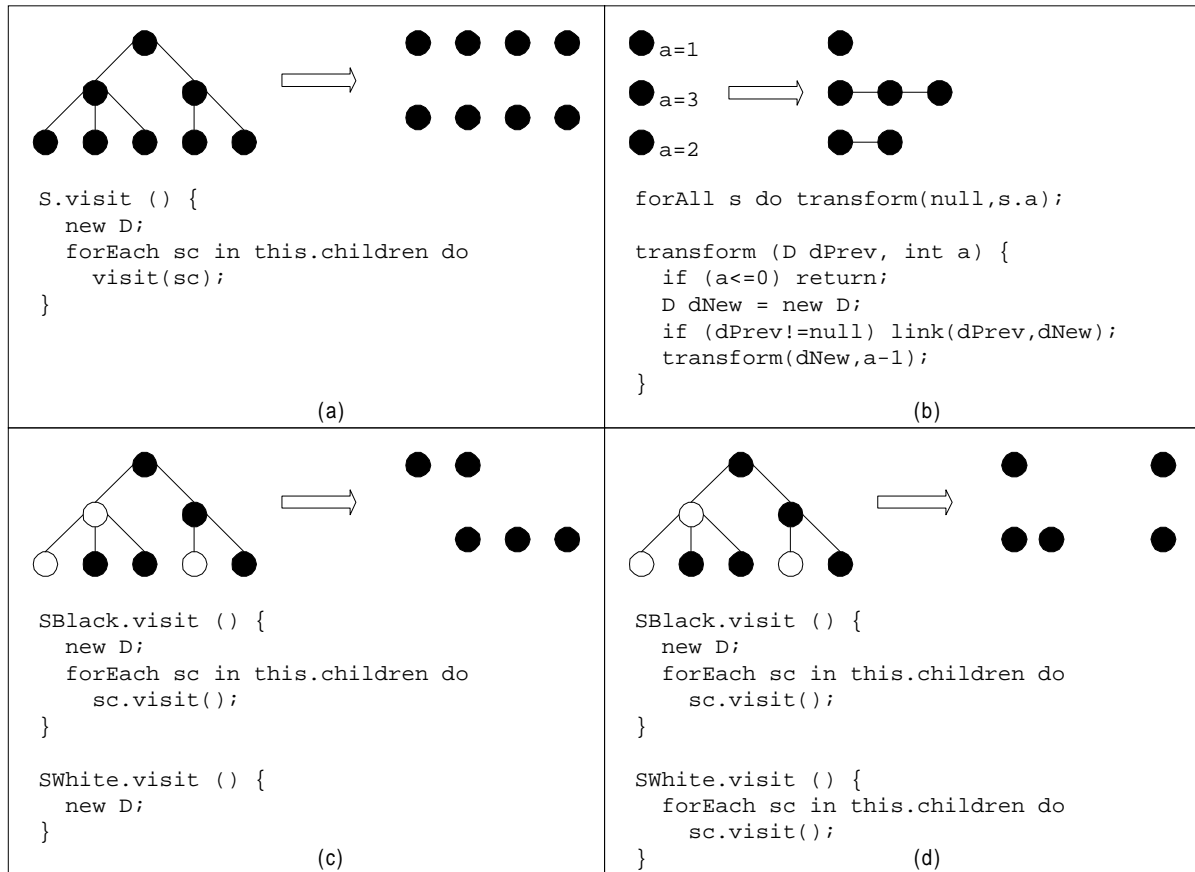
Zbog svoje prirodne sličnosti, ova dva aspekta imaju i slične parametre. Ovde su istaknuti sledeći parametri:

- Način specifikacije transformacije (programski, vizuelno ili na drugi način).
- Da li specifikacije dozvoljavaju rekurziju. Slično kao i kod izvorišnog modela, ukoliko odredišni model poseduje svojstvo rekurzivnosti, dobro je da metoda podrži rekurzivnu specifikaciju generisanja strukture odredišnog modela.
- Da li specifikacije podržavaju polimorfizam. Opet se pod ovim podrazumeva da se za neki tip elementa iz izvorišnog domena može definisati način generisanja dela odredišnog modela, a takođe taj način i redefinisati za specijalizovane tipove.
- Način organizacije i dekompozicije ovih specifikacija.

Razlike u značenju ovih parametara između dva navedena aspekta – specifikaciji obilaska strukture i specifikaciji transformacije, iako su teže uočljive, ipak postoje. Naročito je teško uočiti razlike kod drugog i trećeg parametra (rekurzija i polimorfizam). Ove razlike istaknute su na sasvim jednostavnim primerima na slici 3.2.

Na slici 3.2a prikazana je simbolično jedna izvorišna struktura (levo) koja je rekurzivna, pa je način njenog obilaska najjednostavnije i definisati rekurzivno, kao što je to prikazano datim pseudo-kodom. Sa druge strane, međutim, odredišna struktura (desno) može biti sasvim proizvoljna.

Na slici 3.2b je pak prikazan slučaj kada je redosled obilaska izvorišne strukture (levo) nebitan (prikazan je pseudo-naredbom `forAll`). Odredišna struktura (desno), međutim, treba da se generiše rekurzivno pomoću datog pseudo-koda u kome se koriste vrednosti atributa `a` tipa `s` iz izvorišnog domena.



Slika 3.2: Razlike između nekih parametara specifikacije obilaska izvorišne strukture (a i c) i specifikacije transformacije (b i d). (a) Rekurzija u specifikaciji obilaska. (b) Rekurzija u specifikaciji transformacije. (c) Polimorfizam u specifikaciji obilaska. (d) Polimorfizam u specifikaciji transformacije.

Na slici 3.2c prikazan je slučaj kada se način obilaska izvorišne strukture definiše polimorfno. U izvorišnom modelu postoje instance dva tipa, *SWhite* (prikazane belim kružićima) i *SBlack* (prikazane crnim kružićima), koji predstavljaju izvedene tipove nekog zajedničkog osnovnog tipa *S*. Iako se za oba tipa generiše ista struktura (instanca tipa *D*) u odredišnom modelu, način obilaska susednih instanci je za njih različito definisan.

Najzad, suprotan slučaj je prikazan na slici 3.2d gde je način obilaska suseda isti, dok je način generisanja odredišnog modela različit i obavlja se polimorfno.

Može se primetiti da su navedeni primeri veoma jednostavni i da kod njih možda i nije potrebno isticati ove razlike. Dva aspekta (specifikacija obilaska i način transformacije) su u ovim primerima sjedinjeni zbog korišćenja programskog pristupa pri specifikaciji. Međutim, u opštem slučaju, ova dva aspekta mogu biti sasvim razdvojena. Takođe se može primetiti da su u datim primerima rekurzije veoma jednostavne i mogu se (kao i u opštem slučaju) pretvoriti u iteracije. Međutim, može se očekivati da je za složene rekurzivne strukture obilazak ili transformaciju mnogo jednostavnije definisati rekurzivno nego iterativno, pa je dobro da metoda to podržava.

Navedeni aspekti analiziranih metoda sumirani su u tabeli 3.5. Slično kao i kod strategija obilazaka, podrška rekurziji postoji u praktično svim metodama, dok je polimorfizam podržan samo u metodama koje se oslanjaju na objektno orijentisane koncepte. Jedan od osnovnih nedostataka je generalno slaba podrška dekompoziciji i organizaciji složenih specifikacija transformacija, uključujući i mogućnost lakog definisanja različitih transformacija za iste parove izvorišnih i odredišnih domena. Najzad, važno je uočiti da

Metoda	Rekurzija	Polimorfizam	Način specifikacije	Organizacija i dekompozicija
Programski (skriptovanje)	Zavisno od jezika, uglavnom da	Zavisno od jezika, uglavnom ne	Programski	Zavisno od jezika, uglavnom proceduralna
Tekstualni šabloni	Ne	Ne	Tekstualni	Podela u datoteke
Obrazac <i>Visitor</i>	Da	Da	Programski	Objektna
Apstraktne specifikacije	Da	Da	Programski	Objektna
Tradicionalni prevodioci (generisanje koda)	Da	Ne	Programski	Nema direktne podrške
Transformatori bazirani na gramatikama	Da	Ne	Pomoću podržanih operacija	Nema direktne podrške
Intencionalno programiranje	Da	Ne	Pomoću operacija transformacija	Nema direktne podrške
Prevodioci za vizuelne jezike (parsiranje)	Da	Ne	Pozicionom gramatikom	Nema direktne podrške

Tabela 3.5: Pregled karakteristika načina specifikacije transformacija za analizirane metode

nijedna metoda ne podržava vizuelne specifikacije transformacija, kao što to čini metoda predložena u ovom radu.

IV Suština predloženog rešenja

Geneza i osnovni elementi ideje

Analiza problema transformacije modela i postojećih rešenja donela je mnoge značajne zaključke o nedostacima tih rešenja. Ta analiza ukazala je i na mogući put rešavanja ovog problema. U ovom poglavlju je najpre naveden suštinski uzrok složenosti procesa specifikacije preslikavanja koji leži u konceptualnoj udaljenosti domena koji se preslikavaju. Kao rešenje se predlaže uvođenje međudomena za koje je znatno lakše definisati preslikavanja. Zatim je objašnjena osnovna ideja i namena metode preslikavanja domena koja se predlaže u ovom radu.

Preslikavanje udaljenih domena i uvođenje međudomena

U glavi "Pregled postojećih rešenja" bio je prikazan demonstrativni primer generisanja tekstualnog modela (izvornog koda na jeziku C++) iz izvorišnog domena konačnih automata zadatog odgovarajućim metamodelom. Na slici 2.2 prikazan je taj primer u kontekstu četvoroslojne arhitekture metamodelovanja. Diskutovane su osnovne mane jednog od najzastupljenijih pristupa u generisanju tekstualnog izlaza koji se sastojao u za tu namenu specijalno programiranom generatoru koda.

Analiza ovog primera ukazuje na suštinski uzrok postojanja navedenih nedostataka. Problem je u tome što su izvorišni i odredišni domeni na veoma udaljenim nivoima apstrakcije, pa je izuzetno teško definisati njihovo preslikavanje. Izvorišni domen je bio veoma apstraktan, a osim toga oslanjao se na objektno orijentisani meta-metamodel. Odredišni domen je konceptualno udaljen od njega, a osim toga je i sekvencijalan. Ovakva konceptualna udaljenost ogleda se i u tome što jedan element izvorišnog modela, npr. jedan događaj, ima mnogo posledica na potpuno različitim mestima u ciljnom modelu (operacije interfejsne klase, osnovne klase stanja i izvedenih klasa stanja). Dakle, ova konceptualna udaljenost, odnosno značajna razlika u nivou apstrakcije između domena, kako se pokazuje i na drugim primerima, predstavlja osnovni otežavajući faktor u preslikavanju domena.

Direktno preslikavanje između domena jako različitih nivoa apstrakcije zapravo poseduje iste nedostatke kao i slična preslikavanja u drugim oblastima softverskog inženjerstva. Na primer, danas se s pravom smatra da je direktno pisanje izvornog koda za kompleksan sistem na nekom objektno orijentisanom programskom jeziku kao što je C++, veoma složeno i nepogodno bez prethodnog modelovanja na višem nivou apstrakcije, pomoću nekog jezika za vizuelno objektno modelovanje kao što je UML. Ovakva teškoća leži u tome što je jezik za vizuelno objektno modelovanje kao što je UML daleko bliži načinu razmišljanja projektanta nego što je to tekstualni programski jezik. Zbog toga je projektantu generalno lakše da najpre definiše model na ovako apstraktnom jeziku, a da ga zatim transformiše u ciljni tekstualni programski jezik. Ako je ta transformacija još i formalna, onda se ona može izvršiti i automatski, pa je proces projektovanja utoliko jednostavniji.

Slično važi i za demonstrativni primer generisanja koda konačnih automata. Umesto da se iz modela konačnog automata direktno generiše C++ kôd, može se najpre formirati jedan posredni međumodel baziran na metamodelu višeg nivoa apstrakcije. Taj posredni metamodel može sadržati osnovne objektno koncepte (klasa, atribut, operacija, asocijacija, nasleđivanje), odnosno biti jedan mali podskup metamodela UML. Sa druge strane, pošto se može očekivati da generator koda iz UML modela već postoji ne samo za C++ nego i druge jezike, on se može iskoristiti i za proces generisanja koda iz formiranog međumodela. Čak da

takav generator koda i ne postoji, njega je znatno lakše napraviti za konceptualno bliži domen jezika UML nego za konačne automate. U tom slučaju on može biti ponovno upotrebljen i za druge namene.

Prema tome, ideja se sastoji u pravljenju instanci apstrakcija iz međudomena korišćenjem postojećeg metamodela UML. Posle toga se za tako napravljeni međumodel može pozvati postojeći generator koda opšte namene. Ovakvo formiranje međumodela može se obaviti programski na sledeći način:

```
void StateMachine::generateCode () {
    // Privremeni paket za međumodel:
    Package& pck = Package::create();

    // Međumodel:
    // Osnovna klasa stanja:
    Class& baseState = Class::create(pck);
    baseState.name = this->name+"State";

    // Konstruktor osnovne klase stanja:
    Method& baseStateConstr = Method::create(pck);
    baseStateConstr.name = this->name+"State";
    MMLink::create("members",baseState,baseStateConstr);

    //... i još mnogo, mnogo detalja ...

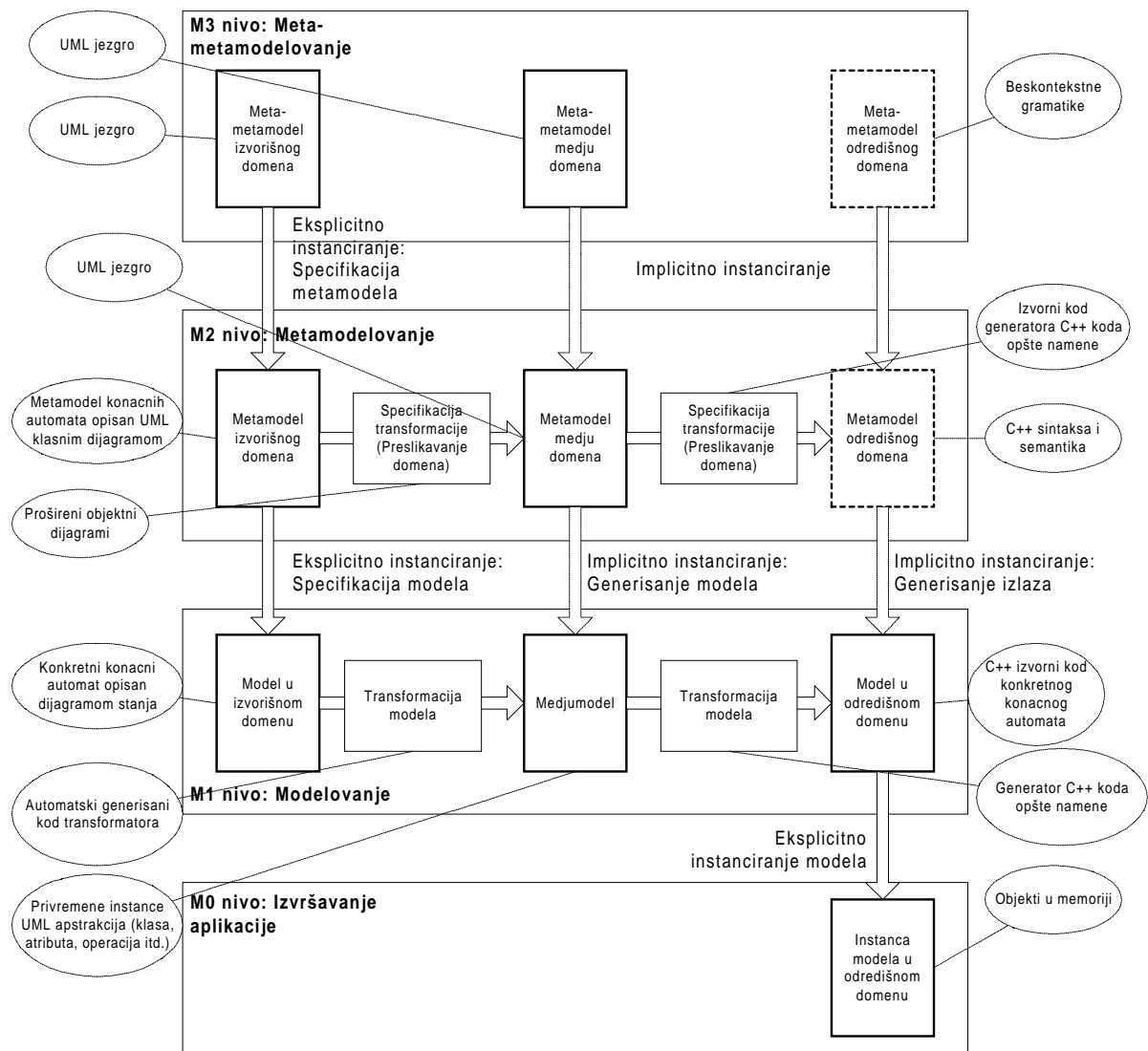
    // Generisanje koda za međumodel:
    pck.generateCode();
    deleteFromModel(pck);
}
```

Ovaj isečak koda prikazuje pravljenje instance sa imenom `FSMState` u međumodelu koja će se u konačnom kodu preslikati u istoimenu klasu `FSMState` na jeziku C++. To se obavlja pravljenjem instanci apstrakcija `Class` i `Method` iz UML metamodela, korišćenjem dostupnog programskog interfejsa tog metamodela (npr. operacija `Class::create()`). Potom se postavljaju vrednosti atributa tih instanci. Najzad, uspostavljaju se veze između tih instanci (npr. operacija `MMLink::create()`, pri čemu klasa `MMLink` predstavlja veze na M1 nivou). Sve ove instance pakuju se u privremeni UML paket (engl. *package*) za koji se na kraju generiše kôd.

Ovaj pristup prevazilazi neke nedostatke prvobitnog rešenja. Prvo, on eliminiše problem konceptualne udaljenosti izvorišnog i odredišnog domena uvođenjem međudomena. Ovim se proces generisanja izlazne forme rastavlja na dva koraka, pri čemu je drugi korak najčešće podržan postojećim generatorom koda, dok su oba koraka znatno jednostavnija za implementaciju nego prvobitno direktno preslikavanje. Prvo preslikavanje se više ne bavi specifičnostima sintakse i semantike ciljnog jezika C++, niti tehničkim detaljima (izlazne datoteke). Pored toga, preslikavanje iz UML domena u C++ domen je manje-više standardizovano. Sa druge strane, preslikavanje ostalih proizvoljnih domena višeg nivoa apstrakcije koje definiše korisnik u programski kôd ne može, niti treba da bude standardizovano. Primer konačnih automata je samo jedan primer, dok glava "Primeri upotrebe" prikazuje neke druge interesantne primere koji potvrđuju ove iskaze.

Ovo su razlozi zbog kojih mnogi komercijalni alati za modelovanje koriste međudomene i generatore koda upotrebljive u različitim kontekstima, iako je to najčešće sakriveno od korisnika. Prednosti uvođenja preslikavanja izvorišnog domena u međudomen koje je nezavisno od ciljnog implementacionog domena diskutovani su u [F-Sh197].

Međutim, ovakva programska specifikacija je još uvek zamorna i podložna greškama. Osim toga, ovakav kôd može biti veoma kompleksan i nezgodan za održavanje.

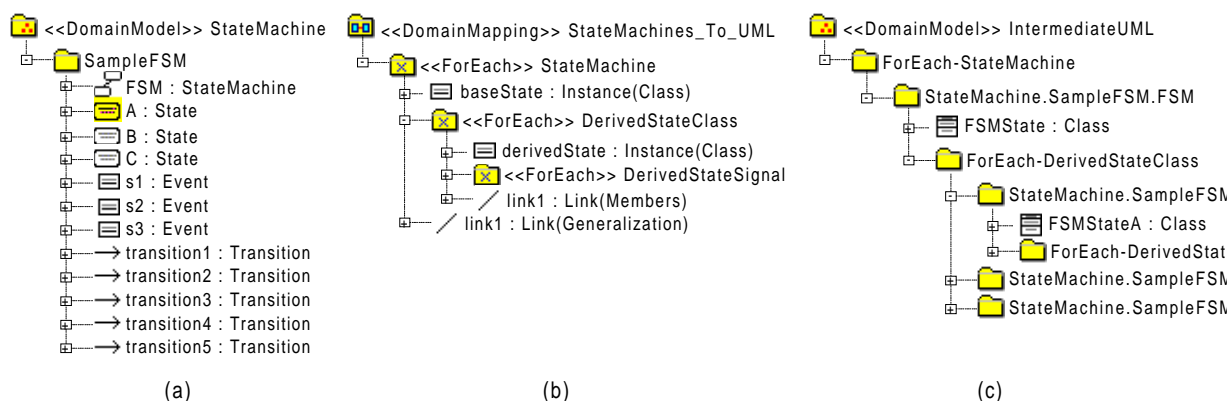


Slika 4.1: Četvoroslojna arhitektura metamodelovanja u kontekstu demonstrativnog primera konačnih automata, sa metodom preslikavanja domena. Preslikavanje iz izvorišnog u odredišni domen rastavljeno je na dva (ili više) koraka da bi se smanjila kompleksnost.

Kako se ovde zapravo radi o pravljenju instanci apstrakcija iz odredišnog domena (u celokupnom procesu to je međudomen) pri obilasku izvorišnog domena, pri čemu se metamodeli oba domena mogu definisati pomoću objektno orijentisanih koncepata, ova specifikacija se može zadati na drugačiji formalni način. Ideja je da se koriste vizuelne specifikacije, po mogućstvu kompatibilne sa UML standardom. Takve specifikacije bi bile lakše za izgradnju, razumevanje i održavanje, a takođe i manje podložne greškama. Na taj način bi one prevazišle sve navedene nedostatke prethodnih pristupa.

Ideja i namena metode preslikavanja domena

Položaj specifikacije preslikavanja domena (engl. *domain mapping*) predložene u ovom radu prikazana je na slici 4.1. Međudomen, odnosno njegov metamodel, uvodi se na M2 nivou, kao i međumodel na M1 nivou. Za demonstrativni primer konačnih automata metamodel



Slika 4.2: (a) Izvorišni model, (b) specifikacija transformacije i (c) generisani međumodel. Izvorišni i generisani model sastoje se od instanci apstrakcija iz odgovarajućih domena, povezanih vezama kao instancama asocijacija iz tih domena. Izvorišni model formira korisnik. Međumodel se generiše automatski pomoću transformatora koji se dobija iz specifikacije preslikavanja domena (b). Elementi generisanog modela grupisani su hijerarhijski u pakete prema repetitivnim elementima specifikacije preslikavanja. Poreklo paketa generisanog repetitivnom zapisano je u njegovom nazivu.

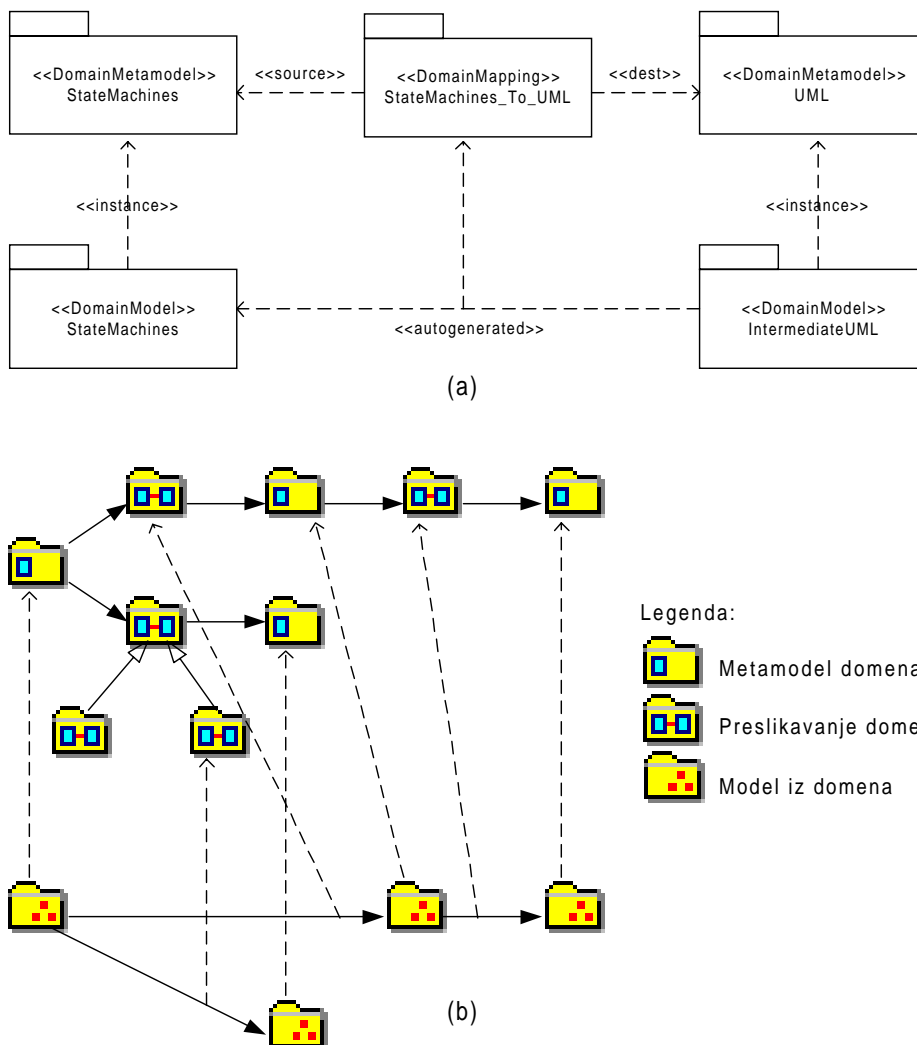
međudomena je podskup UML metamodela. Dobitak je u tome što su pojedinačna preslikavanja u i iz međudomena znatno jednostavnija nego početno direktno preslikavanje, pa ih ja zato lakše i definisati i održavati.

Definicija preslikavanja domena treba da bude formalna i po mogućstvu grafička. Ona treba da specifikuje skup instanci tipova iz odredišnog domena (tj. međudomena) koje treba kreirati za neku instancu iz izvorišnog modela, zajedno sa vezama između njih. Zbog toga se ona može predstaviti UML *objektnim dijagramom* (engl. *object diagram*) [B-Boo99]. Međutim, standardni objektni dijagram nije dovoljan za sve potrebe specifikacije preslikavanja. Potrebni su, između ostalog, i koncepti repetitivnog i uslovnog kreiranja instanci i veza. Ovi koncepti biće opisani u narednom poglavlju.

Prema tome, izvorišni i odredišni model (tj. međumodel) sastoje se od instanci apstrakcija iz odgovarajućih domena (slika 4.2). One su međusobno povezane instancama asocijacija iz tih domena. Izvorišni model (slika 4.2a) formira korisnik eksplicitno. Međumodel (slika 4.2c) generiše se automatski, pomoću transformatora (zapravo generatora) dobijenog iz specifikacije preslikavanja (slika 4.2b). Elementi generisanog modela grupisani su hijerarhijski u pakete prema repetitivnim elementima specifikacije preslikavanja. Ovo obezbeđuje bolju preglednost u generisanom modelu, kao i mogućnost određivanja porekla generisanog elementa. Ovi aspekti biće komentarisani detaljnije u poglavlju "Organizacija specifikacija".

Kao što je već rečeno, strukture modela su tipizirani grafovi instanci povezanih vezama. Hijerarhijska struktura paketa prikazana na slici 4.2 je samo organizacione prirode, formirana u cilju bolje preglednosti složenog modela. Ova hijerarhija predstavlja samo jedan prikaz strukture složenog grafa koja inherentno leži iza tog prikaza. Kako je struktura grafa suštinski različita od sekvencijalne strukture izlaznog teksta, za specifikaciju preslikavanja potrebna je metoda suštinski drugačija od ranije navedenih.

Slika 4.3a prikazuje relacije između modela i njihovih metamodela iskazane na jeziku UML. Metamodeli su predstavljeni paketima sa stereotipom <<DomainMetamodel>>. Specifikacija preslikavanja domena (engl. *domain mapping specification*) je u paketu sa stereotipom <<DomainMapping>>. Njihove međusobne relacije predstavljene su zavisnostima (engl. *dependency*) sa odgovarajućim stereotipovima. Paketi sa stereotipom <<DomainModel>> predstavljaju modele kao instance odgovarajućih metamodela. Međumodel



Slika 4.3: (a) Šema preslikavanja domena. Paketi u gornjem redu sa stereotipom `<<DomainMetamodel>>` predstavljaju metamodele (M2 nivo). Specifikacija preslikavanja domena predstavljena je paketom sa stereotipom `<<DomainMapping>>`. Paketi sa stereotipom `<<DomainModel>>` u donjem redu predstavljaju modele (M1 nivo). Međumodel se generiše automatski iz izvorišnog modela, korišćenjem odgovarajućeg preslikavanja domena. (b) Sukcesivne transformacije (engl. *model pipeline*) u uopštenom pristupu. Modeli se mogu generisati jedan iz drugog sukcesivno, u lancu, pri čemu se svaka transformacija vrši prema određenom preslikavanju domena.

se generiše automatski iz izvorišnog modela, korišćenjem odgovarajućeg preslikavanja domena.

Pored ovoga, postoji i značajno uopštenje predloženog pristupa prikazano šematski na slici 4.3b: modeli se mogu generisati jedan iz drugog sukcesivno, u lancu, pri čemu se svaka transformacija vrši korišćenjem odgovarajuće specifikacije preslikavanja domena. Treba primetiti da specifikacija preslikavanja domena, kao jedna vrsta UML paketa, može biti specijalizovana u nekoj drugoj specifikaciji, npr. u cilju definisanja varijanti preslikavanja istih domena.

Ukoliko je generisani model još uvek na dovoljno visokom nivou apstrakcije da ga korisnik može razumeti a pri tom i želi da u njega unese izmene, pristup bi trebalo da obezbedi mehanizam čuvanja izmena koje je korisnik uneo u generisani model prilikom njegovog ponovnog generisanja (recimo posle izmene izvorišnog modela). Drugim rečima, korisnik može da bude zadovoljan najvećim delom generisanog modela, ali da ipak želi da

učini male modifikacije kako bi taj model prilagodio svojim potrebama. Naravno, te izmene će biti propagirane u narednim sukcesivnim transformacijama tog generisanog modela, sve do željene krajnje implementacije. Ovaj važan mehanizam postoji i u nekim drugim alatima za modelovanje, recimo u slučaju kada se generisani programski kod može promeniti ručno, dok se te izmene čuvaju prilikom ponovnog generisanja koda [D-RSCW]. U ovde predloženom pristupu ovakve manuelne izmene generisanog modela mogu da se beleže unutar alata za modelovanje da bi se prilikom regeneracije ponovo automatski izvršile. To zahteva postojanje jedinstvene identifikacije elemenata generisanog modela, naročito onih koji su dobijeni repetitivnim konstruktima, kao i mogućnost određivanja elementa izvorišnog modela od koga je generisani element potekao. Detalji ovog pristupa biće opisani kasnije.

Mogućnosti primene ove metode su višestruke. Prvo, ona može biti upotrebljena za specifikaciju načina generisanja izlaznih formi u alatima za modelovanje svih nivoa podesivosti. Za fiksne, nepodesive alate za modelovanje ona se može iskoristiti prilikom konstrukcije samog alata, odnosno njegovog dela za generisanje izlaza. Podesivi alat za modelovanje (npr. CASE alat) može da ponudi programski interfejs svojih internih metamodela (npr. UML metamodel, metamodel ciljnog programskog jezika, relacioni metamodel itd.), a korisnik može da definiše preslikavanje. Ona takođe može da bude podržana i u alatu za metamodelovanje gde korisnik može da definiše i metamodele i njihovo preslikavanje. Najzad, ovaj pristup takođe može da doprinese bržem i lakšem konstruisanju specifičnih transformacija unutar specijalizovanih aplikacija.

Drugo, generisanjem modela iz različitih domena može se postići bolje razumevanje problema, odnosno kompletnije modelovanje sistema. Drugim rečima, iz jednog korisnički definisanog modela sistema koji se konstruiše, automatski se mogu na konzistentan način dobiti komplementarni modeli iz različitih domena.

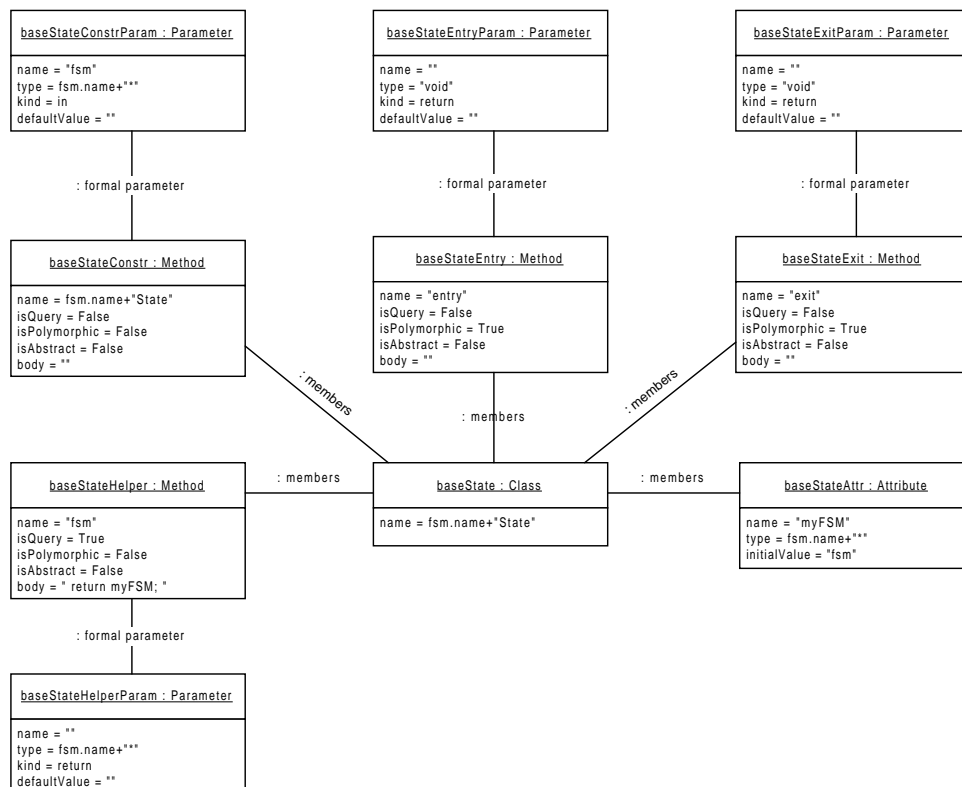
Najzad, takođe automatski se mogu dobiti i modeli na različitim nivoima apstrakcije putem sukcesivnih transformacija. Ovaj proces sukcesivnih transformacija može se posmatrati kao spuštanje nivoa apstrakcije počev od visoko apstraktnog korisničkog modela sve do konačne implementacije na niskom nivou. To spuštanje se vrši automatski, konzistentno i postupno. Osim toga, mogu se definisati i novi domen na višim nivoima apstrakcije, iznad već postojećih. Njihovim preslikavanjem u postojeće niže domene, za koje već postoje sukcesivna preslikavanja do konačne implementacije, brzo se dobija fleksibilna transformacija za novodefinisani domen. Ovo zapravo može da predstavlja novi pristup apstraktnom programiranju prilagođenom specifičnim domenima, uz brzo, fleksibilno i automatsko generisanje željene implementacije. Primeri ovog pristupa biće dati u posebnoj glavi.

Preslikavanje domena

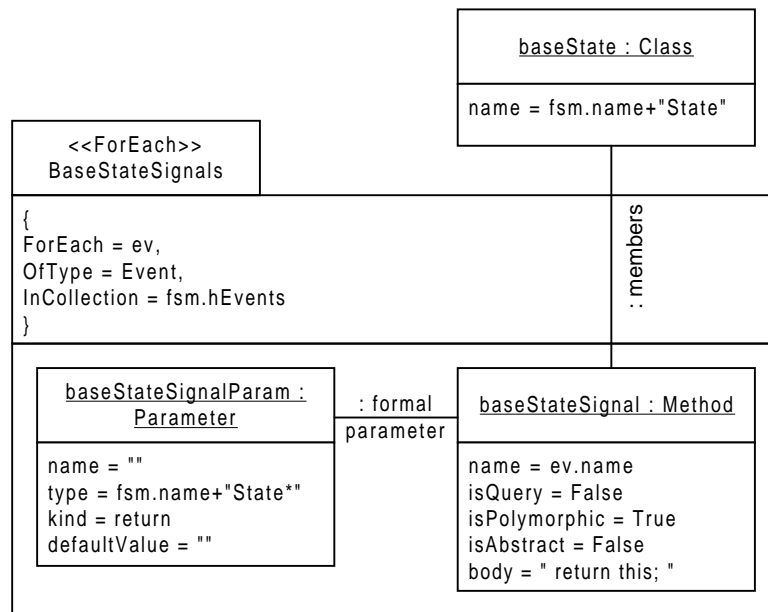
Preslikavanje domena (engl. *domain mapping*) definiše se pomoću hijerarhije paketa koja počinje paketom sa stereotipom <<DomainMapping>>. Ti paketi sadrže instance tipova *Instance* i *Link* iz UML metamodela koje predstavljaju instance i veze u odredišnom modelu koje treba kreirati. Dakle, i preslikavanje domena je zapravo jedan model sa odgovarajućim metamodelom koji je ponovo podskup UML metamodela. Taj model moguće je prikazati pomoću proširenih UML objektnih dijagrama. Ta proširenja uključuju uslovno, repetitivno, uređeno, parametrizovano, rekursivno i polimorfno kreiranje delova odredišnog modela. Od sada pa nadalje će se posmatrati jedno preslikavanje između dva domena koji će biti posmatrani kao izvorišni i odredišni domen tog preslikavanja, čak iako je ono samo jedna karika u dužem lancu preslikavanja. Precizna definicija semantike elemenata preslikavanja biće data u narednoj glavi sa detaljima predloženog rešenja.

Instance, atributi i veze

Kako specifikacija preslikavanja zapravo treba da definiše instance tipova iz odredišnog domena i njihove veze koje treba automatski kreirati, ona se najbolje prikazuje kao UML objektni dijagram [B-Boo99]. Jedan isečak za demonstrativni primer prikazan je na slici 4.4. Pretpostavlja se da je dijagram definisan za jednu instancu tipa *StateMachine* iz izvorišnog



Slika 4.4: Jedan deo objektnog dijagrama iz preslikavanja domena za demonstrativni primer konačnih automata. Dijagram prikazuje samo specifikaciju za kreiranje osnovne klase *fsmState* i njenih članova koji uvek postoje bez obzira na stanja i događaje. Dijagram pripada kontekstu konačnog automata iz izvorišnog modela koji se referiše preko identifikatora *fsm*.



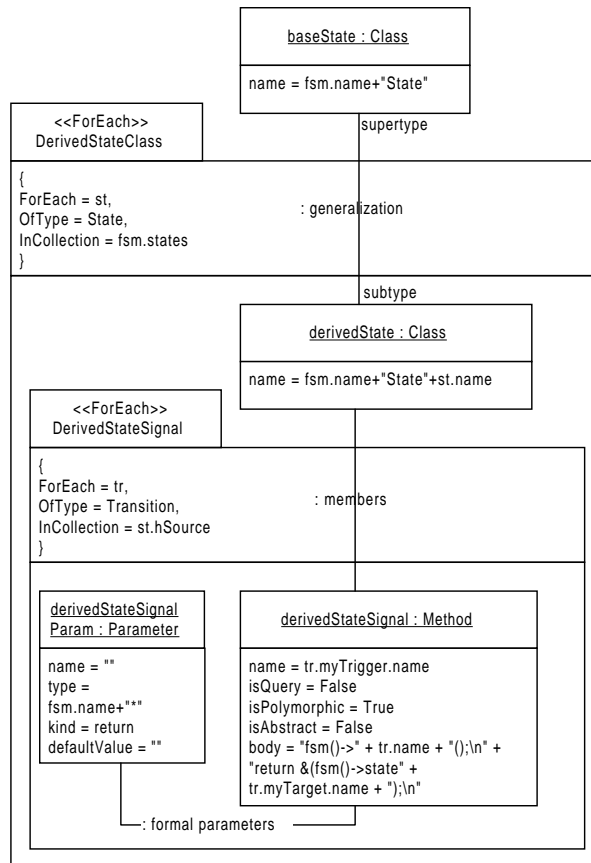
Slika 4.5: Koncept "ForEach" za repetitivno kreiranje objekata. Dijagram prikazuje samo specifikaciju za osnovnu klasu `fsmState` i njene funkcije članice koji se kreiraju za događaje na koje automat reaguje. Dijagram pripada kontekstu konačnog automata iz izvorišnog modela koji se referiše preko identifikatora `fsm`.

modela koja se referiše preko identifikatora `fsm`. Dijagram specifikuje skup instanci tipova iz odredišnog metamodela koje treba kreirati za svaku instancu tipa `StateMachine` iz izvorišnog modela. Dijagram takođe definiše i vrednosti atributa tih instanci, kao i veze koje treba uspostaviti između njih. Veze su instance asocijacija iz odredišnog metamodela.

Vrednosti atributa zadaju se preko izraza koji se izračunavaju u kontekstu izvorišnog modela. Drugim rečima, ovi izrazi referišu elemente izvorišnog modela, odnosno instance i vrednosti njihovih atributa, korišćenjem programske navigabilnosti kroz taj model. Izrazi mogu da koriste i korisnički definisane funkcije koje takođe imaju pristup do elemenata izvorišnog modela. Ove funkcije mogu da izračunavaju vrednosti atributa instanci iz odredišnog modela na način koji se ne može izraziti prostim izrazima ili dijagramom. Prema tome, tradicionalni pristup za programsko izračunavanje vrednosti je i ovde na raspolaganju. Ova mogućnost je posebno značajna ako je vrednost atributa tekstualna struktura. To je slučaj na primer kod tela metoda klasa koja se najčešće izgrađuju na složen način i to parametrizovano u odnosu na elemente izvorišnog modela i vrednosti njihovih atributa.

Repeticije

Standardni objektni dijagram ne zadovoljava sve potrebe preslikavanja. Postoji i potreba za repetitivnim kreiranjem objekata: za demonstrativni primer je potrebno kreirati po jednu operaciju unutar osnovne klase stanja za svaki događaj na koji automat reaguje (slika 2.1). Za ove potrebe ovde se koristi paket sa stereotipom `<<ForEach>>`. Jedan primer prikazan je na slici 4.5. Paket `<<ForEach>>` predstavlja iteraciju kroz kolekciju elemenata izvorišnog modela i kreiranje skupa elemenata odredišnog modela (instanci i veza) za svaki od tih elemenata kolekcije. On sadrži tri označene vrednosti (engl. *tagged value*) koje definišu iteraciju:



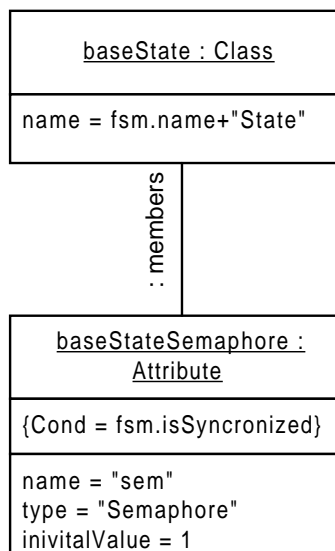
Slika 4.6: Ugnežđivanje `<<ForEach>>` paketa. Dijagram prikazuje deo specifikacije za izvedene klase stanja i njihove operacije koje obrađuju događaje.

- `ForEach`: identifikator koji se uvodi u oblast važenja tog paketa. Ovaj identifikator se može koristiti unutar ovog paketa kao referenca na tekući element kolekcije elemenata izvorišnog modela kroz koju se iterira.
- `OfType`: tip koji predstavlja filter za elemente navedene kolekcije. Iteracija je osetljiva na tip: obrađuju se samo elementi zadate kolekcije koji su datog tipa, dok se ostali preskaču. Nasledjivanje je takođe podržano: ako je tip B izveden iz tipa A, onda je instanca tipa B ujedno i tipa A; tranzitivnost važi. Ovaj tip, naravno, pripada izvorišnom metamodelu.
- `InCollection`: izraz koji vraća kolekciju elemenata izvorišnog modela kroz koju se iterira.

Veza može da povezuje instance koje pripadaju različitim paketima, od kojih bilo koji može biti `<<ForEach>>`. Ovo znači da će u odredišnom modelu biti kreirana veza za svaki par instanci iz ova dva paketa, kreiranih za tekuću iteraciju njihovog prvog zajedničkog okružujućeg `<<ForEach>>` paketa. U specijalnom slučaju kada veza povezuje jednu instancu unutar `<<ForEach>>` paketa i drugu iz njegovog okruženja kao na slici 4.5, svaka repetitivna instanca kreirana iteracijom za taj paket biće povezana sa spoljašnjom instancom.

Za izraze koji definišu kolekciju `<<ForEach>>` paketa može da se koristi bilo koji formalni jezik za navigaciju kroz objektni model. Na primer, može da se koristi jezik *Object Constraint Language* (OCL) [B-OMG99] ukoliko ga okruženje podržava, ili neki drugi specijalizovani ili opštenamenski jezik koga podržava okruženje (ovde će u primerima biti korišćen C++).

`<<ForEach>>` paketi zapravo predstavljaju petlje u kodu generatora odredišnog modela. Zbog toga se oni mogu i ugnežđivati. Jedan primer prikazan je na slici 4.6. Tu je potrebno kreirati jednu izvedenu klasu za svako stanje automata, što je specifikovano



Slika 4.7: Uslovno kreiranje. Dijagram prikazuje specifikaciju za osnovnu klasu `fsmState` i njen podatak član (semafor) koji se generiše za potrebe sinhronizacije, ali samo ako je konačni automat označen kao "sinhronizovan".

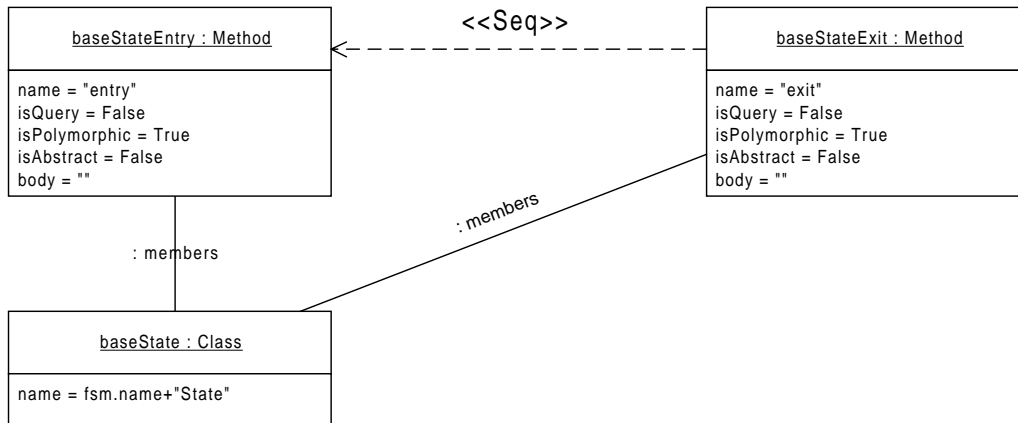
spoljašnjim `<<ForEach>>` paketom. Za svaki od događaja na koji automat reaguje potrebno je opet kreirati po jednu operaciju te izvedene klase, što je specifikovano ugnežđenim `<<ForEach>>` paketom.

Svaki `<<ForEach>>` paket uvodi oblast važenja za identifikatore i izraze koji se definišu unutar njega. Pravila ugnežđivanja oblasti važenja su ista kao i za tradicionalne proceduralne programske jezike tipa Pascal ili C. Izraz koji definiše vrednost atributa instance ili kolekciju u iteraciji može koristiti identifikatore iz svih okružujućih oblasti važenja tj. paketa. Identifikator koji referiše tekući element iteracije u `<<ForEach>>` paketu je lokalna za taj paket i sakriva iste identifikatore iz okružujućih paketa.

Poštujući stil UML notacije, a kako je `<<ForEach>>` zapravo jedna vrsta paketa, dozvoljeno je da se njegov sadržaj prikaže pomoću proizvoljno mnogo dijagrama. Ova mogućnost značajno poboljšava preglednost složenih preslikavanja i olakšava njihovu dekompoziciju i organizaciju. Celo preslikavanje je zapravo organizovano u hijerarhiju paketa koja počinje paketom sa stereotipom `<<DomainMapping>>`. Ovaj koreni paket može sadržati ugneždene obične ili `<<ForEach>>` pakete. `<<ForEach>>` paketi mogu pak da iteriraju kroz sve instance nekog tipa u izvorišnom modelu korišćenjem funkcija koje to podržavaju (npr. funkcija `StateMachine::getAllInstances()` koja vraća kolekciju svih instanci tipa `StateMachine`). Prema tome, svi dijagrami prikazani na slikama 4.4. do 4.6 pripadaju istom `<<ForEach>>` paketu koji iterira kroz sve instance tipa `StateMachine`.

Uslovi

Još jedan potreban koncept jeste uslovno kreiranje. Instanca, veza ili paket može imati označenu vrednost koja predstavlja logički izraz koji se izračunava u kontekstu izvorišnog modela (označena vrednost `Cond`). Ako je rezultat tog izraza "netačno", dati element se neće kreirati. Jedan jednostavan primer prikazan je na slici 4.7. Taj primer podrazumeva da tip `StateMachine` u izvorišnom metamodelu poseduje atribut `isSynchronized`. Ako je vrednost tog atributa `True` za neku instancu u izvorišnom modelu, onda generisana klasa u C++ kodu treba da bude monitor, što znači da njene operacije treba da budu međusobno isključive za konkurentni pristup. To se može postići podatkom članom nekog bibliotečnog tipa `Semaphore` koji se generiše u osnovnoj klasi stanja i koji predstavlja semafor za sinhronizaciju.



Slika 4.8: Sekvencijalno kreiranje. Relacija zavisnosti sa stereotipom `<<Seq>>` eksplicitno definiše redosled kreiranja elemenata. Dijagram prikazuje zahtev da se operacija `entry()` kreira pre operacije `exit()`.

Sekvence

Generisani model je složena struktura elemenata (instanci i veza) koji su hijerarhijski grupisani u pakete prema filozofiji organizacije UML modela (slika 4.2). Svaki paket predstavlja neuređenu kolekciju elemenata koje sadrži (to su instance tipova iz metamodela i ugnežđeni paketi). Preciznije, uređenje elemenata koje paket sadrži implicitno je određeno redosledom njihovog kreiranja koji nije unapred definisan, osim za neke specijalne slučajeve koji će biti navedeni kasnije. Ponekad je, međutim, potrebno eksplicitno definisati uređenje elemenata u odredišnom modelu. Ovo uređenje može da obezbedi potreban redosled prilikom obilaska tih elemenata u narednim transformacijama. To može biti slučaj, na primer, ako je potrebno kasnije generisati sekvencijalnu (tekstualnu) strukturu iz tog modela.

Ako je potrebno da se element x kreira posle elementa y , on se može smatrati zavisnim od elementa y . Ova relacija se na dijagramu predstavlja pomoću relacije zavisnosti od x prema y sa stereotipom `<<Seq>>` (ili potpuno ravnopravno `<<sequence>>`). Kao posledica će kreirani element y prethoditi kreiranom elementu x unutar paketa koji ih sadrži. Precizna pravila konzistentnosti i semantike ovog koncepta sekvencijalne zavisnosti biće definisana u narednoj glavi.

Slika 4.8 prikazuje primer gde je zahtev da operacija `FSMState::entry()` prethodi operaciji `FSMState::exit()` u deklaraciji klase definisan pomoću sekvencijalne zavisnosti. Na ovaj način će generator C++ koda prilikom obilaska generisanog međumodela najpre naići na operaciju `entry()`, pa će nju generisati pre operacije `exit()`.

Parametrizovane podstrukture i rekurzija

Često je potrebno na raznim mestima generisati isti ili donekle različit deo odredišnog modela. Zato je poželjno obezbediti mogućnost da se definiše izgled neke podstrukture modela koja će onda biti generisana na različitim mestima. Eventualne razlike u generisanoj strukturi, kao i njene veze prema okružujućim delovima treba omogućiti odgovarajućom parametrizacijom. Na ovaj način se omogućuje bolja dekompozicija preslikavanja i izbegava ponavljanje istih ili sličnih delova u specifikaciji. Ovaj koncept zapravo predstavlja potpuni ekvivalent koncepta procedure u proceduralnim programskim jezicima.

Način na koji je ovaj koncept podržan u preslikavanju domena prikazan je jednim primerom na slici 4.9. Prvi element ovog koncepta je *definicija podstrukture* (engl. *substructure definition*, slika 4.9a). Ovaj pojam je analogan definiciji procedure u proceduralnom programskom jeziku. Definicija podstrukture predstavlja se paketom sa stereotipom <<Substruct>>. Unutar ovog paketa nalaze se uobičajene specifikacije preslikavanja, odnosno već opisani elementi (instance, veze, <<ForEach>> i obični paketi i relacije zavisnosti).

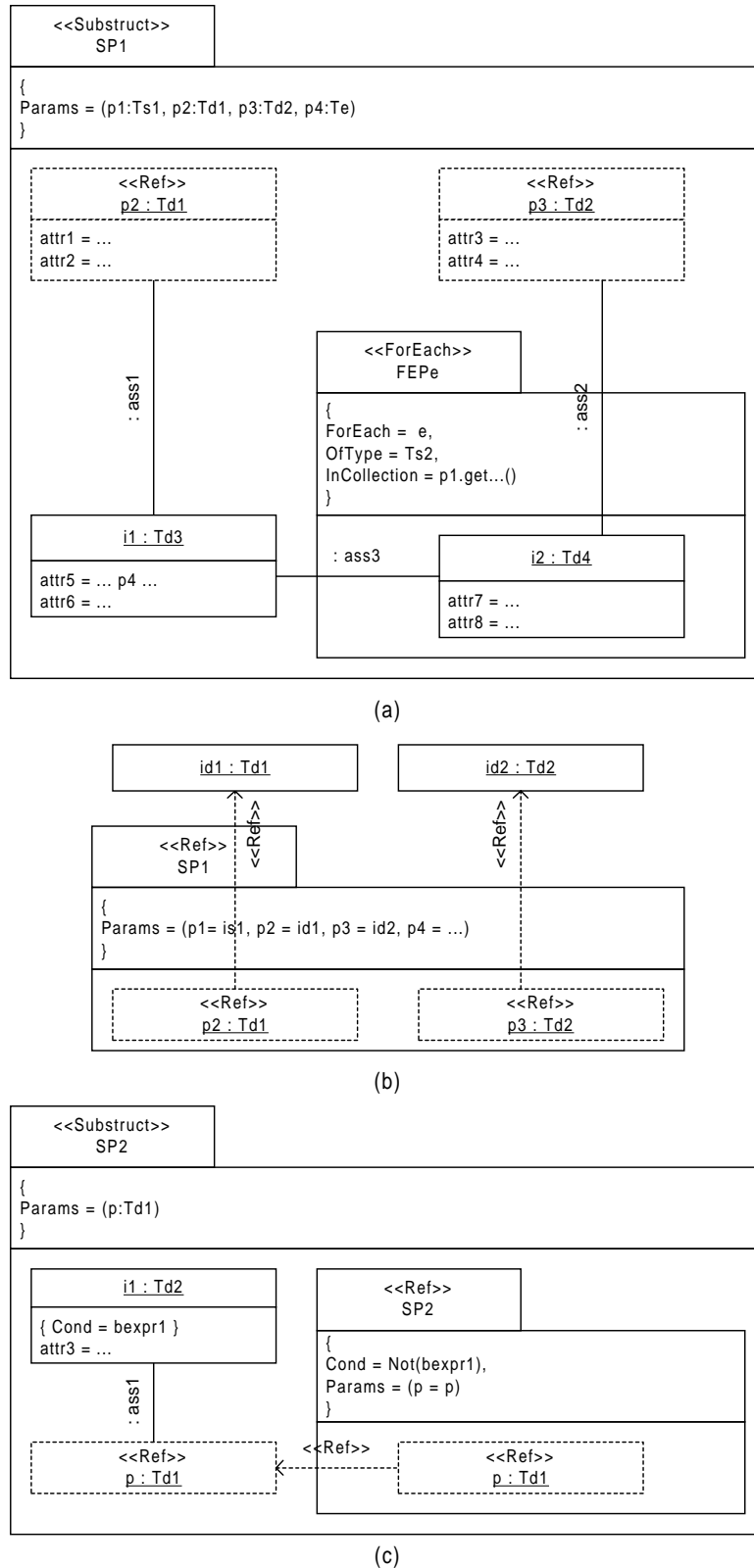
Paket sa stereotipom <<Substruct>> ne može biti sadržan unutar drugog takvog paketa niti sadržati takve pakete. Ovo ograničenje je ekvivalentno ograničenju da se definicije procedura ne mogu (statički) ugnežđivati. Ovo ograničenje uvedeno je iz nekoliko razloga. Prvo, ono znatno olakšava i definiciju semantike i implementaciju. Drugo, ovakav koncept nije toliko značajan da bi ga trebalo podržati. Ni mnogi proceduralni jezici ne podržavaju ugnežđivanje definicija procedura (npr. jezici C i C++ ga ne podržavaju, za razliku od jezika Pascal). Kako je krajnji cilj da se transformator modela implementira na nekom standardnom programskom jeziku (u ovom radu to je C++), pogodnije je da se metoda ne vezuje za ovaj nestandardan koncept.

Drugi element koncepta su *parametri podstrukture*. Naime, potrebna fleksibilnost generisanja podstrukture kao i veza date podstrukture sa okruženjem u kome će biti generisana može se obezbediti putem parametara. U <<Substruct>> paketu se definišu *formalni parametri* (engl. *formal parameters*). Ovi parametri specifikuju se pomoću označene vrednosti `Params` datog paketa. Svaki od formalnih parametara ima svoj tip. Formalni parametri imaju oblast važenja unutar svog paketa. Parametri mogu biti:

- Reference na elemente izvorišnog modela, kada je njihov tip neki tip iz izvorišnog metamodela. U primeru na slici 4.9a to je parametar `p1` tipa `Ts1`. Ovakav parametar može da se koristi u definiciji podstrukture za referisanje elementa izvorišnog modela preko koga se definiše obilazak tog modela prilikom generisanja podstrukture.
- Reference na generisane instance odredišnog modela, kada je njihov tip neki tip iz odredišnog metamodela. U primeru na slici 4.9a su parametri `p1` i `p2` tipa `Td1` i `Td2`, respektivno. Ove reference prilikom generisanja podstrukture ukazuju na konkretne instance odredišnog modela koje su već generisane u okruženju date podstrukture, dok je unutar podstrukture potrebno napraviti veze sa tim instancama (kao veze tipa `ass1` i `ass2` na slici 4.9a).
- Parametri implementacionog tipa. Ovi parametri su pod potpunom kontrolom okruženja i predložena metoda ne ulazi u njihovo značenje. To mogu biti parametri tipova koje nudi dato okruženje i koji služe za implementacione detalje, recimo u izrazima koji određuju vrednosti atributa (npr. atribut `attr5` instance `i1` na slici 4.9a).

Unutar definicije podstrukture, parametri koji su reference na instance odredišnog modela mogu se prikazivati kao instance sa stereotipom <<Ref>>. Ove instance ne predstavljaju specifikaciju za kreiranje instanci u odredišnom modelu, nego samo reference na već kreirane instance. Ove instance mogu imati specifikacije postavljanja vrednosti atributa ili biti povezane vezama sa drugim instancama, pri čemu se sve to odnosi na referencirane instance.

Prema tome, definicija podstrukture, odnosno paket sa stereotipom <<Substruct>> može sadržati sledeće elemente: instance i reference na instance (instance sa stereotipom <<Ref>>), veze, <<ForEach>> i obične ugnežđene pakete. Reference na instance mogu da imaju imena samo nekog od formalnih parametara. Veze ne mogu prelaziti granice ovog paketa. Izrazi koji definišu vrednosti atributa instanci i referenci na instance, kao i izrazi koji definišu kolekcije u <<ForEach>> paketima mogu da koriste sve lokalne identifikatore, pa i formalne parametre koji takođe spadaju u tu kategoriju.



Slika 4.9: Parametrizovane podstrukture i rekurzija. (a) Definicija podstrukture predstavljena je paketom sa stereotipom `<<Substruct>>` i sadrži specifikaciju parametrizovanog kreiranja dela strukture koja se može pojaviti na različitim mestima na isti način. (b) Referenca na podstrukturu (paket sa stereotipom `<<Ref>>`) predstavlja zahtev za kreiranje podstrukture na mestu "poziva". (c) Definicija podstrukture može neposredno ili posredno referisati istu podstrukturu, čime je podržana i rekurzija.

Treći element ovog koncepta je *poziv podstrukture* (engl. *substructure invocation*) ili *referenca na podstrukturu* (engl. *substructure reference*). Ona se može naći u bilo kom paketu, pa i unutar definicije podstrukture, a prikazuje se kao paket sa stereotipom <<Ref>> (slika 4.9b). Ona predstavlja zahtev za generisanje podstrukture zadate odgovarajućom definicijom. Ovaj koncept je potpuni ekvivalent koncepta poziva procedure u proceduralnim jezicima.

Prilikom poziva podstrukture pomoću paketa sa stereotipom <<Ref>>, potrebno je definisati *stvarne parametre* (engl. *actual parameters*) tog poziva. Ovo se vrši pomoću označene vrednosti `Params` paketa sa stereotipom <<Ref>>, kao na slici 4.9b. Postavljanje vrednosti parametara koji su reference na instance određiškog modela predstavlja zapravo vezivanje formalnih parametara-referenci za instance iz konteksta u kome se nalazi poziv. Ovo se može uraditi i grafički, pomoću relacije zavisnosti sa stereotipom <<Ref>> koja polazi od formalnog prema stvarnom parametru (slika 4.9b). Paket sa stereotipom <<Ref>> može da sadrži samo reference na instance koje predstavljaju formalne parametre, i ništa više.

Svaki paket, osim paketa sa stereotipom <<Ref>>, može da sadrži bilo koji paket sa stereotipom <<Ref>>, što znači da se poziv podstrukture može naći i u definiciji druge podstrukture. To u potpunosti odgovara dinamičkom ugnežđivanju poziva potprograma u proceduralnim jezicima. Zbog toga nema nikakve prepreke da se poziv jedne podstrukture može naći direktno ili posredno unutar definicije iste te podstrukture, kao na slici 4.9c. Na ovaj način je u potpunosti podržana i rekurzija sa sasvim uobičajenim značenjem. Naravno, semantička ispravnost same rekurzije, odnosno njena konačnost zavisi samo od specifikacije preslikavanja i okruženje ne treba u to da ulazi.

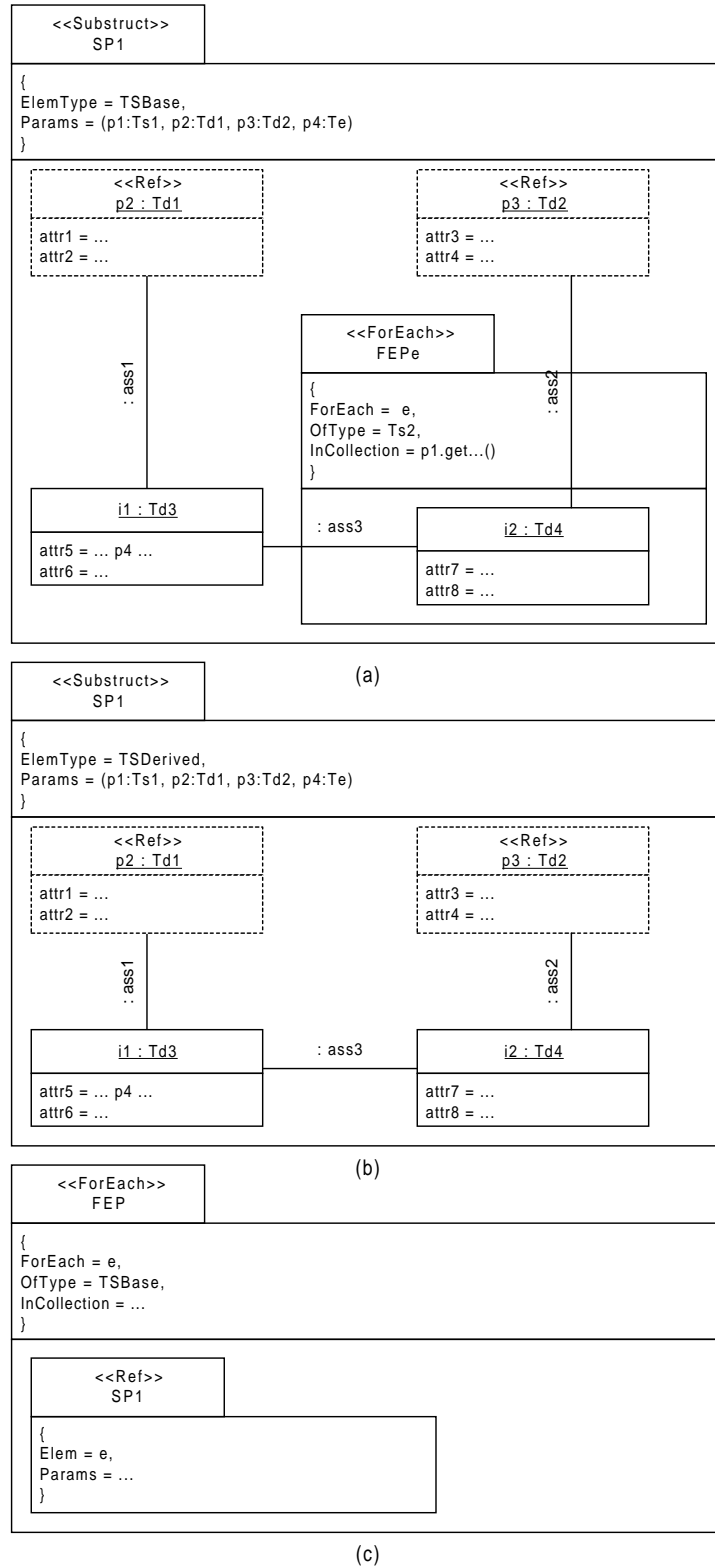
Polimorfizam

Kako je pojam podstrukture u preslikavanju domena ekvivalentan pojmu procedure, sasvim je prirodno i njegovo proširenje koje omogućuje polimorfizam. Ovaj koncept nije šire zastupljen u okruženjima za modelovanje i transformaciju modela, a poseduje izuzetne potencijale, možda još ne sasvim istražene, na koje ukazuje bogato iskustvo u korišćenju polimorfizma u klasičnom objektnom programiranju.

Kao što se u objektno orijentisanom programskom jeziku procedura može pridružiti nekom tipu (kao metoda klase), tako se i ovde definicija podstrukture može pridružiti nekom tipu iz izvoriškog metamodela. To se radi pomoću označene vrednosti `ElementType` paketa sa stereotipom <<Substruct>>. Jedan primer prikazan je na slici 4.10a. Tu je data definicija podstrukture `SP1`, sa odgovarajućim formalnim parametrima, pridružena tipu `TSBase` iz izvoriškog metamodela.

Podstruktura pridružena datom tipu `TSBase` iz izvoriškog metamodela može se redefinisati za neki izvedeni tip `TSDerived`, kao na slici 4.10b. Ta redefinicija podstrukture mora imati isti naziv, kao i broj i tipove formalnih parametara. Pravila su zapravo ista kao uobičajena pravila redefinisavanja operacija (engl. *operation overriding*) u objektno orijentisanim programskim jezicima C++ ili Java.

Najzad, prilikom referisanja podstrukture koja je pridružena nekom tipu, mora se navesti i konkretan element (tačnije referenca na element) iz izvoriškog modela. To se radi pomoću označene vrednosti `Elem` paketa sa stereotipom <<Ref>> kao na slici 4.10c. Generisanje podstrukture vrši se polimorfno, prema onoj definiciji koja odgovara konkretnom tipu objekta koji se krije iza date reference, a ne prema tipu same reference. Opet su pravila potpuno analogna pravilima polimorfizma u objektno orijentisanim programskim jezicima kao što su C++ ili Java.



Slika 4.10: Polimorfizam. (a) Definicija podstrukture može se pridružiti nekom tipu `TSBase` iz izvorišnog metamodela pomoću označene vrednosti `ElemType` parametrizovanog paketa. (b) Za tip iz izvorišnog metamodela `TSDerived` koji je izveden iz tipa `TSBase` može se redefinisati podstruktura sa istim imenom i parametrima. (c) Prilikom referisanja (poziva) podstrukture, generiše se ona podstruktura koja odgovara konkretnom tipu elementa izvorišnog modela koji je referisan označenom vrednošću `Elem`, polimorfno.

Za dati primer na slici 4.10c repeticija `<<ForEach>>FEP` vrši se preko reference `e` na tekući element zadate kolekcije. Ta referenca je tipa `TSBase`, ali to znači da ona može biti vezana i za objekte osnovne klase `TSBase`, ali i iz nje izvedene klase `TSDerived` (jer su i oni tipa `TSBase`). Ako je u nekoj iteraciji referenca `e` vezana za objekat klase `TSBase`, biće pozvana podstruktura definisana na slici 4.10a. Ako se pak iza reference `e` krije objekat klase `TSDerived`, biće pozvana podstruktura definisana na slici 4.10b.

Na ovaj način se u potpunosti postiže efekat da se specifikacije preslikavanja mogu pridruživati tipovima iz izvorišnog metamodela i one redefinisati u hijerarhiji tih tipova, uz polimorfizam. To je u potpunosti analogno definisanju polimorfni operacija u hijerarhiji klasa u objektnom modelovanju. Bitno je ipak uočiti da se definicije podstrukture samo posredno vezuju za te tipove iz izvorišnog metamodela, ali da se ti tipovi ne "prljaju" tim definicijama, odnosno da te definicije ne pripadaju samom metamodelu, nego preslikavanju. Zbog toga je moguće definisati potpuno različita preslikavanja za isti izvorišni metamodel, bez izmene tog metamodela. Ovaj pristup je po tome sličan projektnom obrascu *Visitor*.

Organizacija specifikacija

Preslikavanje domena može biti veoma složeno, pa je organizacija specifikacije preslikavanja i mogućnost njene dobre dekompozicije važan aspekt. Sa druge strane, preslikavanje domena se definiše pomoću instanci apstrakcija kao što su `Instance`, `Link`, `Package` i slično, pa specifikacija preslikavanja predstavlja zapravo model čiji je metamodel podskup UML metamodela. Zbog toga je preslikavanje kao model moguće dekomponovati i organizovati potpuno u skladu sa filozofijom jezika UML [B-Boo99].

Preslikavanje domena, kao model sa objektnim metamodelom, može se zato hijerarhijski organizovati u pakete, kao na slici 4.2b. Hijerarhija počinje paketom sa stereotipom `<<DomainMapping>>`. Taj paket može sadržati sledeće vrste paketa:

- `<<Substruct>>` koji predstavljaju definicije podstrukture. Ovi paketi se mogu nalaziti samo unutar korenog paketa `<<DomainMapping>>` i ne mogu sadržati ugneždene pakete tipa `<<Substruct>>`, ali mogu sadržati druge pakete. Ovo zapravo u potpunosti odgovara pristupu u jeziku C u kome se program sastoji od definicija funkcija koje se ne mogu statički ugnežđivati, pa su sve one ravnopravne i definisane na istom globalnom nivou.
- `<<ForEach>>`, `<<Ref>>` i pakete bez stereotipa. Ovi paketi predstavljaju zahteve za kreiranje odgovarajućih delova odredišnog modela (odnosno podstrukture). Osim što `<<Ref>>` paketi ne mogu sadržati druge pakete, ovi paketi se mogu proizvoljno međusobno ugnežđivati. Unutar korenog `<<DomainMapping>>` paketa može se naći proizvoljno mnogo ovakvih paketa, što je zapravo analogno nizu naredbi unutar glavnog programa na nekom tradicionalnom programskom jeziku.

Potpuno u skladu sa definicijom jezika UML, svaki paket može da sadrži elemente ili da referiše elemente koji pripadaju drugim paketima. Primer kada jedan paket referiše element iz drugog paketa može da bude sledeći. Neka paket A sadrži pakete B i C, i neka paket B sadrži instancu b (to je zapravo instanca tipa `Instance` iz metamodela za preslikavanje domena), a paket C instancu c. Neka je zatim potrebno definisati vezu (instancu tipa `Link` iz metamodela za preslikavanje domena) između b i c. Ta veza pripada paketu A i definisana je u tom paketu tako što su u njega uvezeni elementi b i c, što znači da ih paket A ne sadrži nego samo referiše. Ovo referisanje nema nikakve veze sa pojmom relacije zavisnosti sa stereotipom `<<Ref>>` u preslikavanju, već se odnosi na pojam iz definicije jezika UML [B-OMG99].

Drugi važan element organizacije preslikavanja, ponovo u skladu sa jezikom UML, jesu dijagrami. Specifikacija preslikavanja je zapravo model čiji se različiti delovi mogu prikazati odgovarajućim dijagramima. Ti dijagrami su ovde prošireni objektni dijagrami poput

onih koji su prikazani u navedenim primerima. Dijagram je samo jedan prikaz dela modela pomoću prezentacionih elemenata (grafičkih simbola) koji prikazuju odgovarajuće elemente modela. Jedan element modela, npr. instanca x , može da se prikaže na više različitih dijagrama, što znači da svaki od tih dijagrama sadrži drugi prezentacioni element koji predstavlja isti element modela x . Svaki paket može sadržati proizvoljno mnogo ovakvih dijagrama koji prikazuju kako elemente koje taj paket sadrži, tako i uvezene elemente koje taj paket referiše. Na taj način se preslikavanje može dekomponovati na proizvoljno mnogo dijagrama koji se hijerarhijski organizuju. To omogućuje da dijagrami budu pregledni, nepretrpani, već osmišljeni tako da prikazuju samo po jedan bitan aspekt preslikavanja. Sa druge strane, model preslikavanja koji leži ispod prikaza pomoću dijagrama ostaje konzistentan i jedinstven, jer to obezbeđuje sam alat za modelovanje. Ovakav pristup je u potpunosti usklađen sa filozofijom savremenog vizuelnog modelovanja softvera koji podržava UML [B-Boo99].

Treba dakle jasno razlikovati tri aspekta preslikavanja domena:

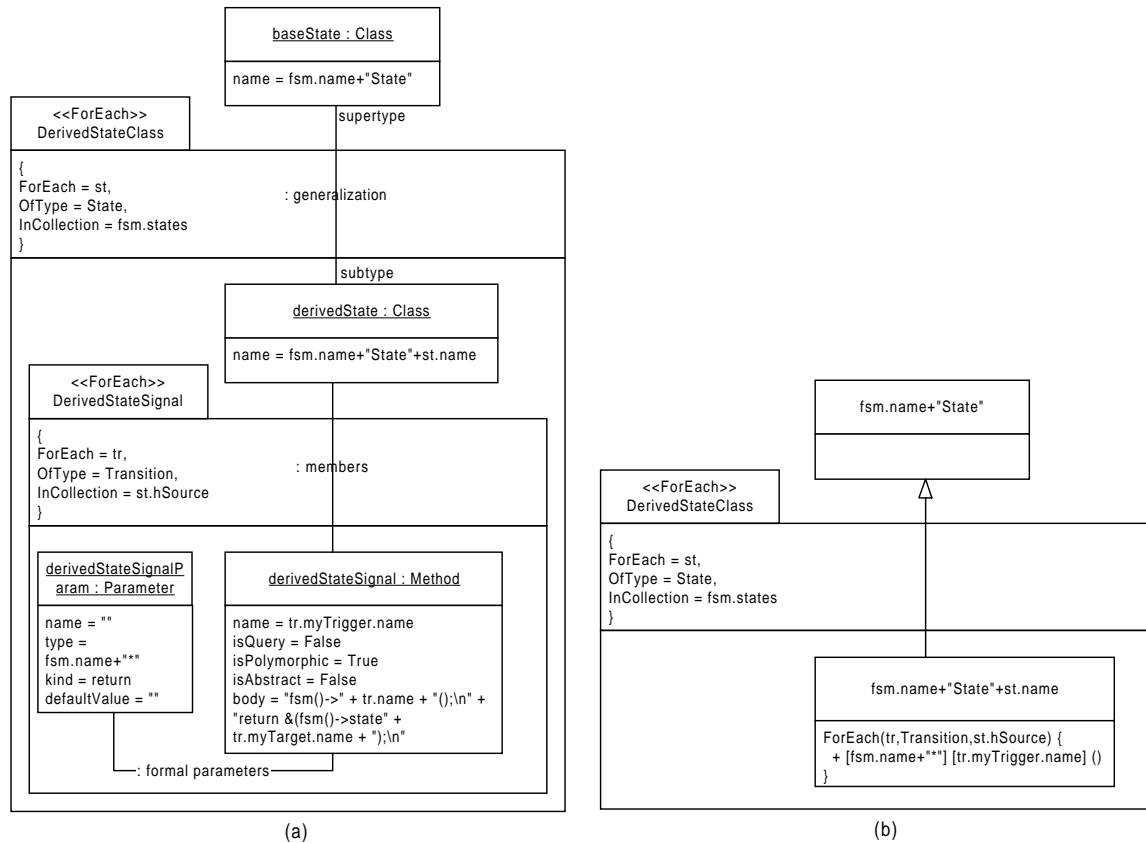
- Preslikavanje domena je zapravo takođe jedan model koji se oslanja na objektno orijentisani metamodel koji je odgovarajući podskup UML metamodela. Prema tome, taj model se sastoji od instanci odgovarajućih apstrakcija (*Instance*, *Link*, *Package* itd.) i njihovih veza kao instanci asocijacija iz metamodela (npr. instanca tipa *Link* povezana je sa dve instance tipa *Instance*, ili instanca tipa *Package* povezana je vezama asocijacije *owns* sa elementima koje sadrži [B-OMG99]). Kao što je već više puta naglašeno, inherentna struktura ovakvog modela je tipizirani graf.
- Preslikavanje domena kao model može se prikazati hijerarhijski pomoću strukture ugnežđenih paketa, kao na slici 4.2b. Bitno je uočiti da je ovakva predstava samo jedan od mogućih prikaza pomenute inherentne strukture grafa samog modela. Naime, hijerarhijska struktura paketa je samo jedno stablo koje se "razapinje" preko strukture grafa preko čvorova odgovarajućeg tipa (*Package* i bilo koji element) i veza odgovarajućih asocijacija (pomenuta asocijacija "sadržavanja" *owns*). Dakle, ovo je samo jedan od mogućih prikaza modela preslikavanja koji je bitan sa dva aspekta. Prvo, on je intuitivno jasan i pregledan za korisnika i drugo, on je bitan za proces interpretacije specifikacije odnosno generisanja automatskog transformatora.
- Dijagrami su drugi važan način predstavljanja modela preslikavanja. Kao što je već opisano, dijagrami grafički prikazuju samo određene delove modela pomoću odgovarajuće notacije.

Svi ovi elementi zajedno obezbeđuju dobru dekompoziciju i lakše snalaženje u složenim preslikavanjima domena.

Korišćenje specifične notacije domena

Jedan nedostatak do sada prikazanog pristupa je to što su dijagrami nedovoljno slikoviti, u smislu da se notacija svodi samo na simbole za pakete, instance i veze. Instance i veze predstavljaju specifikacije kreiranja elemenata modela iz odredišnog domena, pri čemu je veoma čest slučaj da odredišni domen poseduje svoju specifičnu notaciju. Iako korišćenje ovakve opšte notacije za ono što model preslikavanja zapravo predstavlja obezbeđuje precizan formalizam za automatsku interpretaciju preslikavanja, ona ipak uzrokuje relativno slabu izražajnost koja rezultuje složenim dijagramima.

Posmatrajmo primer na slici 4.11a za konačne automate. To je isti primer sa slike 4.6 koji prikazuje specifikaciju generisanja osnovne klase stanja, izvedene klase za svako konkretno stanje automata i po jedne operacije u njoj za svaki događaj na koji automat reaguje. Sve su to instance određenih apstrakcija iz odredišnog domena povezane



Slika 4.11: Korišćenje specifične notacije odredišnog domena u dijagramu preslikavanja za demonstrativni primer konačnih automata. (a) Originalni dijagram objekata sa slike 4.6 koji prikazuje specifikaciju generisanja osnovne klase stanja, izvedene klase za svako konkretno stanje automata i po jedne operacije u njoj za svaki događaj na koji automat reaguje. (b) Ekvivalentni dijagram ali u specifičnoj notaciji odredišnog domena (UML).

odgovarajućim vezama, pa stoga preslikavanje i izgleda onako kako je prikazano dijagramom objekata na slici 4.11a.

Međutim, odredišni domen (UML) u ovom slučaju poseduje svoju specifičnu notaciju. U toj notaciji klasa se prikazuje kao pravougaonik u čijem je gornjem odeljku ispisano njeno ime, a u jednom od odeljaka su deklaracije njenih operacija. Prema tome, ista specifikacija sa slike 4.11a može se prikazati ekvivalentnim dijagramom na slici 4.11b koji koristi ovu specifičnu notaciju odredišnog domena. Naravno, ova notacija mora biti proširena na odgovarajući način kako bi podržala koncepte preslikavanja domena za odgovarajuće elemente. Na primer, u datom primeru je potrebno generisati po jednu operaciju za svaki događaj. Ovo je definisano `ForEach` konstruktom unutar sekcije sa operacijama izvedene klase stanja. Slično, vrednosti atributa instanci iz odredišnog modela, kao što su ime klase ili operacije, dobijaju se pomoću odgovarajućih izraza, pa i to treba omogućiti.

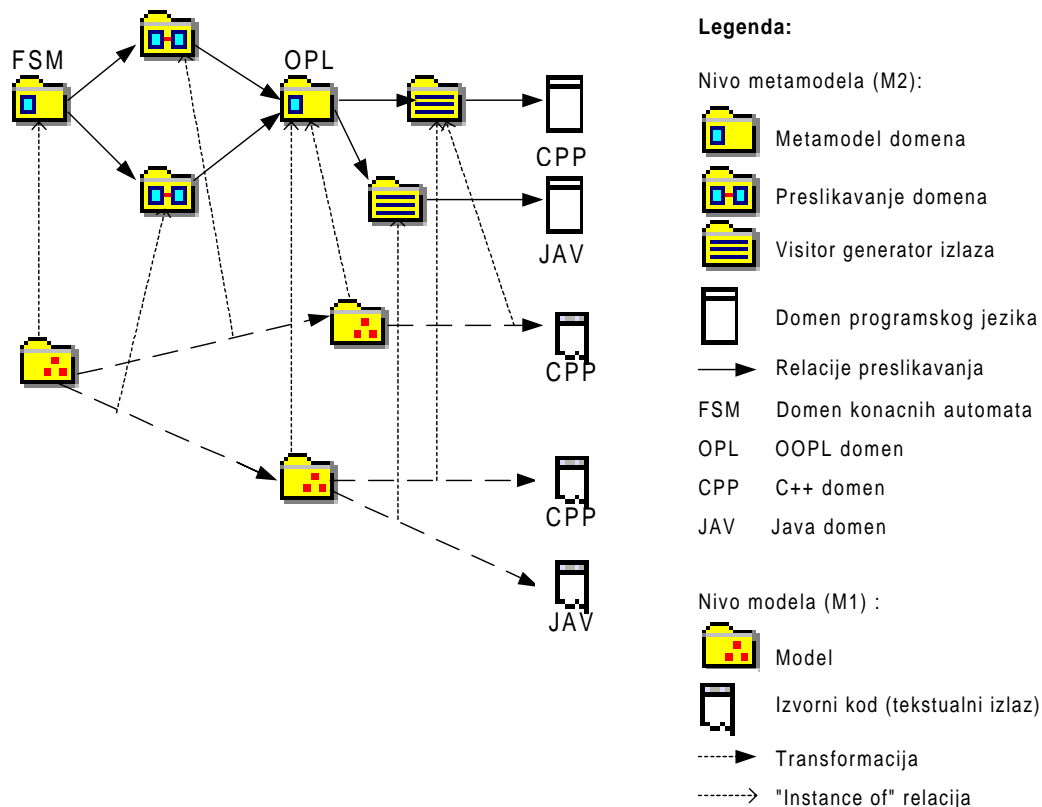
Ovakav pristup predstavlja zapravo samo notacionu pogodnost koja vizuelne specifikacije čini razumljivijim i konciznijim, ali nikako ne menja do sada izloženu semantiku preslikavanja. Način vezivanja prezentacionih elemenata specifične notacije odredišnog domena (kao na slici 4.11b) i elemenata samog modela preslikavanja (kao na slici 4.11a) nije predmet predložene metode. Interesantno je uočiti da bi to vezivanje (zapravo preslikavanje) takođe moglo biti definisano korišćenjem istih predloženih koncepata preslikavanja domena. Ovo je jedan aspekt koji ostaje još uvek otvoren za dalja istraživanja i precizniju formulaciju, ali i koji predstavlja značajan potencijal za poboljšanje izražajnosti predložene metode.

Sukcesivne transformacije modela

Predložena metoda dozvoljava značajno uopštenje koje se sastoji u mogućnosti da se transformacije modela vrše sukcesivno, u lancu, uz izbor odgovarajućih preslikavanja domena. Značajna fleksibilnost postiže se mogućnošću da se za iste domene definišu različita preslikavanja, ali i time što se jedno preslikavanje ili metamodel domena može definisati kao proširenje nekog već postojećeg. Najzad, mehanizam koji podržava restauraciju manualnih korekcija generisanog modela prilikom njegove regeneracije može da bude značajan u postupku spuštanja nivoa apstrakcije korisničkog modela sve do konačne implementacije sistema. Svi ovi elementi opisani su detaljnije u ovom poglavlju.

Lančane transformacije modela

Između dva domena može se definisati proizvoljno mnogo preslikavanja. Ta preslikavanja mogu da budu specifikacije različitih željenih varijacija transformacija. Ovo se jednostavno realizuje definisanjem različitih paketa sa stereotipom `<<DomainMapping>>` koji imaju relacije zavisnosti `<<source>>` i `<<dest>>` prema istim parovima paketa sa stereotipom `<<DomainMetamodel>>`. Na slici 4.12 ove relacije prikazane su punim linijama sa strelicama sa punim vrhom, dok su paketi sa definicijom preslikavanja i metamodeli predstavljeni



Slika 4.12: Primer sukcesivne transformacije modela. U fazi metamodelovanja definišu se metamodeli i njihova preslikavanja. U fazi modelovanja korisnik definiše svoj početni model koji se potom automatski transformiše u jedan ili više ciljnih modela izborom odgovarajućeg lanca preslikavanja.

posebnim simbolima, prema datoj legendi (ovaj pristup definisanja posebnih simbola za određene stereotipove je sasvim u skladu sa jezikom UML).

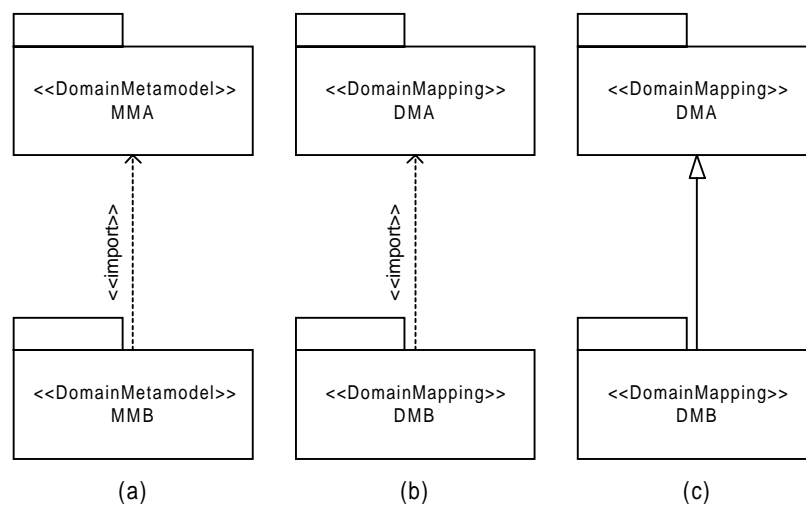
U ovom primeru definisan je poseban domen koji je nazvan OOPL (engl. *object oriented programming language*). Ovaj domen koristi se kao međudomen umesto do sada korišćenog podskupa UML metamodela. U njemu su definisani osnovni objektni koncepti koji se sreću u praktično svim objektno orijentisanim tekstualnim jezicima: klasa, podatak član, funkcija članica, nasleđivanje itd., ali ne i neke složenije apstrakcije koje podržava UML, kao što je asocijacija. Ovo je učinjeno zbog toga što je ovakav domen bolje prilagođen ciljnim domenima u ovom primeru, a to su konkretni programski jezici (C++ ili Java). Oni naime ne podržavaju direktno koncept asocijacije, nego samo koncepte podatka člana i funkcije članice klase. Primeri primene predložene metode ukazali su da je ovaj pristup pogodniji, jer se domen OOPL lako preslikava u tekstualne programske jezike, dok se iznad njega opet mogu lako izgraditi apstraktniji domeni kao što je UML domen, koji se opet lako preslikavaju u njega.

U primeru su takođe definisana dva preslikavanja iz domena konačnih automata (FSM) u domen OOPL. Sa druge strane, definisana su takođe dva preslikavanja iz domena OOPL u domene konkretnih objektno orijentisanih programskih jezika C++ i Java. Ova druga preslikavanja realizovana su pomoću generatora koda baziranih na projektnom obrascu *Visitor*, zato što je to pogodan način za preslikavanje objektnog u tekstualni domen. Na ovaj način je postignuto da različita preslikavanja iz domena FSM u međudomen OOPL budu nezavisna od konkretnog ciljnog programskog jezika. Ta različita preslikavanja mogu da predstavljaju različite željene varijacije implementacije konačnih automata. Sa druge strane, domen OOPL i njegova preslikavanja u ciljne programske jezike su opet potpuno nezavisni od konkretne upotrebe u ovom primeru konačnih automata, pa se mogu ponovno upotrebiti za mnoge druge potrebe.

Model je instanca metamodela i predstavlja se paketom koji je u relaciji zavisnosti stereotipa <<instanceOf>> sa odgovarajućim paketom njegovog metamodela (slika 4.12). Odredišni model dobija se odgovarajućom transformacijom iz izvorišnog modela. Transformacija je na slici 4.12 predstavljena isprekidanom linijom sa strelicom sa punim vrhom od izvorišnog prema odredišnom modelu. Svaka transformacija određena je tačno jednim preslikavanjem između odgovarajućih metamodela.

Prema tome, modeli se mogu transformisati sukcesivno, lančano (engl. *model pipelining*). Ovakvom lančanom transformacijom se od nekog početnog, korisnički definisanog modela može automatski dobiti jedan ili više ciljnih modela koji predstavljaju željenu implementaciju datog sistema koji se konstruiše. Međumodeli u lancima transformacija predstavljaju stepenice u spuštanju nivoa apstrakcije početnog, visoko apstraktnog modela sve do implementacionog modela niskog nivoa apstrakcije koji dalje može biti interpretiran u nekom drugom okruženju. To drugo okruženje su u datom primeru na slici 4.12 standardni, raspoloživi programski prevodioci za jezike C++ i Java. Oni dalje transformišu dobijeni izvorni kod u mašinski program koji se može izvršiti na mašini (opet novi model sistema koga interpretira drugi subjekat – hardver računara) da bi se dobilo željeno ponašanje.

Sve ovo predstavlja jedan fleksibilan pristup za modelovanje na visokom nivou apstrakcije, prilagođen specifičnom domenu, uz korišćenje raspoloživih domena i preslikavanja nižeg nivoa, a sve u cilju efikasnog konstruisanja specifičnih i složenih sistema.



Slika 4.13: Relacije između metamodela i preslikavanja domena. (a) Metamodel MMB *uvozi* (engl. *import*) drugi metamodel MMA (relacija zavisnosti stereotipa <<import>>). (b) Preslikavanje DMB *uvozi* (engl. *import*) drugo preslikavanje DMA (relacija zavisnosti stereotipa <<import>>). (c) Preslikavanje DMB *specijalizuje* (engl. *specialize*) drugo preslikavanje DMA (relacija generalizacije/specijalizacije, tj. nasleđivanja).

Višestruka upotreba preslikavanja

Da bi se postigla potpuna fleksibilnost i što lakša ponovna upotrebljivost (engl. *reusability*) postojećih metamodela i preslikavanja, potrebno je obezbediti mogućnost da se novi metamodel ili preslikavanje definiše na osnovu nekog već postojećeg, njegovim proširivanjem i/ili redefinisanjem. Ovo je jedan od osnovnih elemenata modernog objektno orijentisanog programiranja koji programski jezici podržavaju konceptima nasleđivanja i polimorfizma, pa ga zato u ovom kontekstu takođe treba podržati odgovarajućim relacijama između metamodela i preslikavanja.

U tu svrhu definisana je najpre relacija *uvoženja* (engl. *import*) između metamodela. Ona se predstavlja relacijom zavisnosti stereotipa <<import>> između dva paketa stereotipa <<DomainMetamodel>>, kao što je prikazano na slici 4.13a. Značenje ove relacije je sledeće. Ako metamodel MMB uvozi metamodel MMA (slika 4.13a), onda to znači da metamodel MMB sadrži implicitno sve elemente koji su definisani u MMA, pored onih elemenata koji eksplicitno pripadaju paketu MMB. Treba primetiti da ovo nije relacija specijalizacije paketa kako je definisana jezikom UML [B-Boo99], jer se prema ovako definisanom značenju relacije uvoženja paket MMB ne može upotrebiti svugde gde se očekuje paket MMA. Naime, metamodel MMB može definisati nove apstrakcije, attribute ili asocijacije domena, pa model sa tim metamodelom može da bude nekonzistentno generisan pomoću nekog preslikavanja koje za svoj odredišni domen ima MMA. Na primer, metamodel MMB može da definiše neku novu asocijaciju koja povezuje apstrakciju (klasu) *x* koja postoji u MMA i apstrakciju *y* koja ne postoji u MMA, uz odgovarajuća ograničenja te asocijacije koja definišu pravila konzistentnosti modela. Model generisan pomoću preslikavanja koje ima odredišni domen MMA može biti nepotpun, odnosno nekonzistentan prema pravilima iz metamodela MMB, jer to preslikavanje ne poznaje apstrakciju *y*. To znači da se MMB ne može upotrebiti kao odredišni domen datog preslikavanja.

Između dva preslikavanja takođe može postojati relacija uvoženja, kao na slici 4.13b. Potpuno analogno, ako preslikavanje *DMB* uvozi preslikavanje *DMA* (slika 4.13b), onda to znači da preslikavanje *DMB* implicitno sadrži sve elemente koji su definisani u *DMA*, pored onih elemenata koji eksplicitno pripadaju paketu *DMB*. Ponovo ovo nije specijalizacija, jer *DMB* može imati potpuno druge metamodela kao svoje izvoriste i odredište. Osim toga, *DMB* može uvesti i druge pakete sa preslikavanjima.

Druga relacija koja može postojati između preslikavanja je *generalizacija/specijalizacija* (engl. *generalization/specialization*), odnosno *nasleđivanje* (engl. *inheritance*), kao na slici 4.13c. Značenje ove relacije je sledeće. Prvo, <<DomainMapping>> paket *DMB* koji specijalizuje <<DomainMapping>> paket *DMA* implicitno ima isti par izvorišnog i odredišnog metamodela, pa se oni ne definišu eksplicitno, već se nasleđuju. Drugo, paket *DMB* implicitno uvozi paket *DMA*, što znači da implicitno sadrži i sve elemente iz ovog paketa, prema već definisanom značenju ove relacije uvoženja. Treće, preslikavanje *DMB* može *redefinisati* (engl. *override*) neke ili sve definicije podstrukture iz *DMA*, kako one vezane za tipove iz izvorišnog domena, tako i one nevezane za tipove. Redefinicija podstrukture u *DMB* vrši se prostim definisanjem strukture sa istim imenom i tipovima formalnih parametara. Prema tome, ova relacija u potpunosti zadovoljava opšta značenja pojma generalizacije/specijalizacije u objektno orijentisanoj filozofiji, koja obuhvataju pravila supstitucije (*DMB* se može upotrebiti svugde gde i *DMA* jer ima isti izvorišni i odredišni domen), nasleđivanja (*DMB* implicitno sadrži sve što i *DMA*) i redefinisavanja odnosno polimorfizma (*DMB* redefiniše podstrukture iz *DMA*). Višestruko nasleđivanje dozvoljeno je samo ako nasleđeni paketi imaju iste parove izvorišnih i odredišnih domena.

Prema tome, važno je razlikovati dva činioca preslikavanja:

- izvorišni i odredišni metamodel i
- sadržaj, odnosno elemente koje paket sadrži i koji definišu preslikavanje izvorišnog u odredišni domen.

Kada je potrebno definisati novi paket preslikavanja *DMB* korišćenjem nekog već postojećeg paketa *DMA*, u zavisnosti od toga šta novi paket preslikavanja preuzima od postojećeg a šta ne, između njih postoji i odgovarajuća relacija:

- Ako je potrebno da preslikavanje *DMB* preuzme sve elemente iz *DMA*, eventualno ih proširi, a pri tom ima različit izvorišni i/ili odredišni metamodel, onda *DMB* samo uvozi *DMA* (slika 4.13b).
- Ako *DMB* ima isti izvorišni i odredišni metamodel, a potrebno je da preuzme elemente iz *DMA* i eventualno ih proširi i/ili redefiniše, onda *DMB* specijalizuje *DMA* (slika 4.13c).
- Ako ne važi nijedno od ova dva, preslikavanja nisu u relaciji.

Ovi koncepti omogućuju stvaranje repozitorijuma (engl. *repository*) metamodela i njihovih preslikavanja koji se mogu stalno proširivati i obogaćivati, definisanjem novih metamodela i preslikavanja uz ponovno korišćenje postojećih. Intenzivnija i duža upotreba predložene metode na većem broju praktičnih projekata trebalo bi da oformi ovakve repozitorijume, kao i da ukaže na eventualne izmene, unapređenja i proširenja predloženih koncepata i njihove semantike.

Manuelne modifikacije modela

Kada se generiše odredišni model, okruženje koje podržava predloženu metodu dozvoljava korisniku da ručno unese izmene tog modela. Te izmene okruženje beleži. Istorija tih izmena beleži se od prvog generisanja datog modela kumulativno. Kada korisnik želi da ponovo generiše isti odredišni model, okruženje prvo automatski generiše potpuno novi model

Operacije sa paketima

Operacija	Značenje	Parametri	Preduslovi
<i>create</i>	Kreiranje paketa	<i>parentPckID</i> : ID nadređenog paketa kome će kreirani paket pripadati <i>pckName</i> : ime kreiranog paketa	Staza definisana pomoću <i>parentPckID</i> je korektna. <i>pckName</i> je jedinstven unutar <i>parentPckID</i> .
<i>delete</i>	Brisanje paketa	<i>packageID</i> : ID obrisano paketa	Staza definisana pomoću <i>packageID</i> je korektna.

Operacije sa instancama

Operacija	Značenje	Parametri	Preduslovi
<i>create</i>	Kreiranje instance	<i>parentPckID</i> : ID nadređenog paketa kome će kreirana instance pripadati <i>instName</i> : ime kreirane instance <i>type</i> : referenca na tip instance	Staza definisana pomoću <i>parentPckID</i> je korektna. <i>instName</i> je jedinstven unutar <i>parentPckID</i> za dati tip. Referenca na tip je korektna.
<i>modify</i>	Promena vrednosti atributa instance	<i>instID</i> : ID modifikovane instance <i>attribute</i> : referenca na modifikovani atribut <i>newVal</i> : nova vrednost atributa	Staza definisana pomoću <i>instID</i> je korektna. Referenca na atribut je korektna. Nova vrednost atributa je validna.
<i>delete</i>	Brisanje instance	<i>instID</i> : ID obrisane instance	Staza definisana pomoću <i>instID</i> je korektna.

Operacije sa vezama

Operacija	Značenje	Parametri	Preduslovi
<i>create</i>	Kreiranje veze	<i>inst1ID</i> : ID prve instance <i>inst2ID</i> : ID druge instance <i>assoc</i> : tip veze (asocijacija)	Staze definisane pomoću <i>inst1ID</i> i <i>inst2ID</i> su korektne. Referenca na asocijaciju je korektna.
<i>delete</i>	Brisanje veze	<i>inst1ID</i> : ID prve instance <i>inst2ID</i> : ID druge instance <i>assoc</i> : tip veze (asocijacija)	Staze definisane pomoću <i>inst1ID</i> i <i>inst2ID</i> su korektne. Referenca na asocijaciju je korektna.

Tabela 4.1: Raspoložive manuelne operacije nad elementima generisanog modela koje se beleže i mogu se restaurirati prilikom regenerisanja tog modela.

ispočetka, korišćenjem odgovarajućeg eventualno promenjenog izvorišnog modela i datog preslikavanja. Posle toga, okruženje primenjuje sve zabeležene manuelne izmene po hronološkom redosledu. Za svaku izmenu potrebno je da bude zadovoljen određeni uslov. Ukoliko taj uslov nije zadovoljen, izmena se jednostavno odbacuje (ignoriše). Ovo može da se dogodi ako se izmena odnosi na neki element odredišnog modela koji nije generisan prilikom ponovne transformacije jer je izvorišni model u međuvremenu izmenjen.

Mogućnost beleženja istorije promena modela oslanja se na postojanje jedinstvene identifikacije svakog elementa odredišnog modela. Jedan deo jedinstvene identifikacije elementa modela je svakako njegov tip (paket, klasa instance ili asocijacija veze). Drugi deo identifikacije za instancu ili paket čini njeno puno ime koje se sastoji iz staze u hijerarhiji paketa do neposrednog paketa koji sadrži taj element i imena tog elementa. Veza se identifikuje svojom asocijacijom i dvema instancama koje ona povezuje; ako je veza instance

asocijacione klase, identifikuje se kao instanca te klase. Višestruke veze iste asocijacije između iste dve instance nisu dozvoljene, osim ako su to instance asocijacionih klasa. Svaka izmena modela tretira se kao operacija izvršena na određenom elementu modela uz određene parametre. U Tabeli 4.1 navedene su sve raspoložive operacije, zajedno sa svojim parametrima i preduslovima izvršavanja. *ID* predstavlja opisanu jedinstvenu identifikaciju instance ili paketa.

Kao što je ranije već opisano, upotreba ove pogodnosti ima smisla ukoliko je generisani model još uvek na dovoljno visokom nivou apstrakcije da ga korisnik može razumeti, a da pri tom i želi da u njega unese izmene. Korisnik može da bude zadovoljan najvećim delom generisanog modela, ali da ipak želi da učini male modifikacije kako bi taj model prilagodio svojim potrebama. Naravno, te izmene će biti propagirane u narednim sukcesivnim transformacijama tog generisanog modela, sve do željene krajnje implementacije.

**V Detalji
predloženog
rešenja**

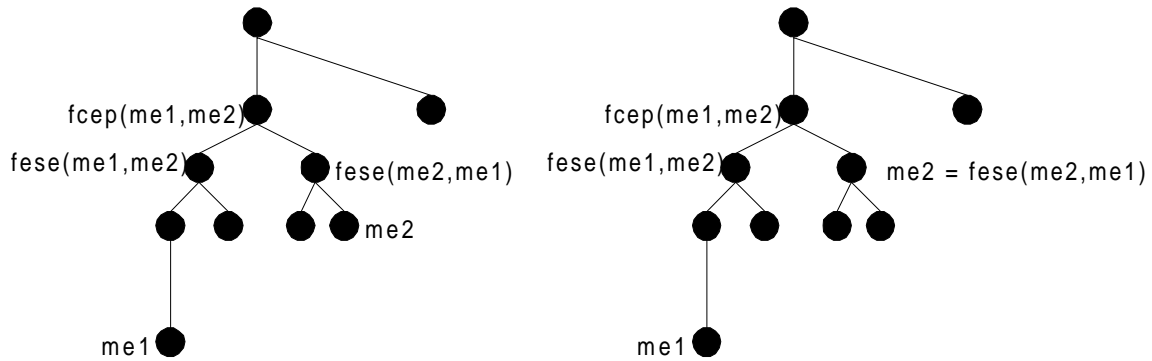
Semantika preslikavanja domena

Tabela 5.1 daje sumarni pregled svih do sada predloženih koncepata preslikavanja domena. Za svaki koncept dato je njegovo značenje u kontekstu kreiranja delova odredišnog modela, kao i odgovarajući pandan u strukturiranim ili objektno orijentisanim programskim jezicima. Ovo poređenje sa tradicionalnim konceptima biće značajno za implementaciju, odnosno formiranje automatskog transformatora, implementiranog u nekom klasičnom programskom jeziku, iz specifikacija preslikavanja.

Do sada je značenje predloženih koncepata izloženo neformalno, bez strogih definicija. U ovoj glavi date su formalne definicije semantike koncepata preslikavanja. Najpre su navedene neke opšte definicije, zatim opis značenja i načina kreiranja implicitnih sekvencijalnih zavisnosti, pravila konzistentnosti preslikavanja domena i, najzad, definicije semantike predloženih koncepata.

Koncept u preslikavanju domena	Značenje	Odgovarajući koncept u programskom jeziku
Instanca bez stereotipa	Specifikacija kreiranja instance u odredišnom modelu	Naredba koja kreira objekat
Instanca <<Ref>>	Referenca na instancu odredišnog modela	Referenca na objekat
Atribut instance	Specifikacija vrednosti atributa instance odredišnog modela	Naredba koja postavlja vrednost atributa objekta
Veza	Specifikacija kreiranja veze između instanci odredišnog modela	Naredba koja kreira vezu između objekata
Zavisnost <<Seq>>	Specifikacija redosleda kreiranja elemenata odredišnog modela	Sekvenca naredbi
Zavisnost <<Ref>>	Vezivanje reference za instancu	Vezivanje reference za objekat
Paket bez stereotipa	Specifikacija kreiranja podstrukture	Blok naredbi
Paket <<ForEach>>	Specifikacija repetitivnog kreiranja podstrukture	Petlja čije je telo blok naredbi
Paket <<Substruct>>	Definicija parametrizovane podstrukture	Procedura
Paket <<Ref>>	Referenca na podstrukturu (poziv podstrukture)	Poziv procedure
Paket <<Substruct>> koji ima ElemType	Definicija podstrukture pridružene tipu iz izvorišnog metamodela	Operacija klase
Paket <<Ref>> koji ima Elem	Polimorfni poziv podstrukture pridružene tipu iz izvorišnog metamodela	Polimorfni poziv operacije klase

Tabela 5.1: Sumarni prikaz koncepata u preslikavanju domena, njihovog značenja i analognih koncepata u tradicionalnom proceduralnom ili objektno orijentisanom programskom jeziku.



Slika 5.1: Različiti slučajevi međusobnih pozicija elemenata modela *me1* i *me2* u hijerarhiji paketa i elementi *fcep* (prvi zajednički okružujući paket) i *fese* (prvi okružujući bratski element) za te slučajeve. Elementi su prikazani kao čvorovi stabla u hijerarhiji. Čvorovi koji imaju potomke su paketi.

Osnovne definicije

Element preslikavanja domena (engl. *domain-mapping element*) je element modela (engl. *model element*) [B-OMG99] preslikavanja domena, tačnije element hijerarhije paketa koja počinje od paketa sa stereotipom <<DomainMapping>>.

Poštujući semantiku jezika UML, svaki element preslikavanja domena, kao i element odredišnog modela, u vlasništvu je tačno jednog nadređenog paketa, osim samog korenog paketa. Ako paket *p* *posедуje* (engl. *owns*) element modela *me*, kaže se i da element *me* *pripada* (engl. *belongs to*) paketu *p*; notacija je: $owns(p, me) \Leftrightarrow belongs(me, p)$.

Relacija "pripada" proširuje se na celu hijerarhiju paketa relacijom *je-unutar* (engl. *is-in*): element preslikavanja domena ili odredišnog modela *me* *je unutar* paketa *p*, u oznaci $isIn(me, p)$, ako i samo ako postoji sekvenca paketa $p_0, p_1, \dots, p_n = p, n \geq 0$, takvih da važi $belongs(me, p_0)$ i $belongs(p_i, p_{i+1}), 0 \leq i < n$. Tzv. *prvi zajednički okružujući paket* (engl. *first common enclosing package*) dva elementa *me1* i *me2* ($me1 \neq me2$) se definiše kao: $fcep(me1, me2) = p$ akko $isIn(me1, p)$ i $isIn(me2, p)$, i ne postoji drugi paket *p'* takav da je $isIn(me1, p')$ i $isIn(me2, p')$ i $isIn(p', p)$. Pošto je hijerarhija paketa stablo, $fcep(me1, me2)$ je jedinstven za svaki par *me1* i *me2*, a takođe je i $fcep(me1, me2) = fcep(me2, me1)$. Prvi zajednički okružujući `ForEach` paket dva elementa preslikavanja domena *dme1* i *dme2*, ako postoji, označava se sa $fcefep(dme1, dme2)$.

Uvodi se takođe i pojam *prvih okružujućih bratskih elemenata* (engl. *first enclosing sibling elements*) dva elementa modela *me1* i *me2*. Prvi okružujući bratski element elemenata *me1* i *me2*, u tom poretku, u oznaci $fese(me1, me2)$, je sam *me1* ako $belongs(me1, fcep(me1, me2))$, odnosno paket *p1* takav da $isIn(me1, p1)$ i $belongs(p1, fcep(me1, me2))$ inače; $fesp(me2, me1)$ se definiše simetrično. Lako je videti da $fese(me1, me2)$ i $fese(me2, me1)$ pripadaju istom paketu $fcep(me1, me2)$, tj. oni su "bratski" elementi (sadržani unutar istog roditeljskog paketa) u hijerarhiji paketa.

Značenje relacije "pripada" preuzeto je iz definicije jezika UML. Specijalno, veza unutar preslikavanja domena koja povezuje dve instance *dmi1* i *dmi2* pripada prvom zajedničkom okružujućem paketu instanci *dmi1* i *dmi2*.

Postoji implicitna relacija *porekla* elementa odredišnog modela *tme* prema elementu preslikavanja domena *dme* čija je *tme* posledica; drugim rečima, *tme* je *generisan* (ili *kreiran*) od *dme* u procesu generisanja odredišnog modela.

U jeziku UML, element modela jednog tipa mora imati jedinstveno ime unutar paketa kome pripada. Isto važi i za elemente preslikavanja domena. Ovo pravilo treba da proverava alat koji podržava metodu. Provera ostalih pravila konzistentnosti, npr. multiplikativnosti veza takođe je moguća, ali se ostavlja u nadležnost implementatoru alata.

Sekvencijalne zavisnosti

Osim eksplicitnih sekvencijalnih zavisnosti koje je definisao korisnik u preslikavanju domena, u cilju korektnog generisanja odredišnog modela, uvode se još neke implicitne sekvencijalne zavisnosti između elemenata preslikavanja. Tako je veza implicitno sekvencijalno zavisna od instanci koje povezuje. Osim toga, eksplicitna sekvencijalna zavisnost ima smisla samo ako povezuje elemente koji pripadaju istom paketu. Ovo važi zbog toga što algoritam generisanja odredišnog modela obilazi hijerarhiju paketa u preslikavanju domena po dubini. Prema tome, ako paket *dmpn* pripada paketu *dmp*, onda će svi elementi odredišnog modela koji se generišu kao posledica elemenata sadržanih u hijerarhiji koja počinje od *dmpn* biti kreirani pre procesiranja sledećeg elementa u *dmp*. Iako sekvencijalna zavisnost može da povezuje dva elementa iz proizvoljnih paketa, ona je automatski zadovoljena ako postoji odgovarajuća sekvencijalna zavisnost između prvih okružujućih bratskih elemenata ova dva elementa. Formalno, eksplicitna sekvencijalna zavisnost od *me1* prema *me2* je zadovoljena ako postoji sekvencijalna zavisnost od *fese(me1,me2)* prema *fese(me2,me1)*.

Prema tome, pre nego što se pristupi generisanju odredišnog modela, uspostavljaju se sledeće implicitne sekvencijalne zavisnosti:

- 1) od svake veze prema instancama koje ona povezuje, a zatim
- 2) za svaku sekvencijalnu zavisnost (eksplicitnu ili implicitno uvedenu prethodnim korakom) od *me1* prema *me2*, uvodi se sekvencijalna zavisnost od *fese(me1,me2)* prema *fese(me2,me1)*.

Dobijena struktura grafa sačinjenog od elemenata koji pripadaju jednom paketu (kao čvorova) i sekvencijalnih zavisnosti između njih (kao usmerenih grana), mora da predstavlja usmereni aciklični graf (usmereni graf bez petlji, engl. *directed acyclic graph*, DAG). Ako posle uvođenja navedenih implicitnih sekvencijalnih zavisnosti u nekom od ovih grafova postoji petlja, preslikavanje domena nije korektno i generisanje modela nije moguće.

Pravila konzistentnosti preslikavanja domena

Sva navedena pravila konzistentnosti preslikavanja domena još jednom su ovde sumirana.

Paket <<Substruct>> može pripadati samo korenom paketu <<DomainMapping>>.

Paket <<Ref>> može sadržati samo instance <<Ref>>.

Instanca <<Ref>> može pripadati ili paketu <<Ref>>, ili paketu unutar hijerarhije paketa koja počinje od paketa <<Substruct>>. Instanca <<Ref>> mora imati ime jednog od formalnih argumenata podstrukture kojoj pripada.

Zavisnost <<Ref>> mora polaziti od instance <<Ref>> koja pripada paketu <<Ref>> i završavati u instanci koja pripada nekom od okružujućih paketa ovog paketa <<Ref>>.

Posle opisane procedure uvođenja implicitnih sekvencijalnih zavisnosti <<Seq>>, graf u kome su elementi koji pripadaju jednom paketu čvorovi, a sekvencijalne zavisnosti između njih (i eksplicitne i implicitne) usmerene grane, mora biti acikličan.

Veza koja povezuje dve instance *dmi1* i *dmi2* pripada prvom zajedničkom okružujućem paketu ove dve instance *fcep(dmi1,dmi2)*.

Instanca, paket bez stereotipa, ili paket `<<ForEach>>` moraju imati jedinstveno ime u skupu instanci odnosno paketa koji pripadaju istom roditeljskom paketu. Paketi `<<Substruct>>` koji imaju isto ime moraju imati ili različitu označenu vrednost `ElemType`, ili različit broj i/ili tipove instanci `<<Ref>>` koje predstavljaju formalne argumente.

Semantika generisanja modela

Ako je preslikavanje domena korektno posle uvođenja implicitnih sekvencijalnih zavisnosti, tj. ako ne postoje ciklične zavisnosti između elemenata koji pripadaju jednom paketu, odredišni model se može generisati. Procedura generisanja obilazi hijerarhiju paketa po dubini. Za svaki od paketa u toj hijerarhiji, najpre se njegovi elementi topološki sortiraju prema sekvencijalnim zavisnostima, a onda se u tom poretku obilaze i za njih generišu delovi odredišnog modela.

U nastavku su date precizne definicije značenja pojedinih koncepata preslikavanja domena, u smislu definicije skupa elemenata odredišnog modela koji će biti generisani kao posledica elementa preslikavanja domena datog tipa.

Paket bez stereotipa

Za prosti paket u preslikavanju domena (bez stereotipa) *dmp*, sa imenom *pName*, jedan paket *tmp* u odredišnom modelu, sa imenom *pName*, biće kreiran samo ako izraz definisan u označenoj vrednosti `Cond` paketa *dmp* ima rezultat `True`. Ako element preslikavanja *dme* pripada paketu *dmp*, onda će svaki element *tme* odredišnog domena, generisan iz *dme* pripadati paketu *tmp* generisanom iz *dmp*.

Paket <<ForEach>>

Za `<<ForEach>>` paket *dmp*, sa imenom *pName*, u odredišnom modelu će biti kreiran jedan paket *tmp* sa imenom "ForEach-" + *pName*. Pored toga, za svaki element *smei* iz izvorišnog modela, do koga se dolazi iteracijom definisanom u *dmp*, i sa punim menom (koje uključuje i stazu u hijerarhiji paketa) *smeiFullName*, biće kreiran jedan paket *tmpi*, sa imenom *smeiFullName*, samo ako izraz definisan označenom vrednošću `Cond` paketa *dmp* ima rezultat `True` za taj *smei*; *tmpi* će pripadati paketu *tmp*. Konačno, ako element preslikavanja *dme* pripada paketu *dmp*, odgovarajući element odredišnog modela *tmei* biće generisan iz *dme* za svaki *smei*, i pripadaće paketu *tmpi*, samo ako je *tmpi* generisan.

Paketi <<Substruct>> i <<Ref>>

Paket `<<Substruct>>` sam po sebi ne predstavlja zahtev za kreiranje dela odredišnog modela. Ovi paketi se zato mogu obilaziti nezavisno, u posebnom prolazu, svaki pojedinačno. Za svaki od ovih paketa može se generisati posebna procedura za kreiranje podstrukture. Ta procedura se generiše na isti navedeni način, tj. obilaskom hijerarhije paketa unutar `<<Substruct>>` paketa po dubini.

Paket `<<Ref>>` predstavlja zahtev za kreiranje podstrukture odredišnog modela. Ovo kreiranje ima potpuno istu semantiku kao da se na mestu `<<Ref>>` paketa u preslikavanju nalazi običan paket (bez stereotipa) koji ima isti sadržaj kao i odgovarajući istoimeni `<<Substruct>>` paket, samo ako izraz definisan označenom vrednošću `Cond` tog paketa ima rezultat `True`.

Ukoliko `<<Ref>>` paket ima definisanu označenu vrednost `Elem` koja referiše element izvorišnog modela *sme*, onda se na njegovo mesto zapravo "ugrađuje" sadržaj onog

istoimenog `<<Substruct>>` paketa koji ima označenu vrednost `ElemType` koja najbliže odgovara tipu elementa *sme*. Značenje termina "po tipu najbliže" je u potpunosti u saglasnosti sa značenjem polimorfizma u klasičnim objektno orijentisanim programskim jezicima i može se osloniti na interpretaciju programskog jezika na kome se implementira transformator modela, kao što će to biti prikazano u narednom poglavlju (Opis implementacije).

Instance

Za instancu bez stereotipa u preslikavanju domena *dmi* sa imenom *dmiName*, a koja referiše apstrakciju iz odredišnog domena *T*, jedna instanca *tmi* tipa *T*, sa imenom *dmiName*, biće kreirana u odredišnom modelu samo ako izraz definisan označenom vrednošću `Cond` instance *dmi* ima rezultat `True`. Atributi instance *tmi* biće tada postavljeni na vrednosti definisane u instanci *dmi*.

Ako instanca sa stereotipom `<<Ref>>` pripada paketu `<<Ref>>`, onda ona predstavlja vezivanje formalnog argumenta podstrukture, koji je referenca na instancu u odredišnom modelu, sa stvarnim argumentom iz okružujućeg konteksta, koji je konkretna instanca u odredišnom modelu. Prilikom "ugrađivanja" odgovarajućeg sadržaja istoimenog paketa `<<Substruct>>`, na način ranije opisan, za instancu *dmi* sa stereotipom `<<Ref>>` koja pripada paketu `<<Substruct>>`, neće biti kreirana posebna instanca u odredišnom modelu, već samo referenca na instancu koja se koristi u procesu generisanja modela. Ta referenca će referisati odgovarajuću instancu iz okruženja koja predstavlja stvarni argument, samo ako je sama referisana instanca kreirana; inače referenca ima nevažeću vrednost (engl. *null*, ne referiše ništa). Ako instanca *dmi* sa stereotipom `<<Ref>>` pripada nekom paketu unutar hijerarhije paketa koja počinje paketom `<<Substruct>>`, i ako izraz definisan označenom vrednošću `Cond` te instance ima rezultat `True`, onda će atributi referisane instance (stvarnog argumenta) biti postavljeni na vrednosti definisane u instanci *dmi*.

Veza

Neka je *dml* veza u preslikavanju domena koja povezuje dve instance bez stereotipa *dmi1* i *dmi2*, i koja referiše asocijaciju *A* iz odredišnog domena. Neka je *dmp* prvi zajednički okružujući paket za *dmi1* i *dmi2* (*dmp* = *fcep*(*dmi1*, *dmi2*)). Kao što je ranije već rečeno, *dml* implicitno pripada paketu *dmp*. Neka je *tmp* neki paket u odredišnom modelu koji je generisan iz *dmp*. Za svaku instancu u odredišnom modelu *tmi1* koja je generisana iz *dmi1* u *tmp*, i za svaku instancu u odredišnom modelu *tmi2* koja je generisana iz *dmi2* u *tmp*, po jedna veza *tml* asocijacije *A* će biti kreirana u odredišnom modelu, tako da povezuje *tmi1* i *tmi2*, i to samo ako izraz definisan označenom vrednošću `Cond` veze *dml* ima rezultat `True`. Izraz definisan u `Cond` može da referiše instance koje data veza povezuje.

Ukoliko je neka od instanci koje veza *dml* u preslikavanju domena povezuje ima stereotip `<<Ref>>`, onda se veza kreira na isti način ali tako da povezuje referisanu instancu u odredišnom modelu. Uslov je, naravno, da referenca ima važeću vrednost, tj. da je referisana instanca kreirana.

se pojavljuju apstrakcije `Package`, `Instance` i `Link` koje su preuzete iz UML metamodela [B-OMG99]. Klasa `DMElement` je apstraktna klasa koja služi samo da ujedini zajednička svojstva svih apstrakcija koje se pojavljuju u ovom domenu, a koja apstrakcije UML domena ne poseduju. To je najpre atribut `Cond` koji predstavlja uslov za dati element (istoimena označena vrednost). Ovaj atribut je tipa niza znakova i okruženje ne ulazi u njegovu interpretaciju, nego ga jednostavno prepisuje u kôd transformatora, tako da predstavlja izraz na ciljnom programskom jeziku. Drugo, tu je i polimorfna (u ovoj klasi apstraktna) operacija `generateCode()` koja je odgovorna za generisanje koda transformatora na ciljnom programskom jeziku. Nju će definisati konkretne izvedene klase.

Klasa `DMPackage` predstavlja paket bez stereotipa u preslikavanjima domena. Ova apstrakcija izvedena je iz UML apstrakcije `Package` i pomenute apstraktne klase `DMElement`. Iz ove klase izvedena je klasa `DMForEach` koja predstavlja pakete sa stereotipom `<<ForEach>>`, kao i klasa `DMSubstruct` koja predstavlja pakete sa stereotipom `<<Substruct>>`. Atributi ovih klasa predstavljaju odgovarajuće označene vrednosti.

Na sličan način su definisane i apstrakcije `DMInstance`, koja predstavlja instancu bez stereotipa, `DMInstanceRef`, koja predstavlja instancu sa stereotipom `<<Ref>>`, i `DMLink`, koja predstavlja vezu. One su izvedene iz UML klasa `Instance`, odnosno `Link`, i pomenute klase `DMElement`.

Apstrakcija `DMPackageRef` predstavlja paket sa stereotipom `<<Ref>>`. Ona nije izvedena iz UML apstrakcije `Package`, jer zapravo ima mnogo više ograničenja i razlika nego zajedničkih svojstava sa apstrakcijom UML paketa. Naime, ona može da sadrži samo instance sa stereotipom `<<Ref>>`, pa je ovo sadržavanje uvedeno u metamodel kao eksplicitna asocijacija `actual params`. Dakle, ova apstrakcija ne nasleđuje svojstvo UML paketa da može da sadrži bilo koje elemente, pa, zapravo, osim notacije, nema drugih zajedničkih svojstava sa UML paketom. Ovo je izuzetak od pravila da stereotip predstavlja izvedeni tip u metamodelu, učinjen u cilju jednostavnije implementacije.

Najzad, zavisnosti `<<Seq>>` i `<<Ref>>` nisu prikazane kao posebne klase izvedene iz UML klase `Dependency` zbog konciznosti slike. Slično, deo metamodela koji opisuje koncept definicije vrednosti atributa instance u potpunosti je preuzet iz UML metamodela, kao i asocijacije koje povezuju `Instance` i `Link`, pa ovi elementi takođe nisu prikazani na slici.

Generisanje transformatora

Iz modela sa datim metamodelom može se automatski generisati kôd transformatora koji odgovara datom preslikavanju domena. Interesantno je primetiti da ovaj postupak zapravo opet predstavlja generisanje izlazne forme, u ovom slučaju tekstualnog koda na ciljnom programskom jeziku iz polaznog objektnog modela. Za ovakav slučaj, kao što je ranije već diskutovano, može se izabrati jedan od najpogodnijih metoda definisanja načina generisanja sekvencijalnog izlaza, recimo pomoću projektnog obrasca *Visitor*.

Prilikom generisanja koda transformatora, struktura modela preslikavanja obilazi se na već opisani način: hijerarhija paketa (koja predstavlja stablo) obilazi se po dubini, a elementi paketa obilaze se u topološkom redosledu prema sekvencijalnim zavisnostima. Za svaki element koji se posećuje, generiše se odgovarajući deo koda transformatora.

Procedura generisanja koda prilikom obilaska elementa datog tipa (klase iz metamodela na slici 5.2) biće ovde prikazana kao operacija `generateCode()` odgovarajuće klase. Detaljan pseudo-kôd ovih operacija koje generišu kôd na jeziku C++ dat je u Prilogu A. Ovde su u nastavku dati samo okvirni opisi tih operacija, sa objašnjenjima načina na koji se dobija željeni kôd transformatora.

Paket bez stereotipa

Za paket bez stereotipa u preslikavanju domena treba generisati kôd koji će u odredišnom modelu kreirati istoimeni paket, naravno samo ukoliko je odgovarajući uslov `Cond` ispunjen. Potom treba običi elemente koje taj paket sadrži i za njih generisati odgovarajući kôd, kao ugnežđeni blok naredbi.

Na primer, neka posmatrani paket u preslikavanju domena ima ime `Package_1` i atribut `Cond` postavljen na vrednost `"expr"`. Tada za ovaj paket treba generisati sledeći deo koda transformatora:

```
// Package: Package_1
Package* pckPackage_1 = 0;
if (expr) {
    pckPackage_1 = Package::create(pckTarget);
    pckPackage_1->dmOrgName = "Package_1";
    pckPackage_1->setName("Package_1");

    // Ovde dolazi kôd generisan za elemente koje dati paket sadrži
}
```

Operacija `Package::create()`, kao i istoimene operacije drugih klasa iz metamodela, jeste statička operacija koja kreira objekat date klase. Ona kao argument prima referencu na paket kome će kreirani element pripadati. U ovom slučaju pretpostavlja se da je to neki paket iz odredišnog modela na koga ukazuje referenca (tačnije pokazivač) `pckTarget`. Iz ovoga se vidi da je u toku procesa generisanja koda potrebno pamtiti ime reference na tekući roditeljski paket kome će pripadati novokreirani elementi. Ovde će promenljiva tipa niza znakova koja pamti to ime reference biti nazivana `parentPckName`. Prilikom generisanja koda za navedeni paket, potrebno je zato najpre sačuvati prethodnu vrednost promenljive `parentPckName`, zatim pre obilaska sadržanih elemenata postaviti novu vrednost te promenljive na ime reference na novokreirani paket (u ovom slučaju `"pckPackage_1"`), kako bi dalje kreirani elementi pripadali tom paketu, i, najzad, po završetku obilaska sadržanih elemenata, restaurirati zapamćenu vrednost promenljive `parentPckName`. Ovi koraci mogu se pronaći u pseudo-kodu procedura generisanja koda transformatora u Prilogu A.

Paket <<ForEach>>

Za paket sa stereotipom <<ForEach>> treba generisati kôd koji će najpre u odredišnom modelu kreirati jedan paket koji će sadržati sve generisane potpakete, zatim petlju koja će iterirati kroz kolekciju definisanu označenom vrednošću `InCollection`, i u svakoj iteraciji kreirati po jedan paket u odredišnom modelu za datu iteraciju, ukoliko je zadati uslov zadovoljen. Najzad, treba ponovo za sadržane elemente paketa generisati kôd kao i kod običnog paketa bez stereotipa.

Na primer, neka se paket sa stereotipom <<ForEach>> zove `Package_2`, i neka njegove označene vrednosti imaju sledeći sadržaj: `Cond = "true"`, `ForEach = "mel"`, `OfType = "ModelElement"`, `InCollection = "p1->getOwnedElements()"`. Neka je, takođe, ovaj paket `Package_2` sadržan unutar paketa `Package_1` iz prethodnog odeljka. Tada za ovakav paket treba generisati sledeći kôd:

```
// ForEach Package: Package_2
Package* pckForEachPackage_2 = Package::create(pckPackage_1);
pckForEachPackage_2->dmOrgName = "Package_2";
pckForEachPackage_2->setName("ForEach-Package_2");
```

```

ForEach(mel, ModelElement, p1->getOwnedElements())
    Package* pckPackage_2 = 0;
    if (true) {
        pckPackage_2 = Package::create(pckForEachPackage_2);
        pckPackage_2->dmOrgName = "Package_2";
        pckPackage_2->setName(mel->getFullPathName());

        // Ovde dolazi kôd generisan za elemente koje dati paket sadrži
    }
EndForEach(mel)

```

U ovom kodu, `ForEach` i `EndForEach` su C++ makroi koji se pri prevođenju razvijaju u odgovarajuće naredbe za iteraciju kroz elemente kolekcije u izvorišnom modelu. Ta iteracija je osetljiva na tip tekućeg elementa, što znači da se preskaču oni elementi kolekcije koji nisu zadatog tipa.

Instanca bez stereotipa

Za instancu bez stereotipa treba generisati kôd koji će u odredišnom modelu kreirati odgovarajuću instancu, naravno pod uslovom definisanim označenom vrednošću `Cond`, a zatim postaviti vrednosti njenih atributa.

Na primer, neka je ime instance `inst1`, neka je njena `Cond` vrednost jednaka "`expr`", neka je ona instanca klase `x` iz odredišnog metamodela i neka su joj atributi `attr1`, `attr2` i `attr3` redom postavljeni na vrednosti "`val1`", "`val2`" i "`val3`". Tada generisani kôd izgleda ovako:

```

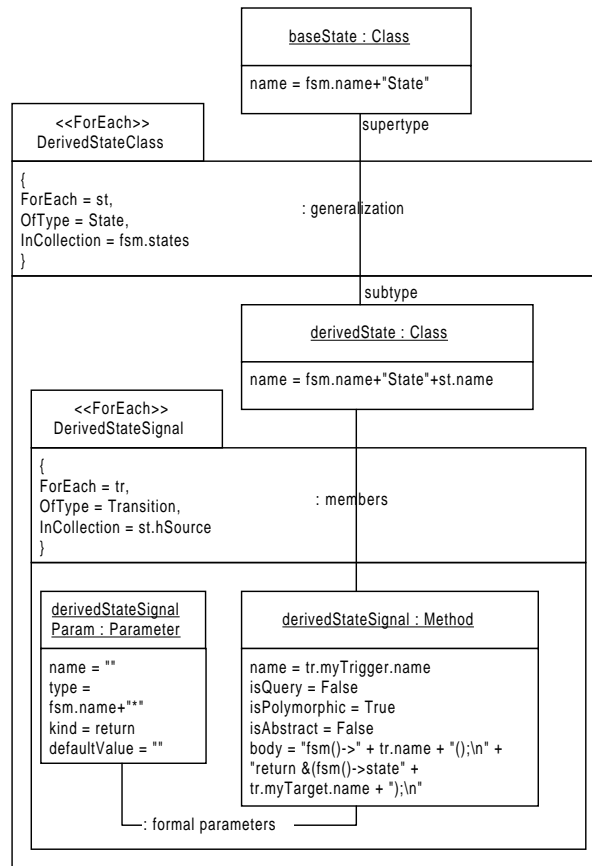
// Instance: inst1
X* inst1 = 0;
if (expr) {
    inst1 = (X*)X::create(pckPackage_2);
    inst1->setName("inst1");
    inst1->dmOrgName = "inst1";
    inst1->attr1 = val1;
    inst1->attr2 = val2;
    inst1->attr3 = val3;
}

```

Veza

Veza (kao instanca klase `Link`) implicitno pripada prvom zajedničkom okružujućem paketu instanci koje povezuje. Osim toga, za vezu se uvode implicitne sekvencijalne zavisnosti prema instancama koje povezuje, koje se tokom pripreme generisanja koda transformatora prevode u sekvencijalne zavisnosti od veze prema prvim okružujućim bratskim elementima tih instanci. Na primer, za vezu tipa `members` na slici 5.3 postoje implicitne sekvencijalne zavisnosti prema instanci `derivedState` i <<`ForEach`>> paketu `DerivedStateSignal`. Zbog toga će data veza biti posećena prilikom generisanja koda transformatora posle obilaska i generisanja koda za ova dva elementa.

Kako semantika kreiranja veza zahteva da se veza kreira za sve parove instanci odredišnog modela koje su generisane kao posledica instanci u preslikavanju domena koje data veza povezuje, potrebno je generisati kôd koji će iterirati kroz sve takve kreirane instance u odredišnom modelu. Ukoliko instanca koju veza povezuje pripada istom paketu kao i veza, kao instanca `derivedState` na slici 5.3, onda je referenca na nju u generisanom kodu transformatora direktno dostupna u istom opsegu važenja. Ukoliko je pak instanca u



Slika 5.3: Primer <<ForEach>> paketa za prikazani generisani kod transformatora. Isti primer sa slike 4.2. Posmatra se veza asocijacije members.

(proizvoljno duboko) ugnežđenom paketu, kao instanca `derivedStateSignal` na slici 5.3, onda treba generisati kôd koji obilazi datu podhijerarhiju paketa, iterira kroz pakete generisane `ForEach` iteracijama, i konačno pristupa ugnežđenim instancama. Za dati primer veze tipa `members` na slici 5.3 potrebno je tako generisati sledeći kôd:

```
// Link: link1 : Members
{
  // First side:
  Class* derivedState = (Class*)(pckDerivedStateClass->
    getOwnedElement("derivedState", "Class"));
  // Second side:
  Package* pckForEachDerivedStateSignal = pckDerivedStateClass->
    getOwnedElement("ForEach-DerivedStateSignal", "Package");
  ForEach(pckDerivedStateSignal, Package,
    pckForEachDerivedStateSignal->getOwnedElements())
  Method* derivedStateSignal = (Method*)(pckDerivedStateSignal->
    getOwnedElement("derivedStateSignal", "Method"));
  // Link:
  if (derivedState && derivedStateSignal && (expr)) {
    M1Link* link1 = M2Association::
      createLink("Members", derivedState, derivedStateSignal);
    link1->dmOrgName = "link1";
  }
  EndForEach(pckDerivedStateSignal)
}
```

Način na koji se dolazi do ovog koda prikazan je u Prilogu A, u funkciji `DMLink::generateCode()`. Pomoćna funkcija `DMLink::generateInstanceAccess()` generiše kôd za pristup do ugnežđenih instanci, uz eventualne iteracije ukoliko se u datoj podhijerarhiji nalaze <<ForEach>> paketi.

Organizacija koda transformatora

Svi do sada izloženi delovi koda transformatora treba da budu deo jedne procedure koja generiše određeni model prema elementima preslikavanja domena koji su do sada navedeni. Ovde će biti pokazano kako se taj kôd organizuje i kako izgleda jedno moguće okruženje pomenute procedure. Ovakva organizacija potrebna je u cilju implementacije preostalih koncepata preslikavanja, paketa <<Substruct>> i <<Ref>>. Naime, kako će za pakete <<Substruct>> biti neophodno generisati procedure koje se kasnije mogu redefinisati prilikom nasleđivanja preslikavanja domena, organizacija transformatora opisana ovde se prirodno nameće kao logično rešenje. Ovakav pristup primenjen je i u prototipskom alatu koji podržava predloženu metodu.

Za jednu specifikaciju preslikavanja domena, tj. za jedan paket sa stereotipom <<DomainMapping>>, generiše se jedna klasa. To omogućuje da se jednostavno implementira koncept nasleđivanja preslikavanja domena, uz eventualnu redefiniciju njegovih delova. Ta klasa nasleđuje u alat već ugrađenu klasu `M2GenDomainMapper`, ukoliko dati <<DomainMapping>> paket ne nasleđuje drugi takav paket; ukoliko pak nasleđuje, generisana klasa takođe nasleđuje klasu generisanu za ovaj drugi paket.

Generisana klasa je *Singleton* (ima samo jednu instancu) [B-Gam95], pa poseduje zaštićeni konstruktor, destruktor i statičku operaciju `Instance()` kao režijske operacije.

Glavna operacija ove klase je operacija `generate()`, u čijem se telu nalazi kôd generisan obilaskom svih elemenata korenog paketa <<DomainMapping>>, osim paketa <<Substruct>>, kao i njihovih elemenata, rekurzivno. Prema tome, C++ kôd generisan za jedan paket <<DomainMapping>> sa imenom `x` može da izgleda ovako:

```
class M2GenDMX : public M2GenDomainMapper {
public:

    static M2GenDMX* Instance ();
    virtual ~M2GenDMX () {};

    virtual void generate (Package* pckSource, Package* pckDest);

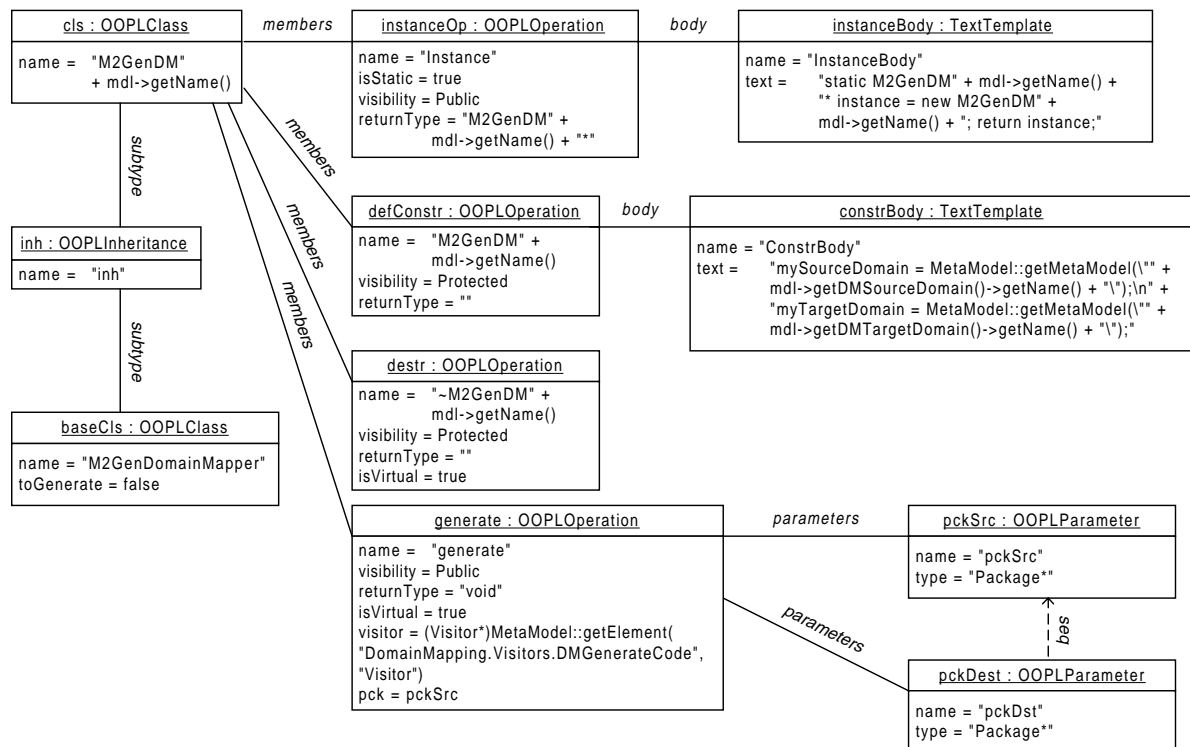
protected:

    M2GenDMX ();

};

M2GenDMX* M2GenDMX::Instance () {
    static M2GenDMX* instance = new M2GenDMX;
    return instance;
}

void M2GenDMX::generate (Package* pckSource, Package* pckDest) {
    // Ovde dolazi kôd generisan obilaskom elemenata preslikavanja domena x
}
```



Slika 5.4: Preslikavanje domena DomainMapping u domen OOPL. Dijagram prikazuje specifikaciju generisanja jedne *Singleton* klase čija operacija `generate()` služi za generisanje odredišnog modela. Nasleđivanje preslikavanja, radi konciznosti, nije podržano.

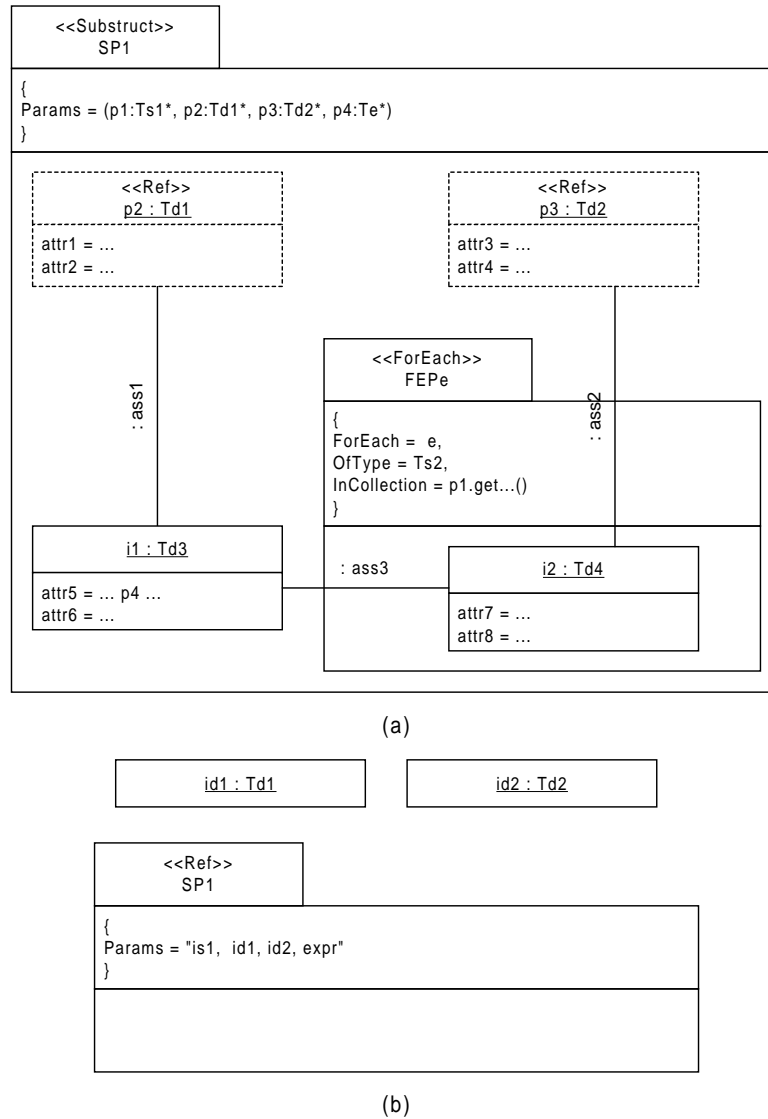
Na ovaj način je nasleđivanje `<<DomainMapping>>` paketa jednostavno podržano. Na primer, ukoliko paket `Y` nasleđuje prethodno definisani paket `X`, onda će klasa generisana za `Y` izgledati ovako:

```
class M2GenDMY : public M2GenDMX {
//...
};
```

```
void M2GenDMY::generate (Package* pckSource, Package* pckDest) {
    M2GenDMX::generate (pckSource, pckDest);
    // Ovde dolazi kôd generisan obilaskom elemenata preslikavanja domena Y
}
```

Tako se postiže zahtevani efekat da izvedeni paket `Y` "nasleđuje" sve elemente preslikavanja iz osnovnog paketa `X`, što zapravo znači da njegov transformator vrši sva kreiranja definisana u osnovnom paketu `X`, a potom i kreiranja definisana u samom izvedenom paketu `Y`.

Navedeni kôd je zahtevani izlaz koji treba generisati za zadati `<<DomainMapping>>` paket. Kako ovaj izlaz predstavlja kôd u ciljnom objektno orijentisanom programskom jeziku koji treba generisati iz objektnog modela preslikavanja domena, svi raniji zaključci o načinu specifikacije generisanja takvog izlaza važe i ovde. Prema tome, za specifikaciju ovog izlaza pogodno je uvesti međudomen koji opisuje osnovne koncepte objektno orijentisanih programskih jezika, ranije nazvan domenom OOPL. Iz modela u tom domenu može se dobiti ciljni programski kôd korišćenjem već postojećih generatora koda realizovanih pomoću



Slika 5.5: Primer za generisanje koda za pakete `<<Substruct>>` i `<<Ref>>`. Isti primer sa slike 4.9a i b. (a) Definicija podstrukture. (b) Referenca na tu podstrukturu.

projektnog obrasca *Visitor*. Sa druge strane, preslikavanje iz `DomainMapping` domena u domen OOPL definisano je pomoću predložene metode.

Ovo preslikavanje prikazano je na slici 5.4. Radi konciznosti slike, nasleđivanje preslikavanja nije podržano. Interesantan detalj je način generisanja tela operacije `generate()`. Naime, domen OOPL omogućuje da se telo neke operacije generiše pomoću obrasca *Visior*, jer je to jedan od najčešće korišćenih metoda u praksi. Upravo ta mogućnost iskorišćena je ovde, pa je za generisanje koda operacije `generate()` u alatu definisana klasa `DMGenerateCode` koja je izvedena iz odgovarajuće *Visitor* klase. Ovaj *Visitor* generiše kôd transformatora na način opisan u prethodnim odeljcima, odnosno procedurama prikazanim u Prilogu A.

Paket `<<Substruct>>` bez vrednosti `ElemType`

Ukoliko se transformator realizuje klasom kao što je upravo pokazano, onda se paketi `<<Substruct>>` mogu jednostavno implementirati kao operacije te klase, a paketi `<<Ref>>`


```
    attr2 = ...;
}
```

Ovaj deo preslikavanja za pakete `<<Substruct>>` u domen OOPL prikazan je na slici 5.6, a pseudokod operacije `generateCode()` za instance `<<Ref>>` dat je u Prilogu A.

Paket <<Ref>> bez vrednosti Elem

Za paket `<<Ref>>` bez definisane označene vrednosti `Elem` generiše se jednostavni poziv operacije klase transformatora. Pri tome se postavljaju vrednosti stvarnih argumenata prema označenoj vrednosti `Params` tog paketa. Odnos vrednosti `Params` i instanci `<<Ref>>` paketa, kao i rešavanje njihove eventualne nekonzistentnosti, ostavljaju se samoj implementaciji. Načini na koje neko okruženje (alat) za podršku može da rešava međusobni odnos ova dva elementa su sledeći:

- Instance `<<Ref>>` unutar paketa `<<Ref>>` ne moraju uopšte da budu podržane. Okruženje jednostavno uzima vrednost `Params` i nju koristi za postavljanje stvarnih argumenata poziva.
- Instance `<<Ref>>` unutar paketa `<<Ref>>` i vrednost `Params` su komplementarne i moraju biti disjunktne, u smislu da se unutar `Params` ne smeju naći postavljanja argumenata predstavljenih instancama `<<Ref>>`. Oba elementa se uzimaju zajedno za definisanje stvarnih argumenata kao različiti i nezavisni.
- Instance `<<Ref>>` i vrednosti `Params` se mogu preklapati, ali moraju biti konzistentni. To znači da u vrednosti `Params` mogu da se pojave instance sa istim imenom kao i instance `<<Ref>>`, ali vezane za iste stvarne parametre. Za ovaj najsloženiji slučaj, okruženje treba da obezbedi proveru važenja navedenih pravila za definisani `<<Ref>>` paket u preslikavanju.

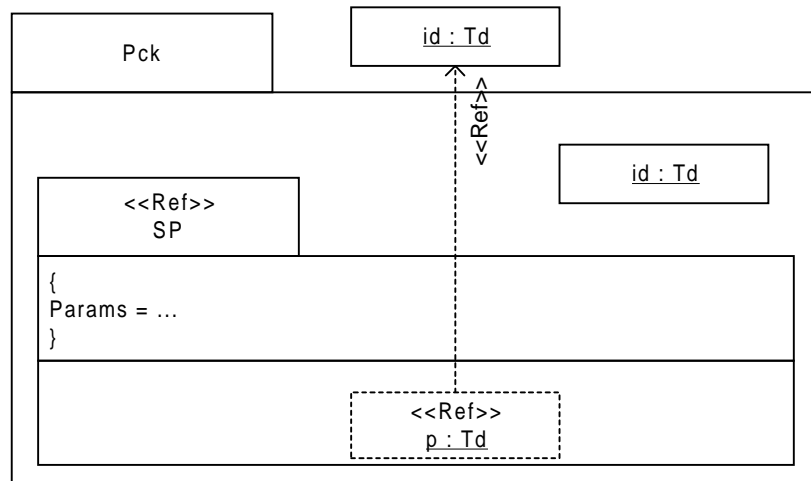
Pored toga, implementaciji se ostavlja da podrži specifikaciju stvarnih argumenata po imenu ili po poziciji. Naime, u dosadašnjim primerima su formalni argumenti bili referisani po imenu, što znači da je redosled njihovog postavljanja u vrednosti `Params` paketa `<<Ref>>` bio proizvoljan. Kako ciljni programski jezici tipično ne podržavaju ovakav koncept za argumente procedura, okruženje u tom slučaju mora da poreda stvarne argumente onako kako to odgovara redosledu definisanja odgovarajućih formalnih argumenata. U jednostavnijem slučaju okruženje ne mora uopšte da podrži ovaj koncept, već korisnik mora poredati stvarne argumente u vrednosti `Params` onako kako to odgovara redosledu formalnih argumenata. U tom slučaju nije potrebno ni imenovanje tih formalnih argumenata u paketu `<<Ref>>`.

Za najjednostavniji slučaj kada okruženje ne podržava instance `<<Ref>>`, već stvarne argumente postavlja samo uzimanjem označene vrednosti `Params`, i to po poziciji, kôd generisan za primer sa slike 5.5b izgleda ovako (vrednost `Cond` je "expr"):

```
if (expr) {
    SP1(pckCur, is1, id1, id2, expr);
}
```

Pokazivač `pckCur` ukazuje na trenutno aktivni paket u odredišnom modelu u kome se kreiraju elementi podstrukture. Ime ovog pokazivača je zapravo vrednost ranije pomenute promenljive `parentPckName` u proceduri za generisanje koda. Pseudokod operacije `generateCode()` za paket `<<Ref>>` dat je u Prilogu A.

Ukoliko okruženje u potpunosti podržava definisanje instanci `<<Ref>>` unutar paketa `<<Ref>>`, onda pravolinijsko generisanje koda kao što je do sada vršeno unosi potencijalni problem sa semantikom. Primer ovog problema prikazan je na slici 5.7. Ovde se formalni argument `p` podstrukture `SP` vezuje za instancu `id` iz spoljnog paketa, dok u prvom



Slika 5.7: Problem imenovanja instanci kod referisanja podstrukture. Formalni argument `p` podstrukture `SP` treba da referiše instancu `id` iz spoljnjeg paketa, a ne istoimenu instancu `id` iz prvog okružujućeg paketa `Pck`. Direktno preslikavanje u ugneždene blokove u ciljnom programskom jeziku nema željenu semantiku.

okružujućem paketu postoji istoimena instanca koju formalni argument ne treba da referiše. U modelu preslikavanja domena ovo ne predstavlja dvosmislenost jer je sasvim određeno za šta je vezana instanca `<<Ref>>` koja predstavlja formalni parametar. Međutim, ukoliko se ovaj model preslika u neki klasični proceduralni programski jezik, gde se prikazani paketi preslikavaju u ugneždene blokove naredbi, a instance u identifikatore lokalnih referenci, postoji problem zbog toga što identifikator `id` u ugnežđenom bloku sakriva isti identifikator iz okružujućeg bloka. Tako dobijeni kôd ima drugačiju semantiku od željene:

```
...
Td* id = ...;
...
Package* pckPck = 0;
if (...) {
    pckPck = Package::create(...);
    ...
    Td* id = ...;
    ...
    SP(pckPck,id); // Ovo nije željeni id!
}
```

Ovaj problem može se rešiti tako što se instance ne preslikavaju u istoimene identifikatore u ciljnom kodu, već se na te identifikatore dodaje neki automatski generisani dodatak koji ih čini globalno jedinstvenim. To može da bude interna identifikacija samog elementa modela preslikavanja koja je globalno jedinstvena u celom modelu. Taj dodatak alat treba da pridružuje svakoj instanci kako bi u svakom trenutku mogao da dobije identifikator koji joj u ciljnom kodu odgovara. Uz ovakav pristup, generisani kôd za dati primer više ne nosi problem jer izgleda poput:

```
...
Td* id_0125644 = ...; // Sufiks je automatski generisan
...
Package* pckPck = 0;
if (...) {
    pckPck = Package::create(...);
```

```

...
Td* id_0125645 = ...;
...
SP(pckPck,id_0125644); // Ovo sada jeste željeni id!
}

```

Paketi <<Substruct>> i <<Ref>> vezani za tip

Ukoliko su paketi <<Substruct>> i <<Ref>> vezani za neki tip iz izvorišnog domena, odnosno imaju definisane vrednosti ElemType i Elem, generisanje koda je nešto složenije. Naime, u ovom slučaju potrebno je podržati polimorfizam i u odnosu na hijerarhiju tipova u izvorišnom domenu, i u odnosu na hijerarhiju nasleđivanja klasa transformatora. Kako nijedan klasični objektno orijentisani programski jezik ne podržava višestruki polimorfizam, ovaj problem zahteva složenije rešenje. Pored toga, nije moguće osloniti se na implementaciju polimorfizma podstruktura u odnosu na tipove iz izvorišnog domena kroz operacije klasa koje implementiraju njegov metamodel, jer bi to značilo proširivanje tih klasa operacijama specifičnim za preslikavanja. To nije prihvatljivo rešenje jer bi svako definisanje novog preslikavanja, ili čak definisanje nove podstrukture u nekom preslikavanju, zahtevalo izmene klasa metamodela izvorišnog domena dodavanjem operacija koje realizuju te podstrukture.

Ovaj problem sličan je problemu razdvajanja strategije obilaska neke strukture od same hijerarhije strukture koji se uspešno rešava projektnim obrascem *Visitor*. Zbog toga se suština tog rešenja može primeniti i ovde. Suština pristupa je da se polimorfizam na strani tipova iz izvorišnog domena rešava u tim klasama, ali nezavisno od preslikavanja. Pri tome je klasa transformatora izvedena iz neke osnovne klase koja je zajednička za sve transformatore koji imaju isti izvorišni domen. Ta osnovna klasa ima po jednu operaciju za svaki tip u izvorišnom domenu, kao kod obrasca *Visitor*. Na primer, neka klase A i B predstavljaju apstrakcije iz izvorišnog domena SD, i neka je osnovna klasa za sve transformatore sa tim izvorišnim domenom klasa VisitorSD. Tada ove klase izgledaju ovako:

```

class A {
public:
    virtual void accept (VisitorSD* v) { v->visitA(this); }
    ...
};

class B : public A {
public:
    virtual void accept (VisitorSD* v) { v->visitB(this); }
    ...
};

class VisitorSD {
public:
    virtual void visitA(A* elem) {}
    virtual void visitB(B* elem) { visitA(elem); }
    ...
};

```

Neka je za ovaj izvorišni domen definisano preslikavanje DM koje ima neku definisanu podstrukturu SP sa pridruženim tipom A. Neka ta podstruktura ima dva formalna argumenta x i y tipa x^* i y^* , respektivno. Neka je, zatim, ta podstruktura redefinisana za tip B. Tada se za ovo preslikavanje generiše klasa transformatora na sledeći način:

```

class M2GenDM : public VisitorSD {
public:
    ...
    virtual void SP (Package* pckOwner, A* elem, X* x, Y* y);
    virtual void SP (Package* pckOwner, B* elem, X* x, Y* y);
    ...
    virtual void visitA(A*);
    virtual void visitB(B*);
private:
    String substructName;
    Package* pckOwner;

    // Substruct: SP
    X* SP_x;
    Y* SP_y;
    ...
};

```

Kao što se vidi, za svaki formalni argument podstrukture generiše se odgovarajući podatak član klase. Ovaj podatak član služiće za privremeni smeštaj stvarnog argumenta prilikom razrešavanja polimorfnog poziva te podstrukture, odnosno operacije klase. Pored toga, generiše se i jedan podatak član tipa niza znakova koji će čuvati ime pozvane operacije.

Sada je potrebno razrešiti polimorfizam poziva ove podstrukture SD u odnosu na tip iz izvorišnog modela, odnosno pozvati pravu operaciju SP, ali tako da se dodavanjem novih redefinicija iste podstrukture ne menjaju klase A i B izvorišnog metamodela. To se može rešiti na sledeći način. Prilikom poziva podstrukture potrebno je generisati kôd koji će najpre stvarne argumente poziva smestiti u odgovarajuće podatke članove klase transformatora, a zatim pozvati operaciju `accept()` elementa izvorišnog modela koji je definisan vrednošću Elem u paketu <<Ref>>, tako da se potom poziva ona `visit...()` operacija koja odgovara konkretnom tipu tog elementa. Unutar te operacije klase transformatora poziva se konkretna operacija SP te klase, sa argumentima koji su restaurirani iz podataka članova. Kôd operacija `visitA()` i `visitB()` koji to obezbeđuje izgleda ovako:

```

void M2GenDM::visitA (A* elem) {
    if (substructName=="SP") {
        SP(pckOwner,elem,SP_x,SP_y); // Poziva se SP(...,A*,...)
    } else
        if (substructName==...) ... // Za ostale podstrukture definisane za A
    else
        VisitorSD::visitA(this); // Inače, nasleđeno ponašanje
}

void M2GenDM::visitB (B* elem) {
    if (substructName=="SP") {
        SP(pckOwner,elem,SP_x,SP_y); // Poziva se SP(...,B*,...)
    } else
        if (substructName==...) ... // Za ostale podstrukture definisane za B
    else
        VisitorSD::visitB(this); // Inače, nasleđeno ponašanje
}

```

Poziv podstrukture SP sa stvarnim argumentima `ax` i `ay`, i vrednošću Elem jednakom "e", prevodi se u sledeću sekvencu:

```

// Substruct Ref: SP
substructName = "SP";
pckOwner = ...;

```

```
SP_x = ax;
SP_y = ay;
e->accept(this);
```

Na ovaj način, hijerarhija klasa metamodela izvorišnog domena ostaje nezavisna od klasa koje implementiraju transformatore. Ukoliko se i osnovna klasa transformatora `VisitorSD` smatra delom implementacije domena, a ne preslikavanja, što je i logično jer je ona zajednička za sve transformatore tog izvorišnog domena, onda važi i obrnuto. Na primer, ako se u hijerarhiju klasa izvorišnog domena doda klasa `C` izvedena iz klase `B`, dok se u preslikavanjima ne redefiniše nijedna podstruktura za taj novi tip `C`, onda samo u osnovnu klasu transformatora treba dodati operaciju `visitC()`:

```
class VisitorSD {
public:
    virtual void visitA(A* elem) {}
    virtual void visitB(B* elem) { visitA(elem); }
    virtual void visitC(C* elem) { visitB(elem); }
    ...
};
```

Tako će se preko operacije `visitC()` pozvati definicija te podstrukture za tip `B`, što i jeste bio cilj. Ukoliko je pak neka podstruktura (npr. `SP`) redefinisana za `C`, što opet predstavlja izmenu samog preslikavanja, onda se i implementacija transformatora menja tako što se u klasi transformatora generiše operacija `visitC()`:

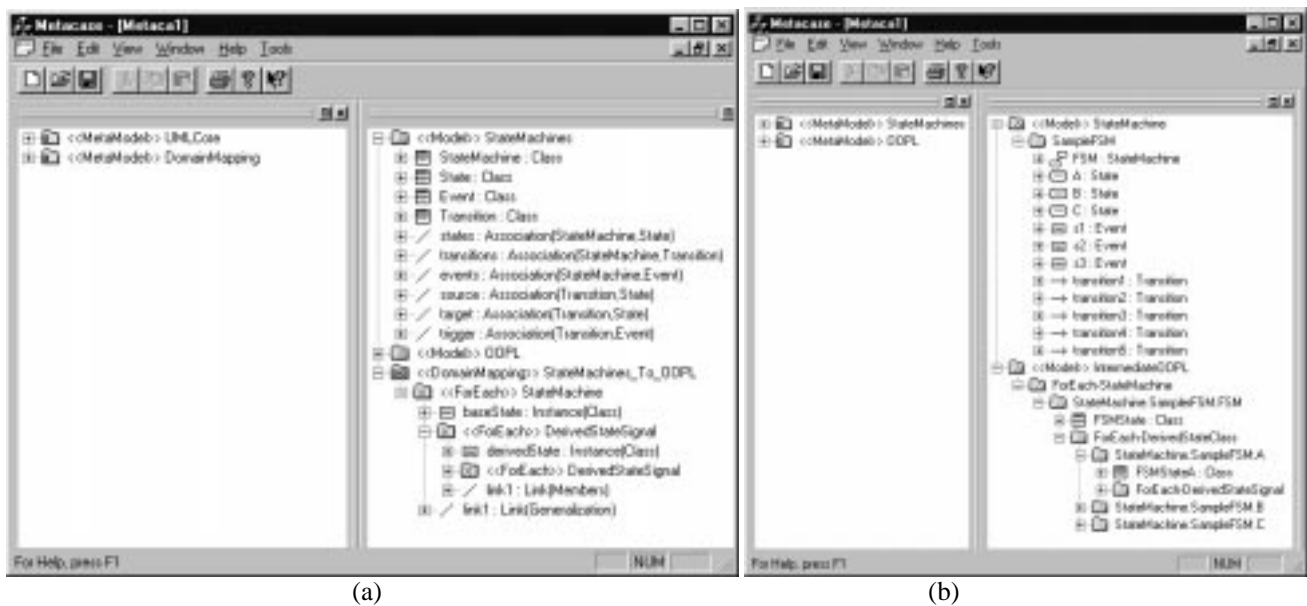
```
void M2GenDM::visitC (C* elem) {
    if (substructName=="SP") {
        SP(pckOwner,elem,SP_x,SP_y); // Poziva se SP(...,C*,...)
    } else
        if (substructName==...) ... // Za ostale podstrukture definisane za C
        else
            VisitorSD::visitC(this); // Inače, nasleđeno ponašanje
}
```

Slično, ukoliko se definiše novo preslikavanje `DMD` izvedeno iz postojećeg preslikavanja `DM`, a kako su pozivi operacija `visit...()` polimorfni, pozivaće se i odgovarajuće verzije operacije `SP`. Na primer, ukoliko izvedeno preslikavanje ne redefiniše `SP`, onda će poziv funkcije `visit...()` obezbediti poziv nasleđene verzije:

```
void M2GenDMD::visitA (A* elem) {
    if (substructName==...) ... // Za ostale podstrukture definisane za A
    else
        M2GenDM::visitA(elem); // Inače, poziv nasleđene verzije
}
```

Ako pak izvedeno preslikavanje redefiniše `SP` za npr. `A` ali ne i za `B`, biće uvek pozvana odgovarajuća verzija operacije `SP`:

```
void M2GenDMD::visitA (A* elem) {
    if (substructName=="SP") {
        SP(pckOwner,elem,SP_x,SP_y); // Poziva se M2GenDMD::SP(...,A*,...)
    } else
        if (substructName==...) ... // Za ostale podstrukture definisane za A
        else
            M2GenDM::visitA(elem); // Inače, poziv nasleđene verzije
}
```



Slika 5.8: Demonstrativni primer u prototipskom alatu za metamodelovanje. (a) Faza metamodelovanja. (b) Faza modelovanja. U oba slučaja levi deo prikazuje metamodele (domene), a desni modele za odgovarajuću fazu. Svaki model u desnom delu je instanca tačno jednog metamodela iz levog dela.

```
void M2GenDMD::visitB (B* elem) {
    if (substructName==...) ... // Za ostale podstrukture definisane za B
    else
        M2GenDM::visitB(elem); // Inače, poziv nasledene verzije
}
```

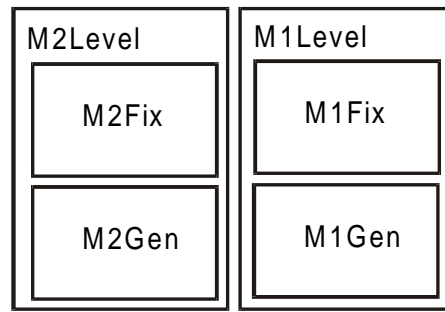
Konstrukcija alata za metamodelovanje

Kao podrška predloženoj metodi implementiran je prototip alata za metamodelovanje (slika 5.8). Alat je namenjen za formiranje i manipulaciju metamodela specifičnih domena, formiranje i manipulaciju modela koji pripadaju tim domenima, definisanje preslikavanja domena i automatsku transformaciju modela prema predloženoj metodi.

Alat predstavlja generičko okruženje za manipulaciju modelima sa različitim metamodelima. Isto generičko okruženje koristi se i u fazi metamodelovanja specifičnih domena, kao i u fazi modelovanja u tim domenima. Ovakav pristup omogućen je osnovnom konceptijskom odlukom da okruženje zapravo ima informacije o metamodelima domena u kojima se vrši modelovanje. Na taj način je okruženje u stanju da interpretira te informacije i vrši generičke strukturne operacije sa modelima.

Međutim, za razliku od većine drugih raspoloživih alata za metamodelovanje, ovaj alat se ne oslanja isključivo na interpretaciju podataka o metamodelima, nego pored tih podataka, alat za metamodelovanje generiše i deo izvornog koda alata za modelovanje koji se prevodi i povezuje sa ostalim generičkim delovima u izvršnu verziju alata za modelovanje. Ovakav pristup omogućuje praktično proizvoljnu fleksibilnost alata i proširivost specifičnostima datog domena modelovanja. Ovi aspekti biće detaljnije opisani u nastavku.

U ovoj prototipskoj verziji alat nema grafički editor simbola niti bilo kakvu drugu podršku definisanju i korišćenju specifične grafičke notacije za modelovanje u domenu. Modelima se za sada isključivo manipuliše preko dijaloga za pristup do elemenata modela, kao i preko hijerarhijskih prikaza delova modela u vidu stabla (engl. *tree view browser*, slika 5.8). Pored toga, sadašnja verzija alata ne podržava neke naprednije koncepte preslikavanja



Slika 5.9: Osnovna struktura alata. M2Level deo sadrži podatke o metamodelima. M1Level deo manipuliše modelima.

domena, kao što su podstrukture i polimorfizam, jer su ovi koncepti nastali u skorije vreme, posle implementacije prototipa, kao posledica primene predložene metode u praksi. Ova proširenja predviđaju se za naredne verzije, a neki početni rezultati već postoje.

Alat je projektovan korišćenjem jezika UML, uz obimnu primenu projektnih obrazaca [B-Gam95], a implementiran je u potpunosti na jeziku C++. U razvoju su korišćeni alati Rational Rose C++ i Microsoft Visual C++. Za crtanje dijagrama preslikavanja korišćen je alat Visio, jer alat Rational Rose nema dovoljnu fleksibilnost za ove potrebe.

Arhitektura alata

Jedna od osnovnih projektnih odluka prilikom razvoja alata, koja je ujedno i jedna od njegovih značajnih prednosti u odnosu na većinu raspoloživih alata za metamodelovanje, je ta da se okruženje za modelovanje ne oslanja isključivo na interpretaciju *podataka* o metamodelu datog domena koji su generisani u fazi metamodelovanja, nego da se kao posledica definisanog metamodela generiše i izvorni (a posle prevođenja i povezivanja i izvršni) *kôd* koji obezbeđuje ponašanje specifično za apstrakcije domena.

Ukratko, to znači sledeće. Neka su u metamodelu nekog domena definisane apstrakcije A, B i C, sa svojim atributima i operacijama, i nekim relacijama nasleđivanja i asocijacije između njih. Kao posledica toga, ovo okruženje za modelovanje posedovaće:

- Podatke o metamodelu manifestovane kroz objekte ugrađenih klasa Class, Attribute, Association, Generalization itd. Ove klase su zapravo apstrakcije jezgra UML metamodela koje je iskorišćeno u tu svrhu. Za navedeni primer, u okruženju za modelovanje postojeće objekti klase Class čiji atributi name imaju vrednosti "A", "B" i "C", kao i objekti klase Attribute, Association i Generalization na odgovarajući način povezani sa tim objektima.
- Izvorni kôd klasa A, B i C, generisan na ciljnom programskom jeziku (C++) prema definicijama ovih apstrakcija. Ove klase sadrže kôd za attribute i operacije, kao i sve ostale manifestacije elemenata metamodela koje podržava ciljni programski jezik. Instance apstrakcija kreirane u modelu biće zapravo objekti tih klasa u ciljnom jeziku.

Posledica ovakvog pristupa jeste i osnovna arhitekturna struktura alata prikazana na slici 5.9. Osnovna podela je na tzv. M2Level i M1Level delove, nazvane prema nivoima u hijerarhiji metamodelovanja koje predstavljaju. Deo M2Level sadrži podatke o metamodelima, dok deo M1Level manipuliše modelima. U implementacionom smislu, ovi delovi predstavljaju komponente izvornog koda celokupnog okruženja za modelovanje.

Svaki od ovih delova podeljen je na dva dela koji nose oznake Fix i Gen. Delovi Fix sadrže elemente koji su isti i nepromenljivi za svaki metamodel, odnosno model. Oni predstavljaju neku vrstu bibliotečnih, fiksnih komponenti koje se povezuju sa ostatkom koda, da bi se dobilo konkretno okruženje za modelovanje u specifičnim domenima. Ti promenljivi delovi su označeni sa Gen, jer su generisani za konkretne domene.

Konkretnije, deo M2Fix sadrži definicije klasa i relacija koje predstavljaju jezgro UML metamodela. Te klase nose prefiks "M2" (M2Class, M2Attribute, M2Association, M2Generalization, itd.). Te klase su modelovane u alatu Rational Rose i za njih je generisan implementacioni C++ kôd na uobičajeni način.

Ove klase iz dela M2Fix instanciraju se u delu M2Gen. Ovaj deo sadrži kôd generisan za domene za koje se želi okruženje za modelovanje. Taj kôd zadužen je da kreira instance klasa iz M2Fix dela, povezane na odgovarajući način, tako da one odslikavaju metamodela datih domena. Te instance predstavljaju, kao što je rečeno, podatke o metamodelima koje okruženje za modelovanje interpretira da bi obavljalo generičke strukturne operacije nad modelima. Te generičke operacije su kreiranje, modifikacija i brisanje instanci apstrakcija i veza asocijacija iz metamodela.

Deo M1Fix sadrži definicije nekih osnovnih klasa potrebnih za generičko ponašanje alata. To su najpre neke osnovne apstraktne klase preuzete iz UML metamodela, *Element* i *ModelElement*, zatim klasa *Package* za opšte grupisanje elemenata, kao i neke druge. Tu su značajne klase *M1Instance*, iz koje je izvedena (direktno ili indirektno) svaka klasa generisana za apstrakciju iz domena, kao i klasa *M1Link* čiji objekti predstavljaju veze između instanci apstrakcija.

Konačno, deo M1Gen sadrži kôd na ciljnom programskom jeziku za klase generisane za apstrakcije iz domena. Kôd za ove klase generisan je specifično, kako bi se obezbedio generički, ali i programski pristup do atributa klase i veza sa drugim instancama. Kôd ovih klasa sadrži i operacije koje obezbeđuju specifično ponašanje objekata tih klasa.

Zahvaljujući ovakvom pristupu, potpuno isto generičko okruženje može se koristiti i u fazi metamodelovanja, i u fazi modelovanja. Naime, delovi Fix su potpuno isti za ove dve faze, dok se jedino razlikuju delovi Gen. U fazi metamodelovanja (slika 5.8a), u kojoj se definišu metamodeli domena i njihova preslikavanja, koriste se njihovi meta-metamodeli. To su u ovom slučaju jezgro UML, koje se uvek koristi kao meta-metamodel za definisanje metamodela domena, kao i metamodel za preslikavanje domena. U ovoj fazi alat na svom levom delu (slika 5.8a) prikazuje elemente ovih metamodela. Na desnoj strani se prikazuje i manipuliše modelima iz tih domena. Ti modeli predstavljaju metamodela određenih domena, kao i preslikavanja između njih.

Kada se želi okruženje za modelovanje u definisanim domenima, alat generiše sledeće. Za metamodel domena (paket sa stereotipom <<MetaModel>>), generiše se kôd za delove M2Gen i M1Gen za taj domen. Kôd za M2Gen sastoji se iz jedne *Singleton* klase koja u svom konstruktoru ima kôd za kreiranje objekata M2Fix klase koji predstavljaju metamodel datog domena. Kôd za M1Gen sastoji se iz klasa na ciljnom programskom jeziku generisanih za apstrakcije iz datog metamodela. Pored toga, unutar M1Gen dela generiše se i osnovna *Visitor* klasa za dati domen, koja definiše zajednički interfejs (prema obrascu *Visitor*) specifičnih klasa za obilazak modela iz tog domena. Konkretno izvedene *Visitor* klase korisnik može kasnije samostalno da definiše za svoje potrebe, kao specifično proširenje okruženja. Za svako preslikavanje (paket sa stereotipom <<DomainMapping>>) se pak generiše jedna konkretna *Visitor* klasa koja je pridružena datom izvorišnom domenu u okruženju za modelovanje, kao što je prikazano ranije.

Kada se generisani kôd prevede i poveže sa Fix delovima, dobija se okruženje za modelovanje kao na slici 5.8b. U tom okruženju se na levoj strani opet prikazuju objekti iz dela M2Gen koji opisuju metamodela domena. Na desnoj strani se prikazuju instance klasa dela M1Gen, čijim atributima i vezama korisnik manipuliše kroz generičke ili specifične dijaloge.

Na slici 5.8 prikazan je izgled okruženja u fazama metamodelovanja (slika 5.8a) i modelovanja (slika 5.8b) za demonstrativni primer konačnih automata. U fazi metamodelovanja, na raspolaganju su domeni *UMLCore* i *DomainMapping*. Korišćenjem

njihovih metamodela, definisani su (meta-)modeli `StateMachines`, koji predstavlja metamodel željenog domena konačnih automata, `OOPL`, koji predstavlja opšte korišćeni međudomen objektno orijentisanih programskih jezika, i preslikavanje iz `StateMachines` u `OOPL`. U fazi modelovanja su na raspolaganju upravo ova dva domena. Na slici 5.8b je prikazan jedan model konačnih automata koji je definisao korisnik, ali i jedan model iz domena `OOPL` koji je generisan automatskom transformacijom pomoću *Visitor* transformatora dobijenog iz preslikavanja sa slike 5.8a. Iz ovog međumodela može se dobiti ciljani programski kôd opet pomoću *Visitor* transformatora koji je ranije već implementiran za `OOPL` domen, nezavisno od specifičnog domena konačnih automata, i koji generiše sekvencijalni (tekstualni) izlaz na željenom programskom jeziku (C++ u ovom slučaju).

Postupak konstrukcije alata

Uprkos svojoj značajnoj funkcionalnosti i visokom nivou apstrakcije, ovaj alat je prilično jednostavan softver. To je posledica izuzetno visokog stepena redundantnosti koja se krije u prirodi samog problema (metamodelovanja, generičke manipulacije modelima, transformacije modela i slično), a koja je prilično uspešno otkrivana i apstrahovana. Jedna od interesantnih manifestacija te redundanse je činjenica da se okruženje za metamodelovanje može dobiti generisanjem iz samoga sebe, ukoliko se njegov metamodel unese u njega kao model. Ovo je jedan od najčešće isticanih aspekata svih okruženja za metamodelovanje. Tako je, posle izuzetno dugog i pažljivog projektovanja i osmišljavanja (reda nekoliko godina), sama prototipska implementacija dobijena izuzetno brzo (reda jedan čovek-mesec).

Ceo postupak razvoja alata bio je usmeren na dobijanje okruženja za metamodelovanje kao što je prikazano na slici 5.8a. Iako je to okruženje samo jedan specijalni slučaj istog tog generičkog okruženja za modelovanje koje se, ukoliko ono već postoji, može dobiti i automatski unošenjem metamodela za domene `UMLCore` i `DomainMapping`, bilo je potrebno ove domene implementirati ručno, jer okruženje, naravno, nije postojalo. Ipak, pažljivim manuelnim korišćenjem istih postupaka koje okruženje sprovodi automatski, a naročito principa transformacije modela (bilo u tekstualni, bilo u objektni oblik), postigla se pravilna arhitektura okruženja i kôd koji, iako je napravljen "klasičnim" postupcima, izgleda potpuno isto kao da je dobijen automatski iz istog okruženja. Naravno, time se i maksimalno izbegavalo ponavljanje, pa je dobijen softver malih dimenzija.

Postupak razvoja alata je ukratko tekao na sledeći način. Najpre su projektovani delovi `M1Fix` i `M2Fix`. Oni su najpre modelovani na jeziku UML, a zatim i implementirani na klasičan način na jeziku C++, korišćenjem navedenih alata. Najveći deo ovog modela preuzet je iz jezgra UML metamodela [B-OMG99], uz izvesne modifikacije.

Zatim je definisan metamodel UML jezgra (`UMLCore`) koji se koristi kao domen u okruženju za metamodelovanje. Njegova struktura je skoro ista kao i struktura modela u `M2Fix`, pa je ovaj deo modela upotrebljen ponovo, uz male izmene. Posle toga su definisani metamodeli domena `OOPL` i `DomainMapping`, koji su takođe uneseni u alat Rational Rose.

Potom je definisan način na koji se od nekog modela iz domena `UMLCore` dobija izvorni kôd za `M2Gen` i `M1Gen` delove okruženja, za slučaj kada taj model predstavlja zapravo metamodel nekog domena. Ova definicija izvedena je kao preslikavanje domena `UMLCore` u domen `OOPL` i služila je kao podsetnik za manuelnu implementaciju (posredstvom podešavanja C++ generatora koda ugrađenog u Rational Rose) istih tih domena `UMLCore` i `OOPL`. Za generisanje tela konstruktora *Singleton* klase koja predstavlja metamodel domena u delu `M2Gen` iskorišćeni su praktično u potpunosti algoritmi iz Priloga A. Oni su implementirani kroz jedan *Visitor* transformator iz `UMLCore` modela u kôd tela operacije na jeziku C++. Cilj ovog postupka je bio da se dobiju Gen delovi okruženja za metamodelovanje

na isti način kao što bi se to radilo pri generisanju okruženja za modelovanje u navedenim domenima.

Zatim su implementirani i *Visitor* generatori C++ koda iz modela iz domena OOPL, kao i iz modela iz domena DomainMapping. Pored toga, definisano je i preslikavanje iz domena DomainMapping u domen OOPL koje generiše potrebnu *Visitor* klasu transformatora, pri čemu se telo njegove operacije `generate()` generiše posredstvom navedenog *Visitor* transformatora sa implementiranim algoritmima iz Priloga A. Najzad, specifikacije preslikavanja iz domena UMLCore i DomainMapping u domen OOPL unesene su kao modeli sa metamodelom DomainMapping u dobijeni alat i iz njih su posredstvom *Visitor* generatora C++ koda iz domena DomainMapping dobijene konačne implementacije generatora M2Gen i M1Gen delova, odnosno transformatora modela u okruženju za modelovanje iz definisanog metamodela, odnosno preslikavanja. Na taj način sklopljen je kompletan alat za metamodelovanje, uključujući i metamodele UMLCore, DomainMapping i OOPL, njihove M2Gen i M1Gen implementacije, generatore M2Gen i M1Gen koda alata za modelovanje iz definisanog metamodela, kao i *Visitor* transformatora iz preslikavanja domena.

U cilju procene veličine celog softvera, ovde su navedeni brojevi klasa po delovima: delovi M2Fix i M1Fix imaju ukupno tridesetak klasa, implementacije domena UMLCore, OOPL i DomainMapping po oko trinaest, što sa nekoliko ostalih pomoćnih klasa ukupno čini oko osamdeset klasa u celom projektu (bez korisničkog interfejsa). Veličina klasa varira od samo nekoliko članova, do dvadesetak članova za neke fundamentalne klase iz Fix delova (prosek je svakako manji od deset). Veličina koda operacija klasa je u najvećem broju slučajeva manja od desetak linija, osim u specijalnim slučajevima *Singleton* klasa M2Gen dela (po jedna za domen) čija su tela uglavnom generisana automatski.

Ceo postupak je, dakle, koristio nekoliko ključnih obrazaca, postupaka i modela na mnogo mesta:

- Metamodel UML jezgra iskorišćen je za definiciju dela M2Fix, kao i za metamodel domena UMLCore.
- Algoritmi u Prilogu A iskorišćeni su za implementaciju generatora koda za M2Gen *Singleton* klasu (tačnije za telo njenog konstruktora), kao i za *Visitor* transformator modela iz domena DomainMapping u telo operacije `generate()` transformatora.
- Predložena metoda preslikavanja domena iskorišćena je za specifikaciju preslikavanja iz domena UMLCore u domen OOPL, kao definicija i implementacija generisanja delova M2Gen i M1Gen okruženja za modelovanje iz modela iz domena UMLCore u okruženju za metamodelovanje, kao i iz domena DomainMapping u domen OOPL, za definiciju i implementaciju transformatora modela iz definisanog preslikavanja.
- Projektni obrazac *Visitor* korišćen je više puta za implementaciju svih transformatora u različitim fazama razvoja i delovima alata.
- Više drugih projektnih obrazaca [B-Gam95] korišćeno je u različitim delovima implementacije (*Singleton*, *Command*, *Composite*, *Prototype*).

Glavne mogućnosti i pogodnosti alata

Projektna odluka da se okruženje za modelovanje dobija prevođenjem izvornog koda generisanog iz okruženja za metamodelovanje i njegovim spajanjem sa fiksnim delovima, povlači neke veoma značajne pogodnosti opisanog alata koje ne postoje u većini drugih:

- Alat za modelovanje je proizvoljno proširiv bilo kakvim funkcionalnostima koje se mogu implementirati na ciljnom programskom jeziku (C++), što znači da se okruženje može prilagoditi specifičnim potrebama modelovanja u određenom domenu. Drugim rečima, okruženje za modelovanje može biti proizvoljno složen i specifičan softver, a ne samo generičko okruženje ograničeno na ugrađenu funkcionalnost.

- Metamodel nekog domena, definisan u okruženju za metamodelovanje, manifestuje se u dva oblika, M2Gen i M1Gen. Deo M1Gen predstavlja implementaciju apstrakcija (klasa) domena na ciljnom programskom jeziku, tako da su objekti tih klasa programski dostupni na uobičajeni način. To znači da oni mogu da poseduju proizvoljno ponašanje definisano operacijama i strukturu definisanu atributima, onoliko složeno koliko to podržava ciljni programski jezik, a ne ograničenu ugrađenim mogućnostima okruženja.
- M1Gen datog domena može da sadrži i elemente (klase, attribute i operacije) koji nisu manifestovani u M2Gen delu, što znači i da ne podležu generičkim manipulacijama okruženja. Takvi elementi tipično služe za implementaciju složenijih struktura i ponašanja apstrakcija domena. Oni se mogu proizvoljno kombinovati sa elementima koji su manifestovani u M2Gen delu i koje generički deo alata prepoznaje.
- Ukoliko je generičko ponašanje alata dovoljno za manipulaciju modelima u specifičnom domenu, korisnik ne mora da definiše nikakve posebne operacije niti elemente korisničkog interfejsa. Pod ovim se podrazumevaju operacije kreiranja, brisanja i modifikacije instanci i veza kroz generičke dijaloge koje alat podrazumevano obezbeđuje. Međutim, ukoliko je to potrebno, korisnik može ovo ponašanje potpuno da redefiniše obezbeđivanjem sopstvenih operacija kao i elemenata korisničkog interfejsa (dijaloga) koji su prilagođeni semantici i potrebama datog domena.

Sve ovo u potpunosti podržava sledeće važne principe čijem se ispunjenju težilo pri razvoju alata. Prvo, alat za modelovanje je proizvoljno složen softver, koji može imati proizvoljno složenu i specifičnu funkcionalnost. Sa druge strane, postoje neki opšti, zajednički principi modelovanja koje treba generički podržati alatima za metamodelovanje, kako se oni ne bi implementirali za svaki domen ispočetka. Prema tome, okruženje za metamodelovanje treba da predstavlja *podršku* za brzu i jednostavnu implementaciju tih opštih elemenata alata za modelovanje, ali ne i *ograničenje* koje sprečava implementaciju specifične funkcionalnosti. Navedenim pristupom okruženje ne nameće svoja ograničenja, već predstavlja proširiv softver pisan na ciljnom programskom jeziku (C++). Zato su njegova eventualna ograničenja zapravo posledica ograničenja tog jezika, što je praktično zanemarljivo, jer okruženje može da obezbedi svaku funkcionalnost koja se može implementirati na tom jeziku.

Drugo, neko okruženje (engl. *framework*), pa i okruženje za modelovanje, treba da obezbedi podrazumevane ponašanje gde god je to moguće, tako da korisnik *ne mora* da definiše ponašanje ukoliko mu ono podrazumevano odgovara. Međutim, svako takvo ponašanje mora da bude dostupno za *redefinisavanje*, što znači da korisnik može (ali ne mora) da definiše specifično ponašanje samo za neki posebni slučaj od interesa, na proizvoljan način.

Struktura samog alata je konzistentna, jer se zasniva samo na nekoliko ključnih koncepata: metamodel, model, vizitor i transformator. Za svaki metamodel definisan u okruženju za metamodelovanje, alat generiše osnovnu *Visitor* klasu namenjenu za implementaciju različitih obilazaka strukture modela. Transformatori su samo jedna vrsta ovakvih vizitora koji se generišu iz preslikavanja domena. Pored njih, korisnik može da definiše proizvoljne specifične vizitore. Svi vizitori, kako transformatori, tako i oni korisnički definisani, na raspolaganju su korisniku alata za modelovanje kao elementi metamodela, tako da korisnik može aktivirati bilo koji od njih za bilo koji model iz odgovarajućeg domena. Na ovaj način se mogu aktivirati i transformatori modela dobijeni iz preslikavanja domena, ali i bilo koji drugi specifični vizitori, npr. generatori tekstualnih izveštaja, vizitori zaduženi za proveru konzistentnosti modela i slično.

Svi modeli i metamodeli prikazuju se korisniku na isti konzistentan način. Na primer, čak i izveštaj o greškama prilikom provere konzistentnosti modela, koja se inače za ugrađene domene obavlja ugrađenim vizitorima, predstavlja model u posebnom domenu koji je za to

namenjen (sadrži apstrakcije greške, upozorenja i slično). Kao što je već rečeno, i preslikavanja domena su modeli iz domena `DomainMapping`. Generisanje koda na nekom ciljnom objektno orijentisanom programskom jeziku može se vršiti korišćenjem ugrađenog domena `OOPL`, za koji je već realizovan generator C++ koda, ponovo kao vizitor.

Sve navedene karakteristike čine realizovani alat fleksibilnim, ali ipak jednostavnim softverom koji se može upotrebljavati za (meta)modelovanje u najrazličitijim domenima, i koji se značajno razlikuje od svih trenutno dostupnih alata te vrste.

VI Primeri upotrebe

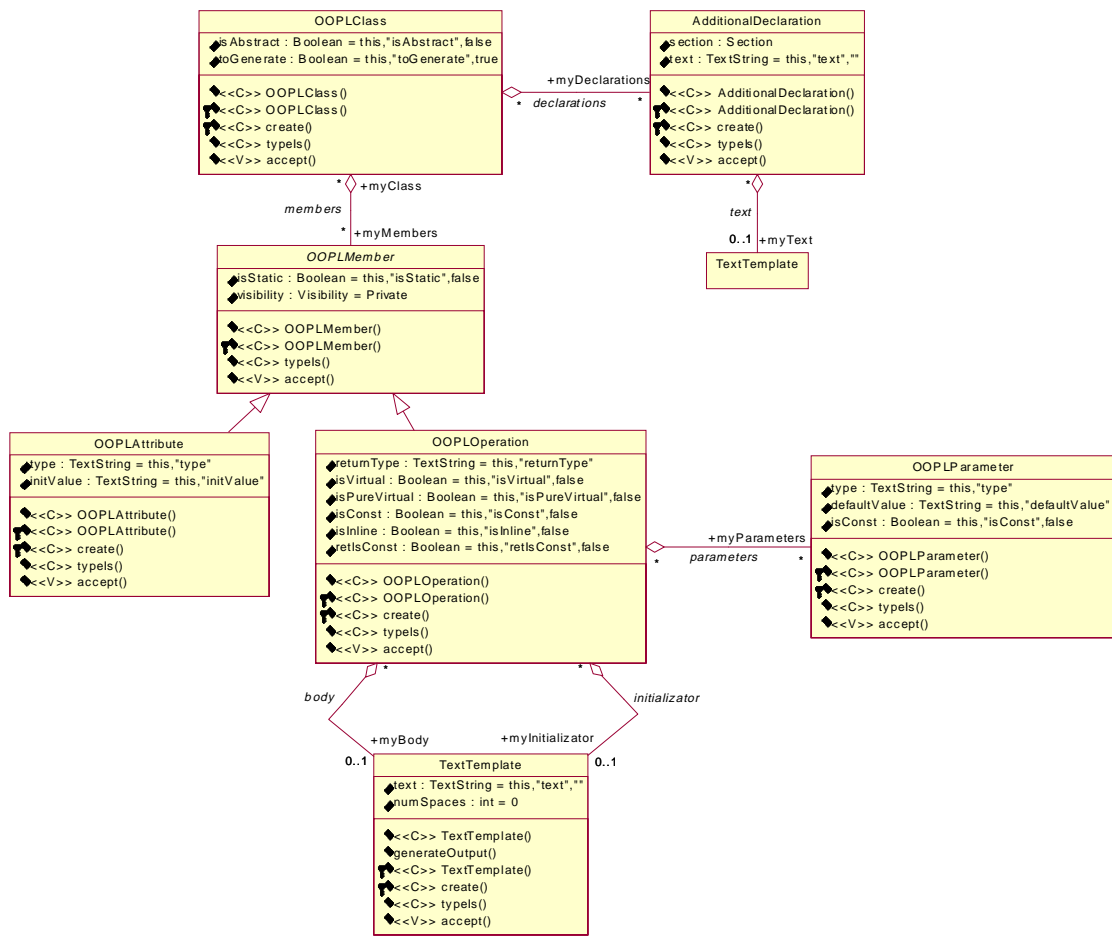
Konstrukcija alata za metamodelovanje

Prvi primer upotrebe predložene metode bio je u realizaciji samog prototipskog alata za metamodelovanje koji tu metodu podržava. Pri tom, specifikacije preslikavanja domena u prvoj fazi razvoja nisu korišćene za automatsko generisanje transformatora, jer okruženje koje bi to uradilo nije ni postojalo, već samo kao podsetnik za implementaciju određenih delova modela alata na ciljnom programskom jeziku (C++). Preglednost specifikacija je omogućila veoma jednostavnu i brzu željenu implementaciju sa malim brojem grešaka. Time je pogodnost upotrebe predložene metode, čak i bez podrške alata, potvrđena na jednom konkretnom netrivialnom primeru.

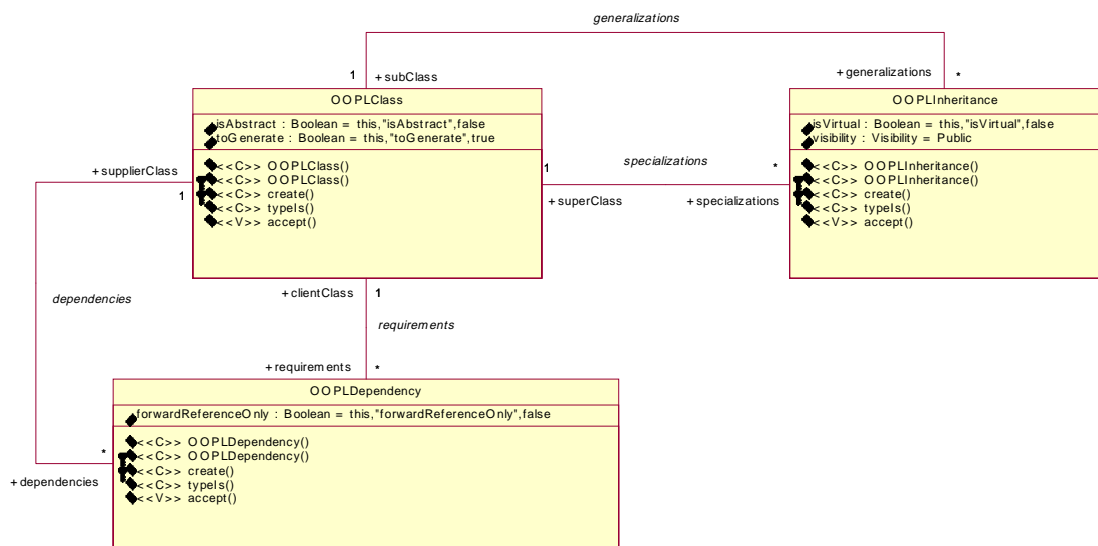
Kao što je već ranije opisano, u alatu je najpre definisan domen OOPL koji uključuje najznačajnije apstrakcije koje se sreću u popularnim objektno orijentisanim programskim jezicima. Ovaj domen služi kao međudomen za generisanje ciljnog C++ koda, jer se njegovim korišćenjem postižu sve pogodnosti navedene na početku ovoga rada. Za ovaj domen napravljen je i generator C++ koda. Zatim je definisano preslikavanje iz domena UMLCore u domen OOPL, kojim se opisuje način na koji se metamodel nekog domena (definisan pomoću koncepata domena UMLCore) manifestuje u M2Gen i M1Gen delovima generisanog alata za modelovanje u tom domenu. Najzad, definisano je i preslikavanje iz domena DomainMapping u OOPL, kojim se opisuje način na koji se preslikavanje domena manifestuje u M2Gen i M1Gen delovima generisanog alata za modelovanje koji izvršava definisanu transformaciju. Detalji metamodela domena OOPL kao i dva navedena preslikavanja opisani su u nastavku.

Metamodel domena OOPL

Metamodel domena OOPL prikazan je na slici 6.1. Na slici 6.1a prikazane su osnovne apstrakcije domena: klasa (OOPLClass), član klase (OOPLMember), atribut (podatak član klase, OOPLAttribute), operacija (funkcija članica klase, OOPLOperation), i parametar operacije (OOPLParameter). Na dijagramu 6.1b prikazani su koncepti relacija zavisnosti (OOPLDependency) i izvođenja (OOPLInheritance) klasa u ciljnom programskom jeziku. Treba primetiti da su to ujedno i jedine relacije koje podržava ciljni programski jezik kakav je C++, odnosno da relacija asocijacije nije direktno podržana. Apstrakcija asocijacije je koncept iz domena višeg nivoa, kakav je UMLCore, a koji se u implementaciji objektnog modela na programskom jeziku manifestuje kao odgovarajući podatak član klase. Ovakvo razdvajanje koncepata po domenima i uključivanje samo onih koncepata koje programski jezik direktno podržava, pokazalo se kao ključna olakšica u definisanju kako generatora C++ koda iz domena OOPL, tako i preslikavanja iz drugih domena u ovaj.



(a)



(b)

Slika 6.1: Metamodel domena OOPL. (a) Glavni dijagram koji prikazuje osnovne apstrakcije klase, člana klase, atributa i operacije sa parametrom. Telo operacije zadaje se preko objekta klase `TextTemplate`. (b) Dijagram koji prikazuje koncepte relacija zavisnosti i nasleđivanja između klasa u objektnom jeziku.

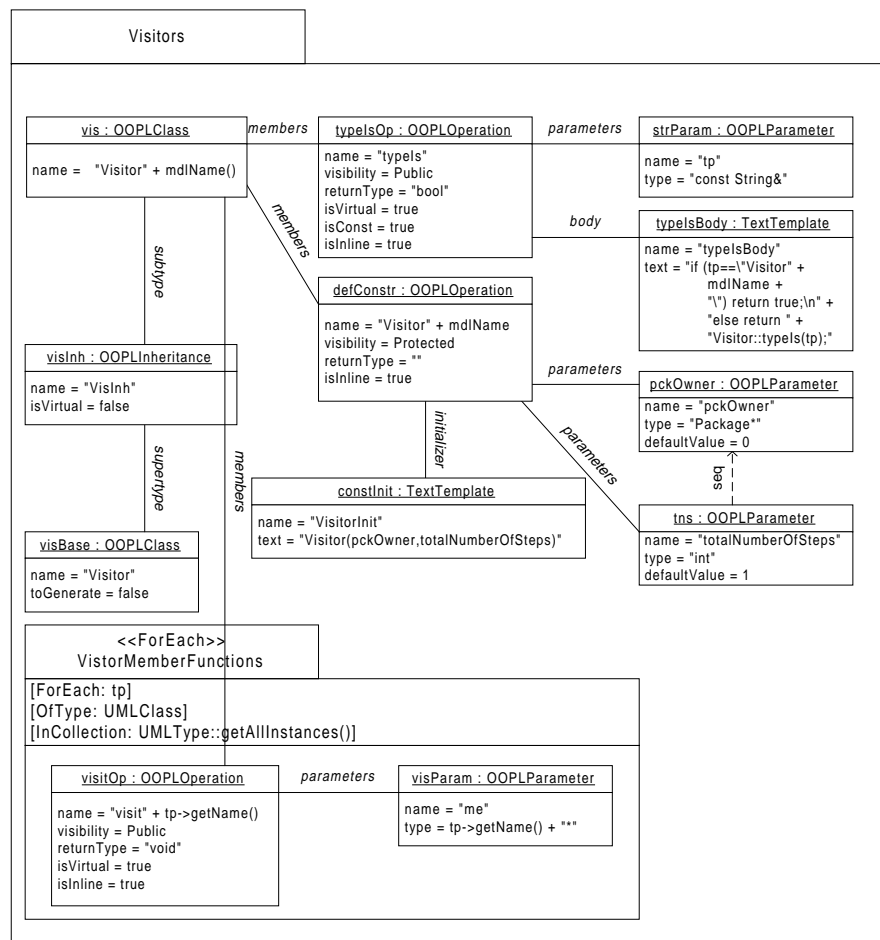
Posebno je interesantan fleksibilan način na koji se može definisati telo funkcije članice. Telo funkcije definiše se kroz objekat klase `TextTemplate`. Objekat klase `TextTemplate` može, ali ne mora da sadrži proizvoljno mnogo objekata klase `TextPart` koji predstavljaju delove teksta koji se generiše. Ukupan generisani tekst dobija se spajanjem teksta definisanog u objektu `TextTemplate` i svim objektima `TextPart` koji mu pripadaju. Ovakav pristup posebno je pogodan za iterativno generisanje delova tela funkcije koje se može sresti u preslikavanjima domena. Osim toga, `TextTemplate` podržava i parametrizovano generisanje teksta (pomoću šablona), kao i generisanje teksta pomoću obrasca *Visitor*. Na ovaj način postignuto je potpuno fleksibilna specifikacija generisanja sekvencijalne (tekstualne) forme iz objektnog modela, jer su podržane sledeće mogućnosti:

- Objekat klase `TextTemplate` ne mora da sadrži delove (objekte `TextPart`), već se izlazni tekst dobija samo iz njega.
- Objekat klase `TextTemplate` može da sadrži delove (objekte `TextPart`), kada se izlazni tekst dobija spajanjem izlaznog teksta dobijenog iz njega i pridruženih objekata klase `TextPart`.
- I objekti klase `TextTemplate` i objekti klase `TextPart` mogu svoj tekstualni sadržaj da definišu ili kao tekstualni šablon u atributu `text`, ili kao tekstualni šablon u ulaznoj tekstualnoj datoteci (zadatoj imenom ili referencom na `istream`), ili se on dobija generisanjem pomoću pridruženog objekta klase `Visitor`.
- Ukoliko se tekstualni sadržaj dobija kao šablon (iz atributa `text` ili iz datoteke), on može sadržati reference na parametre koji se definišu kao objekti klase `TextParameter` pridružene klasi `TextTemplate`. Na taj način se sadržaj definiše kao parametrizovani šablon, gde se sve reference na parametre u tekstualnom šablonu zamenjuju konkretnim vrednostima zadatim objektima klase `TextParameter`.

Takođe treba uočiti atribut `toGenerate` klase `OOPLClass`. Ukoliko je vrednost ovog atributa `False`, data klasa neće biti generisana u ciljnom kodu. Ovo je pogodno za definisanje relacija (nasleđivanja i zavisnosti) od klasa koje treba generisati prema klasama koje već postoje u fiksni delovima okruženja, ili u bibliotekama na primer. Te postojeće klase se u specifikacijama preslikavanja predstavljaju instancama tipa `OOPLClass` sa atributom `toGenerate` postavljenim na `False`.

Preslikavanje iz UMLCore u OOPL

Preslikavanje iz domena `UMLCore` u domen `OOPL` definiše način na koji se neki metamodel, definisan u fazi metamodelovanja kao model u domenu `UMLCore`, manifestuje u `M2Gen` i `M1Gen` delovima generisanog alata za modelovanje. U Prilogu B, na slikama B.1 i B.2 dato je kompletno preslikavanje. Ovde je, radi konciznosti, na slici 6.2 prikazan samo dijagram koje definiše generisanje *Visitor* klase i njenih funkcija članica. Po jedna osnovna apstraktna *Visitor* klasa generiše se podrazumevano za svaki domen, a u njoj se, pored ostalih režijskih operacija, generiše i po jedna funkcija članica za svaku apstrakciju iz metamodela. Apstrakcije se javljaju kao instance tipa `UMLClass` iz domena `UMLCore`.



Slika 6.2: Način na koji se generiše *Visitor* klasa i njene funkcije članice u alatu za modelovanje, za svaki metamodel definisan kao model u domenu *UMLCore*.

Preslikavanje iz DomainMapping u OOPL

Preslikavanje iz domena *DomainMapping* u domen *OOPL* definiše način na koji se neko preslikavanje domena, definisano u fazi metamodelovanja kao model u domenu *DomainMapping*, manifestuje u generisanom alatu za modelovanje. U Prilogu B, na slici B.3 prikazano je ovo preslikavanje. Ono definiše generisanje jedne *Singleton* klase koja će predstavljati transformator zadat datim preslikavanjem. Telo njene operacije `generate()` dobija se pomoću odgovarajuće *Visitor* klase koja implementira algoritme generisanja koda transformatora prikazane u Prilogu A.

Transformacija objektnog u relacioni model

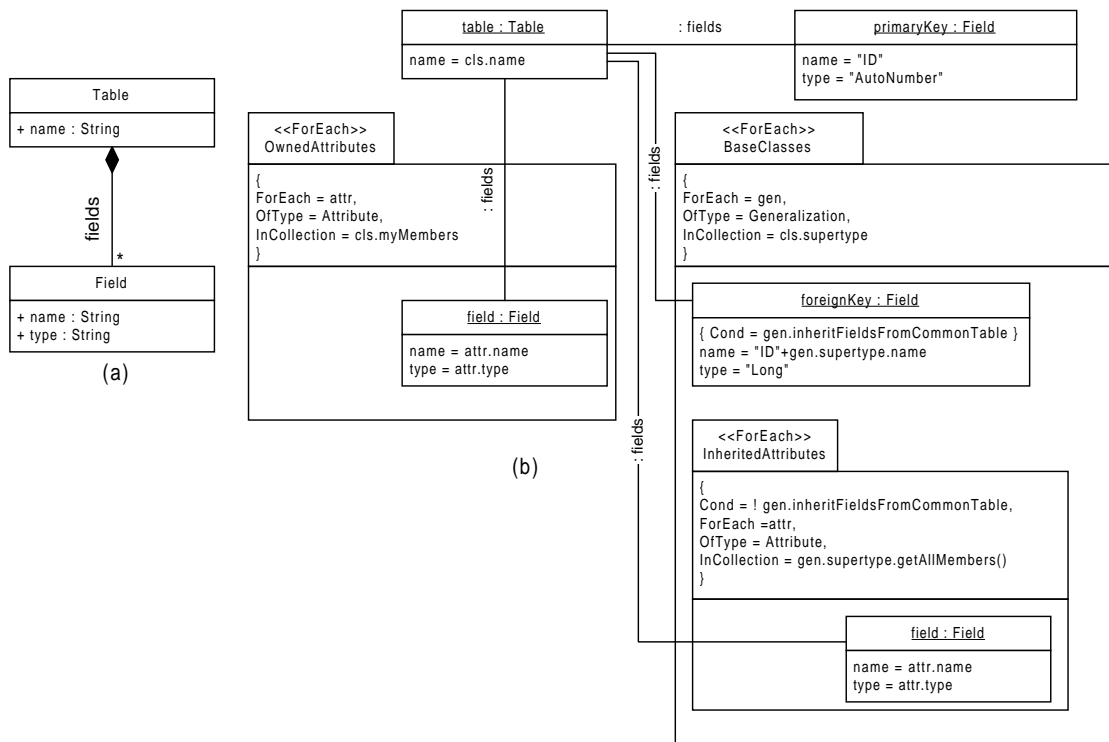
Ovaj primer bavi se problemom transformisanja objektno orijentisanog strukturnog modela u šemu relacione baze podataka. Ova transformacija je uobičajeni zadatak u mnogobrojnim slučajevima aplikacija u kojima se perzistencija objekata obezbeđuje preko relacione baze podataka. Apstrakcije izvorišnog domena koje su bitne za preslikavanje su: klase, atributi, asocijacije i nasleđivanja (generalizacije). U ciljnom domenu (relacioni model) postoje veoma jednostavne apstrakcije – tabele i njihova polja (kolone). Osnovni principi ove transformacije su opšte poznati [B-Mil01b]. Klase se preslikavaju u tabele, a njihovi atributi u polja tih tabela. U cilju identifikacije objekata, tipično se u tabeli za klasu obezbeđuje posebno polje koje predstavlja primarni ključ (engl. *primary key*) i jedinstvenu identifikaciju sloga tabele (vrste, engl. *record*), odnosno instance date klase. Ovde će ta polja imati ime "ID". Asocijacije se najjednostavnije preslikavaju u posebne tabele koje imaju po dve kolone za spoljne ključeve (engl. *foreign key*) koji predstavljaju identifikacije vezanih instanci. Preslikavanje asocijacionih klasa je nešto složenije i nije značajno za ovaj primer.

Preslikavanje nasleđivanja predstavlja poseban problem. U principu, postoje dva najčešće korišćena načina realizacije nasleđivanja klasa u relacionoj bazi. Prvi pretpostavlja da izvedena klasa ima svoju nezavisnu tabelu koja poseduje polja za sve attribute, uključujući i one nasleđene. Drugim rečima, svi atributi koje poseduju objekti te klase (i direktni i nasleđeni) predstavljeni su poljima iste tabele te klase. U tom pristupu, objekat se predstavlja jednim slogom iz jedne tabele svoje klase. U drugom pristupu, izvedena klasa ima svoju tabelu u kojoj su samo polja za attribute koji su direktno deklarirani u toj klasi, ali ne i za one nasleđene. Zbog toga se objekat izvedene klase predstavlja pomoću više slogova, i to iz tabele za njegovu klasu i tabela za osnovne klase u celoj hijerarhiji nasleđivanja. Ukoliko ID polje sadrži vrednost koja je globalno jedinstvena identifikacija objekta, ti slogovi su povezani preko polja ID. Ukoliko pak polje ID sadrži vrednost jedinstvenu samo u okviru date tabele, tabela izvedene klase mora imati spoljni ključ kojim se povezuje sa slogom osnovne klase (ili više njih ako je dozvoljeno višestruko nasleđivanje) [B-Mil01b].

Ovaj primer koncentriše se samo na problem realizacije nasleđivanja, jer su ostali elementi preslikavanja znatno jednostavniji. U ovom primeru pretpostavlja se da su korisniku na raspolaganju obe strategije realizacije nasleđivanja koje može da bira za svaku takvu relaciju postavljanjem posebnog atributa `inheritFieldsFromCommonTable`. Ako je ovaj atribut generalizacije postavljen na `True`, izabran je drugi pristup, pri čemu se pretpostavlja da vrednost polja ID nije globalno jedinstvena, pa u tabeli treba generisati i spoljni ključ.

Izvorišni domen je ovde `UMLCore`. Krajnji odredišni domen je zapravo tekstualni jezik SQL (engl. *Structured Query Language*) u kome se mogu deklarirati (naredbama `CREATE TABLE`) tabele i njihova polja. Međutim, direktno preslikavanje iz domena `UMLCore` u ciljni domen SQL naredbi relativno je složeno, pa se i ovde uvodi međudomen sa objektnim metamodelom. Ovaj domen poseduje osnovne apstrakcije relacionog modela, kao što su tabele i polja. Njegov metamodel u najjednostavnijem obliku prikazan je na slici 6.3a. Za ovakav domen veoma je jednostavno realizovati generator SQL deklaracija.

Preslikavanje izvorišnog domena u ovaj relacioni domen prikazano je na slici 6.3b. Ono definiše delove odredišnog modela koji se generišu za svaku klasu (tj. instancu tipa `Class`) u izvorišnom modelu. Za oba načina preslikavanja tabela generisana za klasu sadrži primarni ključ sa imenom "ID" i tipa "AutoNumber", kao i skup polja koji su posledica



Slika 6.3: Primer generisanja šeme relacione baze podataka iz UML strukturnog modela. Primer se koncentriše na nasleđivanje. Metamodel izvornog domena je UML jezgro (nije prikazan ovde). (a) Metamodel određivanja domena (relacioni model). (b) Preslikavanje. Operacija `getAllMembers()` vraća kolekciju svih članova jedne instance tipa `GeneralizableElement` (`Class` u ovom slučaju), kako direktnih, tako i onih nasleđenih.

atributa definisanih u toj klasi. U prvom pristupu, tabela takođe treba da poseduje i polja za sve nasleđene attribute date klase, i to za svako nasleđivanje kod koga je `inheritFieldsFromCommonTable = False`. Skup ovih nasleđenih atributa dobija se pozivom operacije `getAllMembers()` koja predstavlja specifičnu operaciju apstrakcije `Class` u izvorišnom domenu. Ovaj primer takođe potvrđuje svrsishodnost opredeljenja da implementacija apstrakcija u okruženju za modelovanje treba da predstavlja proizvoljno proširiv i programski dostupan kôd na ciljnom programskom jeziku. Uz tu mogućnost, sasvim je jednostavno realizovati navedenu operaciju kao članicu klase koja se u okruženju za modelovanje generiše kao M1Gen manifestacija apstrakcije `Class`. U drugom pristupu, međutim, tabela treba da ima još samo spoljni ključ tipa "Long" sa imenom "ID"<ime osnovne klase>.

Metoda ROOM

Ovaj primer prikazuje primenu predložene metode u realizaciji generatora koda za poznatu metodu modelovanja ROOM [I-Sel94]. ROOM (engl. *Real-Time Object-Oriented Modeling*) je metoda za objektno orijentisano modelovanje softverskih sistema za rad u realnom vremenu. Ova metoda poslužila je kao dobar poligon za eksperimente sa primenom predložene metode na jednom realnom i složenom domenu. Cilj je bio da se ispita upotrebljivost predložene metode za specifikaciju različitih varijanti generatora izvornog koda koji se dobija iz ROOM modela. Ovi eksperimenti vršeni su u okviru diplomskog rada opisanog u [L-Zar00]. Ovde će biti ukratko opisani samo najznačajniji koncepti ROOM domena i osnovni rezultati tih eksperimenata. Detalji vezani za metodu ROOM mogu se naći u [I-Sel94], a detalji vezani za metamodel i generisanje koda u [L-Zar00].

U okviru ovog primera obuhvaćeni su samo koncepti ROOM domena koji služe za opis strukture sistema. Koncepti vezani za opis ponašanja sistema nisu uzimani u obzir, jer se u metodi ROOM opis ponašanja oslanja isključivo na koncept konačnih automata, koji su bili predmet drugog rada [L-Laz99b]. Domen konačnih automata je inače korišćen u ovom radu i kao demonstrativni primer. Zadatak je bio formirati metamodel ROOM domena na osnovu opisa ove metode raspoloživog u [I-Sel94], zatim definisati nekoliko željenih varijanti oblika C++ koda generisanog iz apstraktnih ROOM modela i, najзад, definisati preslikavanja kojim se automatski mogu dobiti generatori koda. Cilj je bio ispitati efikasnost predložene metode u procesu definisanja varijanti generatora koda, odnosno stepen potrebnih modifikacija preslikavanja da bi se dobile te varijante.

U nastavku su najpre opisani osnovni strukturni koncepti ROOM domena i njegov metamodel, a zatim i najznačajnija preslikavanja za nekoliko varijanti generatora koda. Detaljnije specifikacije preslikavanja nalaze se u Prilogu C.

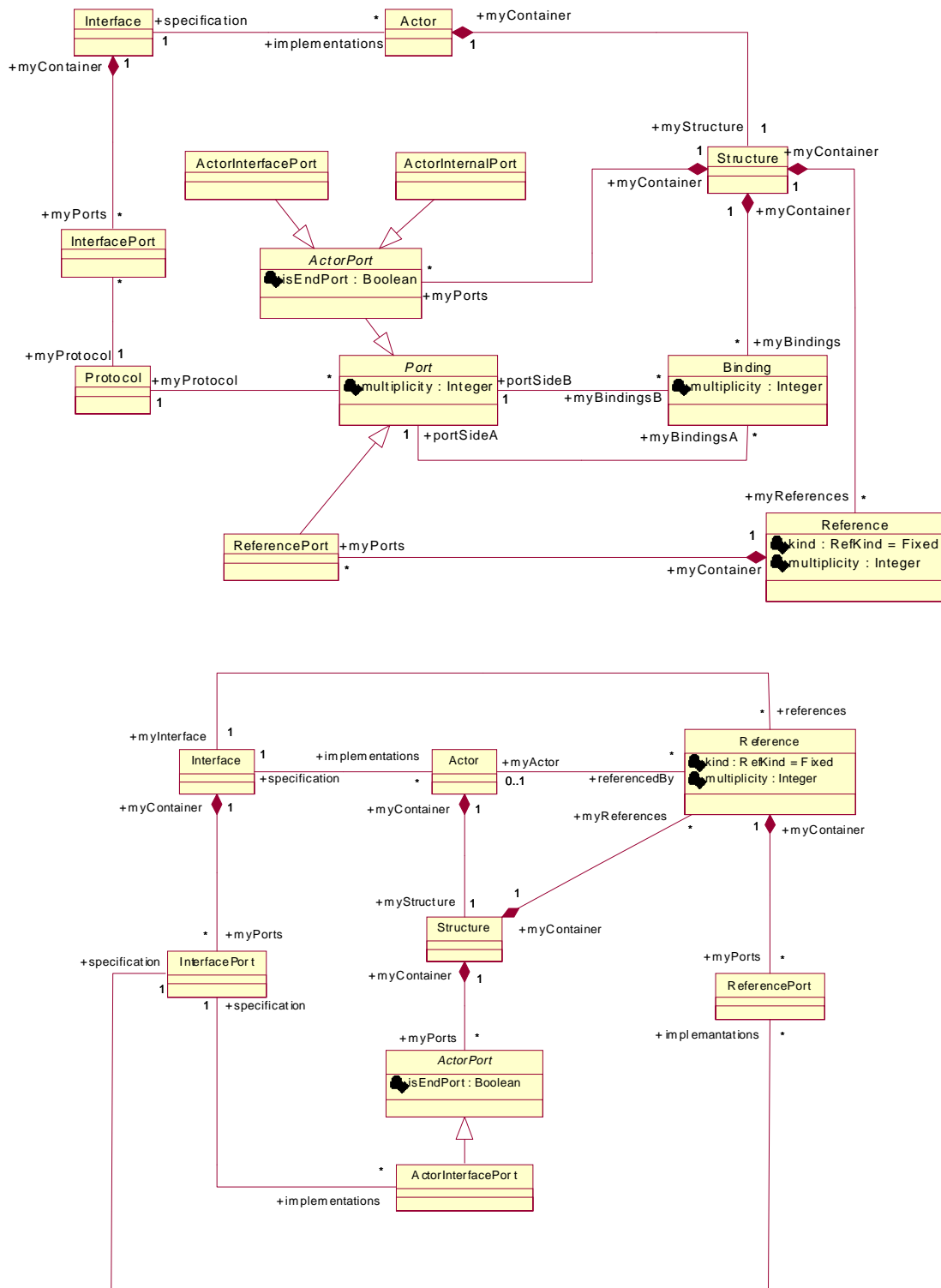
Osnovni koncepti domena ROOM

Osnovni koncept metode ROOM je tzv. *aktor* (engl. *actor*). Aktori su kompleksni, aktivni, potencijalno distribuirani i konkurentni arhitekturni objekti. Aktor ima svoju strukturu koja je enkapsulirana i koja nema direktne veze sa okruženjem aktora. Komunikaciju sa spoljnim svetom aktor obavlja isključivo preko svog *interfejsa* (engl. *interface*).

Aktori međusobno komuniciraju razmenom *poruka* (engl. *message*). Interfejs aktora je odgovoran za prijem poruka od drugih aktora, kao i za distribuciju poruka drugim aktorima koja je posledica internog ponašanja aktora. Aktor može, ali ne mora imati definisano ponašanje. Ponašanje aktora definiše se pomoću konačnog automata.

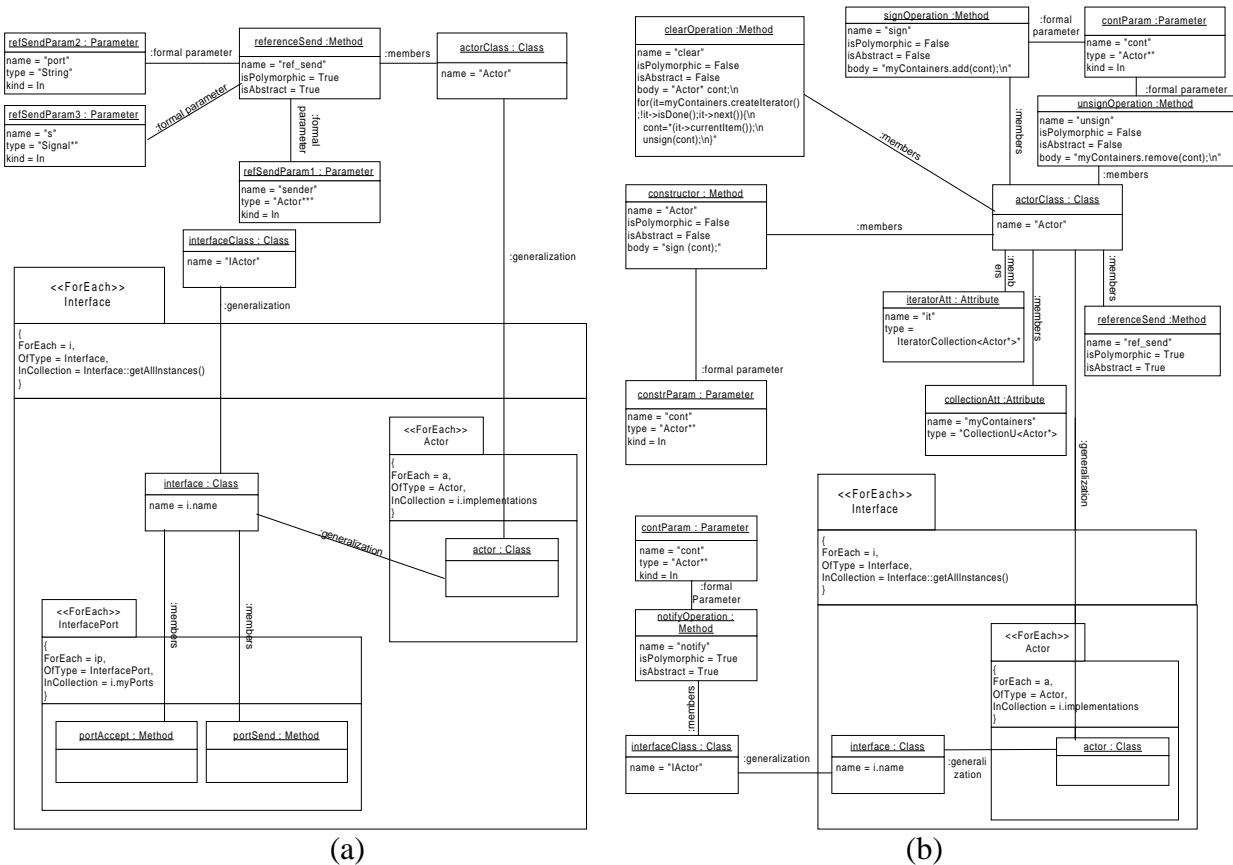
Ovako opisani aktori zapravo su *instance* klasa aktora (u metodi ROOM one se nazivaju *referencama* na klase aktora). Klase aktora služe kao šabloni za kreiranje instanci. Klasa aktora definiše strukturu, ponašanje i interfejs skupa svojih instanci. Nasleđivanje klasa aktora je takođe podržano u metodi ROOM.

Struktura klase aktora definiše način na koji su aktori ugrađeni jedni u druge. Aktori koji su ugrađeni u strukturu drugog aktora nazivaju se *referencama* na druge klase aktora. Te reference u nekim slučajevima mogu biti samo reference na interfejse, a ne na konkretne klase aktora. Drugi aspekt strukture je način na koji su ugrađeni aktori međusobno povezani. Ove veze definišu komunikacione puteve između aktora.



Slika 6.4: Deo metamodela domena ROOM.

Interfejs aktora sastoji se iz *portova* (engl. *port*), koji predstavljaju komunikacione objekte kroz koje protiču poruke. Port realizuje određeni *protokol* (engl. *protocol*). Protokol prvenstveno definiše skup poruka. *Poruka* se sastoji od imenovanog *signala* i proizvoljnog *podatka*. Protokol takođe može da definiše i dozvoljenu sekvencu poruka koje se mogu



Slika 6.5: Dva osnovna dijagrama preslikavanja domena ROOM u domen OOPL. (a) Preslikavanje za prvu, najjednostavniju verziju generatora koji podržava samo osnovne koncepte. (b) Preslikavanje za treću, najsloženiju verziju generatora koji podržava i koncepte relejnog porta i uvežene reference.

razmenjivati između aktora. Port je fizički objekat koji realizuje određeni protokol, odnosno koji se ponaša u skladu sa pravilima definisanim protokolom.

Veza (engl. *binding*) je fizički objekat koji služi kao kanal između portova. Veza služi kao medijum za prenos poruka. Veza je uvek deo strukture okružujućeg aktora: ona povezuje ili sadržane reference na aktore, ili referencu na aktor i njen okružujući aktor.

Ovo je samo kratak opis metamodela strukturnog dela domena ROOM čiji je najznačajniji deo prikazan na slici 6.4. Detalji se mogu naći u [L-Zar00].

Generatori koda i preslikavanja

Drugi deo eksperimenta obuhvatao je definiciju tri varijante generatora C++ koda iz ROOM modela. Prva varijanta podržavala je samo osnovne koncepte iz ROOM domena. U drugoj varijanti dodat je koncept tzv. *relejnog porta* (engl. *relay port*), a u trećoj i koncept *uveženih referenci* (engl. *imported reference*) [I-Sel94]. Za sve tri varijante realizovana su preslikavanja iz domena ROOM u već postojeći domen OOPL i posmatrane su razlike između ovih preslikavanja, odnosno modifikacije koje je trebalo uneti u dijagrame prethodne varijante u cilju dobijanja složenijih generatora.

Kao ilustracija, na slici 6.5 prikazana su dva osnovna dijagrama preslikavanja za prvu (najjednostavniju) i treću (najsloženiju) varijantu. Još neki izabrani dijagrami dati su u Prilogu C, a kompletno preslikavanje dato je u [L-Zar00]. Iz prikazanih dijagrama (a isti zaključak je izveden i za ostale dijagrame preslikavanja) mogu se jasno uočiti razlike u

preslikavanjima za različite varijante generatora. Te razlike su sasvim pregledne, pa njihovo unošenje nije zahtevalo veliki trud.

Otežavajuća okolnost prilikom definisanja preslikavanja je bio nedostatak alata sa vizuelnim grafičkim editorom dijagrama. Zbog toga su preslikavanja crtana u programu Visio. I pored toga, njihova dvodimenzionalna grafička priroda, kao i visok nivo apstrakcije, znatno su olakšali modifikacije preslikavanja u odnosu na klasičan programski pristup koji je bio jedina alternativa u ovom slučaju. Osim toga, koncepti nasleđivanja i redefinisavanja preslikavanja nisu bili izmišljeni u vreme realizacije ovih preslikavanja (oni su nastali upravo kao posledica ovih i sličnih eksperimenata, kada je uočena potreba za neznatnim proširivanjima i redefinisanjima postojećih preslikavanja). Logično je očekivati da bi primena ovih koncepata, uz potpunu podršku alata, dovela do još efikasnijeg i lakšeg razvoja generatora koda.

Logičko projektovanje hardvera

U ovom primeru predložena metoda je primenjena na domen koji se ne bavi modelovanjem softvera, već hardvera. I ovaj primer rađen je u okviru jednog diplomskog rada [L-Dor00], gde je zadatak bio da se definiše metamodel domena logičkog projektovanja hardvera, kao i preslikavanja iz kojih se može automatski dobiti generator koda koji simulira projektovana logička kola. Cilj je bio proveriti primenljivost predložene metode na ovaj nesoftverski, ali veoma značajan i poznat domen. Ovde je najpre ukratko opisan metamodel domena, a zatim i preslikavanja koja definišu generator koda. Detalji se mogu naći u [L-Dor00].

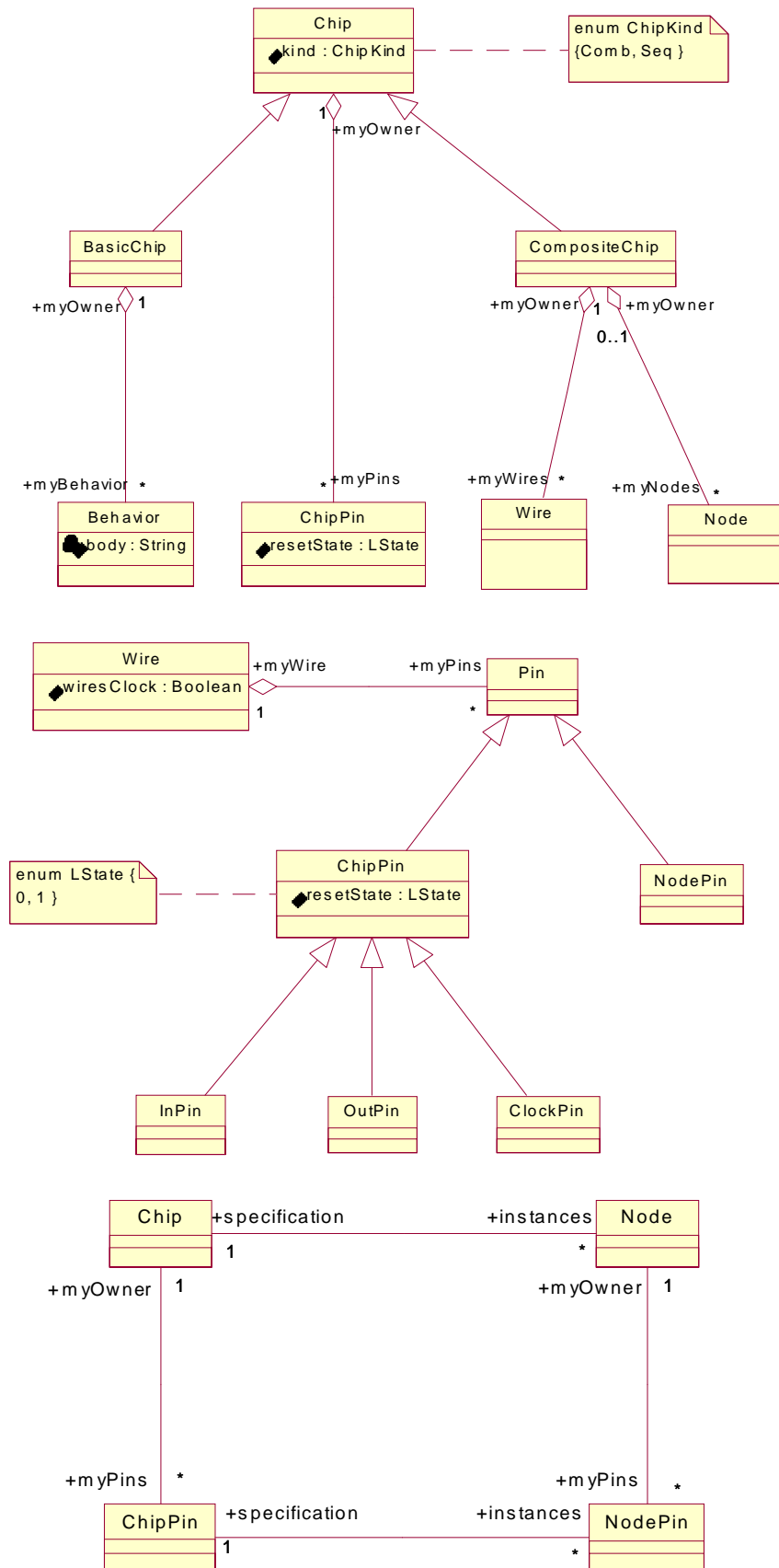
Metamodel domena logičkog projektovanja hardvera

Metamodel domena logičkog projektovanja prikazan je na slici 6.6. Apstrakcija *čip* (Chip) predstavlja tip hardverskog bloka koji može biti ili *prost* (BasicChip) ili *složen* (CompositeChip). Prosti čip predstavlja "crnu kutiju" koja nema svoju internu strukturu i čije se ponašanje (Behavior) može definisati pomoću koda na ciljnom programskom jeziku. Složeni čip je blok koji ima svoju internu strukturu, što znači da je hijerarhijski organizovan. Interna struktura složenog čipa sastoji se iz *čvorova* (Node) i *žica* (wire). Čvor je zapravo referenca na neki drugi čip i označava da će instanca nadređenog čipa sadržati instancu referisanog čipa (sadržanog čvora). Drugim rečima, čip i čvor su u asocijaciji koja predstavlja dihotomiju "tip-instanca".

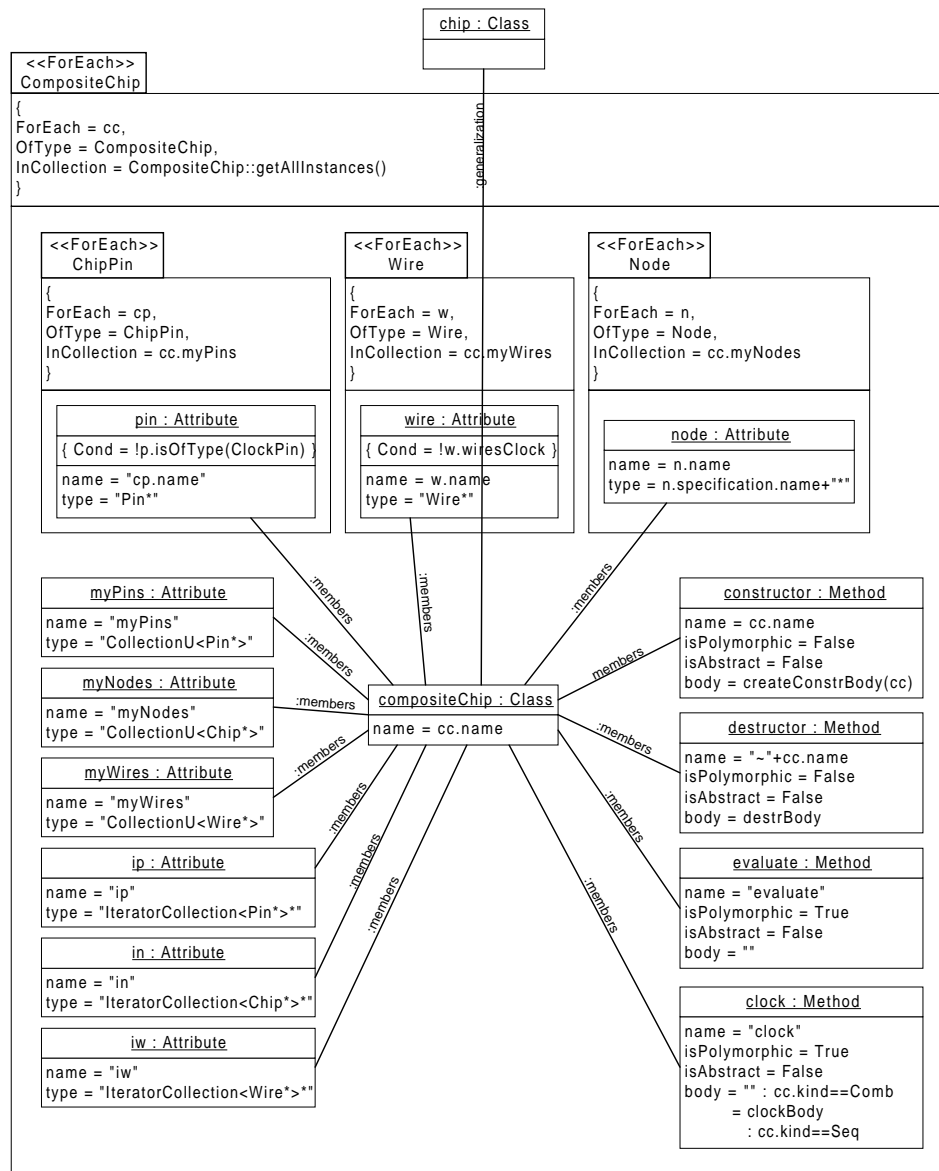
Svaki čip poseduje svoje *pinove* (ChipPin). Pin predstavlja deo interfejsa čipa. Čvor takođe poseduje pinove (NodePin). Pin čipa i pin čvora su takođe u asocijaciji "tip-instanca". U složenom čipu žice povezuju pinove čvorova. Pin može biti ulazni, izlazni, ili pin za signal takta. Značenje ovih koncepata je sasvim uobičajeno i intuitivno jasno, jer je preuzeto iz domena standardnih okruženja za logičko projektovanje hardvera.

Bitno je uočiti fleksibilnost koju pružaju koncepti složenog i prostog čipa. U toku projektovanja hardvera, modelovanje u ovom domenu ima za cilj generisanje koda na ciljnom programskom jeziku (C++ u ovom slučaju), čijim se izvršavanjem može simulirati projektovani hardver. Projektant može najpre da napravi svoj dizajn na vrlo visokom nivou apstrakcije. Taj dizajn bi podrazumevao podelu sistema na podsisteme i definisanje veza između njih. Ti podsistemi bi predstavljali proste čipove u čiju se podstrukturu projektant ne upušta, već čije ponašanje definiše neposredno programskim kodom. Dakle, prostim čipom se mogu opisati složeni delovi sistema u ranim fazama projektovanja.

U kasnijim fazama, posle izvršenih simulacija i testiranja, prosti čipovi mogu se dalje dekomponovati, čime postaju složeni. Ta dekompozicija može ići do nivoa elementarnih jedinica (npr. logičkih kola ili flip-flopova), koji sada ponovo predstavljaju proste čipove i čije se ponašanje definiše programskim kodom. Prostim čipovima se, dakle, u kasnijim fazama projektovanja predstavljaju najmanji i nedeljivi elementi sistema. Podsystem definisan kao prosti čip u ranoj fazi projektovanja, sa zadatim ponašanjem, mogao bi tako biti dodeljen drugom projektantu čiji bi zadatak bio da napravi složeni čip od realnih komponenti (prostih bibliotečnih čipova). Jedan primer ovog pristupa prikazan je u [L-Dor00].



Slika 6.6: Metamodel domena logičkog projektovanja hardvera.



Slika 6.7: Preslikavanje iz domena logičkog projektovanja hardvera u domen OOPL. Dijagram prikazuje specifikaciju generisanja koda za složene čipove.

Preslikavanje u domen OOPL

Jedan od ciljeva modelovanja u opisanom domenu je generisanje koda na ciljnom programskom jeziku (C++ u ovom slučaju) čijim se izvršavanjem projektovani hardver može simulirati i testirati. Preslikavanja su zato ponovo definisana za odredišni domen OOPL.

U ovom primeru se za svaki čip generiše po jedna klasa izvedena iz bibliotečne osnovne klase `chip`. Ta osnovna klasa ima polimorfnu operaciju `evaluate()`, koju treba da redefinišu prosti kombinacioni čipovi, i operaciju `clock()`, koju treba da redefinišu prosti sekvencijalni čipovi. Pinovi čipa manifestuju se kao podaci članovi bibliotečnog tipa `Pin`, žice kao podaci članovi bibliotečnog tipa `wire`, a čvorovi kao ugrađeni objekti (podaci članovi) odgovarajuće klase generisane za referisani čip. Dijagram preslikavanja koji opisuje način generisanja koda za složeni čip prikazan je na slici 6.7. Dijagram koji opisuje generisanje koda za prosti čip dat je u Prilogu D, a svi detalji mogu se naći u [L-Dor00].

Modelovanje Web aplikacija

Ovaj primer je samo jedan od mnogih koji pokazuju kako se primenom modelovanja u specifičnom domenu i predložene metode za preslikavanje domena može efikasno doći do rešenja relativno složenog problema u praksi. Osim toga, on predstavlja netrivialni slučaj preslikavanja između domena od kojih nijedan nije OOPL, odnosno gde cilj transformacije modela nije generisanje koda na nekom programskom jeziku, već specifičnog opisa aplikacije koga interpretira specijalizovano izvršno okruženje. Najzad, on prikazuje i način na koji se može koristiti princip sukcesivnih transformacija modela u cilju brzog dobijanja aplikacije iz visoko apstraktnih specifikacija, korišćenjem već realizovanih metamodela i preslikavanja. Primer je preuzet sa projekta *Socratenon* [K-Nik00], koji je imao za cilj razvoj složene Web aplikacije za edukaciju. Zbog svog praktičnog značaja ovaj primer opisan je detaljnije.

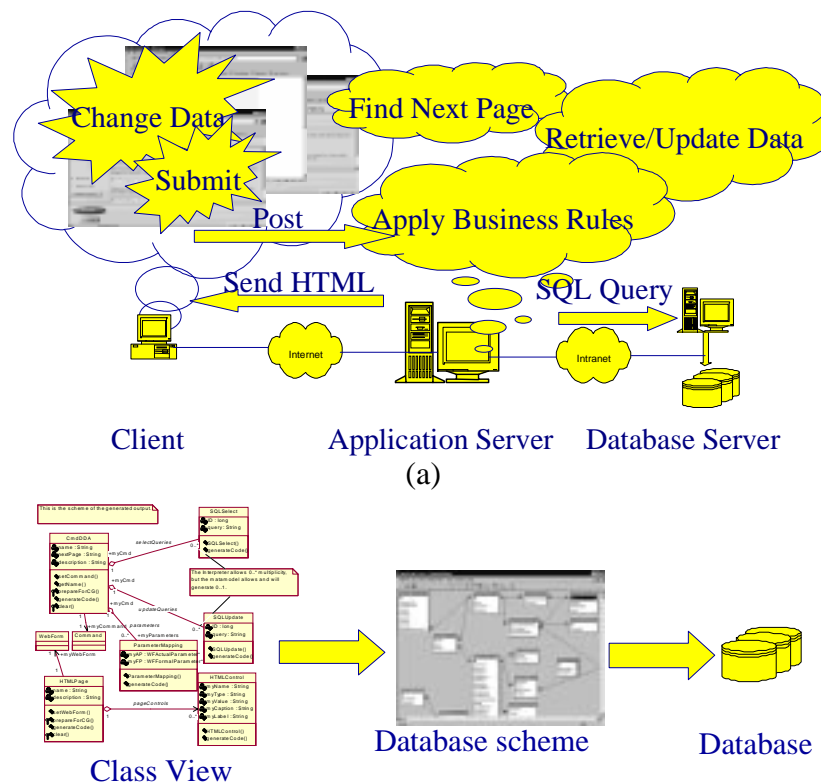
Socratenon je kompleksni sistem namenjen za edukaciju korisnika u industriji i školstvu. On se zasniva na složenom konceptualnom modelu koji omogućuje podešavanje nastavnih planova kognitivnim sposobnostima i preferencama pojedinih učenika. Njegova bogata funkcionalnost i korisnički interfejs stvaraju utisak postojanja "virtuelne učionice".

Socratenon je razvijen na Elektrotehničkom fakultetu u Beogradu, u saradnji sa Univerzitetom u Salernu, Italija. Specifičnost ovog razvoja bila je ta što je na raspolaganju za razvoj ove složene aplikacije (tačnije njene prve isporuke) bilo veoma kratko vreme (dva meseca) i samo tri programera. U prilog proceni složenosti, prva verzija sistema imala je oko dve stotine Web stranica. Zbog ovako složenih uslova bilo je neophodno razviti jednostavnu, ali efikasnu metodu za brzu proizvodnju Web aplikacija oslonjenih na bazu podataka, koju je, uz to, u datom vremenskom roku još bilo moguće podržati razvojnim i izvršnim okruženjem. Tako je nastala tehnika projektovanja Web aplikacija, u čijem su razvoju u potpunosti primenjena znanja iz oblasti metamodelovanja, modelovanja u specifičnom domenu, kao i rezultati predložene metode preslikavanja domena. Ovde će ta tehnika, kao i primena predložene metode preslikavanja domena, biti ukratko opisana. Na kraju ovog poglavlja data je jedna moguća generalizacija ove tehnike koja primenjuje predloženi princip sukcesivnih transformacija modela.

Predložena tehnika modelovanja Web aplikacija

Posmatrano iz perspektive infrastrukture i izvršavanja, jedan tipičan model Web aplikacije oslonjene na bazu podataka, kakav je i *Socratenon*, prikazan je na slici 6.8a. Na klijentskoj mašini prikazuje se neka HTML stranica. Stranica može biti formular koji sadrži kontrole (tekst-okvire, liste, dugmad itd.) i koji omogućuje pregled i modifikaciju podataka iz baze. Korisnik može da promeni prikazane podatke i pokrene operaciju podnošenja ("Submit"). Ova akcija uzrokuje poziv standardne *post* metode koja se sastoji iz slanja niza elemenata sa klijentske na serversku mašinu. Svaki od tih elemenata odgovara jednoj kontroli sa stranice i nosi ime i vrednost te kontrole. Aplikacioni server je odgovoran za primenu tzv. poslovnih pravila aplikacije, uzimanje i smeštanje podataka u bazu, kao i za iznalaženje naredne HTML stranice koja će biti poslata klijentu. U komunikaciji sa serverom baze podataka aplikacioni server izvršava standardne SQL upite.

Posmatrano iz konceptualne perspektive (slika 6.8b), a sledeći opšteprihvaćene principe objektno orijentisanog modelovanja, tipična Web aplikacija je zapravo samo implementacija objektnog modela, pri čemu je šema baze podataka dobijena iz modela klasa

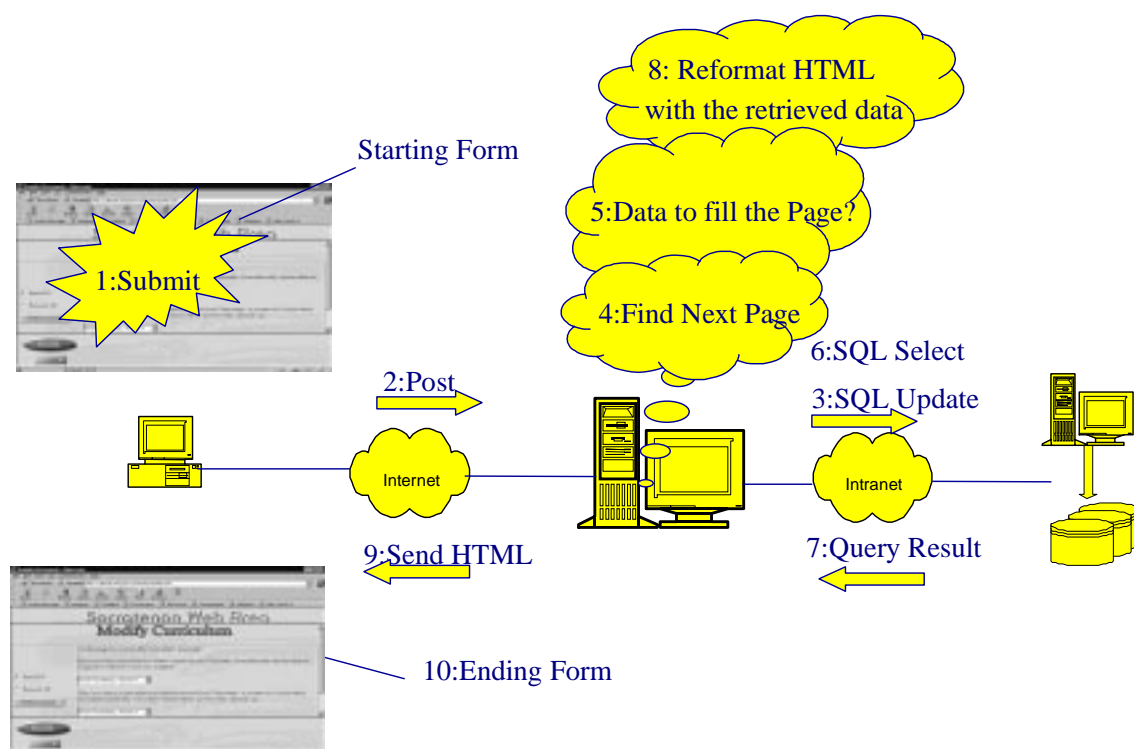


Slika 6.8: Tipičan model Web aplikacije oslonjene na bazu podataka. (a) Iz perspektive infrastrukture i izvršavanja. (b) Iz konceptualne perspektive.

na način opisan u jednom od prethodnih primera. Model klasa predstavlja strukturni konceptualni model datog problema za koji se razvija aplikacija. Sa druge strane, aspekt ponašanja aplikacije modeluje se slučajevima upotrebe (engl. *use case*) [B-Boo99]. Međutim, iskustvo pokazuje da se većina slučajeva upotrebe jedne tipične poslovne aplikacije može okarakterisati kao "strukturni" (engl. *structural use case*). To podrazumeva operacije kreiranja, modifikacije (vrednosti atributa) i brisanja instanci klasa iz konceptualnog modela aplikacije, kao i kreiranja i brisanja veza kao instanci asocijacija između tih klasa. Najčešće se veoma mali broj slučajeva upotrebe može smatrati specifičnim slučajevima ponašanja (engl. *behavioral use case*) koji koriste datu strukturu da bi obavili neku funkciju potpuno specifičnu za datu aplikaciju. Osim toga, strukturni slučajevi upotrebe mogu se projektovati korišćenjem različitih obrazaca, oslanjajući se na semantiku definisanog strukturnog modela, kao što će to biti ukratko prikazano na kraju ovog poglavlja. Prema tome, tehnika modelovanja Web aplikacija mora da podrži modelovanje na visokom nivou apstrakcije, da omogući brz razvoj tipičnih strukturnih slučajeva upotrebe, ali i da bude otvorena za specifične slučajeve ponašanja.

Ideja predloženog rešenja

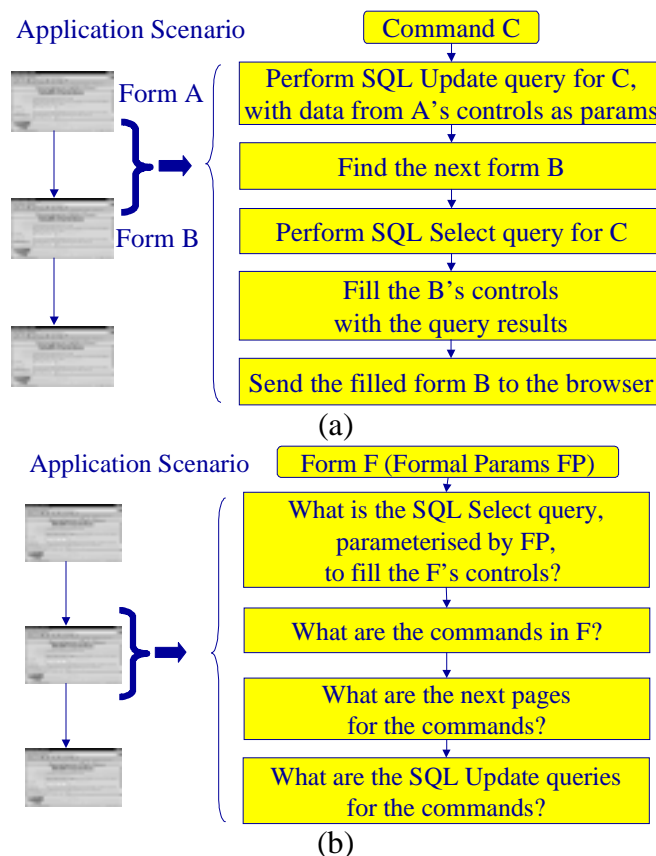
Ideja predloženog rešenja zasniva se na posmatranju tipičnog ciklusa interakcije između čoveka i mašine prikazanog na slici 6.9. U toku izvršavanja aplikacije korisnik modifikuje podatke na nekom polaznom Web formularu. Ova operacija ne uključuje nikakvu interakciju sa serverima, zato što se ona izvršava u potpunosti u okviru Web pretraživača (engl. *Web browser*). Ciklus počinje kada korisnik podnese dati formular (engl. *submit*). Ova operacija podnošenja može se uopšteno posmatrati kao izdavanje jedne od *komandi* (engl. *command*)



Slika 6.9: Tipičan ciklus interakcije između čoveka i mašine u Web aplikaciji oslonjenoj na bazu podataka.

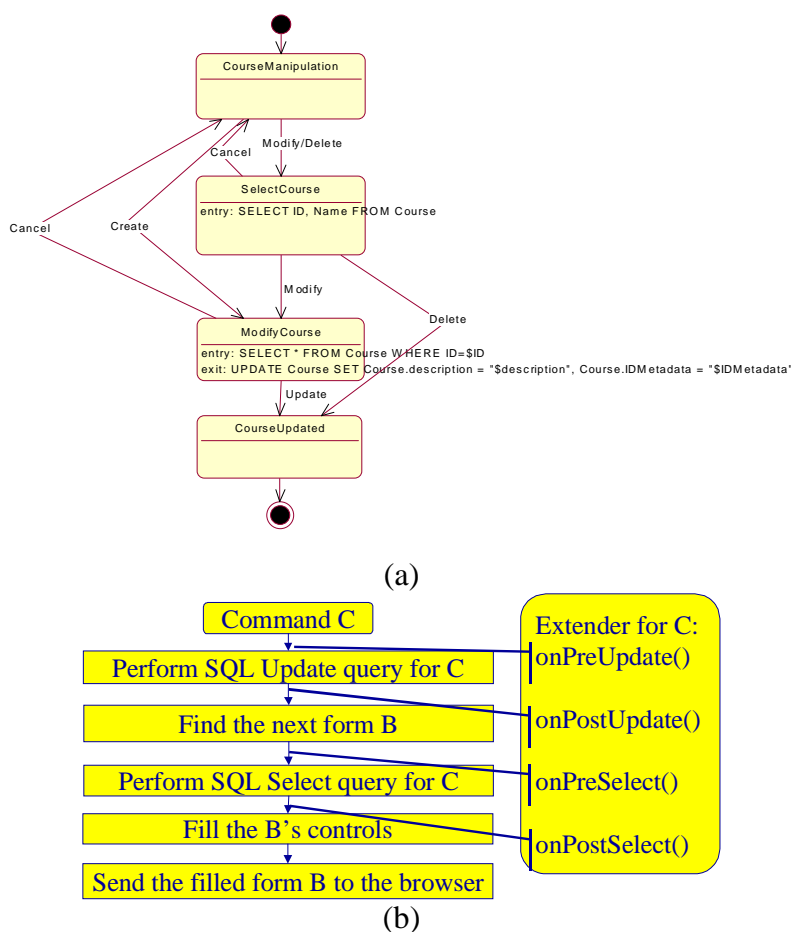
koje su ponuđene na formularu. Tada se podaci sa formulara prenose do servera. Aplikacioni server tada treba da ažurira bazu podacima dobijenim sa klijenta. Ova operacija biće posmatrana kao izvršavanje "SQL-Update" upita, iako ti upiti mogu da budu bilo koji upiti ažuriranja baze ("Insert", "Delete" ili nijedan). Potom server treba da pronade narednu HTML stranicu za izdatu komandu. Kako i ta stranica može da bude formular čije kontrole treba popuniti podacima iz baze pre slanja klijentu, server mora da izvrši "SQL-Select" upite da bi pročitao podatke iz baze. Po prijemu rezultata tih upita iz baze, server mora da popuni konkretne vrednosti kontrola na HTML stranici podacima dobijenim iz baze. Na kraju, server šalje odredišnu HTML stranicu klijentu, pa se ciklus ponavlja.

Prema tome, moguće je definisati konceptualni pogled na prikazani ciklus koji je bitan za izvršavanje kao na slici 6.10a. Kada se izda komanda, potrebno je izvršiti skup SQL-Update upita. Svaki taj upit može biti parametrizovan podacima koji se dobijaju sa polaznog formulara. Na primer, ako korisnik definiše vrednost polja ID nekog sloga u bazi na polaznom formularu, onda upit može biti parametrizovan na sledeći način: "UPDATE ... WHERE ID=\$ID", gde je \$ID referenca na vrednost iz kontrole sa imenom "ID" u polaznom formularu. Server treba da zameni ove reference konkretnim vrednostima dobijenim iz polaznog formulara pre izvršenja upita. Posle toga, server pronalazi odredišnu HTML stranicu pridruženu izdatoj komandi. Ta stranica može opet da sadrži kontrole čije su vrednosti ponovo definisane kao reference na rezultate SQL-Select upita pridruženih komandi. Takva HTML stranica se obično naziva šablonom. Prema tome, server izvršava SQL-Select upite (koji takođe mogu biti parametrizovani vrednostima kontrola sa polaznog formulara) i zamenjuje reference konkretnim vrednostima rezultata upita. Na kraju se ovako popunjena stranica šalje klijentu. Iz ovoga sledi da aplikacioni server može da izvršava program koji interpretira specifikaciju aplikacije koja je definisana pomoću pojmova stranica (kao šablona), komandi i parametrizovanih upita pridruženih komandama.



Slika 6.10: Konceptualni pogled na predloženo okruženje. (a) Ono što je bitno računaru za izvršavanje: ciklus tranzicije. (b) Ono što je bitno korisniku i projektantu aplikacije: formulari, njihovi parametri, upiti i komande.

Međutim, opisani model je orijentisan na izvršavanje i nalazi se na suviše niskom nivou apstrakcije. Za korisnika i projektanta aplikacije mnogo je pristupačniji konceptualni model na višem nivou apstrakcije, prilagođen samom postupku projektovanja aplikacije. Za tu svrhu je izgrađen konceptualni model prikazan na slici 6.10b. Ovaj model može se automatski transformisati u prethodni, kao što će to biti prikazano kasnije. Aplikacija se tako definiše kao skup *Web formulara*, pri čemu svaki formular može da ima svoje *formalne parametre*. SQL-Select upit je pridružen formularu i služi za popunjavanje njegovih kontrola. Vrednosti kontrola mogu da referišu kako rezultate SQL-Select upita, tako i formalne parametre formulara. Formalni parametri i vrednosti kontrola predstavljaju skup "lokalnih promenljivih" formulara, kao što su to formalni parametri i lokalne promenljive u proceduralnim jezicima. Pored toga, formularu je pridružen i SQL-Update upit koji može biti parametrizovan lokalnim promenljivima formulara. Najzad, formularu se pridružuje i skup *komandi*. Svakoj komandi pridružen je određeni formular. SQL-Update upit može biti pridružen i komandi. Komanda takođe prosleđuje *stvarne parametre* određinom formularu. Ti stvarni parametri mogu imati vrednost lokalnih promenljivih polaznog formulara. Prema tome, ovaj konceptualni model je veoma sličan tradicionalnom proceduralnom modelu, gde formulari odgovaraju procedurama.



Slika 6.11: Elementi tehnike modelovanja. (a) UML dijagram stanja za specifikaciju navigacije kroz aplikaciju. (b) Koncept "proširivača" (engl. *extender*) za implementaciju specifičnih slučajeva upotrebe.

Tehnika modelovanja

Adekvatan UML koncept za specifikaciju opisanih elemenata aplikacije je *dijagram stanja* (engl. *state transition diagram*), pri čemu stanja predstavljaju formulare, a tranzicije predstavljaju komande (slika 6.11a). Na taj način se navigacija kroz aplikaciju jednostavno definiše pomoću standardnih UML koncepata. Sledeći UML stil modelovanja, strukturni slučajevi upotrebe se definišu pomoću UML dijagrama slučajeva upotrebe (engl. *use-case diagram*) [B-Boo99] i dijagrama stanja.

Kao što je navedeno u uvodu, tehnika treba da omogući jednostavan način za implementaciju strukturnih slučajeva upotrebe, ali i da bude otvorena za specifične slučajeve ponašanja. Ovi specifični slučajevi upotrebe treba da se implementiraju na uobičajen način, korišćenjem standardnih tehnika objektnog modelovanja i programiranja. Za tu svrhu je uveden koncept tzv. *proširivača* (engl. *extender*). Proširivač je objekat u ciljnom programskom jeziku (Java u ovom slučaju) koji se može pridružiti svakoj komandi. Ovaj objekat poseduje operacije koje se pozivaju u odgovarajućim koracima ciklusa interpretacije komande (slika 6.11b). Na ovaj način korisnik može da pridruži Java proširivač nekoj komandi i definiše njegove operacije na proizvoljan način. Interpreter će pozivati njegove operacije posle svakog koraka glavnog algoritma interpretacije komande. Ugrađeni proširivač sa svim praznim operacijama pridružuje se komandi podrazumevano.

bilo SQL upitima za strukturne, bilo Java kodom za slučajeve ponašanja. Dijagrami stanja se zatim automatski transformišu u specifikaciju aplikacije koja se interpretira na serveru kao što je opisano. Dijagrami interakcije se pak transformišu u Java kôd koji se inkorporira u operacije proširivača. Na ovaj način se aplikacija modeluje na visokom nivou apstrakcije, korišćenjem standardnih objektno orijentisanih tehnika i jezika UML, a zatim transformiše u izvršni oblik.

Implementacija

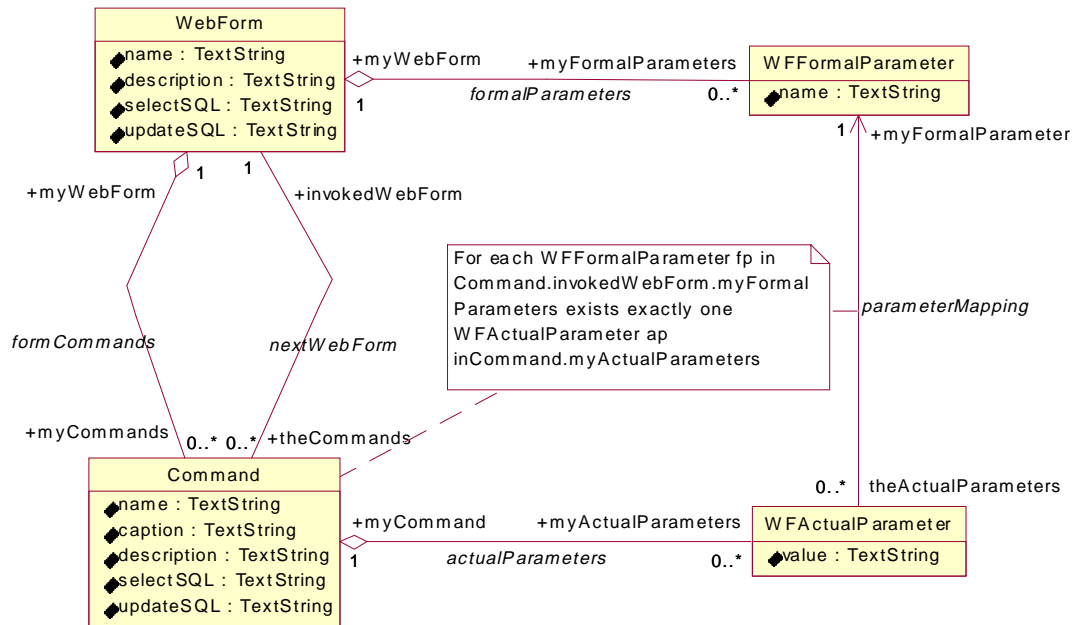
Prema ovim principima realizovano je izvršno okruženje za Web aplikacije razvijene na opisani način. Okruženje se sastoji od interpretera aplikacije i tzv. metabaze podataka. Interpreter je realizovan kao Java program u *servlet* tehnologiji. On prihvata *post* zahteve sa klijenta i izvršava izdate komande prema opisanom algoritmu. Metabaza podataka sadrži specifikaciju aplikacije koju interpretira *servlet*. Ta specifikacija sadrži komande i njima pridružene SQL upite. Na ovaj način se skelet aplikacije jednostavno može modifikovati prostim izmenama sadržaja metabaze podataka, bez potrebe za ponovnim prevođenjem. Ovo čini razvoj aplikacije i ispravljanje grešaka znatno jednostavnijim. Aplikacija je fleksibilna jer se može menjati dinamički, čak i u vreme izvršavanja. Metabaza podataka je realizovana standardnom relacionom bazom, što okruženje čini potpuno prenosivim.

Web stranice se implementiraju kao obične HTML stranice i mogu da se razvijaju pomoću standardnih programa za dizajn HTML stranica. Osim tri posebna elementa, ostali sadržaj stranice može biti proizvoljan, jer ga okruženje ne tretira. Prvi posebni element je komanda koja se izdaje sa stranice. Nju interpreter prepoznaje kao kontrolu sa imenom "Command". Zbog toga su komande koje se nude na stranici implementirane kao grupa radio-dugmadi sa tim imenom. Vrednost izabranog dugmeta prilikom standardne operacije *Submit* definiše komandu koja je izdata. Drugi poseban element predstavlja atribut *value* neke kontrole na stranici. Ako tu kontrolu interpreter treba da popuni vrednošću koja se dobija upitom u bazi podataka, onda njen atribut *value* mora da referiše SQL-Select upit kojim će se ta vrednost dobiti. Ova referenca se jednostavno prikazuje kao niz znakova u obliku "\$<ID SQL-Select upita> \$<Ime polja>". Interpreter će zameniti vrednost atributa *value* svake takve kontrole odgovarajućom vrednošću referisanog polja rezultata zadatog upita, pre nego što datu stranicu pošalje klijentu. Treći poseban element su formalni parametri stranice. Da bi oni bili dostupni SQL-Update upitima komande kada se data stranica napušta, interpreter će njihove vrednosti upisati u skrivene kontrole stranice. Prema tome, svaka stranica mora da sadrži skrivene kontrole za svoje formalne parametre. Svi ovi posebni elementi imaju potpuno standardan HTML format, pa ih standardni programi za dizajn stranica prepoznaju na uobičajeni način.

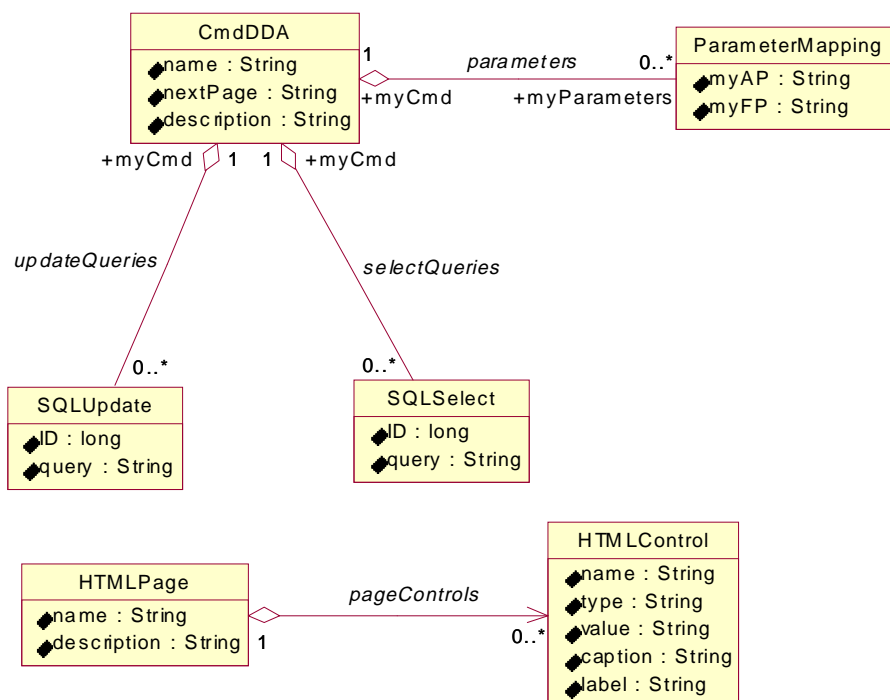
Slično važi i za parametrizovane SQL upite koji su smešteni u metabazu podataka kao obični nizovi znakova. Pre nego što izvrši neki SQL upit, interpreter zamenjuje reference na lokalne promenljive njihovim konkretnim vrednostima. Veza između aplikacije i metabaze podataka je realizovana kroz JDBC standard. Sve ovo čini izvršno okruženje potpuno prenosivim.

Metamodeli i preslikavanja

Opisana tehnika predstavlja izuzetno pogodan primer za primenu predložene metode transformacije modela u cilju brzog razvoja okruženja za apstraktno modelovanje i automatsko generisanje aplikacija. Kao što je opisano, na jednoj strani nalazi se domen modelovanja na visokom nivou apstrakcije, prilagođen načinu razmišljanja projektanta. Taj



(a)

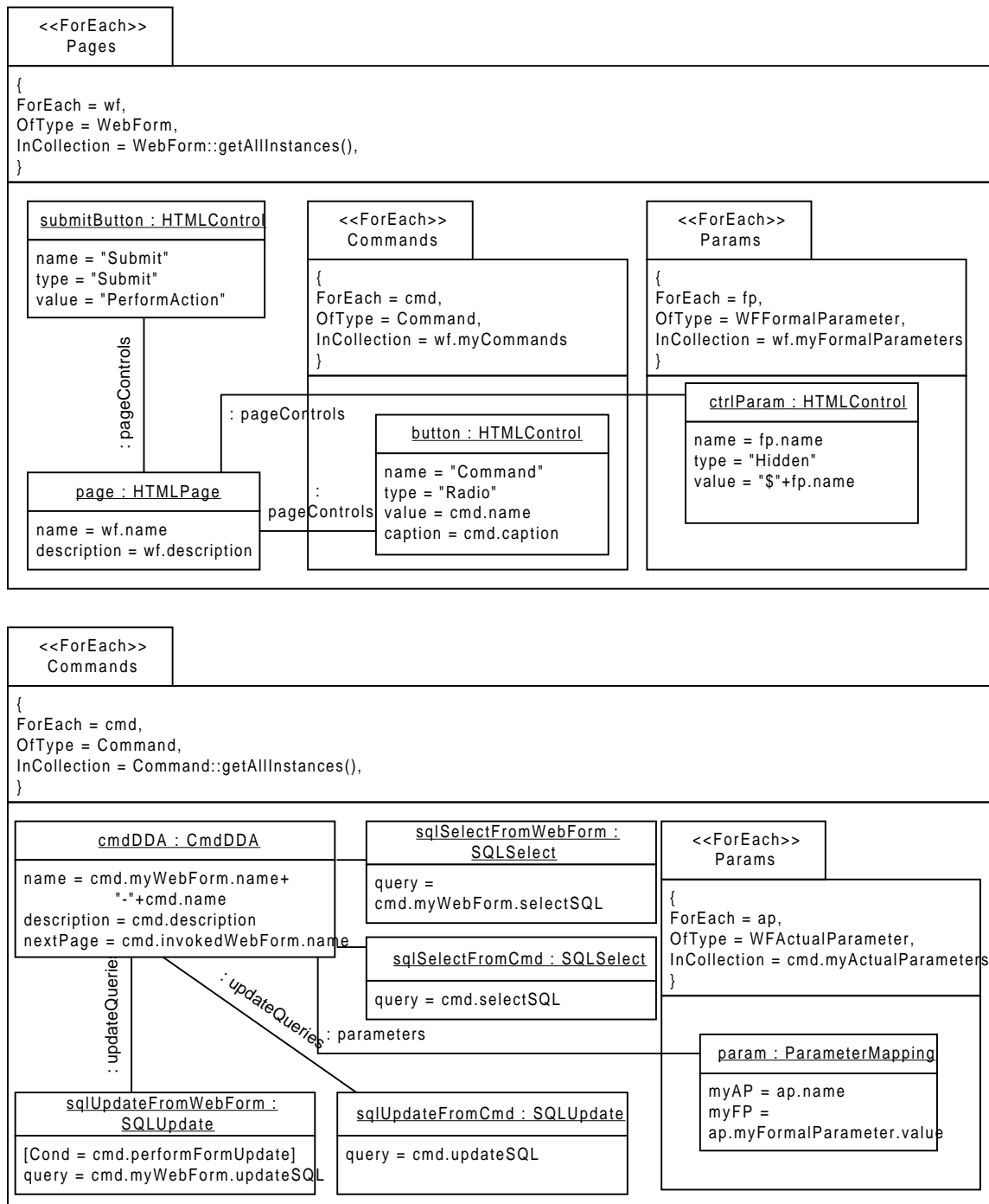


(b)

Slika 6.13: Metamodeli (a) izvorišnog, (b) odredišnog domena za tehniku modelovanja Web aplikacija.

domen sadrži apstrakcije kao što su Web formular, komanda, formalni i stvarni parametar formulara, dok su SQL-Select i SQL-Update upiti pridruženi formularu kao atributi. Metamodel ovog polaznog domena prikazan je na slici 6.13a.

Na drugoj strani nalazi se izvršno okruženje – interpreter i metabaza koji obezbeđuju željeno ponašanje aplikacije. Da bi se aplikacija izvršavala, metabazu treba popuniti



Slika 6.14: Preslikavanje domena za tehniku modelovanja Web aplikacija.

odgovarajućim podacima. Međutim, struktura tih podataka nije u direktnoj vezi sa polaznim domenom. Naime, tu su SQL upiti pridruženi komandama, a ne stranicama, jer interpreter prepoznaje samo komande i njima pridružene upite kao predmet svog delovanja. Pored toga, postojanje tranzicija (komandi) koje polaze iz nekog Web formulara odslikava se kako na postojanje komandi u metabazi, tako i na postojanje radio-dugmadi na odgovarajućoj HTML stranici. Sve ovo zapravo ukazuje na relativnu konceptualnu udaljenost polaznog domena od izvršnog okruženja (tj. strukture podataka u metabazi), što direktno generisanje podataka metabaze iz modela u polaznom domenu čini složenim.

Zbog toga se kao rešenje nameće već predloženi pristup uvođenja objektnog međudomena koji je konceptualno blizak izvršnom modelu, tj. poseduje baš apstrakcije koje

koristi izvršno okruženje (odnosno metabaza). Metamodel tog okruženja prikazan je na slici 6.13b. U njemu je apstrakcija komande nazvana `CmdDDA`, a postoje i koncepti HTML stranice (`HTMLPage`), kontrole na stranici (`HTMLControl`), upita (`SQLSelect` i `SQLUpdate`), kao i vezivanja parametara (`ParameterMapping`). Svi ovi koncepti direktno su podržani odgovarajućom strukturom metabaze ili HTML sintaksom, pa je generisanje podataka metabaze i elemenata HTML stranica iz modela u ovom domenu sasvim jednostavno i direktno.

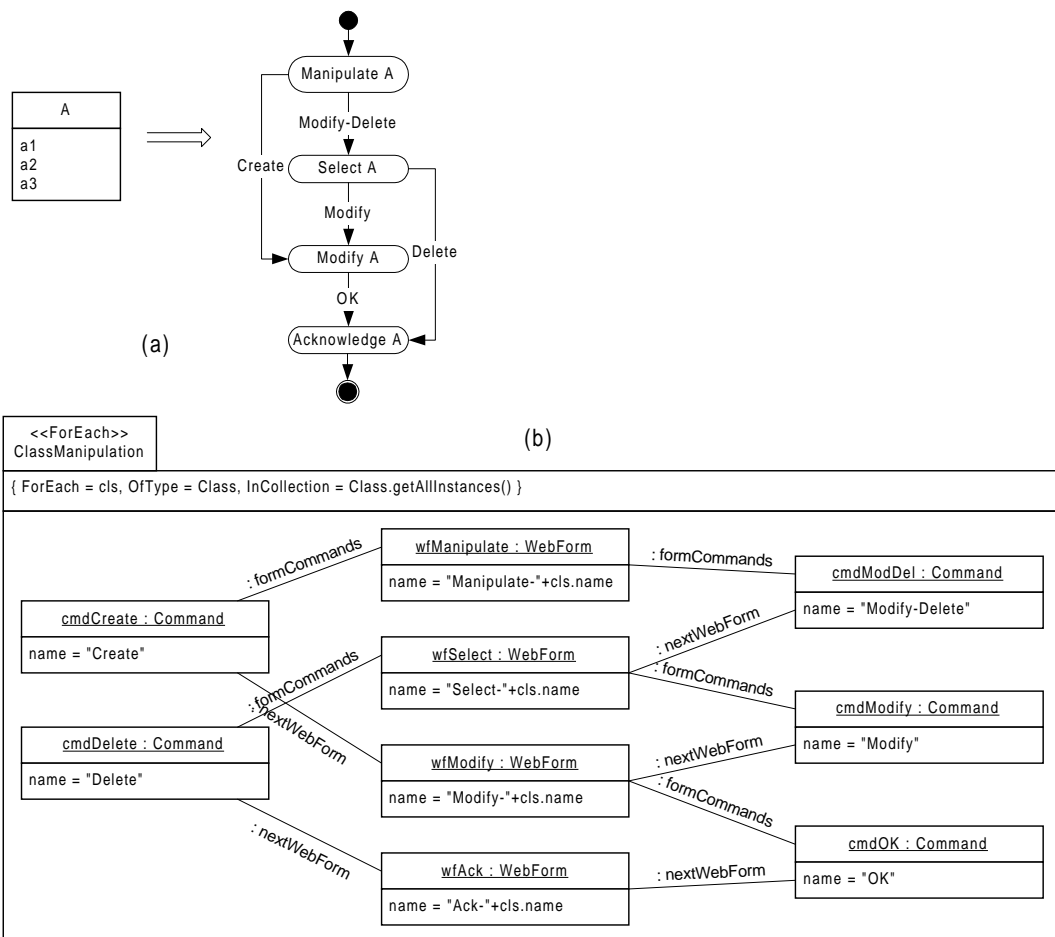
Preslikavanje između ova dva domena relativno je jednostavno (slika 6.14). Gornji paket prikazuje kreiranje HTML stranica i njihovih radio-dugmadi za komande i kontrola za formalne parametre. Donji paket prikazuje kreiranje komandi i njima pridruženih upita.

Sasvim otvoreno rečeno, tokom razvoja opisane metode zapravo je najpre nastao samo ovde opisani odredišni domen sa slike 6.13b, jer je metoda razvijana sa strane izvršnog okruženja. Drugim rečima, prva zamisao je bila upravo koncepcija izvršnog okruženja i domen modelovanja sa slike 6.13b koji mu odgovara. To izvršno okruženje je prvo realizovano i sistem *Socratenon* je u početku razvijan na tom niskom nivou apstrakcije. Tek posle početnih faza razvoja, došlo se do zaključka da se modelovanje može podići na viši nivo apstrakcije, bliži načinu razmišljanja i pogodniji za razvoj, pa je tek tada i tako nastao domen modelovanja prikazan na slici 6.13a. Potom je prirodno nastalo i preslikavanje. Upravo ovaj primer promoviše prirodni postupak razvoja domena "odozdo-nagore" (engl. *bottom-up*), uz pomoć sukcesivnih preslikavanja, koji je predložen u ovom radu. On potvrđuje praktičnu primenljivost tog pristupa koji se sastoji u stalnom podizanju nivoa apstrakcije i jednostavnom dobijanju generatora izvršnih verzija pomoću jednostavnih preslikavanja u već postojeće domene. Naredni odeljak dalje potvrđuje ove zaključke.

Generisanje strukturnih slučajeva upotrebe

Kao što je već rečeno, iskustvo pokazuje da je većina slučajeva upotrebe u tipičnoj aplikaciji orijentisano na strukturu. Ovi slučajevi upotrebe mogu da se realizuju generički, primenom različitih obrazaca scenarija za razne elemente strukture. Kao ilustracija služi primer na slici 6.15a. Za svaku klasu *A* definisanu u modelu može se generisati skup Web formulara koji su zaduženi za kreiranje, modifikaciju ili brisanje instanci klase *A*. Na primer, kreiranje instance klase *A* može da uključuje kreiranje takve instance sa podrazumevanim vrednostima atributa, a zatim modifikaciju te instance korišćenjem formulara za modifikaciju. Modifikacija neke postojeće instance *A* obuhvata najpre selekciju te instance, a zatim modifikaciju njenih atributa korišćenjem istog navedenog formulara za modifikaciju, i slično. Ovakvi obrasci korišćeni su generički na svim mestima u sistemu *Socratenon* gde je to odgovaralo semantici aplikacije.

Prema toj strategiji može se definisati preslikavanje iz domena `UMLCore` u domen Web formulara koji je korišćen do sada kao polazni domen (prikazan na slici 6.13a). To preslikavanje prikazano je uprošćeno na slici 6.15b. Sa druge strane, kao što je već rečeno, iz modela u domenu `UMLCore` može se automatski dobiti šema relacione baze. Prema tome, korišćenjem sukcesivnih transformacija modela može se automatski dobiti kompletna implementacija strukturnih slučajeva upotrebe iz apstraktnog UML klasnog modela aplikacije. Na ovaj način se implementacija dobija postepenim spuštanjem nivoa apstrakcije sve do oblika koji može da se interpretira u izvršnom okruženju.



Slika 6.15: Automatska implementacija strukturnih slučajeva upotrebe. (a) Šematska reprezentacija preslikavanja klase definisane u izvorišnom modelu u obrazac navigacije kroz formulare koji realizuju strukturne operacije sa instancama te klase. (b) Preslikavanje domena.

Naravno, pogrešno je očekivati da navedeni postupak može uvek dovesti do potpune i željene implementacije. Naime, u nekim slučajevima ovakvi generički obrasci realizacije slučajeva upotrebe nisu odgovarajući. Za očekivati je da će korisnik biti zadovoljan većinom takvih generičkih rešenja, ali da će želeti da mali broj njih neznatno izmeni. Za takve slučajeve predložena metoda je još uvek otvorena, jer dozvoljava korisniku da unese proizvoljne specifične delove implementacije u model na bilo kom nivou apstrakcije u sekvenci. Mehanizam pamćenja manualnih modifikacija i njihove restauracije u generisanom modelu podržava ovaj pristup. Ovaj primer dakle jasno ukazuje na potencijale predložene metode i moguće pravce njenog daljeg razvoja i primene.

VII Analiza predloženog rešenja

Zaključci iz primera primene i predlog metodologije

Rezultati primene predložene metode na prikazanim primerima ovde su sumirani, da bi iz njih potom bili izvedeni odgovarajući zaključci. Pored toga, na osnovu tih primera predložena je i metodologija, odnosno postupak sprovođenja prikazanih principa u praksi.

Zaključci iz primera primene

Od početka istraživanja u ovoj oblasti i rada na ovoj tezi, principi modelovanja u specifičnom domenu i predložena metoda primenjeni su na relativno brojnom skupu konkretnih primera i projekata u praksi. Sve ukupno, istraživanja su najpre bila inspirisana, a zatim i potvrđena na slučajevima koji se mogu razvrstati u tri grupe.

Prva grupa obuhvata projekte u ranoj fazi istraživanja, u kojima su primenjeni samo principi modelovanja u specifičnom domenu i metamodelovanja. Na njima je bila potvrđena svrsishodnost ovih principa, ali su uočeni i nedostaci postojećih metoda za metamodelovanje, a naročito za konstrukciju generatora izlaza. Kao rezultat tih iskustava, nastalo je prvo prototipsko okruženje za metamodelovanje [L-Mar99]. Iskustva iz realizacije tog prototipa iskorišćena su za konstrukciju potpuno novog alata opisanog u ovom radu. Ovi projekti bili su ujedno i inspiracija za razvoj predložene metode. Iako ona nije primenjena na ove primere, zbog toga što su oni realizovani pre njenog nastanka, sada je sasvim jasno da bi ona znatno doprinela efikasnosti njihove realizacije. Ti primeri su sledeći:

- Modelovanje distribuiranog, objektno-orijentisanog softvera za rad u realnom vremenu, za softverski sistem telefonske centrale [K-Hir97][K-Mil96]. U realizaciji ovog softvera primenjeni su neki principi iz metode ROOM, vezani za distribuiranost objekata i definisanje ponašanja objekata pomoću konačnih automata. Međutim, ovde je bilo potrebno definisati nekoliko različitih varijanti implementacije tih konačnih automata, u odnosu na konkurentnost, distribuiranost i složenost automata. Kako su te implementacije rađene ručno, uočena je potreba za efikasnom izgradnjom fleksibilnih generatora koda.
- Modelovanje složenog industrijskog sistema za integralnu kontrolu procesa i poslovanja [K-Kru98][K-Mil98a][K-Mil98b]. Za ovaj problem definisana je specifična metoda za kontrolu procesa i protoka materijala i dokumenata u složenom industrijskom okruženju. Takođe je realizovan i prototip izvršnog okruženja koje podržava tu metodu. Međutim, kako je konstrukcija razvojnog okruženja za tu metodu bila složena, a pri tome su potrebe za izmenama i prilagođenjima metode konkretnim specifičnim slučajevima u praksi bile velike i česte, ponovo je uočena potreba za okruženjem za metamodelovanje sa mogućnošću lake realizacije generatora izlaza.

Druga grupa obuhvata primere koji su realizovani primenom predložene metode. U zavisnosti od vremena njihove realizacije, u njima su korišćeni samo neki koncepti te metode. Oni su ukazali na potrebu za konceptima koji su kasnije predloženi (podstrukture, rekurzija, redefinicija podstruktura i polimorfizam). Svi ovi primeri prikazani su u ovom radu:

- Metamodel i generator koda za konačne automate [K-Laz99a][L-Laz99b] koji je korišćen kao demonstrativni primer u ovom radu.
- Metamodel i generator koda za strukturni deo metode ROOM [L-Zar00].
- Metamodel i generator koda za domen logičkog projektovanja hardvera [L-Đor00].

- Metoda za projektovanje Web aplikacija.
- Alat za podršku predloženoj metodi opisan u ovom radu.

U trećoj grupi su primeri upotrebe koji su trenutno u razvoju. Ovi primeri treba da provele upotrebljivost najnaprednijih predloženih koncepata, kao što su podstrukture i polimorfizam, kao i da demonstriraju apstraktan pristup programiranju pomoću sukcesivnih preslikavanja domena. Početni rezultati i ovih primera jesu ohrabrujući, ali do njihove konačne potvrde predstoji ozbiljan rad. To su sledeći primeri:

- Primena predložene metode za, kako se pokazuje, veoma često potrebne realizacije različitih vrsta generisanja jedne strukture objekata od neke druge polazne strukture. Ovakve potrebe uočene su u jednom okruženju za praćenje poslovanja i dokumentacije čiji je prototip razvijen. Prema tome, predložena metoda preslikavanja domena ima značaja ne samo za generisanje izlaznih formi, nego i za specifikaciju i brzu realizaciju automatskog generisanja željenih složenih struktura objekata iz nekih drugih polaznih struktura. Ovaj primer treba da potvrdi upotrebljivost koncepata podstrukture, rekurzije i polimorfizma.
- Drugi primer predstavlja pokušaj uopštenja postupka generisanja strukturnih slučajeva upotrebe. Ideja je da se definiše jedan apstraktni domen u kome bi se na implementaciono nezavisan način definisale realizacije strukturnih slučajeva upotrebe. Te specifikacije (zapravo modeli u odgovarajućem domenu) bi se dalje transformisale u modele u postojećim domenima zavisnim od konkretne implementacije, kao što je opisani domen Web aplikacija. Iznad ovog domena mogli bi se konstruisati još apstraktniji domen u kojima postoje koncepti koji definišu složene strukturne relacije između objekata, i za koje bi se, preslikavanjem u domen slučajeva upotrebe, opisivala njihova semantika. Ukoliko ovaj pokušaj bude uspešan, on će zapravo predstavljati najozbiljniju potvrdu ideje o apstraktnom pristupu programiranju pomoću definisanja specifičnih, visoko-apstraktnih domena, čija se implementacija realizuje preslikavanjem u već postojeće.

Do sada prikupljeni rezultati primene, prvenstveno na primerima iz druge grupe, potvrđuju upotrebljivost i efikasnost metode. Jedan deo primera iz druge grupe realizovao je autor lično, dok su veći deo tih primera realizovali drugi ljudi (studenti-diplomci). Ovo je bilo naročito značajno, jer se, zapravo, upotrebljivost neke metode ne može potvrditi samo tvrdnjama autora da ona odgovara njemu lično, već i lakoćom prihvatanja od strane drugih. U ovom slučaju rezultati su više nego ohrabrujući.

Naime, u svim primerima koje su realizovali studenti, situacija je bila veoma slična. Najveći deo vremena i truda potrošen je na samo navikavanje na razmišljanje u kontekstu modelovanja u specifičnim domenima i metamodelovanja. Taj veoma apstraktni način razmišljanja predstavlja ujedno i najveću teškoću. Taj aspekt, naravno, nema nikakve veze sa predloženom metodom, već prosto sa prirodom problematike. Sa druge strane, kada se osoba jednom navikne na taj način razmišljanja, onda posle više nisu potrebna ulaganja napora, jer on ostaje trajno i veoma moćno sredstvo za rešavanje problema. Posle prvog privikavanja, rešavanje narednih problema postaje mnogo efikasnije. Potvrda ove tvrdnje je i to što su studenti koji su imali kao uzore radove svojih prethodnika ovaj deo zadatka obavljali znatno brže. Okvirno, vreme potrebno za ovaj deo zadatka kretalo se od nekoliko meseci u prvim primerima, do reda nekoliko nedelja u narednim.

Drugi deo zadatka je uvek bio definisanje metamodela specifičnog domena. On je po svojoj prirodi i drugi po težini i dužini vremena potrebnom za realizaciju. Pokazalo se da to vreme u mnogome zavisi od postojanja sličnih uzora. Na primer, u slučaju konačnih automata (gde je osnova metamodela preuzeta iz definicije jezika UML), kao i u slučaju logičkog projektovanja hardvera (koji je relativno jednostavan i gde je uzor bio realizovani ROOM metamodel koji ima mnogo sličnosti), vreme potrebno za metamodelovanje bilo je kratko (reda nekoliko dana), za razliku od slučaja metode ROOM gde je veoma složeni metamodel

pravljen samo na osnovu opisa te metode, dakle potpuno "od nule". U tom slučaju vreme za metamodelovanje bilo je znatno duže (reda više nedelja).

Najzad, poslednji zadatak bila je specifikacija preslikavanja za generatore koda. Ovaj deo, kako se pokazalo, bio je i najlakši i najkraći u svim slučajevima. On nije trajao nikada više od nekoliko dana, čak i za najsloženije slučajeve.

Kao zaključak, eksperimenti primene predložene metode koje su uradili studenti-diplomci imali su za cilj da ispituju i potvrde sledeće aspekte:

(a) **Primenljivost.** Primeri su bili izabrani tako da pokriju različite aspekte i domene: strukturno modelovanje (strukturni ROOM), modelovanje ponašanja (konačni automati) i kombinovan pristup (logičko projektovanje), kao i softverske (ROOM i konačni automati) i nesoftverske (logičko projektovanje) domene. Ovo je potvrdilo široku primenljivost predložene metode.

(b) **Prihvatljivost i složenost učenja.** Eksperimente su izvela tri studenta bez profesionalnog iskustva. Svi su veoma lako prihvatili predloženu metodu, bez ikakve pomoći autora, samo čitajući tehničke izveštaje i proučavajući raspoložive gotove primere. Svi su izjavili kako im je bilo potrebno veoma kratko vreme za prihvatanje i učenje metode, otprilike taman toliko koliko je bilo potrebno da se pročita raspoloživi materijal.

(c) **Vreme proizvodnje.** Da bi se izbegla mogućnost subjektivnosti autora u proceni, kao i uticaj iskustva i skraćanja vremena prilikom izrade svakog sledećeg projekta (tzv. "kriva učenja"), ove eksperimente vršila su tri različita studenta bez ikakve pomoći autora. Vreme koje je bilo potrebno za specifikaciju generatora izlaza bilo je kratko u sva tri slučaja (reda nekoliko dana). To vreme zavisilo je prvenstveno od kompleksnosti samog domena modelovanja i njegovog metamodela. Osim toga, za prvi sprovedeni eksperiment (konačni automati), vreme proizvodnje primenom predložene metode upoređeno je sa vremenom proizvodnje primenom tradicionalnog programskog pristupa. Studentu je bilo potrebno oko deset dana da implementira generator izlaza direktno na jeziku C++, a samo tri dana pomoću predložene metode.

(d) **Održavanje.** U sva tri projekta, studenti su imali zadatak da naprave dve ili više različitih verzija generatora izlaza, menjajući početnu specifikaciju preslikavanja. Svi su izvestili da nisu imali nikakvih problema u ispunjavanju tog zadatka, jer su im specifikacije bile jasne, pregledne, koncizne i lake za modifikaciju i proširivanje. Ovo je posledica mogućnosti jezika UML da prikaze elemenata modela grupiše u dijagrame, a što predložena metoda podržava u potpunosti. Dijagrami mogu da prikažu samo pojedine delove celokupnog preslikavanja, tj. samo one delove koji su relevantni za konkretni kontekst dijagrama. Konzistentnost preslikavanja proverava alat, iako se isti element može prikazati različito na više dijagrama. Zbog toga se dijagrami mogu organizovati tako da odslikavaju koherentne, tesno povezane delove odredišnog modela. Kao posledica, potrebna modifikacija u odredišnom modelu lako se primenjuje jer obično utiče na samo mali broj dijagrama. Ova prednost predložene metode u odnosu na tradicionalni programski pristup ista je kao i prednost vizuelnog modelovanja softvera (na jeziku UML npr.) u odnosu na programiranje na tekstualnim programskim jezicima, kada je održavanje u pitanju.

(e) **Mogućnost ponovne upotrebe** (engl. *reusability*). Domen OOPL, njegov metamodel, kao i generator C++ koda bio je upotrebljen u sva tri eksperimenta. Ovo je skratilo vreme razvoja bar dvostruko. Da je bio korišćen tradicionalni programski pristup, studenti bi morali da prave zasebne generatore koda, potpuno nezavisne i specifične za njihove zadatke, kao i da se bave svim pojedinostima C++ sintakse i semantike. Primenom predložene metode, studenti nisu morali da se bave pojedinostima bilo kog programskog jezika, pa većina specifikacija, izuzev definicija tela operacija, može da se iskoristi i za Java kôd.

(f) Proces. Ovi eksperimenti su doprineli razvoju predloga procesa razvoja okruženja za modelovanje u specifičnom domenu i generatora izlaza. Ovaj predloženi proces opisan je detaljnije u narednom odeljku.

Primeri su ukazali i na određene slabosti predložene metode. Naime, koncepti `ForEach` paketa, uslovnog kreiranja, sekvencijalnih zavisnosti, podstruktura i rekurzije pokrivaju sve fundamentalne strukture kontrole toka. Zbog toga bi se, u principu, predložena tehnika mogla koristiti i za definisanje generatora sekvencijalnih (tekstualnih) izlaza. Međutim, iskustvo pokazuje da se generatori sekvencijalnih struktura znatno lakše konstruišu primenom tradicionalnih programskih ili šablonskih tehnika. Ovo je posebno slučaj kada se definišu tela operacija u ciljnom programskom jeziku. Osim toga, domen OOPL nije dovoljno upotrebljiv za različite ciljne jezike upravo kada su u pitanju tela operacija, jer je njihov izgled prilično zavisn od jezika. Prema tome, predložena tehnika je veoma pogodna za premošćavanje konceptualne udaljenosti između domena, tj. za preslikavanje objektnih domena. Transformacije domena u sekvencijalne forme koje se mogu izvršiti direktno i lako, najbolje se definišu korišćenjem tradicionalnih pristupa.

Osim toga, ideja da se nizom automatskih transformacija od apstraktnog modela može doći do konačne implementacije nosi probleme. Naime, svaki model izostavlja bar neke detalje koji su ipak neophodni u konačnoj implementaciji. Na primer, jezik UML još uvek nema dovoljnu rezoluciju da definiše finije detalje programskih postupaka. Da bi generator proizveo konačnu implementaciju, ti detalji moraju negde biti definisani. Ili će ih generator sam sintetisati, ili će ti detalji morati da se dodaju ručno uz odgovarajući model. Automatsko generisanje detalja, u nekim slučajevima (npr. sistemi za rad u realnom vremenu), nije praktično, pošto dovodi do programa koji su potpuno neprihvatljivi u pogledu performanse ili veličine.

Međutim, nedavni predlog definicije semantike akcija na jeziku UML [B-OMG00] predstavlja predlog rešenja upravo ovih problema. Opredeljenje da se i ponašanje sistema modeluje, do najsitnijih detalja potrebnih da se precizno definiše semantika izvršavanja, korišćenjem metamodela izraženog na jeziku UML, predstavlja odličnu potvrdu ispravnosti celokupne opredeljenosti iskazane u ovom radu. Ovaj kontekst, u kome će se u budućnosti programirati na visokom nivou apstrakcije, pravljenjem modela koji imaju objektnu metamodelu, upravo potvrđuje upotrebnost vrednost predložene metode, jer je ona i namenjena za generisanje modela u drugim domenima iz takvih modela.

Problem efikasnosti automatski generisanog izlaza, sa druge strane, postoji samo u određenim domenima (sistemi za rad u realnom vremenu), dok u drugim domenima ima manji značaj zbog sve intenzivnijeg razvoja hardvera. Ovaj problem sigurno treba rešavati, ali je utisak da je on sličan problemu optimizacije mašinskog koda koji generišu prevodioci za više programske jezike. Kada su ovi jezici nastajali, takođe je postojao problem optimizacije generisanog koda. Kada je teorija programskih prevodilaca sazrela uz odgovarajuće formalne postavke, pokrenuto je i značajno istraživanje u domenu optimizacija koda koje je dalo izuzetno dobre rezultate, tako da se danas malo razmišlja o efikasnosti generisanog mašinskog koda. Predložena metoda može da predstavlja prilog istom pristupu kada su u pitanju visoko apstraktni modeli sa objektnim metamodelima. Može se očekivati da istraživanje u oblasti optimizacije automatski generisanog koda iz ovakvih modela bude znatno lakše ukoliko se principi transformacije postave formalno.

Još jedan izuzetno značajan praktičan problem jeste pitanje inverzne transformacije. Naime, definisane modele potrebno je proveravati na greške, što se najbolje radi simulacijom njihovog izvršavanja, odnosno debugovanjem (engl. *debugging*), ali na samom izvornom nivou. Kada se radi ovakva simulacija, nije praktično raditi sa automatski generisanim kodom, pogotovu ako je on rezultat čitavog niza automatskih transformacija. Problem prenosa informacija iz izvršnog modela u početni, visoko apstraktni model tokom njegove simulacije,

u cilju debugovanja, analogan je problemu debugovanja programa na višem tekstualnom programskom jeziku, jer se u toku debugovanja zapravo izvršava automatski generisani mašinski program, dok korisnik prati simulaciju na visokom nivou. I ovaj problem predstavlja značajan prostor za buduća istraživanja.

Predlog metodologije

Tokom rada na primerima upotrebe uočavane su i neke pravilnosti u postupcima primene predložene metode koje su ovde sumirane. U većini slučajeva problem se svodio na definiciju metamodela specifičnog domena, a zatim i na specifikaciju generatora izlaza. Pri tome je sproveden sledeći postupak koji se i preporučuje kao opšti proces primene.

Najpre treba formirati metamodel domena. U tom postupku primenjuju se sva pravila objektno orijentisanog modelovanja i heuristike koje se u oblasti modelovanja preporučuju. Pored toga, pokazalo se i da iskustvo i posedovanje sličnih uzora može znatno da pomogne.

Pri rešavanju zadatka specifikovanja generatora izlaza korišćeni su, i zato se preporučuju, sledeći postupci. Najpre je potrebno odrediti ciljnu izlaznu formu. Ukoliko je to izvorni kôd na nekom objektno orijentisanom programskom jeziku, onda se za specifikaciju generatora koristi preslikavanje u domen OOPL. U tom slučaju, najpre se definišu eventualne varijante željenog oblika generisanog koda. Te varijante predstavljaju se odgovarajućim karakterističnim primerima. Primeri treba da budu dovoljno jednostavni kako bi bili lako razumljivi, a sa druge strane i dovoljno sadržajni, kako bi ukazali na sve elemente transformacije. Primeri se sastoje iz uglednog modela u izvorišnom domenu, kao i od uglednog izlaza (koda) koji se želi dobiti za taj model. Posle toga definiše se preslikavanje. Postupak definisanja preslikavanja sprovodi se tako što se redom prolazi kroz elemente uglednog ciljnog koda i definišu elementi preslikavanja koji treba da budu odgovorni za njihovo generisanje. Pri formiranju dijagrama, poštuju se sva poznata pravila vizuelnog modelovanja koja doprinose njihovoj preglednosti i dobroj organizaciji [B-Boo99].

Upravo u ovom poslednjem postupku pokazuje se pogodnost predložene metode. Naime, veoma često se jedan element izvorišnog modela manifestuje kroz mnogo elemenata potpuno rasutih po izlaznoj formi. Zbog toga postupak prolaza kroz izlaznu formu i definisanja elemenata u preslikavanju, a naročito preglednost i dobra organizacija dijagrama preslikavanja, znatno pomažu. Zbog tih svojstava se zapravo rasuti elementi u izlaznoj formi često lepo grupišu u koherentne dijagrame preslikavanja, i to postupno i kontrolisano. Ovaj efekat uočljiv je i na primerima prikazanim u prethodnoj glavi i u prilogima.

U slučajevima kada ciljna izlazna forma nije izvorni kôd, već neka druga struktura, posmatra se način na koji se ta struktura može dobiti iz izvorišnog modela. Ponovo je pogodno formirati uzorni primer. Ukoliko je direktna transformacija komplikovana, treba pribeći uvođenju pogodnog međudomena. Taj međudomen treba da bude formiran tako što je po svojoj koncepciji blizak odredišnoj izlaznoj formi. Ukoliko je i tada preslikavanje iz izvorišnog domena u ovaj međudomen složeno, mogu se uvesti i drugi međudomeni. Za definisanje preslikavanja ponovo se koristi ugledni primer na opisani način.

Najzad, poslednji mogući pristup je definisanje apstraktnih domena iznad već postojećih. Naime, često se može desiti da postojanje jednog domena ukazuje na neko uopštenje, odnosno na neki apstraktniji domen koji se onda može preslikati u postojeći. Taj pristup iskorišćen je u opisanom primeru projektovanja Web aplikacija, kao i u primeru apstraktnog domena slučajeva upotrebe koji se trenutno razvija. Za korišćenje ovog postupka neophodna su dodatna iskustva.

Odnos predloženog i postojećih rešenja

U cilju analize odnosa predložene metode preslikavanja domena i postojećih tehnika transformacija modela analiziranih u glavi "Pregled postojećih rešenja", predložena metoda svrstana je u odgovarajuće kategorije prema klasifikacijama datim u toj glavi.

Tabela 7.1 prikazuje prvu klasifikaciju prema tipovima izvorišnih i odredišnih struktura. U pregledu postojećih rešenja naglašeno je da, osim ad-hoc metoda, ne postoje metode koje obezbeđuju transformacije u strukturu grafa. Kako se predložena metoda preslikavanja domena inherentno bazira na strukturi tipiziranog grafa, a kako je i stablo specijalna vrsta grafa, ova metoda pokriva sve nepokrivene i slabo pokrivene oblasti, kao što je prikazano u tabeli (osenčena polja). Te oblasti se odnose na strukture stabla i grafa i to i za izvorišni i za odredišni model.

Tabela 7.2 prikazuje klasifikaciju prema načinu obilaska izvorišne strukture i njenoj specifikaciji. Predložena metoda preslikavanja domena eksplicitno definiše način obilaska pomoću izraza u `ForEach` paketima (programski), odnosno u podstrukturama pridruženim tipu iz izvorišnog domena (simbolički). Metoda takođe, za razliku od mnogih drugih, podržava i rekursiju i polimorfizam u obilasku izvorišne strukture, i to preko koncepata podstrukture i njihovih rekurzivnih poziva, odnosno podstrukture pridruženih tipu iz izvorišnog domena i polimorfizma prema tom tipu. Prema tome, specifikacije podstrukture slede glavne objektno orijentisane principe.

Tabela 7.3 prikazuje klasifikaciju prema načinu specifikacije transformacije. Predložena metoda preslikavanja domena ponovo poseduje sve željene osobine. Ona podržava rekursiju, pomoću rekurzivnih poziva podstrukture, kao i polimorfizam, pomoću redefinicije preslikavanja. Dalje, specifikacije su vizuelne, pomoću dijagrama, dok je organizacija tih specifikacija objektna: elementi specifikacija su hijerarhijski organizovani u pakete koji sadrže dijagrame, dok je njihova implementacija objektna kao i kod obrasca *Visitor*.

Odredišni model				
Izvorišni model		Sekvenca	Stablo	Graf
	Sekvenca	(Posredno: sekvenca-stablo-sekvenca)	<ul style="list-style-type: none"> • Tradicionalni prevodioci (parsiranje) 	-
	Stablo	<ul style="list-style-type: none"> • Programski (skriptovanje) • Tekstualni šabloni • Apstraktne specifikacije • Tradicionalni prevodioci (generisanje koda) 	<ul style="list-style-type: none"> • Transformatori bazirani na gramatikama • Intencionalno programiranje • DM 	<ul style="list-style-type: none"> • DM
	Graf	<ul style="list-style-type: none"> • Programski (skriptovanje) • Tekstualni šabloni • Apstraktne specifikacije 	<ul style="list-style-type: none"> • Prevodioci za vizuelne jezike • DM 	<ul style="list-style-type: none"> • DM

Tabela 7.1: Klasifikacija postojećih tehnika transformacija prema tipu strukture izvorišnog i odredišnog modela. Osenčeni deo predstavlja tipove koje pokriva predložena metoda preslikavanja domena (označena sa DM, od engl. *Domain Mapping*).

Metoda	EksPLICIT- nost	Rekur- zija	Polimor- fizam	Način specifi- kacije	Organizacija i dekompozicija
Programski (skriptovanje)	EksPLICITNO	Zavisno od jezika, uglavnom da	Zavisno od jezika, uglavnom ne	Programski	Zavisno od jezika, uglavnom proceduralna
Tekstualni šabloni	EksPLICITNO	Ne	Ne	Tekstualni	Nema direktne podrške
Obrazac <i>Visitor</i>	EksPLICITNO	Da	Da	Programski	Objektna
Apstraktne specifikacije	EksPLICITNO	Da	Da	Apstraktne specifi- kacije	Objektna
Adaptivno programiranje	EksPLICITNO	Da	Da	Vizuelno	Objektna
Tradicionalni prevodioci	IMPLICITNO	Da	Ne	Preko atributa	Nema direktne podrške
Transformatori bazirani na gramatikama (generisanje koda)	IMPLICITNO	Da	Ne	Ugrađeno u algoritam transfor- macije	Nema direktne podrške
Intencionalno programiranje	EksPLICITNO	Da	Ne	Programski unutar operacija transfor- macija	Nema direktne podrške
Prevodioci za vizuelne jezike (parsiranje)	EksPLICITNO	Da	Ne	Pozicionom gramatikom	Nema direktne podrške
Preslikavanje domena	EksPLICITNO	Da	Da	Programski i simbolički	Objektna

Tabela 7.2: Pregled karakteristika načina obilazaka strukture izvorišnog modela i njihovih specifikacija za analizirane postojeće i predloženu metodu.

Zbog svega ovoga, predložena metoda je pogodna za transformacije objektnih (grafovskih) struktura u iste takve strukture, uz punu podršku svih naprednih koncepata (polimorfizma i rekurzije), kao i dobru objektnu organizaciju vizuelnih specifikacija. Pored toga, ona je pogodna za spuštanje nivoa apstrakcije od domena koji su prilagođeni prirodi problema do nižih, implementacionih domena, i to postupno, uz sukcesivne transformacije modela i premošćavanje konceptualno dalekih domena. Sa druge strane, za domene za koje se želi preslikavanje u sekvencijalne (tekstualne) forme i koji se mogu lako (jedan-na-jedan) preslikati u takve forme, pogodniji su drugi postojeći pristupi. Slično važi i za preslikavanje u obrnutom smeru, iz sekvencijalnih u grafovske strukture, za koje su najpogodniji tradicionalni prevodioci.

Metoda	Rekurzija	Polimorfizam	Način specifikacije	Organizacija i dekompozicija
Programski (skriptovanje)	Zavisi od jezika, uglavnom da	Zavisi od jezika, uglavnom ne	Programski	Zavisno od jezika, uglavnom proceduralna
Tekstualni šabloni	Ne	Ne	Tekstualni	Podela u datoteke
Obrazac <i>Visitor</i>	Da	Da	Programski	Objektna
Apstraktne specifikacije	Da	Da	Programski	Objektna
Tradicionalni prevodioci (generisanje koda)	Da	Ne	Programski	Nema direktne podrške
Transformatori bazirani na gramatikama	Da	Ne	Pomoću podržanih operacija	Nema direktne podrške
Intencionalno programiranje	Da	Ne	Pomoću operacija transformacija	Nema direktne podrške
Prevodioci za vizuelne jezike (parsiranje)	Da	Ne	Pozicionom gramatikom	Nema direktne podrške
Preslikavanje domena	Da	Da	Vizuelni	Objektna

Tabela 7.3: Pregled karakteristika načina specifikacije transformacija za analizirane postojeće i predloženu metodu.

VIII Zaključak

Osnovni doprinosi rada

U ovom radu izučavani su neki osnovni aspekti modelovanja u specifičnom domenu i metamodelovanja, a naročito problem generisanja izlaznih formi. Ovaj problem je najpre precizno definisan i uopšten na problem transformacije modela u okruženjima za modelovanje. Date su definicije najvažnijih pojmova koji se koriste u ovoj oblasti. Uvedena su i neka konceptualna razgraničenja koja značajno doprinose boljem uočavanju i razumevanju problema u ovoj oblasti.

Potom je izvršena analiza i klasifikacija postojećih rešenja, kako onih direktno korišćenih u okruženjima za modelovanje i metamodelovanje, tako i onih koji su u indirektnoj vezi sa problemom transformacije modela. Postojeća rešenja su ukratko opisana, a zatim je predložena njihova klasifikacija na osnovu nekoliko važnih parametara procesa transformacija modela: tipova strukture modela, načina specifikacije obilaska izvorišnog modela i načina specifikacije transformacija.

U radu je zatim predložena originalna tehnika specifikacije transformacije modela koji se zasnivaju na objektnim metamodelima. Tehnika koristi proširene UML objekte dijagrame koji prikazuju instance apstrakcija iz ciljnog domena i veze između njih koje treba automatski kreirati. Standardni UML objektni dijagrami prošireni su konceptima uslovnog, repetitivnog, parametrizovanog, polimornog i rekurzivnog kreiranja.

Pored toga, u radu je predložena generalizacija celog pristupa prema kojoj se transformacije modela mogu vršiti lančano, uz korišćenje različitih raspoloživih gotovih i ponovno upotrebljivih metamodela domena i njihovih preslikavanja. Ovo zapravo predstavlja jedan novi pristup visoko apstraktnom programiranju uz automatsko generisanje aplikacija. Visoka apstraktnost se ogleda u tome što se kao polazni modeli koje definiše korisnik mogu koristiti modeli iz specifičnih domena čiji su metamodeli prilagođeni datoj oblasti primene. Konkretno implementacije dobijaju se izborom željenih transformacija, u zavisnosti od ciljnog okruženja za izvršavanje aplikacije. Kao podrška ponovnoj upotrebljivosti domena i preslikavanja, predloženi su koncepti ugrađivanja metamodela kao i nasleđivanja i polimorfizma preslikavanja.

U radu su dalje date formalne definicije semantike svih predloženih koncepata, kao i opis jedne prototipske implementacije alata koji podržava predloženu metodu. Alat je realizovan korišćenjem jezika UML i C++, kao i alata Rational Rose i Microsoft Visual C++.

Potom je opisan niz konkretnih primera iz prakse na kojima je predložena metoda primenjena. Ukratko su prikazani metamodeli domena od interesa kao i najvažnija preslikavanja. Rezultati ovih primena potvrdili su osnovanost svih pretpostavki koje su postavljene. Prvo, principi modelovanja u specifičnom domenu i metamodelovanja, kao i automatske transformacije modela sigurno imaju značajan prostor primene u praksi. Jedan od doprinosa ovog rada je verovatno i u isticanju tog značaja i pojašnjavanju osnovnih koncepata i problema koji se tu sreću. Drugo, predložena metoda preslikavanja domena je jednostavna za razumevanje i primenu, specifikacije su pregledne i jednostavne za održavanje i modifikaciju, a implementacija transformatora dobija se brzo i automatski. Predložen je i postupak primene ove metode u praksi, uz nekoliko heuristika koje pomažu u projektovanju. Jedna od najbitnijih je uvođenje međudomena u cilju pojednostavljivanja transformacija između konceptualno udaljenih domena. Najzad, predloženi koncepti sukcesivne transformacije modela predstavljaju potencijalno veoma interesantan pristup apstraktnom programiranju čija je upotrebnost preliminarno potvrđena u ovom radu.

Potencijali korišćenja rezultata rada

Rezultati ovog rada mogu se koristiti u različitim kontekstima. Prvo, opšta razmatranja o principima metamodelovanja i transformacije modela mogu biti korišćena u rešavanju raznih problema modelovanja u specifičnom domenu.

Drugo, predložena metoda preslikavanja domena može se koristiti u različitim okruženjima:

- Prilikom projektovanja specijalizovanih alata za modelovanje, tačnije, prilikom specifikacije generatora izlaza i uopšte transformatora modela. Metoda tu može biti od velike pomoći konstruktorima takvih alata bar za specifikaciju i dokumentaciju, ako ne i za automatsku implementaciju generatora.
- U podesivim alatima za modelovanje i alatima za metamodelovanje, gde korisnik može da definiše varijante generisanja izlaza. Primeri su pokazali da je predložena metoda laka za razumevanje i primenu, pa se može očekivati da je u prednosti nad nekim drugim, do sada zastupljenim pristupima ovom problemu.
- U realizaciji aplikacija različitih namena, jer se u praksi često sreće problem generisanja jedne strukture obilaskom neke druge, izvorišne strukture. Rezultati ovog rada (to je skorija praksa već potvrdila) mogu značajno pomoći u uočavanju, klasifikaciji i rešavanju tih problema. Naime, data analiza i klasifikacija postojećih metoda transformacije modela može da pomogne u izboru strategije implementacije transformacije. Ukoliko je problem takav da je predložena metoda najpogodnija, a u radu je ukazano i na kriterijume za tu procenu, ona se može primeniti za specifikaciju i automatsku implementaciju tih transformacija.

Treće, metamodeli domena korišćeni u ovom radu, kao i njihova preslikavanja, sigurno se mogu koristiti i u razne druge svrhe pri rešavanju problema ove vrste.

Konačno, metoda sukcesivnih transformacija otvara velike mogućnosti za definisanje novih, apstraktnih domena za koje se implementacije mogu brzo dobiti korišćenjem već postojećih domena i preslikavanja. Ovo je možda i najveći potencijal predložene metode koji tek treba da bude ispitan.

Pravci daljeg razvoja

Bez obzira na sve rezultate i doprinose, ovaj rad ostavlja mnogo prostora za dalja istraživanja i razvoj. Najpre je neophodno u potpunosti implementirati alat koji podržava predloženu metodu. Iako je implementacija opisanog prototipa bila važna jer je dokazala da se predloženi koncepti mogu realizovati, on još uvek nije sasvim pogodan za praktičnu upotrebu. Njega najpre treba proširiti tako da podrži sve predložene koncepte. Osim toga, treba ga obogatiti vizuelnim grafičkim editorom bez koga je ozbiljniji praktični rad teško izvodljiv.

Dalje, predstoji rad na realizaciji metamodela i preslikavanja drugih raznih specifičnih domena. Taj rad će potencijalno ukazati na neke nedostatke i moguća unapređenja predložene metode.

Možda najambiciozniji budući poduhvat predstavlja ispitivanje primenljivosti ideje o apstraktnom programiranju u specifičnim domenima, uz primenu sukcesivnih preslikavanja. Ovaj poduhvat uključuje uočavanje, definiciju i realizaciju većeg broja često upotrebljivanih domena i preslikavanja (od kojih su neki pomenuti u radu, kao što je OOPL, UML, relacioni model ili Web aplikacije), kao i apstraktnijih, specifičnih domena koji se za sada još ne naziru. Jedan od naročito interesantnih pravaca je već opisani domen uopštenog definisanja realizacije slučajeva upotrebe. Potrebno je, dakle, oformiti dovoljno obiman repozitorijum tih metamodela i preslikavanja, a potom i izvršiti mnogo eksperimenata u okviru praktičnih projekata. Takav repozitorijum trebalo bi da predstavlja pogodan izvor gotovih rešenja za najčešće sretane tipove problema u praksi.

Konačno, rešavanje navedenih problema optimizacije automatski generisanih modela, naročito posle više lančanih transformacija, kao i simulacije, odnosno debugovanja visoko apstraktnih modela, predstavlja zanimljiv predmet daljih istraživanja.

Tek nakon toga moguće je utvrditi da li predloženi pristup predstavlja krupniji korak u razvoju tehnologije proizvodnje softvera.

IX Literatura

Spisak referenci

- [Aho86] Aho, A., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986
- [Ait97] Aitken, W., Dickens, B., Kwiatkowski, P., De Moor, O., Richter, D., Simonyi, C., "Transformation in Intentional Programming," <http://research.microsoft.com/ip>
- [AllIW] Allaire Inc., *Cold Fusion Web Construction Kit*, <http://www.allaire.com/products/ColdFusion/>
- [Anl98] Anlauff, M., Kutter, P. W., Pierantonio, A., "Montages/Gem-Mex: A Meta Visual Programming Generator," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998
- [Art95] Artsy, S., "Meta-modeling the OO Methods, Tools, and Interoperability Facilities," *OOPSLA'95 Workshop in Metamodeling in OO*, Oct. 1995
- [AST95] Advanced Software Technologies, Inc., *Graphical Designer Language*, 1995, <http://www.advancedsw.com>
- [ASTW] Advanced Software Technologies, Inc., *Graphical Designer*, <http://www.advancedsw.com>
- [Bel95] Bell, R., Sharon, D., "Tools to Engineer New Technologies into Applications," *IEEE Software*, Mar. 1995, pp. 11-16
- [Boo94] Booch, G., *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, Redwood City, Calif., 1994
- [Boo99] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley Longman, Reading, Mass., 1999
- [BorW] Borland, *Delphi Client/Server Suite*, <http://www.borland.com/delphi/>
- [Bur95] Burnett, M., McIntyre, D., "Visual Programming," *IEEE Computer*, Mar. 1995, Special issue on Visual Programming, pp. 14-16
- [Cha95] Chang, S.-K., Costagliola, G., Pacini, G., Tucci, M., Tortora, G., Yu, B., Yu, J.-S., "Visual-Language System for User Interfaces," *IEEE Software*, Mar. 1995, pp. 33-44
- [Con99] Conallen, J., "Modeling Web Application Architectures with UML," *Communications of the ACM*, Vol. 42, No. 10, Oct. 1999, pp. 63-70
- [Cos95] Costagliola, G., Tortora, G., Orefice, S., De Lucia, A., "Automatic Generation of Visual Programming Environments," *IEEE Computer*, Vol. 28, No. 3, Mar. 1995, pp. 56-66
- [Cos97] Costagliola, G., De Lucia, A., Orefice, S., Tortora, G., "A Parsing Methodology for the Implementation of Visual Systems," *IEEE Trans. Software Engineering*, Vol. 23, No. 12, Dec. 1997, pp. 777-799
- [Cox98] Cox, P., Smedley, T., "A Model for Object Representation and Manipulation in a Visual Design Language," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998

- [Dem99] Demeyer, S., Ducasse, S., Tichelaar, S., "Why FAMIX and not UML?" *Proc. UML'99 Conf.*, Springer-Verlag Lecture Notes in Computer Science, 1999
- [Dia95] Diaz, A., Isakowitz, T., Maiorana, V., Gilabert, G., "RMC: A Tool to Design WWW Applications," *Proc. 4th World Wide Web Conf.*, Dec. 1995
- [Dor00] Dorđević, I., "Logičko projektovanje: metamodel i generisanje koda," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 2000.
- [EIGW] *The CDIF Metamodel*, <http://www.eigroup.org/cdif>
- [Fra97] Fraternali, P., Paolini, P., "A Conceptual Model and a Tool Environment for Developing More Scalable, Dynamic, and Customizable Web Applications," Tech. Rep. 1997-X, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Aug. 1997, <http://www.ing.unico.it/autoweb>
- [Fra98] Fraternali, P., "Web Application Development: Tools and Approaches," *Proc. 7th World Wide Web Conf.*, Apr. 1998, ftp://ftp.elet.polimi.it/pub/Piero.Fraternali/www_docs/www7/webtools.html
- [Gam95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley Longman, Reading, Mass., 1995
- [Gar86] Garlan, D., Krueger, C. W., Staudt, B. J., "A Structural Approach to the Evolution of Structure-Oriented Environments," *Proc. ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. Practical Softw. Development Environments*, Dec. 1986
- [Gar92] Garlan, D., Cai, L., Nord, R. L., "A Transformational Approach to Generating Application-Specific Environments," *Proc. Fifth ACM SIGSOFT Symp. Softw. Development Environments*, Dec. 1992, pp. 68-77
- [Gel99] Gellersen H.-W., Gaedke, M., "Object-Oriented Web Application Development," *IEEE Internet Computing*, Vol. 3, No. 1, Jan./Feb. 1999, pp. 60-68
- [Geo95] Georgakopoulos, D., Hornick, M., Sheth, A. "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases*, 3, Kluwer Ac. Publishers, 1995, pp. 119-153
- [Gon97] Gong, M., Scott, L., Offen, R., "MetaBuilder: a Generic CASE Tool Builder," *Proc. 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC '97/ICSC '97)*, 1997
- [Gra97] Gray, J. P., Ryan, B., "Applying the CDIF Standard in the Construction of CASE Design Tools," *Proc. Australian Software Engineering Conference (ASWEC '97)*, 1997
- [Hab86] Habermann, A. N., Notkin, D. S., "Gandalf: Software Development Environments," *IEEE Trans. Software Engineering*, Vol. 12, No. 12, Dec. 1986, pp. 1117-1127
- [Hah96] Hahn, E., "Metamodeling in ConceptBase Demonstrated on Fusion," semestral paper, Faculty of Computer Science, Technische Universität München, Oct. 1996
- [HAHW] HAHT Software, *HahtSite*, <http://www.haht.com/>
- [Har87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987, pp. 231-274
- [Har96a] Harel, D., Gery, E., "Executable Object Modeling with Statecharts," *Proc. 18th Int'l Conf. Software Engineering*, Berlin, IEEE Press, Mar. 1996, pp. 246-257

- [Har96b] Harel, D., Naamad, A., "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Engineering Methodology*, Vol. 5, No. 4, Oct. 1996
- [Hil98] Hillegersberg, J. V., Kumar, K., "Using Metamodelling to Analyze the Fit of Object-Oriented Methods to Languages," *Proc. 31st Hawaii International Conference on System Sciences (HICSS'98)*, Jan. 1998
- [Hir97] Hiršl, V., Antonić, A., Milićev, D., "Realizacija konačnih automata u konkurentnom okruženju," *ETRAN*, Zlatibor, Jun 1997.
- [Hop69] Hopcroft, J. E., Ullman, J. D., *Formal Languages and Their Relation to Automata*, Addison-Wesley, 1969
- [Jac92] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Longman, Reading, Mass., 1992
- [Kar94] Karrer, A. S., Scacchi, W., "Meta-Environments for Software Production," *Report from the ATRIUM Project*, University of Southern California, Los Angeles, CA, Dec. 1994, <http://www2.umassd.edu/SWPI/Atrium/localmat.html>
- [Kar98] Karsai, G., Sztipanovits, J., Franke, H. "Towards Specification of Program Synthesis in Model-Integrated Computing," *Proc. IEEE ECBS'98 Conf.*, pp. 226-233, 1998
- [Kar99] Karsai, G. "Structured Specification of Model Interpreters," *Proc. IEEE ECBS'99 Conf.*, pp. 84-91, Mar. 1999
- [Kes95] Kessler, M., "A Schema-Based Approach to HTML Authoring," *Proc. 4th World Wide Web Conf.*, Dec. 1995
- [Kos98] Koskimies, K., Systa, T., Tuomi, J., Mannisto, T., "Automated Support for Modeling OO Software," *IEEE Software*, Vol. 15, No. 1, Jan./Feb. 1998, pp. 87-94
- [Kru98] Krunić, V., Milićev, D., Orčić, Z., "Primena Mesh metodologije u projektovanju informaciono-upravljačkih sistema," *YU Info*, Kopaonik, Apr. 1998., pp. 177-181
- [Lau98] Lauder, A., Kent, S., "Two-Level Modeling," *Proc. 31st IEEE Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS'98)*, 1998
- [Laz99a] Lazarević, Lj., Milićev, D., "Automatsko generisanje C++ koda konačnih automata," *Telfor*, Beograd, Nov. 1999.
- [Laz99b] Lazarević, Lj., "Automatsko generisanje koda za ITU-T preporuke," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 1999.
- [Lie96] Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996
- [Liu98] Liu, S., Offutt, A. J., Ho-Stuart, C., Sun, Y., Ohba, M., "SOFL: A Formal Engineering Methodology for Industrial Applications," *IEEE Trans. on Software Engineering*, Vol. 24, No. 1, Jan. 1998, pp. 24-45
- [LSLW] Lincoln Software Ltd., *IPSYS ToolBuilder*, <http://www.ipsys.com>
- [Man99] Manola, F., "Technologies for a Web Object Model," *IEEE Internet Computing*, Vol. 3, No. 1, Jan./Feb. 1999, pp. 38-47

- [Mar98] Marjanović, D., Milićev, D., "Ravan model perzistencije objekata," *Informacione tehnologije*, Žabljak, Mar. 1998.
- [Mar99] Marjanović, D., "UML semantika metaCASE alata za objektno orijentisanu analizu i projektovanje," *Magistarski rad*, Elektrotehnički fakultet u Beogradu, 1999.
- [MCCW] MetaCase Consulting, *MetaEdit+ Method Workbench*, <http://www.metacase.com>
- [MetW] MetaModel.com, *Metamodeling Glossary*, <http://www.metamodel.com>
- [MGSW] MicroGold Software Inc., *WithClass Scripting Tool*, <http://www.microgold.com>
- [Mil00] Milutinović, V., "Mini-Track on System Support for E-Business on the Internet," *Proc. HICSS-33*, Maui, Hawaii, USA, Jan. 2000, pp. 159.1-159.3
- [Mil00a] Milićev, D., "Extended Object Diagrams for Transformational Specifications in Modeling Environments," *Proc. Second International Symposium on Constructing Software Engineering Tools (CoSET2000)*, Limerick, Ireland, June 2000
- [Mil00b] Milićev, D., "Customizable Output Generation in Modeling Environments Using Pipelined Domains," *ACM SIGSOFT Software Engineering Notes*, Vol. 25, No. 3, May 2000, pp. 46-50
- [Mil01a] Milićev, D., Lazarević, Lj., Marušić, J., *Objektno orijentisano programiranje na jeziku C++*, *Skripta sa praktikumom*, Mikro knjiga, Beograd, 2001
- [Mil01b] Milićev, D., Zarić, M., Piroćanac, N., *Objektno orijentisano modelovanje na jeziku UML*, *Skripta sa praktikumom*, Mikro knjiga, Beograd, 2001
- [Mil95] Milićev, D., *Objektno orijentisano programiranje na jeziku C++*, Mikro knjiga, Beograd, 1995
- [Mil96] Milićev, D., Miletić-Vidaković, M., "Pristup objektno-orijentisanom projektovanju softvera za rad u realnom vremenu," *TELFOR*, Beograd, Nov. 1996.
- [Mil98a] Milićev, D., Krunić, V., "Hijerarhijska metodologija modelovanja sistema upravljanja procesom proizvodnje," *YU Info*, Kopaonik, Apr. 1998., pp. 73-78
- [Mil98b] Milićev, D., Krunić, V., "Mesh: Metod za integralno upravljanje poslovnim i proizvodnim sistemima u naftnoj industriji," *Yung-Info*, Zlatibor, Dec. 1998.
- [Min98] Minas, M., "Automatically Generating Environments for Dynamic Diagram Languages," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998
- [Mip98] mip GmbH, *ALFABET Technology Overview*, <http://www.alfabet.de>
- [MipW] mip GmbH, *Alfabet*, <http://www.alfabet.de>
- [Mit94] Mitchell, K. J., Kennedy, J. B., Barclay, P. J., "A Notation Independent Object Modelling Environment," Technical Report, Napier University, Edinburgh, Scotland, UK, 1994
- [MSCWa] Microsoft Corporation, *Access*, <http://www.microsoft.com/access/>
- [MSCWb] Microsoft Corporation, *Active Server Pages*, <http://msdn.microsoft.com/workshop/server/asp/ASPover.asp>
- [MSCWc] Microsoft Corporation, *Visual InterDev*, <http://msdn.microsoft.com/vinterdev/>

- [Myl90] Mylopolous, J., Borgida, A., Jarke, M., Koubarakis, M., "Telos: A Language for Representing Knowledge about Information Systems," *ACM Trans. on Information Systems*, Vol. 8, No. 4, 1990
- [NetW] NetDynamics, *NetDynamics*, <http://www.netdynamics.com/product/overview/>
- [Nik00] Nikolić, N., Trajković, M., Milićević, M., Milićev, D., Marjanović, D., Sokić, I., Milutinović, V., De Santo, M., Salerno, S., Ritrovato, P., Marsella, M., "Socratenon — A Web-Based Training System with an Intellect," *Proc. HICSS-33*, Maui, Hawaii, USA, Jan. 2000, pp. 160.1-160.10.
- [Nor98a] Nordstrom, G., Sztipanovits, J., Karsai, G., "Meta-Level Extension of the Multigraph Architecture," *Proc. IEEE ECBS'98 Conf.*, 1998
- [Nor98b] Nordstrom, G., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," Area paper, Vanderbilt University, Sep. 1998
- [Nor99] Nordstrom, G., Sztipanovits, J., Karsai, G., Ledeczi, A., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proc. IEEE ECBS'99 Conf.*, Mar. 1999
- [OMG99] Object Management Group (OMG) Inc., *OMG UML Specification*, Ver. 1.3, Jun. 1999, <http://www.omg.org>
- [OMG00] Object Management Group (OMG) Inc., *Response to OMG RFP ad/98-11-01, Action Semantics for the UML*, Ver. 16, Sep. 2000, <http://www.omg.org>
- [OMGW] The Object Management Group, <http://www.omg.org>
- [OPEW] *The OPEN/OML Metamodel*, <http://www.open.org.au>
- [OraWa] Oracle, *Designer*, <http://www.oracle.com/tools/designer/index.html>
- [OraWb] Oracle, *Developer*, <http://www.oracle.com/tools/developer/index.html>
- [Pap95] Papazoglou, M. P., Russell, N., "A Semantic Meta-Modelling Approach to Schema Transformation," *Proc. ACM Conf. Information and Knowledge Management (CIKM'95)*, 1995
- [Pir00] Piroćanac, N., "Softverski alat za projektovanje Web aplikacija," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 2000.
- [Pla97] Platinum Technology, Inc., *Evaluation of Initial Submissions to the OMG's Object Analysis & Design Facility*, Ver. 1.1, Mar. 1997, <http://www.platinum.com/clrlake/omgrfp/oadeval.html>
- [PltW] Platinum Technology, *Paradigm Plus*, <http://www.platinum.com/clearlake>
- [Rep95] Repenning, A., Sumner, T., "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer*, Mar. 1995, pp. 17-25
- [RSCW] Rational Software Corporation, *Rational Rose*, <http://www.rational.com>
- [Sch95] Schwabe, D., Rossi, G., "The Object-Oriented Hypermedia Design Model," *Communications of the ACM*, Vol. 38, No. 8, Aug. 1995, pp. 45-46
- [SeaW] Seagate, *Crystal Report Print Engine*, <http://www.seagatesoftware.com/products/CrystalReports/>
- [Sel94] Selic, B., Gullekson, G., Ward, P., *Real-Time Object-Oriented Modeling*, John Wiley & Sons Inc., 1994

- [Sel99] Selic, B., "Turning Clockwise: Using UML in the Real-Time Domain," *Communications of the ACM*, Vol. 42, No. 10, Oct. 1999, pp. 46-54
- [Shl97] Shlaer, S., Mellor, S. J., "Recursive Design of an Application-Independent Architecture," *IEEE Software*, January 1997
- [Sim96] Simonyi, C., "Intentional Programming - Innovation in the Legacy Age," presented at IFIP WG 2.1 meeting, Jun. 1996, <http://research.microsoft.com/ip>
- [Sim98] Simons, A., "Use Cases Considered Harmful," *Proc. 31st IEEE Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS'98)*, 1998
- [Sno98] Snoeck, M., Dedene, G., "Existence Dependency: The Key to Semantic Integrity Between Structural and Behavioral Aspects of Object Types," *IEEE Trans. Software Engineering*, Vol. 24, No. 4, Apr. 1998, pp. 233-251
- [Ste97] Stewart, D. B., Volpe, R. A., Khosla, P. K., "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Trans. on Software Engineering*, Vol. 23, No. 12, Dec. 1997, pp. 759-776
- [SunW] Sun Microsystems, *Java Server Pages*, <http://www.java.sun.com>
- [SybW] Sybase, *PowerBuilder*, <http://www.powersoft.com/products/powerbuilder/>
- [Szt95] Sztipanovits, J. et al. "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proc. IEEE ICECCS'95*, Nov. 1995, pp. 361-368
- [Szt97] Sztipanovits, J., Karsai, G., "Model-Integrated Computing," *IEEE Computer*, Vol. 30, No. 4, Apr. 1997, pp. 110-112
- [Tol93] Tolvanen, J.-P., Marttiin, P., Smolander, K., "An Integrated Model for Information Systems Modeling," *Proc. 26th HICSS*, Nunamaker, J., and Sprague, H., (Ed.), Vol. 3, IEEE Computer Society Press, Los Alamitos., pp. 470-479, Jan. 1993
- [UAlW] University of Alberta, *MetaView*, <http://www.cs.ualberta.ca/news/CS/1998/research/software.html>
- [VigW] Vignette, *StoryServer*, <http://www.vignette.com/>
- [VUnW] Vanderbilt University, *Multigraph Architecture*, <http://www.isis.vanderbilt.edu>
- [WMC94] Workflow Management Coalition, *The Workflow Reference Model*, <http://www.wfmc.org>, 1994
- [WonWa] Wonderware Corporation, *Wonderware InTouch Getting Started*, Irvine, Calif., 1997
- [WonWb] Wonderware Corporation, *Wonderware InTrack User's Guide*, Irvine, Calif., 1997
- [Zar00] Zarić, M., "The ROOM Method: Metamodeling and Automatic Code Generation," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 2000.
- [Zha98] Zhang, D.-Q., Zhang, K., "VisPro: A Visual Language Generation Toolset," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998
- [Zhu93] Zhu, Y., *Multiple Views for Integrated CASE Environments*, Master thesis, University of Alberta, Sep. 1993
- [Zhu94] Zhuang, Y., *Object-Oriented Modeling in Metaview*, Master thesis, University of Alberta, 1994

Klasifikovana bibliografija

A Softversko inženjerstvo i razno

- [Aho86] Aho, A., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986
- [Bel95] Bell, R., Sharon, D., "Tools to Engineer New Technologies into Applications," *IEEE Software*, Mar. 1995, pp. 11-16
- [Har87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987, pp. 231-274
- [Hop69] Hopcroft, J. E., Ullman, J. D., *Formal Languages and Their Relation to Automata*, Addison-Wesley, 1969

B Objektno orijentisano modelovanje i UML

- [Boo94] Booch, G., *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, Redwood City, Calif., 1994
- [Boo99] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley Longman, Reading, Mass., 1999
- [Gam95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley Longman, Reading, Mass., 1995
- [Har96a] Harel, D., Gery, E., "Executable Object Modeling with Statecharts," *Proc. 18th Int'l Conf. Software Engineering*, Berlin, IEEE Press, Mar. 1996, pp. 246-257
- [Jac92] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Longman, Reading, Mass., 1992
- [Kos98] Koskimies, K., Systs, T., Tuomi, J., Mannisto, T., "Automated Support for Modeling OO Software," *IEEE Software*, Vol. 15, No. 1, Jan./Feb. 1998, pp. 87-94
- [Lie96] Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996
- [Mil95] Milićev, D., *Objektno orijentisano programiranje na jeziku C++*, Mikro knjiga, Beograd, 1995
- [Mil01a] Milićev, D., Lazarević, Lj., Marušić, J., *Objektno orijentisano programiranje na jeziku C++*, *Skripta sa praktikumom*, Mikro knjiga, Beograd, 2001
- [Mil01b] Milićev, D., Zarić, M., Piroćanac, N., *Objektno orijentisano modelovanje na jeziku UML*, *Skripta sa praktikumom*, Mikro knjiga, Beograd, 2001

- [OMG99] Object Management Group (OMG) Inc., *OMG UML Specification*, Ver. 1.3, Jun. 1999, <http://www.omg.org>
- [OMG00] Object Management Group (OMG) Inc., *Response to OMG RFP ad/98-11-01, Action Semantics for the UML*, Ver. 16, Sep. 2000, <http://www.omg.org>
- [Sim98] Simons, A., "Use Cases Considered Harmful," *Proc. 31st IEEE Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS'98)*, 1998
- [Sno98] Snoeck, M., Dedene, G., "Existence Dependency: The Key to Semantic Integrity Between Structural and Behavioral Aspects of Object Types," *IEEE Trans. Software Engineering*, Vol. 24, No. 4, Apr. 1998, pp. 233-251

C Metamodelovanje

- [Art95] Artsy, S., "Meta-modeling the OO Methods, Tools, and Interoperability Facilities," *OOPSLA'95 Workshop in Metamodeling in OO*, Oct. 1995
- [AST95] Advanced Software Technologies, Inc., *Graphical Designer Language*, 1995, <http://www.advancedsw.com>
- [Dem99] Demeyer, S., Ducasse, S., Tichelaar, S., "Why FAMIX and not UML?" *Proc. UML'99 Conf.*, Springer-Verlag Lecture Notes in Computer Science, 1999
- [EIGW] *The CDIF Metamodel*, <http://www.eigroup.org/cdif>
- [Gra97] Gray, J. P., Ryan, B., "Applying the CDIF Standard in the Construction of CASE Design Tools," *Proc. Australian Software Engineering Conference (ASWEC '97)*, 1997
- [Gon97] Gong, M., Scott, L., Offen, R., "MetaBuilder: a Generic CASE Tool Builder," *Proc. 4th Asia-Pacific Software Engineering and International Computer Science Conference (APSEC '97/ICSC '97)*, 1997
- [Hah96] Hahn, E., "Metamodeling in ConceptBase Demonstrated on Fusion," semestral paper, Faculty of Computer Science, Technische Universität München, Oct. 1996
- [Hil98] Hillegersberg, J. V., Kumar, K., "Using Metamodelling to Analyze the Fit of Object-Oriented Methods to Languages," *Proc. 31st Hawaii International Conference on System Sciences (HICSS'98)*, Jan. 1998
- [Kar94] Karrer, A. S., Scacchi, W., "Meta-Environments for Software Production," *Report from the ATRIUM Project*, University of Southern California, Los Angeles, CA, Dec. 1994, <http://www2.umassd.edu/SWPI/Atrium/localmat.html>
- [Lau98] Lauder, A., Kent, S., "Two-Level Modeling," *Proc. 31st IEEE Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS'98)*, 1998
- [MetW] MetaModel.com, *Metamodeling Glossary*, <http://www.metamodel.com>
- [Mip98] mip GmbH, *ALFABET Technology Overview*, <http://www.alfabet.de>
- [Mit94] Mitchell, K. J., Kennedy, J. B., Barclay, P. J., "A Notation Independent Object Modelling Environment," Technical Report, Napier University, Edinburgh, Scotland, UK, 1994

- [Myl90] Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M., "Telos: A Language for Representing Knowledge about Information Systems," *ACM Trans. on Information Systems*, Vol. 8, No. 4, 1990
- [Nor98a] Nordstrom, G., Sztipanovits, J., Karsai, G., "Meta-Level Extension of the Multigraph Architecture," *Proc. IEEE ECBS'98 Conf.*, 1998
- [Nor98b] Nordstrom, G., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," Area paper, Vanderbilt University, Sep. 1998
- [Nor99] Nordstrom, G., Sztipanovits, J., Karsai, G., Ledeczi, A., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proc. IEEE ECBS'99 Conf.*, Mar. 1999
- [OMGW] The Object Management Group, <http://www.omg.org>
- [OPEW] *The OPEN/OML Metamodel*, <http://www.open.org.au>
- [Pla97] Platinum Technology, Inc., *Evaluation of Initial Submissions to the OMG's Object Analysis & Design Facility*, Ver. 1.1, Mar. 1997, <http://www.platinum.com/clrlake/omgrfp/oadeval.html>
- [Szt95] Sztipanovits, J. et al. "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proc. IEEE ICECCS'95*, Nov. 1995, pp. 361-368
- [Szt97] Sztipanovits, J., Karsai, G., "Model-Integrated Computing," *IEEE Computer*, Vol. 30, No. 4, Apr. 1997, pp. 110-112
- [Tol93] Tolvanen, J.-P., Marttiin, P., Smolander, K., "An Integrated Model for Information Systems Modeling," *Proc. 26th HICSS*, Nunamaker, J., and Sprague, H., (Ed.), Vol. 3, IEEE Computer Society Press, Los Alamitos., pp. 470-479, Jan. 1993
- [Zhu93] Zhu, Y., *Multiple Views for Integrated CASE Environments*, Master thesis, University of Alberta, Sep. 1993
- [Zhu94] Zhuang, Y., *Object-Oriented Modeling in Metaview*, Master thesis, University of Alberta, 1994

D Podesivi CASE i metaCASE alati

- [ASTW] Advanced Software Technologies, Inc., *Graphical Designer*, <http://www.advancedsw.com>
- [LSLW] Lincoln Software Ltd., *IPSYS ToolBuilder*, <http://www.ipsys.com>
- [MCCW] MetaCase Consulting, *MetaEdit+ Method Workbench*, <http://www.metacase.com>
- [MGSW] MicroGold Software Inc., *WithClass Scripting Tool*, <http://www.microgold.com>
- [MipW] mip GmbH, *Alfabet*, <http://www.alfabet.de>
- [PltW] Platinum Technology, *Paradigm Plus*, <http://www.platinum.com/clearlake>
- [RSCW] Rational Software Corporation, *Rational Rose*, <http://www.rational.com>
- [UAIW] University of Alberta, *MetaView*, <http://www.cs.ualberta.ca/news/CS/1998/research/software.html>
- [VUnW] Vanderbilt University, *Multigraph Architecture*, <http://www.isis.vanderbilt.edu>

E Vizuelni jezici

- [Anl98] Anlauff, M., Kutter, P. W., Pierantonio, A., "Montages/Gem-Mex: A Meta Visual Programming Generator," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998
- [Bur95] Burnett, M., McIntyre, D., "Visual Programming," *IEEE Computer*, Mar. 1995, Special issue on Visual Programming, pp. 14-16
- [Cha95] Chang, S.-K., Costagliola, G., Pacini, G., Tucci, M., Tortora, G., Yu, B., Yu, J.-S., "Visual-Language System for User Interfaces," *IEEE Software*, Mar. 1995, pp. 33-44
- [Cos95] Costagliola, G., Tortora, G., Orefice, S., De Lucia, A., "Automatic Generation of Visual Programming Environments," *IEEE Computer*, Vol. 28, No. 3, Mar. 1995, pp. 56-66
- [Cos97] Costagliola, G., De Lucia, A., Orefice, S., Tortora, G., "A Parsing Methodology for the Implementation of Visual Systems," *IEEE Trans. Software Engineering*, Vol. 23, No. 12, Dec. 1997, pp. 777-799
- [Cox98] Cox, P., Smedley, T., "A Model for Object Representation and Manipulation in a Visual Design Language," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998
- [Har87] Harel, D., "Statecharst: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987, pp. 231-274
- [Min98] Minas, M., "Automatically Generating Environments for Dynamic Diagram Languages," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998
- [Rep95] Repenning, A., Sumner, T., "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *IEEE Computer*, Mar. 1995, pp. 17-25
- [Zha98] Zhang, D.-Q., Zhang, K., "VisPro: A Visual Language Generation Toolset," *Proc. 14th IEEE Symp. Visual Languages*, Halifax, Canada, Sep. 1998

F Transformacije modela

- [Ait97] Aitken, W., Dickens, B., Kwiatkowski, P., De Moor, O., Richter, D., Simonyi, C., "Transformation in Intentional Programming," <http://research.microsoft.com/ip>
- [Gar86] Garlan, D., Krueger, C. W., Staudt, B. J., "A Structural Approach to the Evolution of Structure-Oriented Environments," *Proc. ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. Practical Softw. Development Environments*, Dec. 1986
- [Gar92] Garlan, D., Cai, L., Nord, R. L., "A Transformational Approach to Generating Application-Specific Environments," *Proc. Fifth ACM SIGSOFT Symp. Softw. Development Environments*, Dec. 1992, pp. 68-77
- [Hab86] Habermann, A. N., Notkin, D. S., "Gandalf: Software Development Environments," *IEEE Trans. Software Engineering*, Vol. 12, No. 12, Dec. 1986, pp. 1117-1127

- [Kar98] Karsai, G., Sztipanovits, J., Franke, H. "Towards Specification of Program Synthesis in Model-Integrated Computing," *Proc. IEEE ECBS'98 Conf.*, pp. 226-233, 1998
- [Kar99] Karsai, G. "Structured Specification of Model Interpreters," *Proc. IEEE ECBS'99 Conf.*, pp. 84-91, Mar. 1999
- [Pap95] Papazoglou, M. P., Russell, N., "A Semantic Meta-Modelling Approach to Schema Transformation," *Proc. ACM Conf. Information and Knowledge Management (CIKM'95)*, 1995
- [Shl97] Shlaer, S., Mellor, S. J., "Recursive Design of an Application-Independent Architecture," *IEEE Software*, January 1997
- [Sim96] Simonyi, C., "Intentional Programming - Innovation in the Legacy Age," presented at IFIP WG 2.1 meeting, Jun. 1996, <http://research.microsoft.com/ip>

G Projektovanje Web aplikacija

- [Con99] Conallen, J., "Modeling Web Application Architectures with UML," *Communications of the ACM*, Vol. 42, No. 10, Oct. 1999, pp. 63-70
- [Dia95] Diaz, A., Isakowitz, T., Maiorana, V., Gilabert, G., "RMC: A Tool to Design WWW Applications," *Proc. 4th World Wide Web Conf.*, Dec. 1995
- [Fra97] Fraternali, P., Paolini, P., "A Conceptual Model and a Tool Environment for Developing More Scalable, Dynamic, and Customizable Web Applications," Tech. Rep. 1997-X, Politecnico di Milano, Dipartimento di Elettronica e Informazione, Aug. 1997, <http://www.ing.unico.it/autoweb>
- [Fra98] Fraternali, P., "Web Application Development: Tools and Approaches," *Proc. 7th World Wide Web Conf.*, Apr. 1998, ftp://ftp.elet.polimi.it/pub/Piero.Fraternali/www_docs/www7/webtools.html
- [Gel99] Gellersen H.-W., Gaedke, M., "Object-Oriented Web Application Development," *IEEE Internet Computing*, Vol. 3, No. 1, Jan./Feb. 1999, pp. 60-68
- [Kes95] Kessler, M., "A Schema-Based Approach to HTML Authoring," *Proc. 4th World Wide Web Conf.*, Dec. 1995
- [Man99] Manola, F., "Technologies for a Web Object Model," *IEEE Internet Computing*, Vol. 3, No. 1, Jan./Feb. 1999, pp. 38-47
- [Mil00] Milutinović, V., "Mini-Track on System Support for E-Business on the Internet," *Proc. HICSS-33*, Maui, Hawaii, USA, Jan. 2000, pp. 159.1-159.3
- [Sch95] Schwabe, D., Rossi, G., "The Object-Oriented Hypermedia Design Model," *Communications of the ACM*, Vol. 38, No. 8, Aug. 1995, pp. 45-46

H Alati za projektovanje Web aplikacija

- [AllIW] Allaire Inc., *Cold Fusion Web Construction Kit*, <http://www.allaire.com/products/ColdFusion/>
- [BorW] Borland, *Delphi Client/Server Suite*, <http://www.borland.com/delphi/>

- [HAHW] HAHT Software, *HahtSite*, <http://www.haht.com/>
- [MSCWa] Microsoft Corporation, *Access*, <http://www.microsoft.com/access/>
- [MSCWb] Microsoft Corporation, *Active Server Pages*,
<http://msdn.microsoft.com/workshop/server/asp/ASPOver.asp>
- [MSCWc] Microsoft Corporation, *Visual InterDev*, <http://msdn.microsoft.com/vinterdev/>
- [NetW] NetDynamics, *NetDynamics*, <http://www.netdynamics.com/product/overview/>
- [OraWa] Oracle, *Designer*, <http://www.oracle.com/tools/designer/index.html>
- [OraWb] Oracle, *Developer*, <http://www.oracle.com/tools/developer/index.html>
- [SeaW] Seagate, *Crystal Report Print Engine*,
<http://www.seagatesoftware.com/products/CrystalReports/>
- [SunW] Sun Microsystems, *Java Server Pages*, <http://www.java.sun.com>
- [SybW] Sybase, *PowerBuilder*, <http://www.powersoft.com/products/powerbuilder/>
- [VigW] Vignette, *StoryServer*, <http://www.vignette.com/>

I Sistemi za rad u realnom vremenu

- [Har87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987, pp. 231-274
- [Har96a] Harel, D., Gery, E., "Executable Object Modeling with Statecharts," *Proc. 18th Int'l Conf. Software Engineering*, Berlin, IEEE Press, Mar. 1996, pp. 246-257
- [Har96b] Harel, D., Naamad, A., "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Engineering Methodology*, Vol. 5, No. 4, Oct. 1996
- [Sel94] Selic, B., Gullekson, G., Ward, P., *Real-Time Object-Oriented Modeling*, John Wiley & Sons Inc., 1994
- [Sel99] Selic, B., "Turning Clockwise: Using UML in the Real-Time Domain," *Communications of the ACM*, Vol. 42, No. 10, Oct. 1999, pp. 46-54
- [Ste97] Stewart, D. B., Volpe, R. A., Khosla, P. K., "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Trans. on Software Engineering*, Vol. 23, No. 12, Dec. 1997, pp. 759-776

J Modelovanje industrijskih procesa

- [Geo95] Georgakopoulos, D., Hornick, M., Sheth, A. "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases*, 3, Kluwer Ac. Publishers, 1995, pp. 119-153
- [Liu98] Liu, S., Offutt, A. J., Ho-Stuart, C., Sun, Y., Ohba, M., "SOFL: A Formal Engineering Methodology for Industrial Applications," *IEEE Trans. on Software Engineering*, Vol. 24, No. 1, Jan. 1998, pp. 24-45
- [WMC94] Workflow Management Coalition, *The Workflow Reference Model*, <http://www.wfmc.org>, 1994

- [WonWa] Wonderware Corporation, *Wonderware InTouch Getting Started*, Irvine, Calif., 1997
- [WonWb] Wonderware Corporation, *Wonderware InTrack User's Guide*, Irvine, Calif., 1997

K Radovi autora vezani za temu teze

- [Hir97] Hiršl, V., Antonić, A., Milićev, D., "Realizacija konačnih automata u konkurentnom okruženju," *ETRAN*, Zlatibor, Jun 1997.
- [Kru98] Krunic, V., Milićev, D., Orčić, Z., "Primena Mesh metodologije u projektovanju informaciono-upravljačkih sistema," *YU Info*, Kopaonik, Apr. 1998., pp. 177-181
- [Laz99a] Lazarević, Lj., Milićev, D., "Automatsko generisanje C++ koda konačnih automata," *Telfor*, Beograd, Nov. 1999.
- [Mar98] Marjanović, D., Milićev, D., "Ravan model perzistencije objekata," *Informacione tehnologije*, Žabljak, Mar. 1998.
- [Mil00a] Milićev, D., "Extended Object Diagrams for Transformational Specifications in Modeling Environments," *Proc. Second International Symposium on Constructing Software Engineering Tools (CoSET2000)*, Limerick, Ireland, June 2000
- [Mil00b] Milićev, D., "Customizable Output Generation in Modeling Environments Using Pipelined Domains," *ACM SIGSOFT Software Engineering Notes*, Vol. 25, No. 3, May 2000, pp. 46-50
- [Mil01a] Milićev, D., Lazarević, Lj., Marušić, J., *Objektno orijentisano programiranje na jeziku C++*, Skripta sa praktikumom, Mikro knjiga, Beograd, 2001
- [Mil01b] Milićev, D., Zarić, M., Piroćanac, N., *Objektno orijentisano modelovanje na jeziku UML*, Skripta sa praktikumom, Mikro knjiga, Beograd, 2001
- [Mil95] Milićev, D., *Objektno orijentisano programiranje na jeziku C++*, Mikro knjiga, Beograd, 1995
- [Mil96] Milićev, D., Miletić-Vidaković, M., "Pristup objektno-orijentisanom projektovanju softvera za rad u realnom vremenu," *TELFOR*, Beograd, Nov. 1996.
- [Mil98a] Milićev, D., Krunic, V., "Hijerarhijska metodologija modelovanja sistema upravljanja procesom proizvodnje," *YU Info*, Kopaonik, Apr. 1998., pp. 73-78
- [Mil98b] Milićev, D., Krunic, V., "Mesh: Metod za integralno upravljanje poslovnim i proizvodnim sistemima u naftnoj industriji," *Yung-Info*, Zlatibor, Dec. 1998.
- [Nik00] Nikolić, N., Trajković, M., Milićević, M., Milićev, D., Marjanović, D., Sokić, I., Milutinović, V., De Santo, M., Salerno, S., Ritrovato, P., Marsella, M., "Socratenon — A Web-Based Training System with an Intellect," *Proc. HICSS-33*, Maui, Hawaii, USA, Jan. 2000, pp. 160.1-160.10.

L Radovi saradnika vezani za temu teze

- [Dor00] Dorđević, I., "Logičko projektovanje: metamodel i generisanje koda," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 2000.

- [Laz99b] Lazarević, Lj., "Automatsko generisanje koda za ITU-T preporuke," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 1999.
- [Mar99] Marjanović, D., "UML semantika metaCASE alata za objektno orijentisanu analizu i projektovanje," *Magistarski rad*, Elektrotehnički fakultet u Beogradu, 1999.
- [Pir00] Piroćanac, N., "Softverski alat za projektovanje Web aplikacija," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 2000.
- [Zar00] Zarić, M., "The ROOM Method: Metamodeling and Automatic Code Generation," *Diplomski rad*, Elektrotehnički fakultet u Beogradu, 2000.

X Prilozi

Prilog A

Algoritmi za generisanje koda transformatora

U ovom prilogu dat je pseudokôd pomoću koga je moguće generisati C++ kôd transformatora polazeći od specifikacija preslikavanja domena. Podela je izvršena na operacije klase iz metamodela preslikavanja domena prikazanog na slici 5.2. Dati pseudokôd se može lako izmeniti tako da generiše kôd na nekom drugom ciljnom programskom jeziku.

DMPackage::generateCode()

```
String pckName = "pck" + name;
output: "Package* " + pckName + " = 0;";

output: "if (" + cond + ") {";

output: pckName + " = Package::create(" + parentPckName + ");";
output: pckName + "->dmOrgName = \"" + name + "\";";
output: pckName + "->setName(\"" + name + "\");";

String parentPckNameSaved = parentPckName;
parentPckName = pckName;

Topologically sort the owned elements, regarding to sequence dependences;
Traversing the owned elements in the topological order,
for each element e do
    e->generateCode();

output: "}";

parentPckName = parentPckNameSaved;
```

DMForEachPackage::generateCode()

```
String pckName = "pckForEach" + name;
output: "Package* " + pckName + " = Package::create(" + parentPckName +
");";
output: pckName + "->dmOrgName = \"" + name + "\";";
output: pckName + "->setName(\"ForEach-" + name + "\");";

output: "ForEach(" + forEach + "," + ofType + "," + inCollection + ")";

String pckNestedName = "pck" + name;
output: "Package* " + pckNestedName + " = 0;";

output: "if (" + cond + ") {";

output: pckNestedName + " = Package::create(" + pckName + ");";
output: pckNestedName + "->dmOrgName = \"" + name + "\";";
output: pckNestedName + "->setName(" + forEach + "->getFullParthName());";
```

```

String parentPckNameSaved = parentPckName;
parentPckName = pckNestedName;

Topologically sort the owned elements, regarding to sequence dependences;
Traversing the owned elements in the topological order,
for each element e do
    e->generateCode();

output: "}";

output: "EndForEach(" + forEach + ")";

parentPckName = parentPckNameSaved;

DMInstance::generateCode()
String tpNm = type->name;
output: tpNm + "* " + name + " = 0;";

output: "if (" + cond + ") {";

output: name + "=(\"+tpNm+"*)" + tpNm + "::create(" + parentPckName + ");";
output: name + "->dmOrgName = \"\" + name + \"\";";
output: name + "->setName(\"\" + name + \"\");";

ForEach(attr,AttributeSetting,getAttributeSettings())
    output: name + "->" + attr->name + " = " + attr->value + ";";
EndForEach(attr)

output: "}";

// Helper function:
DMLink::generateInstanceAccess(Package* p, String& pckName)
if (p->isForEach()) {

    // ForEach package
    String pckForEachName = "pckForEach" + p->name;
    String pckNewName = "pck" + p->name;
    output: "Package* " + pckForEachName + " = " + pckName +
        "->getOwnedElement(\"ForEach-\" + p->name + "\",\"Package\");";
    output: "ForEach(" + pckNewName + ",Package,\" +
        pckForEachName + "->getOwnedElements());";
    pckName = pckNewName;

} else {

    // Simple package
    String pckNewName = "pck" + p->name;
    output: "Package* " + pckNewName + " = " + pckName +
        "->getOwnedElement(\"\" + pckNewName + "\",\"Package\");";
    pckName = pckNewName;
}

```


DMLink::generateCode()

```

Let dmi1 = mySideA, dmi2 = mySideB;
Let tpName1 = dmi1->type->name, tpName2 = dmi2->type->name;
Let dmp = myOwnerPackage;
Let pList1 = listOfPackages(dmp,dmi1->myOwnerPackage); // Excluding dmp
Let pList2 = listOfPackages(dmp,dmi2->myOwnerPackage); // Excluding dmp
output: "{";

// Access to the first side:
String pckName1 = dmp->name;
ForEach (p1,Package,pList1)
    generateInstanceAccess(p1,pckName1);
EndForEach(p1)

String ptrName1 = dmi1->name;
output: tpName1 + "*" + ptrName1 + " = (" + tpName1 + ")( " +
    pckName1 + "->getOwnedElement(\"" +
    dmi1->name + "\",\"" + tpName1 + "\");";

// Access to the second side:
String pckName2 = dmp->name;
ForEach (p2,Package,pList2)
    generateInstanceAccess(p2,pckName2);
EndForEach(p2)

String ptrName2 = dmi2->name;
output: tpName2 + "*" + ptrName2 + " = (" + tpName2 + ")( " +
    pckName2 + "->getOwnedElement(\"" +
    dmi2->name + "\",\"" + tpName2 + "\");";

// Link creation:
output: "if (" + ptrName1 + " && " + ptrName2 + "(" + cond + ")) {";
output: "M1Link* " + name + " = M2Association::createLink(\"" +
    type->name + "\" + dmi1->name + "," + dmi2->name + ");";
output: name + "->dmOrgName = \"" + name + "\";";
output: "}";
output: "}";

Generate "EndForEach" clauses for ForEach packages in pList2 (reverse
order);
Generate "EndForEach" clauses for ForEach packages in pList1 (reverse
order);
output: "}";

```

DMInstanceRef::generateCode()

```

output: "if (" + name + "!=0 && (" + cond + ")) {";
ForEach(attr,AttributeSetting,getAttributeSettings())
    output: name + "->" + attr->name + " = " + attr->value + ";";
EndForEach(attr)
output: "}";

```

DMPackageRef::generateCode()

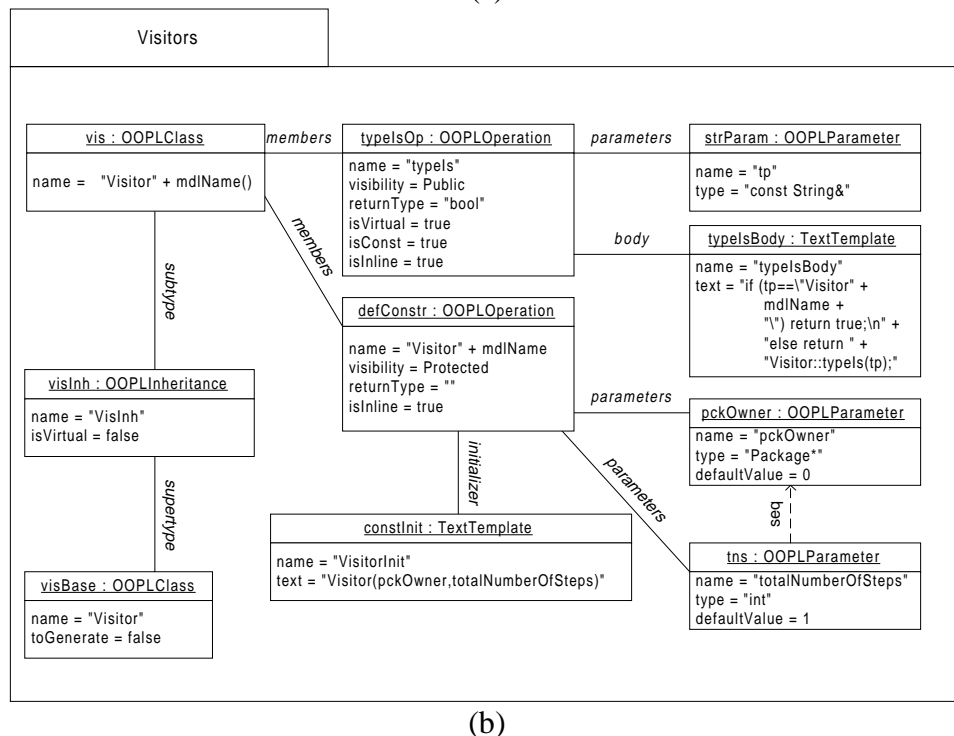
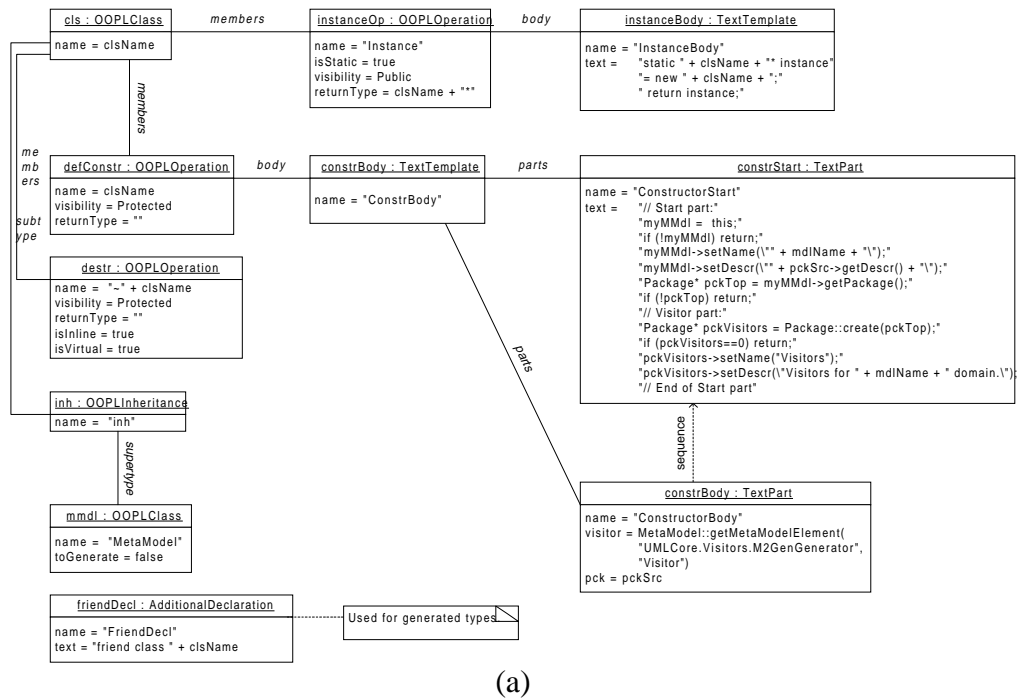
```

output: "if (" + cond + ") {";
output: name + "(" + parentPckName + ", " + params + ");";
output: "}";

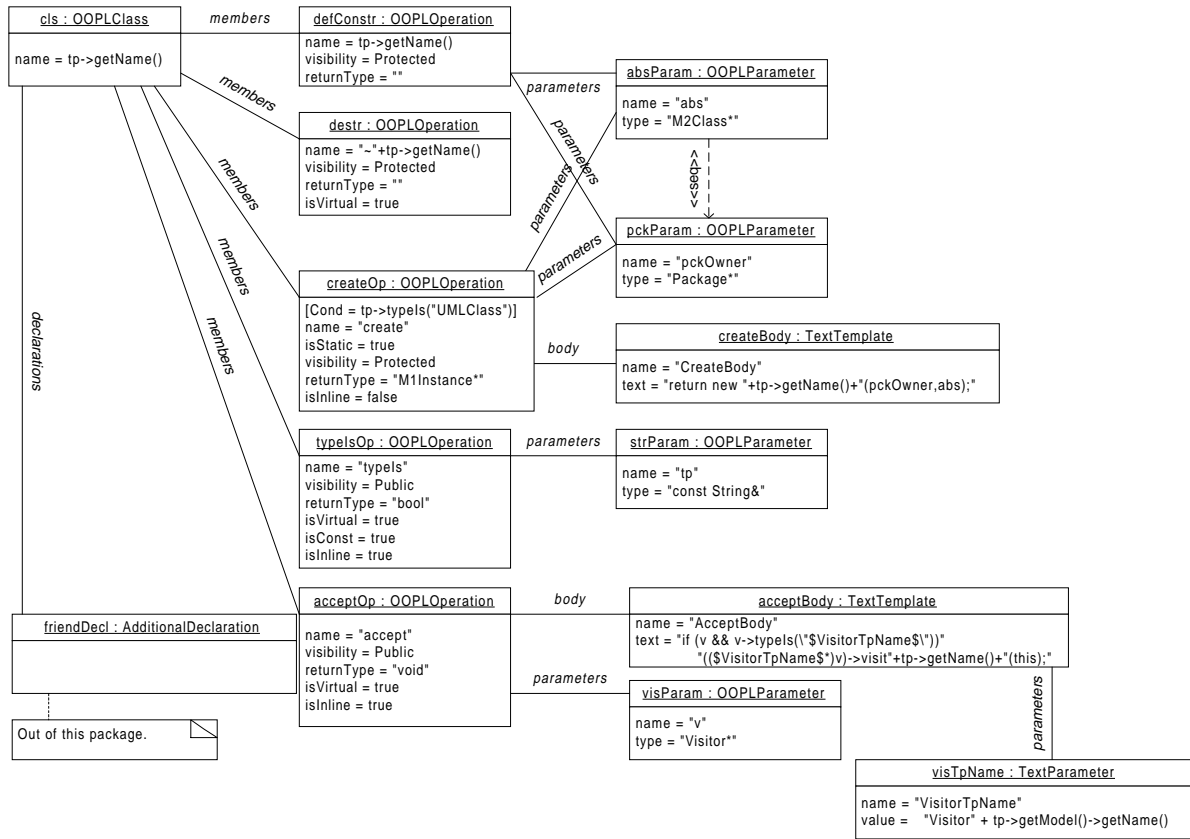
```

Prilog B

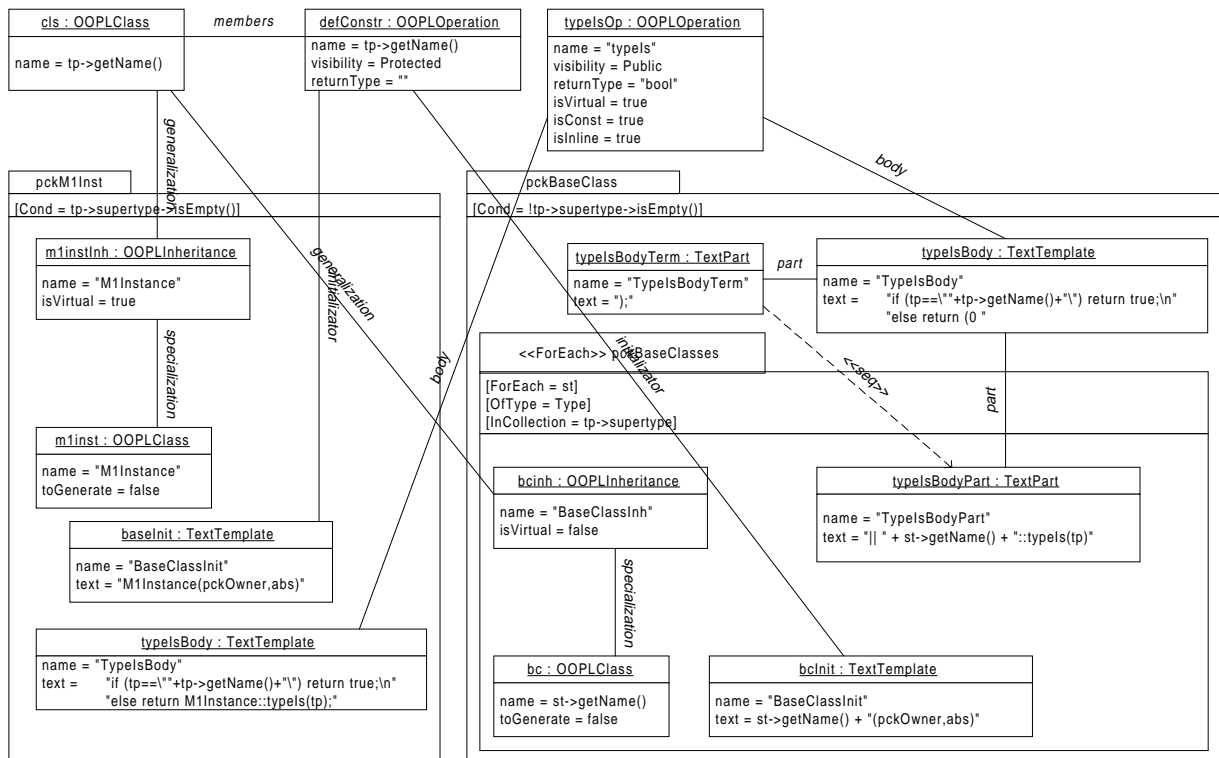
Specifikacije preslikavanja u alatu za metamodelovanje



Slika B.1: Način na koji se metamodel definisan kao model u domenu UMLCore manifestuje u alatu za modelovanje. (a) M2Gen manifestacija: *Singleton* klasa u čijem se konstruktoru kreiraju objekti koji predstavljaju metamodel. (b) *Visitor* koji se generiše za svaki domen.

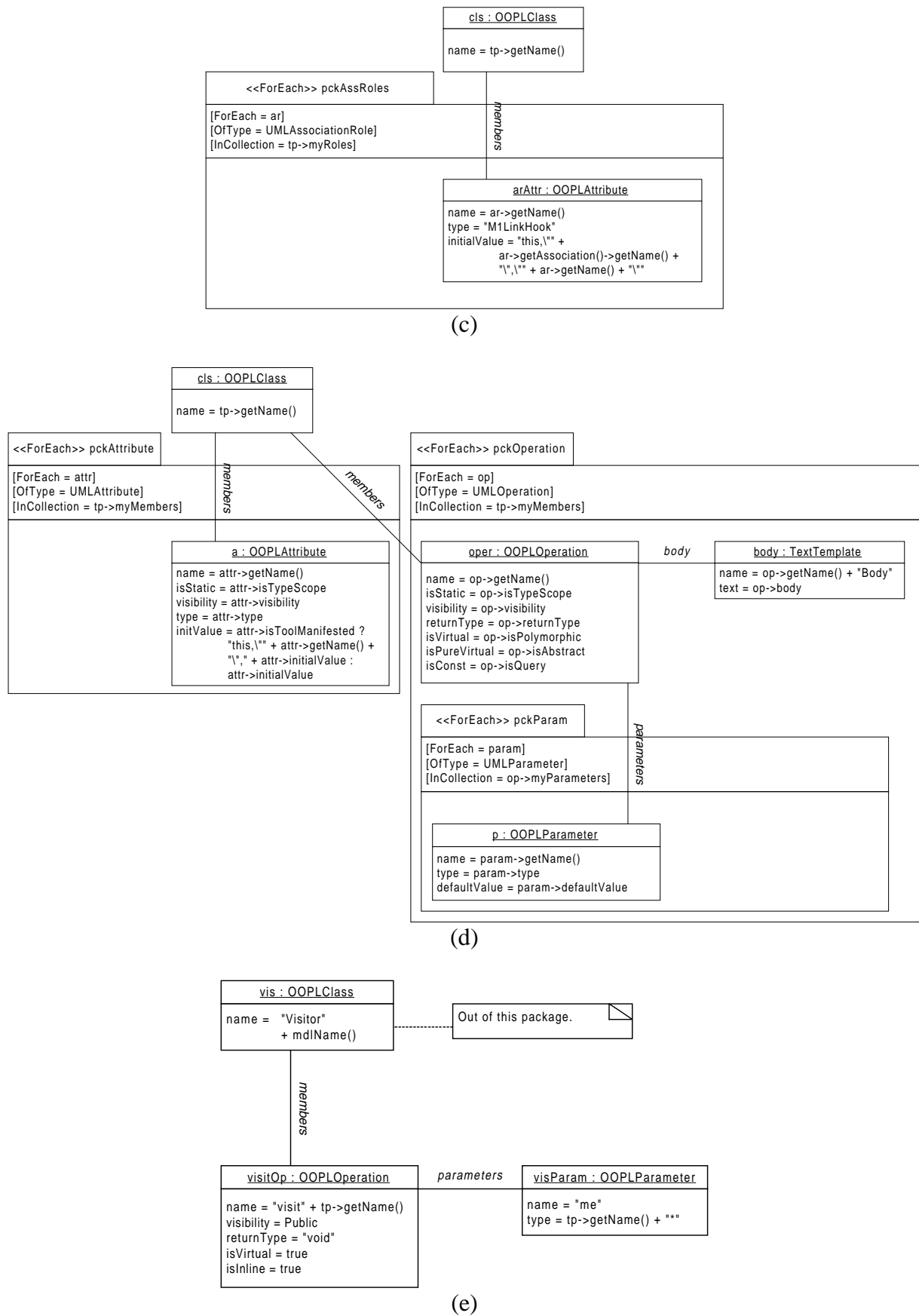


(a)

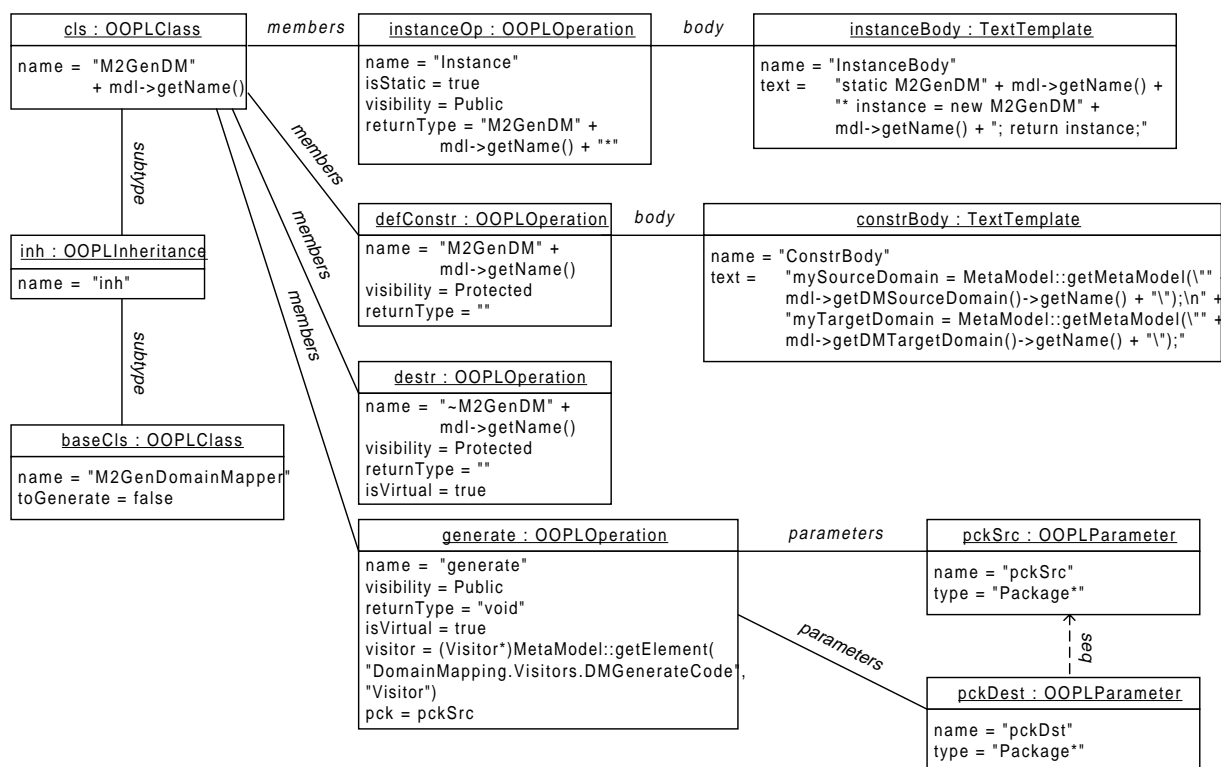


(b)

Slika B.2: Način na koji se svaka apstrakcija metamodela, definisana kao instanca tipa `UMLClass` u modelu u domenu `UMLCore`, manifestuje u `M1Gen` delu generisanog alata za modelovanje. (a) Generisana klasa i osnovne operacije. (b) Manifestacija generalizacije. (Nastavak na sledećoj strani)



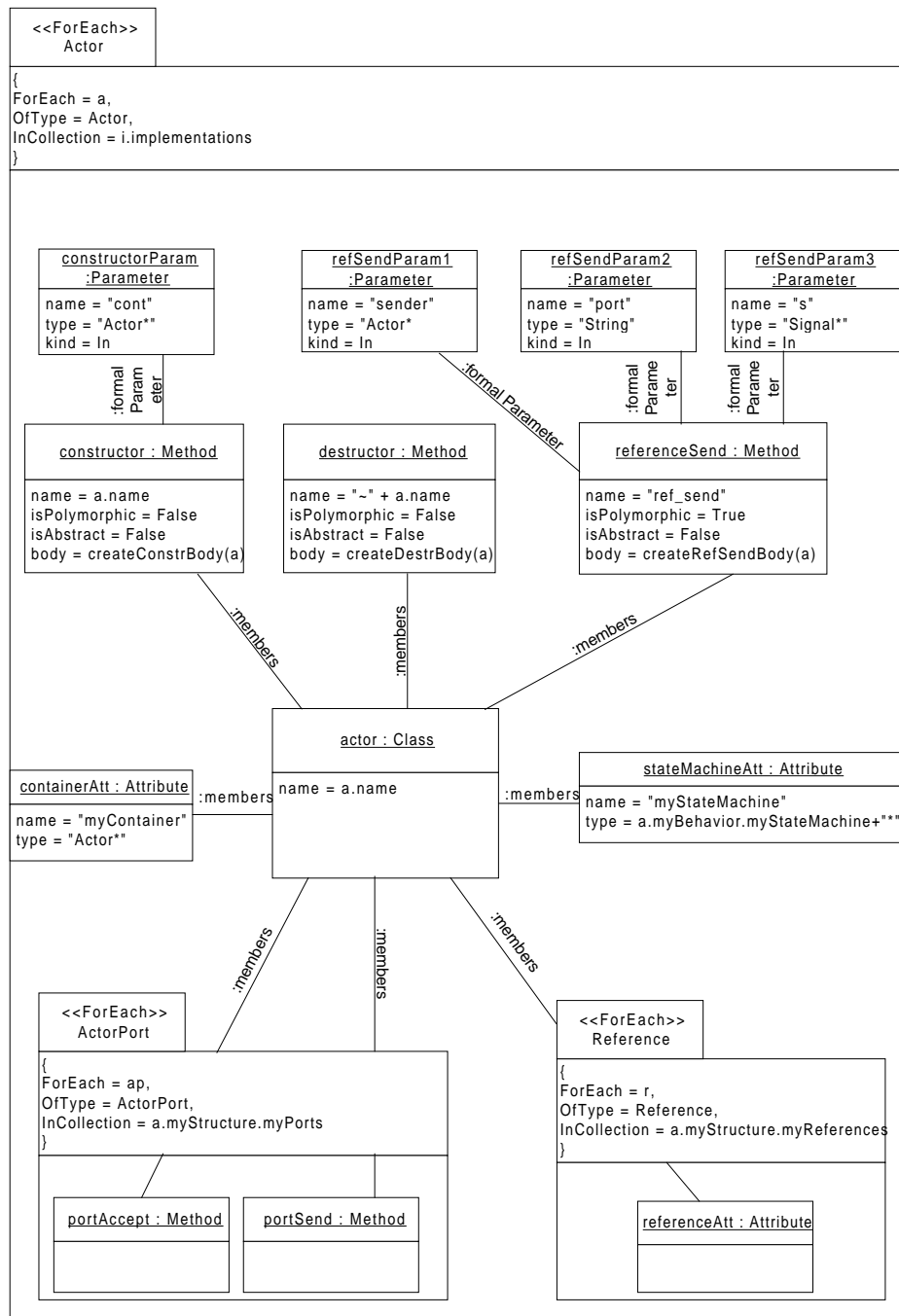
Slika B.2 (nastavak): Način na koji se svaka apstrakcija metamodela, definisana kao instanca tipa `UMLClass` u modelu u domenu `UMLCore`, manifestuje u `M1Gen` delu generisanog alata za modelovanje. (c) Manifestacija uloge asocijacije (engl. *Association Role*) kao podatak član generisane klase. (d) Manifestacija članova klase (atributi i operacije). (e) Generisanje operacije članice *Visitor* klase za svaku apstrakciju metamodela.



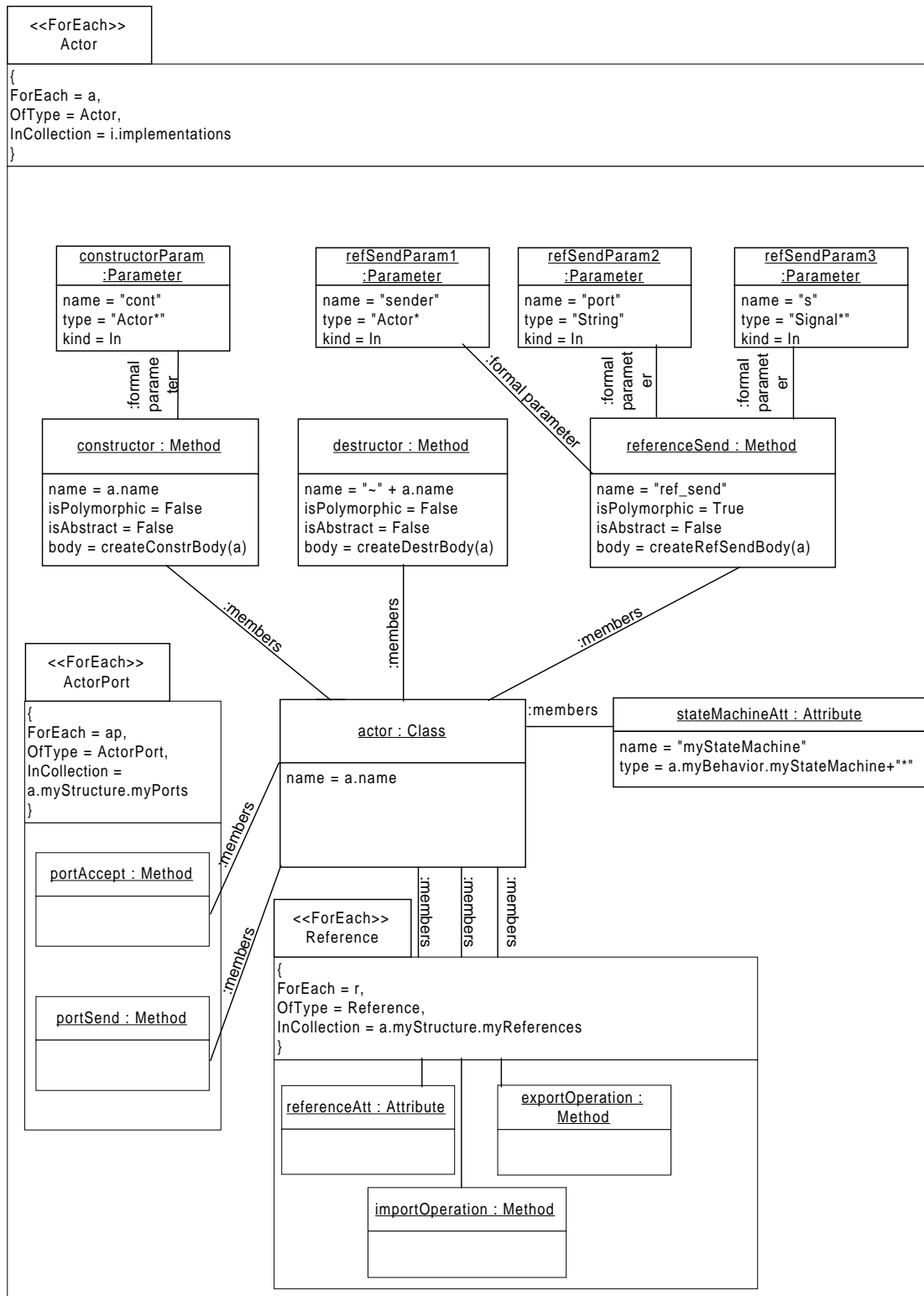
Slika B.3: Način na koji se preslikavanje domena, definisano kao model u domenu DomainMapping, manifestuje u M2Gen delu generisanog alata za modelovanje.

Prilog C

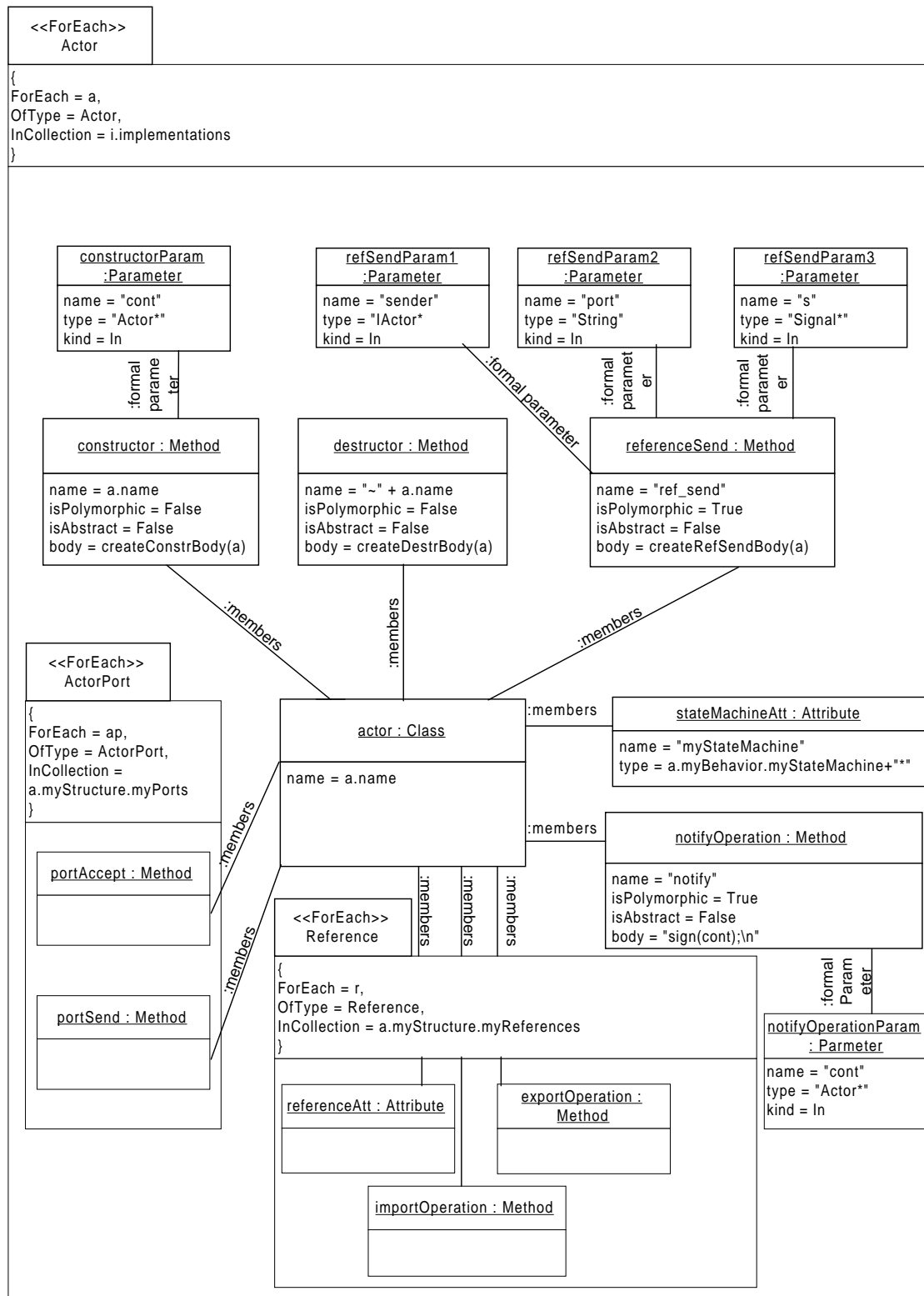
Specifikacije preslikavanja za primer metode ROOM



Slika C.1: Dijagram preslikavanja za `<<ForEach>>` paket koji iterira kroz sve instance tipa `Actor` u izvorišnom modelu, za prvu verziju generatora.



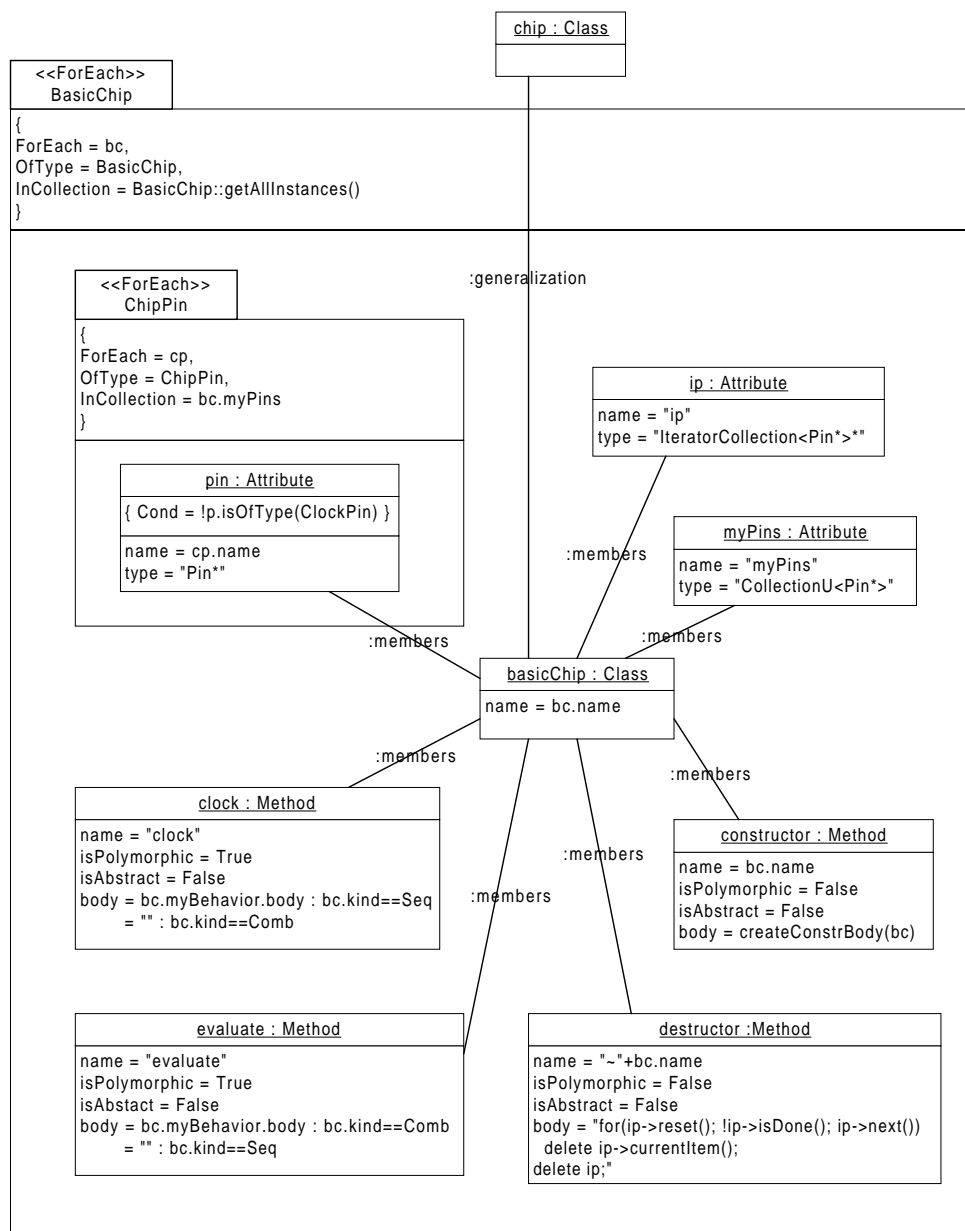
Slika C.2: Dijagram preslikavanja za `<<ForEach>>` paket koji iterira kroz sve instance tipa `Actor` u izvorišnom modelu, za drugu verziju generatora.



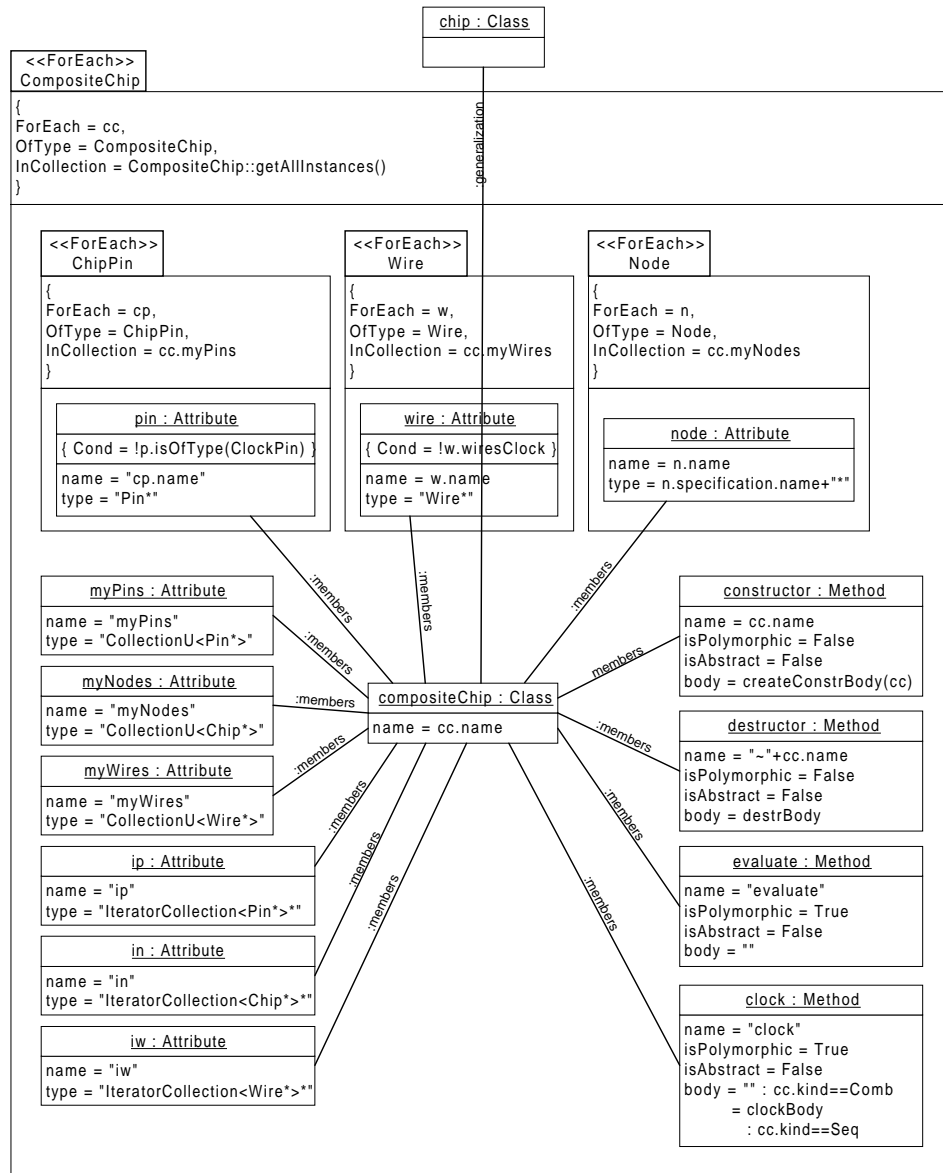
Slika C.3: Dijagram preslikavanja za <<ForEach>> paket koji iterira kroz sve instance tipa Actor u izvorišnom modelu, za treću verziju generatora.

Prilog D

Specifikacije preslikavanja za primer logičkog projektovanja hardvera



Slika D.1: Preslikavanje iz domena logičkog projektovanja hardvera u domen OOP. Dijagram prikazuje specifikaciju generisanja koda za proste čipove.



Slika D.2: Preslikavanje iz domena logičkog projektovanja hardvera u domen OOPL. Dijagram prikazuje specifikaciju generisanja koda za složene čipove.