KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Alfonsas Cirtautas

Programavimo kalbų taikymas modelių transformacijoms realizuoti MDA architektūroje

Magistro darbas

Vadovas
doc. V. Pilkauskas

KAUNAS, 2005

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS FAKULTETAS

TVIRTINU
Katedros vedėjas
prof. habil. dr. H. Pranevičius
2005-05-23

Programavimo kalbų taikymas modelių transformacijoms realizuoti MDA architektūroje

Informatikos magistro baigiamasis darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
dr. Jurgita Mikelionienė
2005-05-20

Recenzentas
dr. doc. Rimantas Butleris

2005-05-23

Vadovas
doc. dr. V. Pilkauskas

2005-05-23

Atliko
IFM-9/1 gr. stud.
A. Cirtautas
2005-05-23

Kaunas, 2005

TURINYS

LENTELĖS

PAVEIKSLAI

SUMMARY

Presented work covers one of the most important areas of OMG's model driven architecture (MDA) – problems of object model transformations. Based on research of OMG specifications and other sources, author analyzes transformation process, states importance of modeling and metamodeling for designing of UML like modeling languages.

Research work describes designed PIM ant PSM metamodels of aggregate systems. Structure and syntax of ATL and MTL model transformation languages was introduced. Author gives a short overview of model editors and graphical representation for these languages, created using EMF framework tools.

Transformation languages have been compared. A result of experimental transformations was summarized by their qualitative and quantitative criteria. Advantages of hybrid ATL transformation language versus imperative MTL language were found. Experimental transformations were successfully produced and executed using ATL and MTL model transformation languages.

1. Įvadas

Informacinės technologijos – viena iš greičiausiai besivystančių paskutinio dešimtmečio mokslo ir pramonės šakų. Buvo pastebėta, kad programinės įrangos abstraktumo lygis atvirkščiai proporcingas kitimo spartai. Programinės įrangos vystymasis pasižymi didžiule kaita: kuriamos naujos ir tobulinamos senos programavimo kalbos. Pastaruoju metu vis didesnis projektuotojų ir programuotojų dėmesys skiriamas sistemų modeliavimui. Įvairūs sistemų modeliavimo būdai jau seniai naudojami kuriant didelius projektus, bet dažniausiai lieka tik dokumentacijos dalimi. Pradinis sistemos modelis dažnai nebeatitinka galutinio produkto, todėl praranda praktinę vertę.

Modeliais pagrįsta architektūra (MDA) (*Model Driven Architecture*)– tai OMG (*Object Management Group*) iniciatyva, kuri pateikia efektyvaus programinės įrangos modelių kūrimo ir panaudojimo strategijas. MDA apibrėžia tokį sistemų specifikavimo būdą, kuris atskiria sistemos funkcionalumo specifikaciją nuo sistemos realizavimo specifikacijos tam tikrai technologinei platformai. MDA nėra nauja architektūra – tai nauja programinės įrangos modelių kūrimo strategija. MDA tikslas – ne vieno uniforminio standarto įdiegimas, o aiškus skirtingų abstrakcijos lygių atskyrimas. MDA architektūros taikymas turėtų iš esmės pakeisti šiuolaikinę programų kūrimo praktiką.
MDA architektūros svarbiausios dalys yra skirtingų abstrakcijų lygių modeliai ir transformacijos tarp jų. Skirtingų abstrakcijos lygių modeliai jau standartizuoti pagal MOF specifikaciją. Mažiausiai ištirta MDA architektūros sritis yra modelių transformavimas. Šiuo metu OMG ruošia modelių transformavimo standartą QVT- RFP (*Query / Views / Transformations - Request For Proposal*). Šiame darbe bus pristatytos dvi programavimo kalbos (ATL – *ATLAS Transformation Language* ir MTL – *Model Transformation Language*) skirtos aprašyti modelių transformacijas.

Šio darbo tikslas – ištirti ATL ir MTL kalbų tinkamumą agregatinių sistemų modelių transformacijoms. Gauti rezultatai galėtų būti tolimesnių tyrimų, detaliau analizuojant transformavimo kalbų galimybes, pradžia.

2. MDA architektūra ir transformacijos

Naudojant tradicinius programinės įrangos projektavimo ir programavimo procesus programos yra kuriamos konkrečiai platformai. MDA pristato aukštesnį abstrakcijos lygį, teikdama galimybę organizacijoms kurti modelius, nepriklausomus nuo konkrečios aparatinės įrangos, operacinės sistemos ar platformos. MDA užtikrina programinės įrangos kūrimą pagrįstą biznio modeliais.

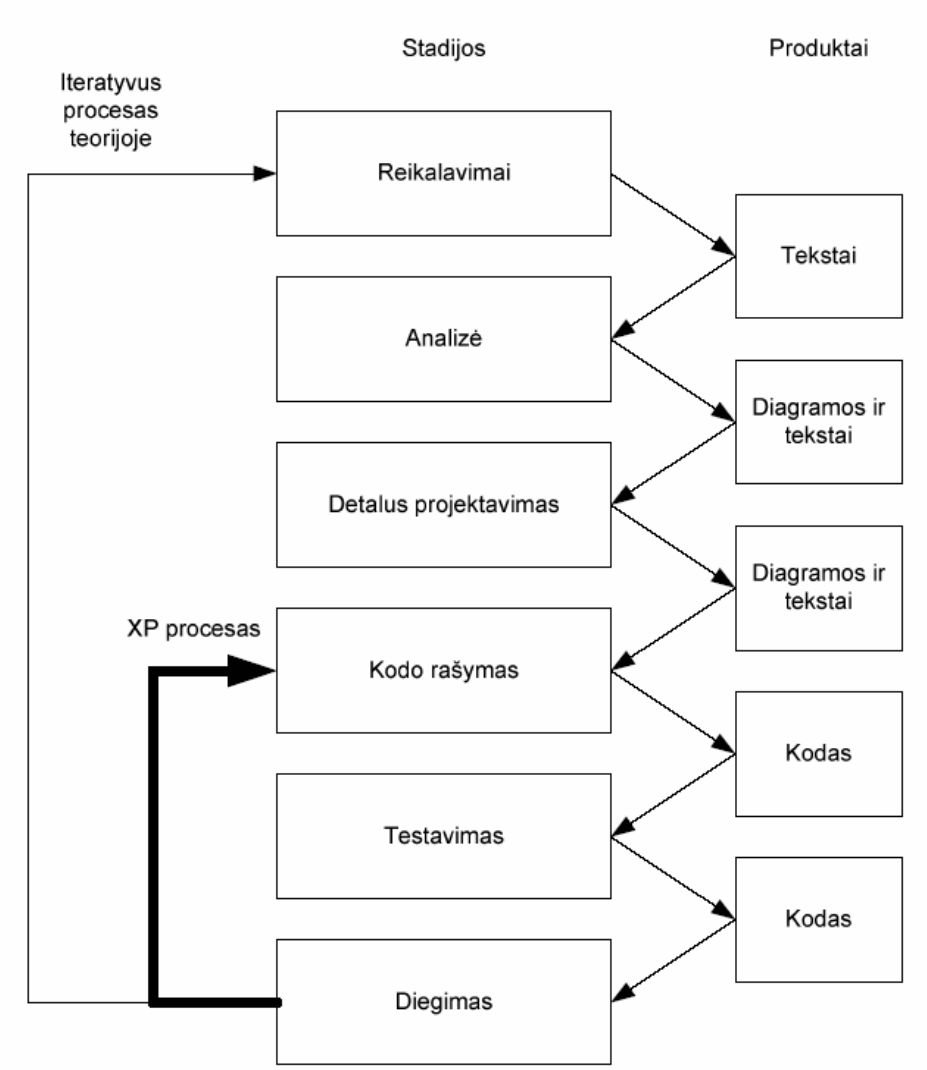
2.1 Tradicinis programų kūrimo procesas

Tradicinis programų kūrimo procesas susideda iš 6 stadijų (žr. 1 pav.):

1. Reikalavimų surinkimas
2. Analizė
3. Projektavimas
4. Kodo rašymas
5. Testavimas
6. Diegimas

Pirmosiose trijose šio proceso dalyse būna sukuriama daug įvairių dokumentų (reikalavimų specifikacijos, funkcinės specifikacijos ir įvairios schemos) ir UML diagramų daug skirtų būsimos sistemos analizei ir projektavimui. Ši dalis reikalauja daug žmoniškųjų ir materialiųjų resursų. Bet prasidėjus 4 stadijai, šių dokumentų vertė ima mažėti, daugėja neatitikimų tarp realizacijos ir projekto. Keičiant ar tobulinant sistemą pakeitimai dažniausiai būna daromi tik programos kodo lygyje.

Ekstremalaus programavimo (XP) (*Extreme Programming*) idėja tapo populiaria ir madinga praktikoje, nes žymiai padidina programų kūrimo greitį ir, savaime suprantama, kaštus. Atsisakius nenaudingų programinės įrangos kūrimo stadijų sutaupoma daug laiko, tačiau tai išsprendžia tik dalį problemų. Problemos iškyla, kai po pirmos versijos išleidimo programą kūrusi komanda būna išformuojama, ir darbą turi tęsti kiti. Turint tik programos kodą ir testus programos tolimesnė priežiūra tampa neįmanoma

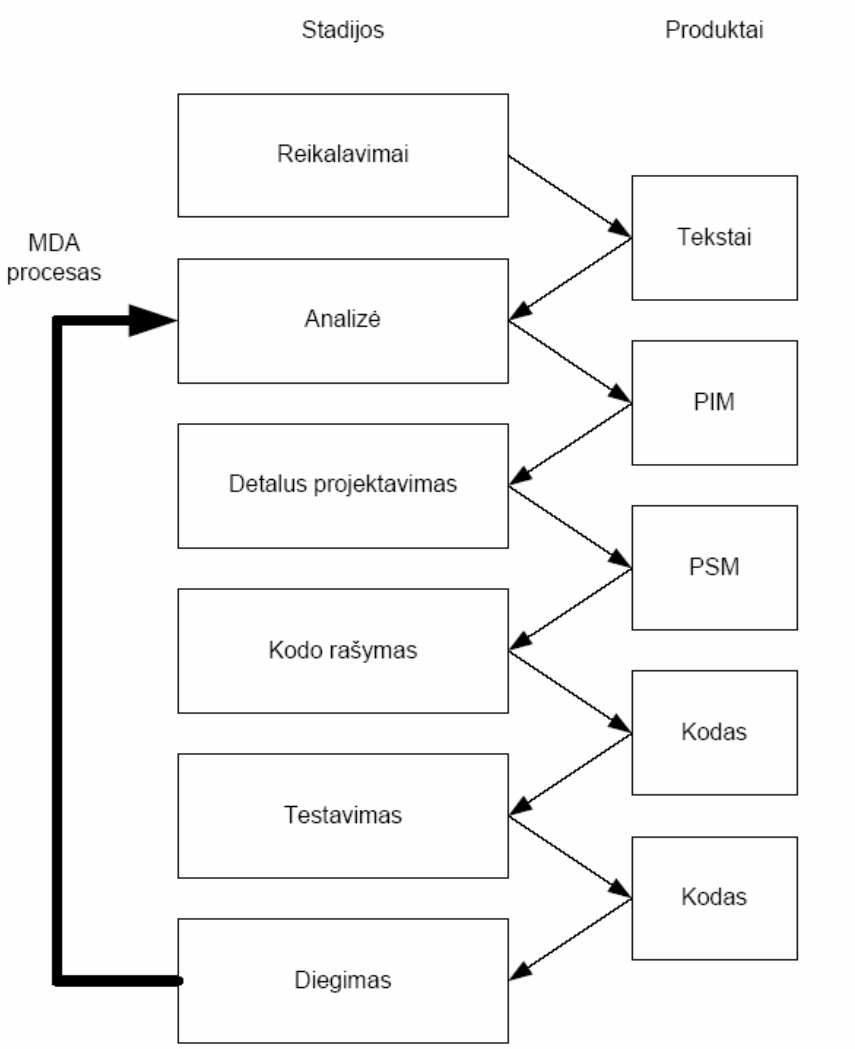


1 pav. Tradicinis programos kūrimo ir gyvavimo ciklas

Kita problema su kuria susiduria programinės įrangos pramonė – tai sparti technologijų kaita. Dar net dalinai neatsipirkus sukurtam produktui atsiranda naujos technologijos arba pasikeičia esamos. Dauguma programavimo priemonių ir bibliotekų kūrėjų palaiko tik paskutines tris versija, todėl neatnaujinus sistemos – rizikuojama likti be palaikymo.

2.2 MDA programų kūrimo procesas

MDA – tai aplinka (*framework*) programų kūrimui, pasiūlyta OMG konsorciumo, kuri turėtų išspręsti bent dalį anksčiau paminėtų programavimo proceso problemų ir suteikti naujas programinių sistemų atnaujinimo ir integravimo galimybes. MDA ir tradicinės programinės įrangos kūrimo stadijos yra labai panašios, tačiau gaunami skirtingi stadijų produktai (žr. 2 pav.).



2 pav. MDA programos kūrimo ir gyvavimo ciklas

MDA pasiūlyta naujovė yra aukštesnio abstrakcijos lygio nuo platformos nepriklausomas modelis PIM (*Platform Independent Model*), skirtas abstrakčiai

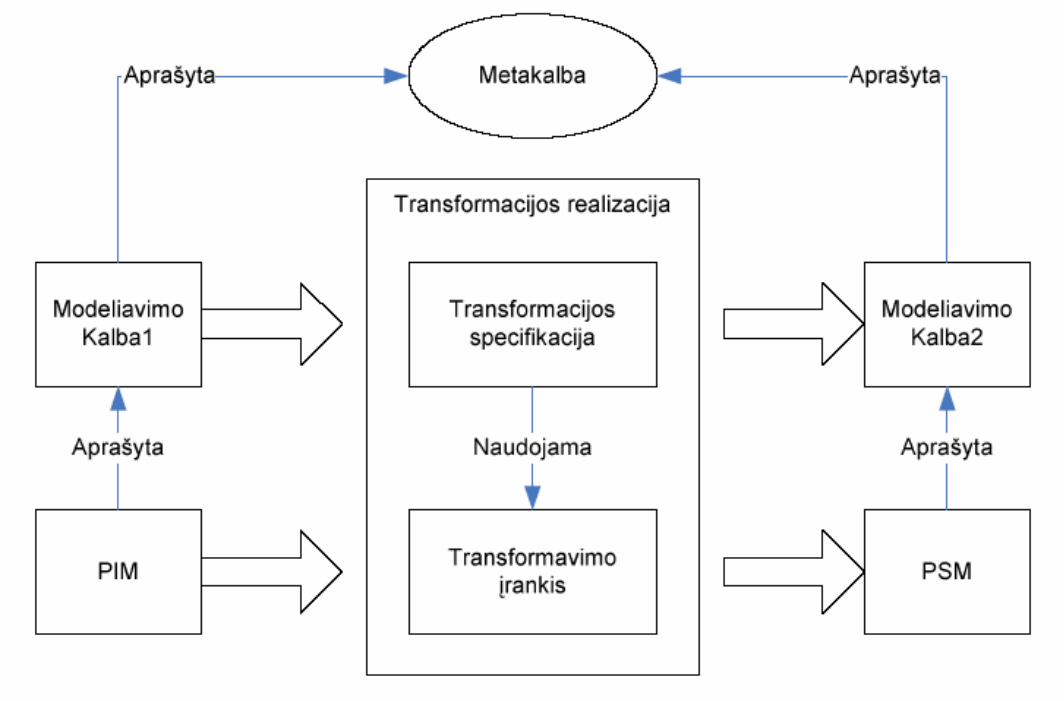
pavaizduoti esminius sistemos elementus, veikimo logiką ir probleminę sritį, neįtraukiant techninių detalių. Detaliai techninei informacijai vaizduoti yra skirtas kitas: nuo platformos priklausantis modelis PSM (*Platform Specific Model*),

2.3 Meta modelių reikšmė transformacijoms

Modelis – tai sistemos arba jos dalies aprašymas formalia modeliavimo kalba. Meta kalbomis vadinamos formalios kalbos naudojamos formalių kalbų aprašymui. Kurti tokiems formaliems meta kalbų it meta duomenų aprašymams OMG konsorciumas sukūrė save aprašančių MOF kalbą, kuri turėjo didelę įtaką kuriant esminius MDA principus.

Priežastys dėl kurių meta modeliavimas yra svarbus modelių transformavimui ir visai MDA architektūrai:

1. Yra reikalinga kalba, kuria būtų galima aiškiai specifikuoti kitas modeliavimo kalbas, kurios vėliau bus naudojamos aprašant PIM ir PSM modelių kalbas
2. Išeities ir tikslo meta modelių apibrėžimas labai palengvina transformacijos taisyklių kūrimą.



3 pav. MDA platforma ir meta modeliai

Modelių transformacijos vykdomos tokia tvarka:

1. Naudojant meta kalbą reikia specifikuoti formalią PIM modeliavimo kalbą, aprašant jos meta modelį.
2. PIM meta modelio pagrindu yra sukuriamas reikiamas sistemos PIM modelis
3. Transformacijos tikslas yra PSM modelis, todėl turime specifikuoti ir PSM meta modelį.
4. Kai jau yra sukurti PIM ir PSM meta modeliai , bei PIM modelis lieka tik aprašyti transformaciją tarp jų

Galime padaryti išvadą, kad modelių transformacijos yra glaudžiai susijusios su meta modeliavimu. Modelių transformacijos yra pačiame MDA architektūros centre. Transformacijų automatizavimas išskiria MDA architektūrą, todėl yra labai svarbu sukurti tinkamus įrankius transformacijoms iš aprašyti ir vykdyti.

2.4 Modelių transformacijų specifikavimo būdai

Transformavimas gali būti išskirtas į dvi pagrindines dalis: transformacijos specifikaciją ir transformavimo įrankį (žr. 3 pav.). Pasitaiko, kad abi šios dalys sudaro bendrą sistemą. Modelių transformacijos gali būti užrašomos trimis pagrindiniais būdais:

1. **Imperatyviai** – naudojant įvairias algoritmines kalbas, tiksliai ir pažingsniui nusakant transformacijos vykdymo eigą.

2. **Deklaratyviai** – kai apibrėžiamos tik atskirų modelių susiejimo taisyklės, o transformavimo įrankis atsako už jų vykdymo eiliškumą.

3. **Mišriai** – kai kartu naudojami abu ankščiau išvardinti būdai.

Vienas iš mano darbo tikslų yra realizuoti eksperimentinę modelių transformaciją, naudojant dvi skirtingas kalbas. Tikslui įgyvendinti pasirinkau:

- **hibridinę ATL** (*Atlas Transformation Language*) transformavimo kalbą

- **imperatyvią MTL** (*Model Transformation Language*) transformavimo kalbą

Toliau savo darbe detaliau apžvelgsiu ATL ir MTL transformavimo kalbas, daugiau dėmesio skirdamas ATL, nes ji yra patrauklesnė ir įdomesnė savo struktūra ir sintakse.

.

3. ATL transformavimo kalba

ATL (*Atlas Transformation Language*) yra modelių transformavimo kalba skirta MDA (*Model Driven Architecture*) ir atitiksianti būsimą QVT-RFP (*Queries Views Transformations – Request For Purposal*) standartą. Jos abstrakti sintaksė yra specifikuota MOF (*Meta-Object Facility*) meta modeliu ir jį atitinkančia konkrečia tekstine sintakse. Egzistuoja ir papildoma grafinė sintaksė, skirta daliniam ATL transformavimo taisyklių atvaizdavimui. Šiame skyriuje bus aprašoma dabartinė ATL versija, kuri vis dar vystosi OMG QVT ir funkcionalumo didinimo linkme.

Po trumpos įžangos apibūdinančios užklausų, vaizdų ir transformacijų valdymą ATL kalboje, detaliau panagrinėsime jos abstrakčią sintaksę. Bus pristatoma ir analizuojama tekstinė ir grafinė kalbos sintaksės.

3.1 Užklausos

ATL kalboje užklausos yra OCL (*Object Constrain Language*) išraiškos, grąžinančios primityvias reikšmes (*Boolean, String, Integer, ...*), modelių elementus, kolekcijas arba bet kokias jų kombinacijas.

Užklausa gali būti vykdomos tarp modelio elementų, leidžiama atlikti ir atskiras užklausų operacijas. Operacijos gali būti aprašomos meta modelyje arba pačioje užklausoje. Operacijų, skirtų užklausoms, sukūrimas atskiriems modelio elementams (naudojant OCL konstruktyvus su <<definition>> stereotipu, kaip aprašyta [11]) suteikia papildomas galimybes, tokias kaip: rekursija, modelių lankytojų (*visitors*) sukūrimas

3.2 Atvaizdai

Tai specifinis transformacijų atvejis, tačiau keletas papildomų transformavimo kalbos savybių galėtų padaryti juos daug naudingesniais:

- Didėjantis (*incremental*) transformacijų palaikymas leistų atnaujinti atvaizdą iš šaltino (*source*), nereikalaujant pakartotino visos transformacijos vykdymo.

- Dvikryptiškumas (*bidirectionality*) galėtų būti naudojamas apibrėžiant kintančius vaizdus ir perduodant pokyčius pradiniams jų modeliams.

Nei viena iš ankščiau paminėtų funkcijų šiuo metu dar nėra realizuota ATL kalboje, bet panašių rezultatų būtų galima pasiekti panaudojant susekamumo (*traceability*) informaciją, generuojamą transformacijos vykdymo metu.

3.3 Transformacijos

ATL transformavimo modelis leidžia išeities (*source*) modelių aibę paversti tikslo (*target*) modelių aibe. Norint vykdyti transformaciją, turi egzistuoti ir būti prieinami kiekvieno modelio (išeities arba tikslo) meta modeliai. ATL supranta visus modelius, apibrėžtus MOF meta modelių standartu. ATL galimybės leidžia transformuoti UML modelį į MOF meta modelį, arba vykdyti užklausas kiekvienam MOF meta meta modelio konstruktyvui. Navigacija tarp modelių yra aprašoma OCL.

3.4 Abstrakčios sintaksės aprašymas

3.4.1 Navigacija:

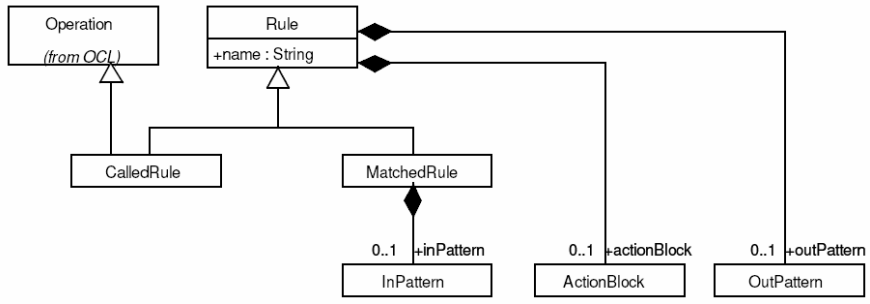
Navigacija galima tik pilnai inicijuotose elementuose, todėl tikslo elementas gali būti pilnai sukurtas tik transformacijos vykdymo pabaigoje. Todėl navigacija ATL kalboje gali būti vykdoma tik tarp išeities modelio ir jo meta modelių elementų. Jei navigacija per tikslo elementus yra reikalinga, tai turi būti daroma kitoje transformacijoje panaudojant pirmosios rezultatus.

3.4.2 Funkcijos ir Operacijos:

OCL leidžia kurti operacijas modelio elementams. ATL turi visas šias galimybes, leisdama modeliuotojui kurti operacijas išeities meta modelio elementams ir pačiam transformacijos modeliui.

3.4.3 Transformavimo taisyklės:

ATL kalboje egzistuoja skirtingos transformavimo taisyklių rūšys, skirstomos pagal jų užrašymo būdą ir grąžinamus rezultatus. Dalis ATL meta modelio, aprašančio taisykles pateikiama 4 paveiksle.

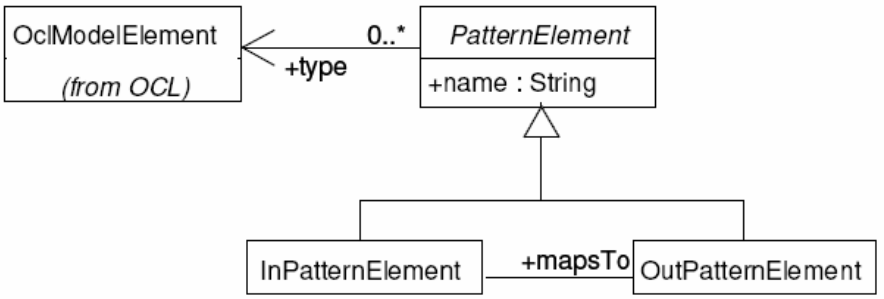


4 pav. Taisyklės ATL kalboje

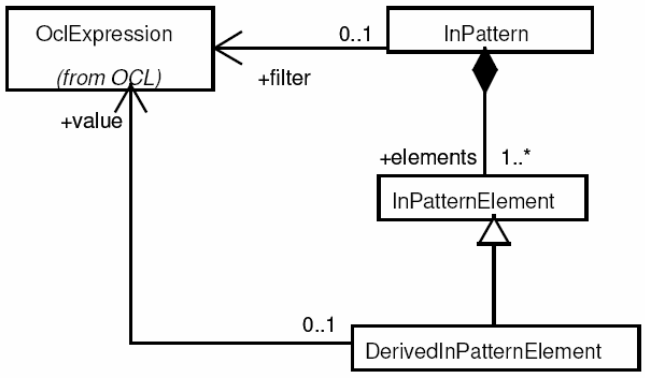
Taisyklė gali būti vykdoma nurodant jos pavadinimą ir reikiamus parametrus (*Called Rule*), arba iškviečiama kaip metodo inPattern atpažinimo rezultatas tikslo (target) modelyje (*Matched Rule*). Operacijos vykdymo rezultatas gali būti: aprašomas naudojant OutPattern, realizuotas imperatyvioje dalyje, arba abiem šiais būdais kartu.

Taisyklės su inPattern ir outPattern vadinamomis **deklaratyviomis** (*declarative*) (o taisyklės visiškai neturinčios imperatyvios sekcijos – **pilnai deklaratyviomis** (*fully declarative*). Taisyklė turinti vardą, formalius parametrus, imperatyviąją sekciją ir be outPattern vadinama **procedūra** (*procedure*)**.** Kitos kombinacijos vadinamos **hibridinėmis** (*hybrid*) taisyklėmis.

Todėl ATL galima vadinti hibridine kalba. OMG QVT [7] teigiama, kad tokios kalbos pagrindas yra aprašomas naudojant deklaratyvų priėjimą (*declarative approach*) taisyklėms parinkti, kuriose imperatyviai nusakoma kaip tiksliai turi būti atlikti veiksmai. ATL kalba turi tokias taisykles, bet leidžia naudoti ir pilnai deklaratyvias arba imperatyvias.



5 pav. Šablono elementai



6 pav. Išeities šablonai

3.4.4 Išeities šablonas (*Source pattern*)

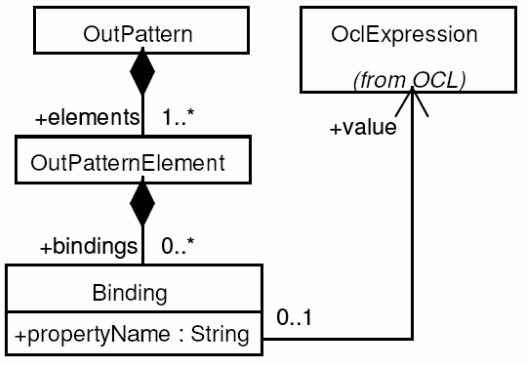
ATL pritaikyta (*matched*) taisyklė aprašo išeities šabloną (*source pattern*) (arba *inPattern*, arba *input pattern*) kaip aibę tipų paimtų iš išeities meta-modelių priskiriant jiems vardus ir esant poreikiui papildomai filtruotų naudojant OCL logines išraiškas. Šis filtras pasiekia įėjimo elementus (*in elements*) per jų vardus ir gražina teigiamą rezultatą (*true*), jei atitinkama aibė yra priimtina taisyklei. Išeities šablonas taip pat yra ir išeities meta modelio mazgų (*nodes*) aibė, turinčių specifinius ryšius, kurios patikrina filtras. ATL meta modelį atitinkantis vaizdas pateiktas 6 paveiksle, o šablono elementai apibrėžiami 5 paveiksle.

Specialus atvejis gaunamas tada, kai visi elementai atpažinti taisyklės inPattern dalyje priklauso tam pačiam modeliui. Dažnai pasitaiko, kad pografis (*sub graph*) suformuotas iš šių elementų gali ir dažnai būna susijęs, tai reiškia, kad bet kokia informacija gauta navigacijos metu iš vieno elemento, gali būti gaunama ir iš bet kurio kito. Tai galėtų

reikšti, kad tokiais atvejais papildomi elementai tampa nereikalingi ir užtenka pasilikti tik vieną. Bet reiktų įvertinti keletą papildomų sąlygų:

- Kai kurios navigacijos išraiškos gali būti patogiau sudarytos iš vieno mazgo, nei iš kitų.
- Kadangi transformavimo programa turi išanalizuoti visus galimus šablonus ir pritaikyti jiems diskriminatorių, skaičiavimų rezultatai gauti iš sutrumpintų navigacijos išraiškų, gali būti prarasti.
- Jei taisyklė sukuria keletą skirtingų elementų iš vieno turimo, kiekvienas naujai sukurtas elementas turės būti priskirtas jai. Kartais gali būti naudinga susieti dalį tikslo elementų su kitais specifiniais išeities šablonais.

ATL siūlomas sprendimas leidžia aprašyti išeities elementų paveldimumą. Tokiu būdu, jei du šaltinio elementai yra sujungti, vienas iš jų yra išskaičiuojamas navigacijos būdu, bet vis tiek gali būti susietas su specifiniu tikslo elementu.



7 pav. Tikslo šablonai

3.4.5 **Tikslo šablonas** (*target pattern*)

Tikslo šablonas (kitaip dar *outPattern* arba *output pattern*), kaip pavaizduota 7 paveiksle, yra tipų aibė paimta iš tikslo meta modelių priskiriant jiems vardus ir sąryšius (*bindings*). Vykdant taisyklę su tikslo šablonu, sukuriami specifinių tipų tikslo elementai. Sąryšis (*binding*) nusako pradinę elemento reikšmę naudojamą inicijuojant elemento egzempliorių (*instance*). Sąryšis nusako kintamojo reikšmę naudojamą sukuriant egzempliorių.

Tikslo šablonas yra mazgų aibė, kurie gali būti tarpusavyje susieti sąryšiais .Tikslo modelio kraštai (*edges*), vaizduojami grafiškai, yra sukuriami panaudojant sąryšius ir gali būti nustatomi taisyklėje arba tarp taisyklių, naudojant „tikslo į išeities“ (*target-to-source*) sąsajos algoritmą, kurį detaliau nagrinėsime vėliau, kaip vykdymo semantikos dalį.

3.4.6 Imperatyvus blokas

ATL taisyklės imperatyvus blokas nusako instrukcijų seką, kurios bus vykdomos po outPattern (jei toks egzistuoja). Aprašymui naudojama kalba yra skirta būti suderinama tarp atitinkamų transformacijų, su Veiksmų Semantika (*Action Semantics*).
Pastaroji jau yra standartizuota. Egzistuoja tiesioginės sintaksinės projekcijos tarp ATL abstrakcijų ir konkrečių kalbų skirtų imperatyvioms instrukcijoms.

3.4.7 Taisyklių paveldimumas

Taisyklė gali papildyti kitą taisyklę. Naujoji taisyklė gali aprašyti papildomus elementus arba apribojimus ankstesniajai. Apribojimai gali būti aprašomi naujų filtrų forma (kurie būtų logiškai pridedami prie ankstesniųjų) arba iš naujo priskiriant tikslo elementą su tipu praplečiančiu senąjį. Lygiai taip pat galima nurodyti ir papildomus tikslo elementus bei sąryšius.

3.4.8 Abstrakčios taisyklės

Abstrakčia vadiname taisyklę, kuri negali būti vykdoma, kol nėra papildoma.

3.4.9 **Vykdymo semantika** (*execution semantics*)

ATL transformacijos modelis yra vykdomas keliais žingsniais. Jei modelis yra duotas tekstiniame formate, jis pirmiausia turi būti analizuojamas (*parsed*) ir verčiamas į modelį apibrėžtą ATL meta modeliu. Šis modelis vėliau yra statiškai tikrinamas ieškant semantinių klaidų lyginant su ATL meta modeliu ir išeities bei tikslo meta modeliais. Sekantis žingsnis gali būti interpretavimas arba kompiliavimas. Abiem atvejais ATL transformacijos taikymas vykdomas pagal toliau šioje dalyje aprašomą semantiką.

Jei egzistuoja iškviečiamoji (*called*) taisyklė pažymėta kaip įeities taškas (*entry point*), kuris gali būti tik vienas, tada ji yra vykdoma pirmiausiai. Ši taisyklė taip pat gali pritaikyti neribotą skaičių kitų iškviečiamų taisyklių iki pabaigos.

Vykdant pritaikomą taisyklę (*matched rule*), pirmiausia yra sukuriami išeities elementai. Su kiekviena taisykle yra tikrinamos visos elementų kombinacijos, atitinkančios jų inPattern tipus ir filtrus. Jei grąžinamas rezultatas yra teigiamas (*true*), vadinasi šablonas buvo atpažintas ir taisyklė tinka konkrečiai elementų aibei.

Kiekvieną kartą atpažinus deklaratyvią taisyklę yra sukuriami jos tikslo elementai. Tai yra daroma elementariai sukuriant visus elementus iš jos outPattern. Tuo metu yra sukuriami vykdomieji (*run-time*) ryšiai tarp taisyklės, atpažintų ir naujai sukurtų elementų. Šis ryšys susieja vieną išėjimo elementą su kiekvienu įėjimo elementu ir tampa numatytu (*default*) elementu numatomam „tikslo į išeities“ sprendimui. Gautasis šaltinio elementas negali būti priskirtas daugiau negu vienam inPattern, nes kitu atveju gautumėme vykdymo klaidą. Šių klaidų negalima aptikti statinės analizės metu, nebent bendru atveju tikrintumėme, ar kai kurie filtrai kartu negrąžina teigiamo rezultato (*true*).

Po to pritaikant sąryšiai, sukuriami visi išėjimo elementai. Priklausomai nuo reikiamos nustatyti reikšmės tipo ir daugialypumo (*multiplicity*) gali būtu taikomi įvairūs veiksmai:

- Jei tipas primityvus (*String, Integer, Boolean, Double*) ir daugialypumo viršutinė riba lygi 1, kairiosios dalies skaičiavimo rezultatas yra tiesiog priskiriamas kintamajam (kuris yra atributas).

- Jei tipas yra kompleksinis (pvz. Meta modelio klasė) ir daugialypumo viršutinė riba lygi 1, kairysis sąryšio operandas turi būti transformuotas į vieną iš išeities modelio elementų (navigacija tikslo modelyje yra draudžiama). Reikšmė, kuri bus priskiriama kintamajam, negali būti šio modelio elementas nesantis tame pačiame modelyje kaip ir kintamojo šeimininkas (*owner*). Nepaisant to, šiame taške gali egzistuoti nuoroda tarp išeities elemento ir kai kurių tikslo elementų, jei jie priklauso pritaikytosis taisyklės (*matched rule*) (tos pačios arba kitos) pografiui.

Jei taip nėra, gauname vykdymo klaidą, kadangi sąryšis negali tinkamai inicijuoti kintamojo. Jei yra, tai numatytas elementas susietas vykdymo ryšiu su išeities elementu yra naudojamas kintamojo reikšmei nustatyti. Jei yra nurodomas kitas elementas, tai jis ir būna naudojamas vietoje numatytojo. Tai nusako „šaltinio iš tikslo“ (*target-from-source*) algoritmas.

- Jei daugialypumo viršutinė riba didesnė už 1, turime du pasirinkimus
  - Kairysis sąryšio operandas tampa vienu elementu (primityviu arba kompleksiniu), kurio tipas atitinka kintamojo tipą (kitu atveju jis būtų pažymėtas kaip klaidingas statinės analizės metu). Elementas būna įkeliamas į kintamojo elementų kolekciją
  - Kairiajam operandui yra priskiriama OCL elementų (primityvių arba kompleksinių) kolekcija, kurios tipas atitinka kintamojo tipą. Kolekcijos dydis turi atitikti daugialypumą, kitu atveju gausime vykdymo perspėjimą (kuris vėliau galės būtu pataisytas imperatyvioje dalyje). Kiekvienas elementas įkeliamas į kintamojo elementų kolekciją

Vėliau vykdoma pritaikytos (*matched*) taisyklės imperatyvioji dalis. O pabaigoje - iškviečiamoji taisyklė (*called rule*) su žyme *endpoint*, jei tokia yra. Svarbu žinoti, kad taisyklės su outPattern nėra vykdomos kaip pritaikytos. Tikslo elementai yra paprasčiausiai inicijuojami prieš imperatyviosios dalies vykdymą, bet nėra automatiškai susiejami su jokiais išeities elementais.

3.4.10 Atspindėjimas (*reflection*)

Transformacijos vykdymo metu išeities modeliai bei išeities ir tikslo meta-modeliai būna prieinami. Taip pat prieinamas yra ATL meta modelis ir šiuo metu vykdomos transformacijos modelis. Taisyklės ir pilni modeliai gali būti prieinami per savo vardus.

3.4.11 Atsekamumas ATL kalboje (*traceability)*

ATL kalboje atsekamumas realizuojamas gyvendintas išsaugant transformacijos vykdymo darbinę informaciją.

3.5 ATL tekstinė sintaksė

3.5.1 Deklaratyvios konstrukcijos

ATL turi apibrėžtą sintaksę atitinkančią jos meta modelį , kad būtų galima atvaizduoti transformacijos medelį tekstiniu pavidalu.

Tipų pavadinimai rašomi pridedant jų meta-modelių vardus, siekiant išvengti konfliktų tose situacijose, kai yra naudojama daugiau išeities meta-modelių. Pavyzdžiui, jei turime išeities meta-modelį aprašytą UML kalba, ir norime sukurti išeities šabloną, kuris pritaikytų taisyklę kiekvienai porai sudarytai iš klasės ir vieno iš jos atributų, turėtumėme užrašyti taip:

```
from
  c : UML!Class,
  a : UML!Attribute
  (a.owner = c)
```

Šis šablonas atpažįsta du mazgus, Klasę (*Class*) ir vieną iš jos Atributų (*Attribute*). Šioje vietoje iškyla viena problema, susijusi su transformacijos vykdymo greičiu, kadangi pagal šį šabloną programa turėtų palyginti kiekvieną klasės ir atributo porą pagal filtrą. Šios problemos galima nesunkiai išvengti pažymint klasę paveldėt (*derived*).

```
from
   a : UML!Attribute,
   derived c : UML!Class = a.owner
```

panašiai aprašomas ir tikslo šablonas. Surišimui naudojamas operatorius „<−“. Štai kaip aprašome UML klasės ir vieno iš jos atributų egzemplioriaus sukūrimą:

```
to
  c : UML!Class,
  a : UML!Attribute (owner <- c)
```

Deklaratyvios taisyklės aprašomos susiejant išeities šabloną su tikslo šablonu. Paprasta UML klasės projekcija į reliacinės duomenų bazės meta-modelį (RDBMS) galėtų būti aprašoma taip:

```
rule Class2Table {
```

```
  from class : UML!Class
  to
    table : RDBMS!Table
      mapsTo class (
      name <- class.name
    ),
    pk : SimpleRDBMS!Key (
      name <- class.name,
      owner <- class,
      column <- class.attribute->select(e|e.kind = `primary`)
   )
}
rule Attribute2Column {
  from attr : UML!Attribute
  to
    col : RDBMS!Column mapsTo attr (
      name <- a.name,
      owner <- a.owner
    )
}
```

Raktinis žodis *mapsTo* nurodo kuris tikslo elementas yra susietas su kuriuo išeities elementu (asociacija nėra privaloma). Tuo atveju, jei naudosime tikslo-iš-šaltinio sprendimo būdą, kuriame nėra nurodomas tikslo elementas iš kiekvieno konkretaus išeities elemento, reikės įrašyti tikslo elemento kintamojo vardą taisyklėje.

Pavyzdžiui:

```
rule Association2ForeignKey {
  from asso : UML!Association
  to
  fk : RDBMS!ForeignKey
  mapsTo asso (
    refersTo <- [Class2Table.pk] ia.destination,
    owner <- ia.source
  )
}
```

Vykdymo klaida įvyksta tada, kai išeities elemento nepavyksta transformuoti pagal turimą taisyklę.

Problema gali iškilti ir kai keletas taisyklių generuoja tą patį tikslo elementą iš išeities elemento. Ji išsprendžiama paveldint taisykles iš vienos sukurtos taisyklės su abstrakčiu išvedimo elementu.

3.5.2 Imperatyvios instrukcijos

Neprivalomas imperatyvus taisyklės blokas yra sudarytas iš instrukcijų sekos, vykdomų griežtai nustatyta tvarka. Egzistuoja keletas šių instrukcijų rūšių, kurias pristatysime vėliau.

3.5.2.1 Išraiškos

OCL kalbos išraiškos gali būti naudojamos kaip instrukcijos. Paprastų užklausų atveju tokios išraiškos būtų nenaudingos, bet ne užklausos (non-query) operacijų, tokių kaip imperatyvios taisyklės vykdymas yra leidžiamas.

3.5.2.2 Kintamieji

Aprašant kintamuosius yra naudojama OCL sintaksė:

```
let varName : varType = initialValue;
```

Kintamieji yra tipizuoti ir turi būti aprašyti prieš pirmą panaudojimą.

3.5.2.3 Priskyrimas

Surišimo operatorius <- su ta pačia semantika (įskaitant ir automatinį tikslo-iš-šaltinio išrišimą) yra naudojamas ir kaip priskyrimo operatorius. Galima naudoti ir paprastą priskyrimo operatorių := kai nereikalingas automatinis išrišimas, tada kairiajam operandui būna priskiriama dešiniojo reikšmė.

Priskyrimas yra instrukcija, bet ne išraiška.

Tokia instrukcija būtų neleistina:

```
myFunction(myVar <- 'a string');
```

3.5.2.4 Egzempliorių valdymas

Deklaratyviose taisyklėse egzemplioriai yra kuriami automatiškai, o imperatyviajame bloke galima kurti arba naikinti objektus rankiniu būdu

Egzemplioriai kuriami naudojant operatorių **new**:

```
let myClass : UML!Class = new UML!Class();
```

Naikinant egzempliorių naudojamas operatorius **delete**:

```
delete myClass;
delete myClass.contents->select( e|e.oclIsTypeOf(UML!Attribute) );
```

Operacijos delete parametrai turi būti tikslo elementas arba jų kolekcija (šiuo atveju naikinami visi kolekcijos elementai). Reikėtų žinoti, kad pati kolekcija negali būti visiškai ištrinta, kadangi yra tik vykdymo metu egzistuojantis egzempliorius, neišliksiantis jokiame tikslo modelyje. Tokie egzemplioriai dažnai kuriami naudojant OCL (pvz.: ciklo iteratoriai), todėl šioje vietoje galėtų egzistuoti mechanizmas panašus į šiukšlių kolektorių (*garbane collector*).

3.5.2.5 Sąlygos

OCL kalba leidžia naudoti konstrukciją `if-then-else`, tačiau tai tik išraiška, panaši į JAVA arba C++ kalbose naudojamą trinarį operatorių (*ternary operator*):

```
condition ? if-true : if-false;
```

ATL kalboje sąlygos sakinį **`if`** galima užrašyti ir panašia į JAVA ar C forma:

```
if(condition1) {
-- if condition1
} else if(condition2) {
-- if (not condition1)
-- and condition2
} else {
-- if (not condition1)
-- and (not condition2)
}
```

Komutatorius **`switch`** ATL kalboje yra gerokai praplėstas (lyginant su JAVA ar C), nes vykdymo sąlygas leidžiama užrašyti ne tik skaliariniais dydžiais:

```
switch(expression)
{
  case expression1:
  -- if (expression = expression1)
    break;
  case expression2:
  -- if (expression <> expression1)
  -- and (expression = expression2)
    break;
  default:
    -- if (expression <> expression1)
    -- and (expression <> expression2)
    break;
}
```

3.5.2.6 Ciklai

ATL kalboje egzistuoja ciklai **`while`** ir **`do while`** , užrašomi panašiai kaip JAVA ir C:

```
while(condition) {
-- while condition is true
}
```

ir

```
do {
-- executed at least once and while
-- condition is true after that
} while(condition);
```

Ciklas, vykdomas kiekvienam kolekcijos elementui užrašomas taip:

```
foreach element in collection {
}
```

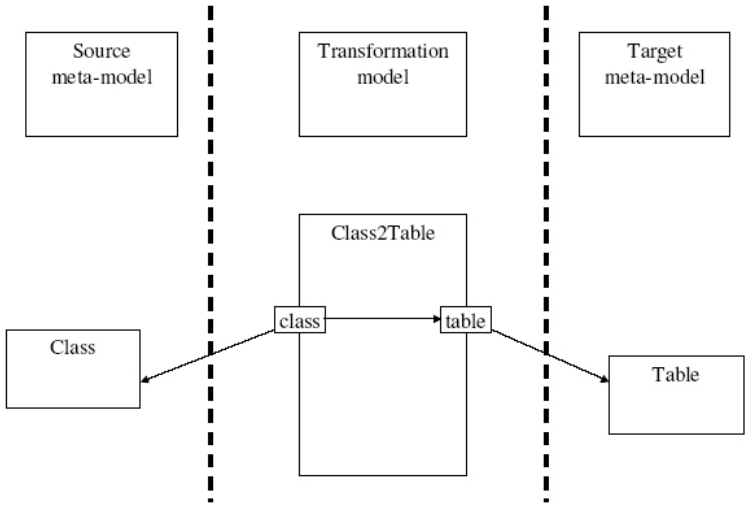
3.6 ATL grafinis atvaizdavimas

Transformavimo modelius ATL kalboje dalinai galime užrašyti ir grafiškai. Ne visos kalbos konstrukcijos turi grafinius simbolius, tik pagrindinės. Pradinė tokio atvaizdavimo egzistavimo mintis slypi poreikyje matyti panaudotus ir susijusius meta modelių elementus.

Vis dėl to geriausias būdas pavaizduoti OCL išraiškas yra standartizuota tekstinė sintaksė, nes negalime nupiešti nei išeities šablonų filtrų, nei tikslo šablonų sąryšių.
8 paveiksle matome, kaip grafiškai būtų pavaizduotos ankščiau užrašytos taisyklės. Grafike pavaizduoti elementų sudarančių atpažinimo šabloną (šiuo atveju `class`) tipai, tikslo elementų tipai (šiuo atveju `table`) ir ryšiai tarp tikslo ir išeities elementų.

Taisyklės kraštuose matomi ir kintamųjų, vykdymo metu saugančių nuorodas tarp elementų vardai, sujunti su reikiamais meta modelio elementais. Akivaizdu, kad `Class` bus transformuota į `Table`.

Ši sintaksė skirta padėti modeliuotojui susidaryti globalų konkrečios transformacijos vaizdą.



8 pav. Grafinės sintaksės pavyzdys

3.7 Dar keletas ATL kalbos savybių

3.7.1 Kryptingumas

ATL kalboje transformacijos yra vienkryptės. Jei sukurtumėme kalbą, kurioje visos transformacijos būtų dvikryptės, susidurtumėme su daugybe papildomų apribojimų. Bet jei transformacijos modelį sudarytumėme vien iš deklaratyvių taisyklių, teoriškai galėtumėme simetriškai sugeneruoti dalį atvirkštinės transformacijos kodo automatiškai, priklausomai nuo surišimui naudojamų išraiškų sudėtingumo.

3.7.2 Transformacijos vietoje (*In-place transformations*)

Tikslo modelis ATL kalboje visada yra naujas, bet egzistuoja sprendimas labai artimas transformavimui vietoje. Pirmiausia išeities modelį reikia nukopijuoti į tikslo modelį, o vėliau pritaikyti jam transformavimo taisykles.

3.7.3 Žingsninės transformacijos (*Incremental transformations*)

ATL kalboje nėra galimybių aprašyti žingsnių transformacijų, bet jei labai reiktų, tai būtų galima realizuoti panaudojant atsekamumą. Šioje srityje dar reiktų atlikti detalesnius tyrimus.

4. MTL transformavimo kalba

MTL (*Model transformation language*) – tai MOF technologija pagrįsta meta kalba. Pagrindinis jos tikslas yra suteikti modeliui įvykdomumą (*executability*). MTL kalbos koncepcija yra panašumas tarp objektinėse kalbose naudojamų klasių ir modelio meta klasių (žr. 1 lent.)

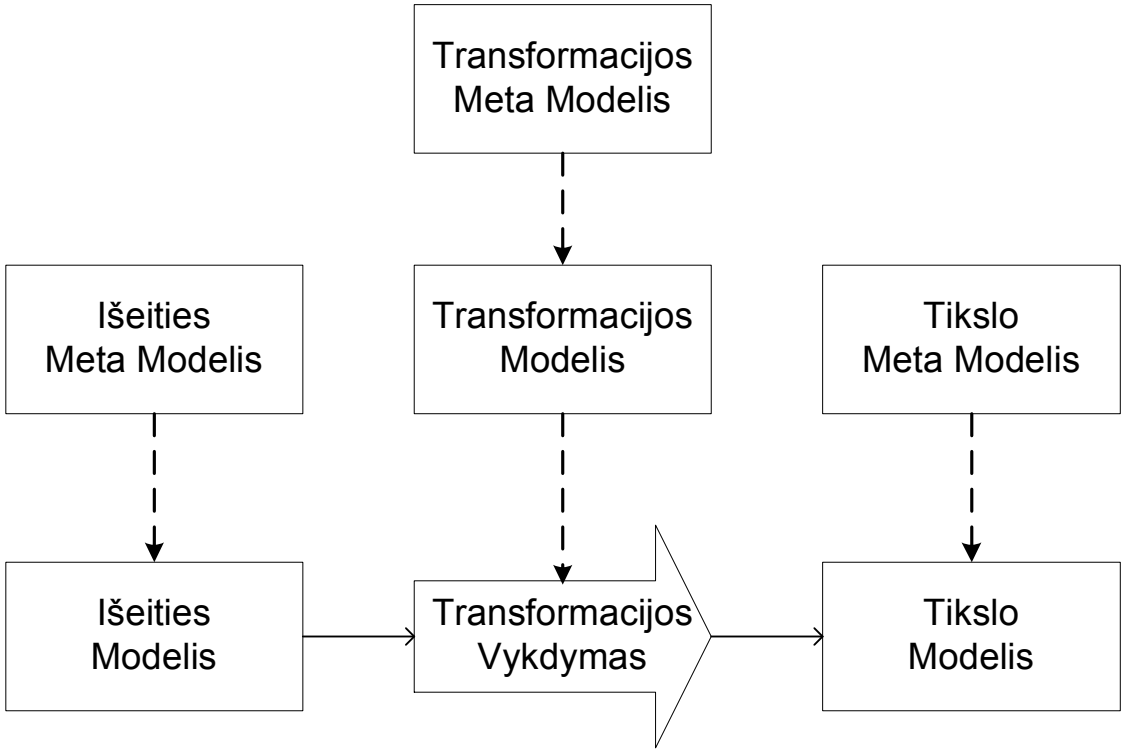
Ši kalba leidžia sukurti daug įvairių programų. Bet pagrindinė MTL kalbos taikymo sritis yra modelių transformacijų aprašymas.

Modelių transformavimo kalboms yra sugalvota daugybė galimų sprendimų, ir darosi sunku apjungti šias idėjas į vieningą standartą. Bet sprendimai šioje srityje reikalingi jau šiandien, todėl INRIA užsibrėžė tikslą realizuoti visas galimas modelių transformacijas MTL kalboje. O ateityje pasirodysiančią QVT kalbą bus galima tiesiog susieti su MTL jos pačios priemonėmis. MTL yra kuriama po truputį papildant BasicMTL kalbą naujomis funkcijomis ir sintakse.

1 lent. Modeliavimo ir MTL objektų atitikmenys

Modelio koncepcija	**MTL koncepcija**
Meta klasė	Klasė
Meta operacija	Operacija
Išplėtimas	Biblioteka
Meta atributas	Atributas
Meta sujungimas	Sujungimas
...	...

MTL yra objektiškai orientuota imperatyvi kalba skirta modelių transformavimui. Todėl MTL kalboje transformacijos yra aprašomos kaip programos naudojant klases, metodus atributus ir t.t. Kaip ir dauguma programavimo kalbų MTL yra kompiliuojama į binarinį kodą.



9 pav. MTL Transformacijos vykdymo principinė schema

Transformacijų vykdymo žingsniai MTL kalboje atitinka MDA architektūrą. (žr. 3,9 pav.) Transformacijos modelis ir vykdymas yra visiškai atskirti. Modelis užrašomas MTL kalba, o transformacijos vykdymu rūpinasi sugeneruota JAVA programa.

4.1 Baziniai tipai ir reikšmės

MTL kalba palaiko daugelį bazinių tipų ir reikšmių. Visi jie saugomi kaip objektai, su atitinkamu funkcionalumu. MTL kalboje nėra primityvių tipų (tokių kaip: *integer, boolean* ir *String*), viskas užrašoma objektais. Tačiau kai kurie standartiniai tipai gali būti paimti iš pagrindinės bibliotekos. Pagrindinėje bibliotekoje be jau anksčiau minėtų standartinių tipų dar yra kolekcijos ir iteratoriai.

4.2 Instrukcijos

MTL yra imperatyvi kalba, todėl visi veiksmai čia atliekami instrukcijų sekomis. Instrukcijomis vadinami veiksmai nuo elementaraus priskyrimo iki vykdymo valdymo , bei operacijų. MTL kalboje visi kintamieji yra tipizuoti, todėl prieš pirmą panaudojimą ar priskyrimą būtina nurodyti tipą. Programos vykdymo valdymas gali būti realizuojamas naudojant ciklus. MLT kalboje yra ir pagrindinės aritmetinės bei loginės operacijos skirtos darbui su primityviais tipais.

4.3 Operacijos

Jos sudaro pagrindinę transformavimo programos struktūra. MTL kalboje visos operacijos turi turėti unikalų pavadinimą klasėje. Parametrai operacijoms yra perduodami su reikšme (*by value*). Operacija grąžina rezultatą pažymėtą po žodelio „***return***”.

4.4 Klasės

Kaip ir bet kuri kita objektiškai orientuota kalba, MTL naudoja klases programos kodo struktūrizavimui. Klasė gali praplėsti (*extend*) kitas klases. Jas galima apjungti į paketus (*packages*). Klasės yra saugomos bibliotekose (*libraries*).

Pagrindinė MTL kalbos koncepcija yra suvienodinti klasių ir meta modelių naudojamą, todėl dauguma veiksmų skirtų MTL klasėms galima naudoti ir modelių meta klasėms. Šiuo metu MTL kalboje dar negalima naudoti konstruktorių, šiam tikslui yra kviečiamos iniciavimo funkcijos.

4.5 Modelių užkrovimas ir saugojimas

Modelių užkrovimas yra priklausomas nuo konkrečios saugyklos (*repository*). Naudojant modelių saugyklas iškyla nemažai problemų, pavyzdžiui užkraunant modelį reikia nustatyti jo naudojamus meta modelius, modelius, jų formatus ir pan. Tos pačios problemos iškyla ir norint išsaugoti modelį.

MTL leidžia minimizuoti šių su modelių saugyklomis susijusias transformavimo programos kodo dalių problemiškumą dviem būdais:

1. Vartotojui leidžiama perduoti modelį kaip specifinės saugyklos bibliotekos parametrą nepriklausomai nuo saugyklos bibliotekai.
2. Vartotojas gali praplėsti nepriklausomą nuo saugyklos biblioteką, papildydamas ją konkrečiai saugyklai reikalingais veiksmais.

Saugyklos tvarkyklė privalo turėti metodą grąžinantį reikiamą modelį. Ji taip pat turėtų pasirūpinti ir rezultato modelio išsaugojimu. MDR ir ModFact tvarkyklės jau siūlo panašias galimybes. Nepaisant to reikalinga informacija yra pateikiama tik modelių užkrovimo metu.

4.6 Navigacija

Navigacija po modelio elementus yra tokia pat paprasta, kaip ir objektuose, kiekvienoje objektiškai orientuotoje kalboje. Modelio užkrovimas ir saugojimas , prijungimas yra priklausomas nuo modelių saugyklos (*repository*). Dabartinėje MTL versijoje standartiškai naudojama MDR saugyklos tvarkyklė.

Paprasčiausias būdas pasiekti prijungto modelio įeities tašką (*entry point*) yra naudoti OCL funkcijas.

5. Eksperimentinis transformacijų vykdymas

Tyrimo vykdymui reikės sukurti išeities ir tikslo modelius bei realizuoti jų transformacijas ATL ir MTL kalbose. Eksperimentui naudosiu agregatinių sistemų PIM ir PSM modelius

Detalus atliekamų darbų planas būtu toks:

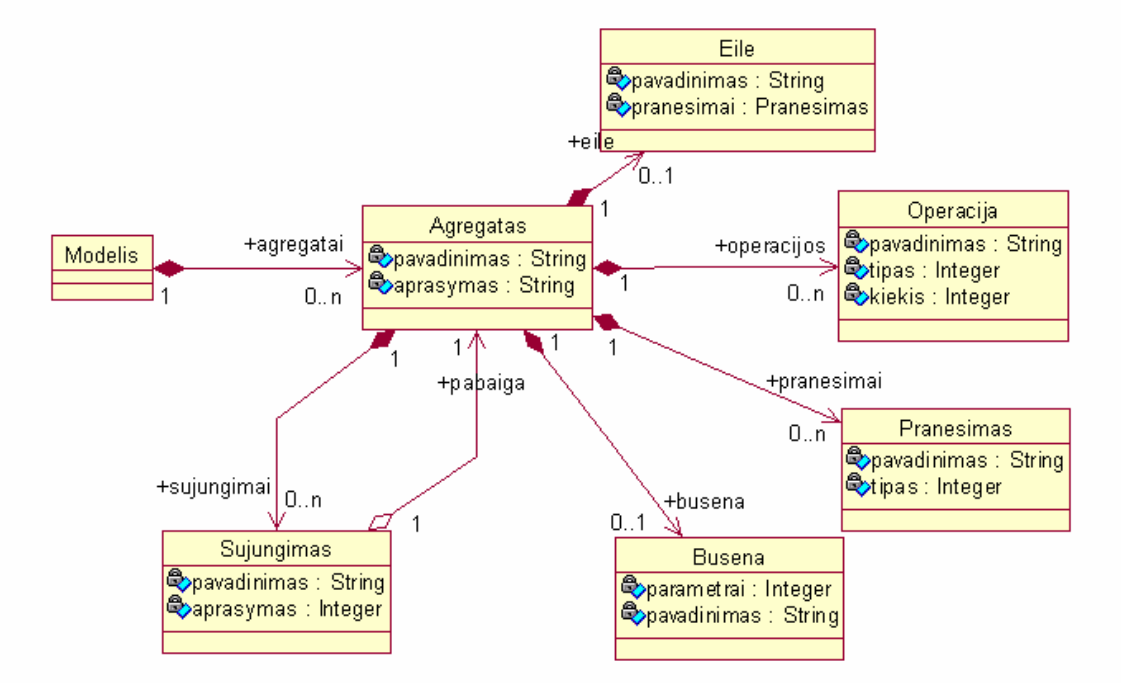
1. Specifikuoti agregatinių sistemų PIM meta modelį
2. Specifikuoti agregatinių sistemų PSM meta modelį jau egzistuojančiai bibliotekai
3. Sukurti bandomąjį PIM modelį
4. Aprašyti ir įvykdyti transformaciją ATL kalboje
5. Aprašyti ir įvykdyti transformaciją MTL kalboje

5.1 Agregatinių sistemų PIM meta modelio sudarymas

Agregatinės sistemos meta modelis buvo kuriamas naudojant UML modeliavimo kalbą. Sukurtas agregatinės sistemos PIM meta modelis, pavaizduotas 9 paveiksle. Meta modelis sudarytas iš 7 tarpusavyje susijusių klasių.

2 lent. PIM meta modelio klasės

Meta modelio klasė	**Aprašymas**
Modelis	Pagrindinis sistemos elementas, turintis neribotą skaičių agregatų, ir apjungiantis juos į bendrą modelį
Agregatas	Svarbiausias sistemos elementas
Operacija	Agregato viduje vykdomos operacijos su duomenimis
Busena	Bendra vieno agregato būsena, sudaryta iš jo vidinių parametrų visumos
Sujungimas	Agregatus tarpusavyje sujungiantis elementas
Pranesimas	Informacija tarp agregatų siunčiama pranešimais
Eile	Vidinis agregato buferis, skirtas pranešimams kaupti



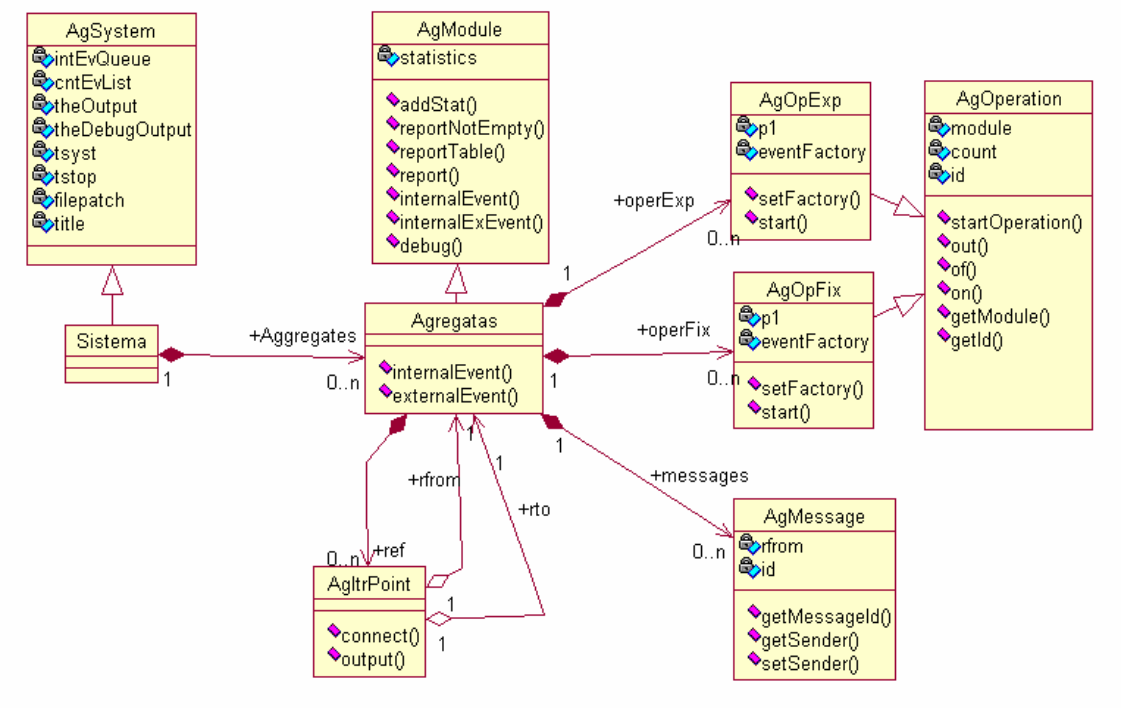
10 pav. Agregatinės sistemos PIM meta modelis

5.2 Agregatinių sistemų PSM meta modelio sudarymas

Šioje eksperimento stadijoje buvo sukurtas agregatinės sistemos PSM meta modelis (žr. 10 pav.), pagal jau egzistuojančią biblioteką. PSM meta modeliai dažnai būna daug sudėtingesni už PIM.

3 lent. PSM meta modelio klasės

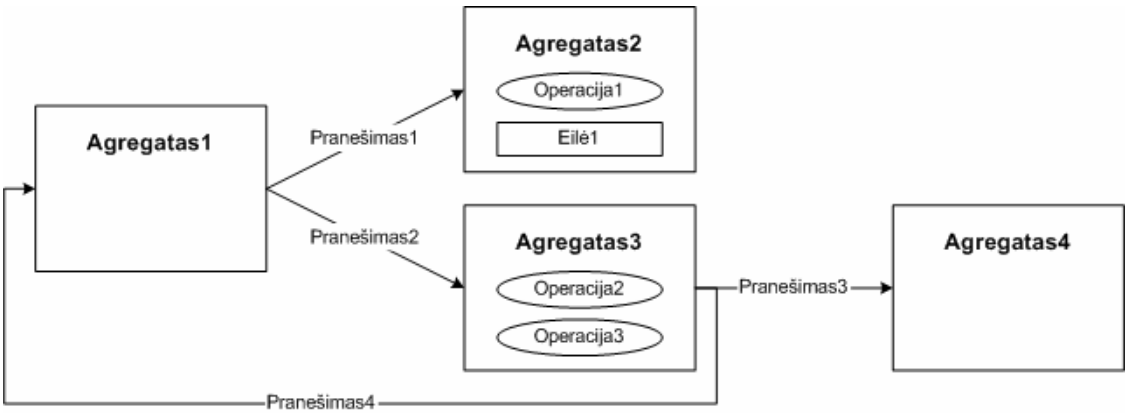
Meta modelio klasė	**Aprašymas**
AgSystem	Agregatinės sistemos bazinė klasė
Sistema	Agregatinės sistemos egzempliorius klasė
AgModule	Agregatų modelio klasė
AgOperation	Bazinė operacijos klasė
AgOpExp	Eksponentinė operacija
AgOpFix	Fiksuota operacija
AgItrPoint	Agregatų sujungimo elementas
AgMessage	Pranešimas



11 pav. Agregatinės sistemos PSM meta modelis

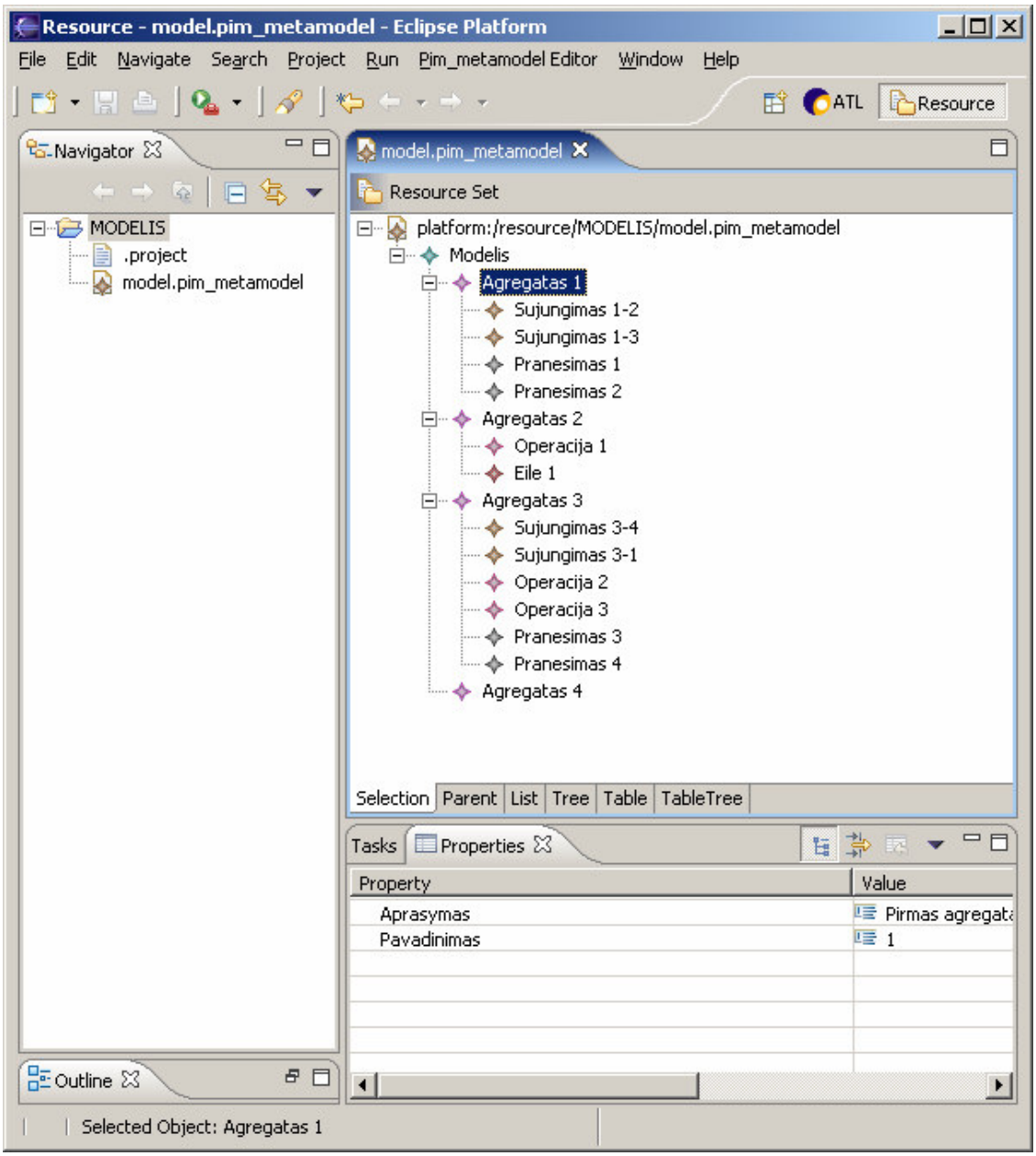
5.3 Agregatinės sistemos PIM modelio sudarymas

Naudojantis EMF (*Eclipse Modeling Framework*) priemonėmis iš ankščiau aprašyto PIM meta modelio UML schemos, buvo sukurtas grafinis PIM modelio redaktorius. Redaktoriumi leido greitai ir patogiai sukurti PIM modelį. 11 paveiksle matote eksperimente naudotą agregatinę sistemą, o 12 paveiksle pateikiu tos pačios sistema redaktoriaus vaizdą



12 pav. Agregatinės sistemos PIM modelis

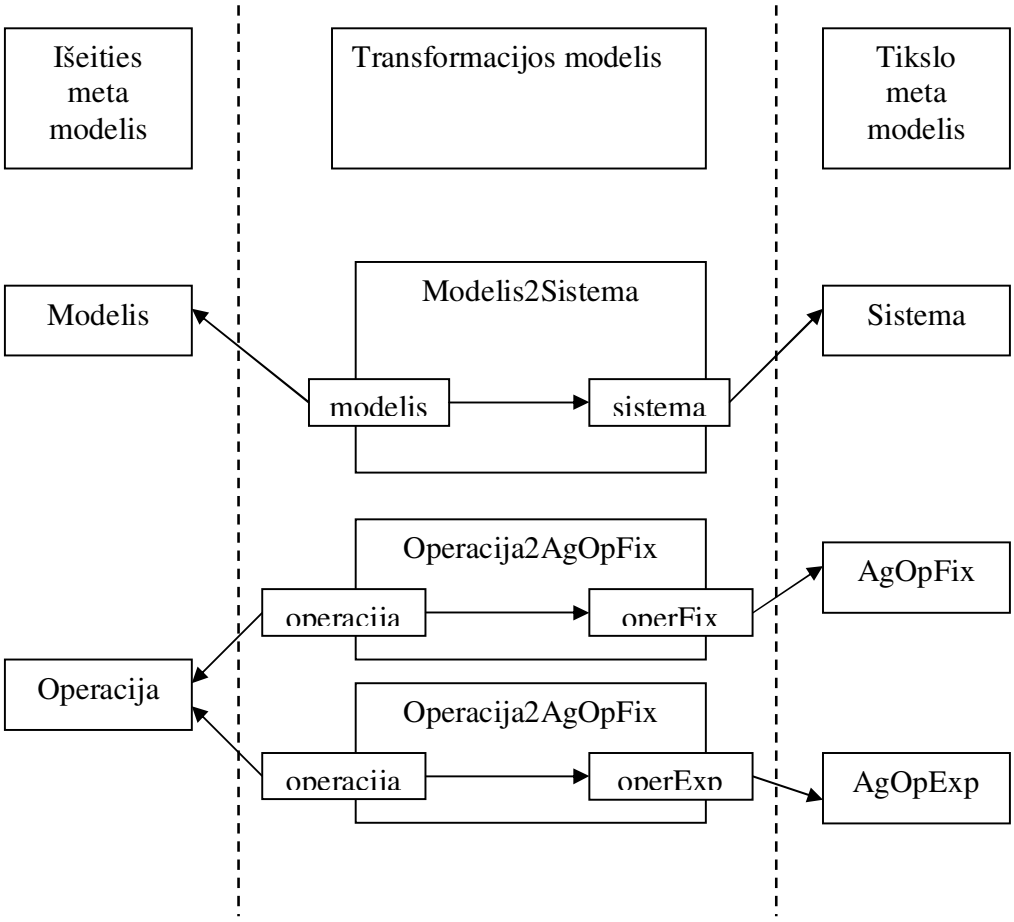
Redaktorius atvaizduoja visą XMI formate saugomo modelio egzempliorių kaip medžio struktūrą. Kiekviena medžio viršūnė atitinka tam tikros meta modelio klasės egzempliorių. PIM modelio XMI failas pateikiamas 2 priede.



13 pav. Agregatinės sistemos PIM modelio redaktorius EMF aplinkoje

5.4 Eksperimentinės PIM į PSM transformacijos ATL kalboje

Kai turime sukurtą PIM modelį ir PIM bei PSM meta modelius, galime pereiti prie svarbiausios šio darbo dalies – transformacijų aprašymo. ATL transformavimo kalba turi dalinę grafinę sintaksę, skirta taisyklėms ir pagrindiniams ryšiams tarp meta modelių elementų pavaizduoti.



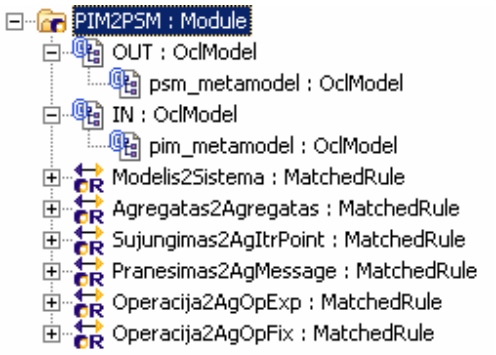
14 pav. Eksperimentinė transformacija pavaizduota grafine ATL sintakse

Bet grafinė sintaksė tinka tik dalinai atspindi transformavimo taisykles, tokiu būdu neįmanoma pavaizduoti vidinių filtrų. ATL vartotojo sąsaja yra integruojama į universalaus redaktoriaus *Eclipse* aplinką. ATL *Eclipse* redaktoriuje galime kurti transformavimo taisykles, nuosekliai tikrinti jų vykdymą (*debug*), gauti pranešimus apie pastebėtas sintaksės klaidas.

14 paveiksle pavaizduota schemos dalis ATL kalbos tekstinėje sintaksėje būtų užrašoma taip:

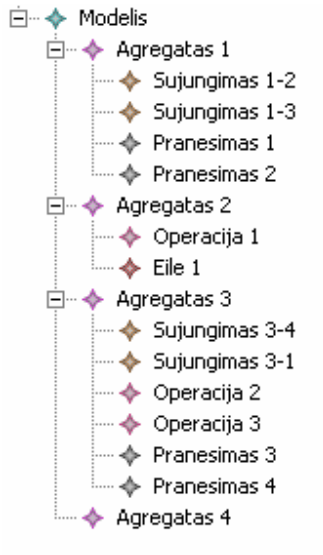
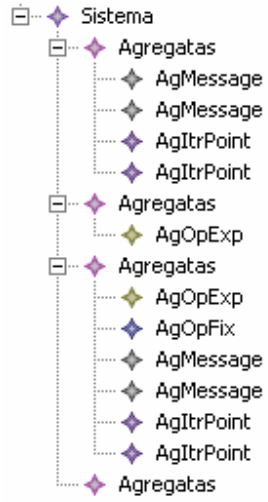
```
module PIM2PSM;
create OUT:psm_metamodel from IN:pim_metamodel;

rule Modelis2Sistema{
  from e : pim_metamodel!Modelis
  to out : psm_metamodel!Sistema mapsTo e (Aggregates <-e.agregatai)
}

rule Operacija2AgOpExp{
  from e : pim_metamodel!Operacija (e.tipas = 1)
  to out : psm_metamodel!AgOpExp mapsTo e (count<-e.kiekis)
}

rule Operacija2AgOpFix{
  from e : pim_metamodel!Operacija (e.tipas = 2)
  to out : psm_metamodel!AgOpFix mapsTo e (count<-e.kiekis)
}
```

Visos eksperimentinei transformacijai reikalingos ir naudojamos taisyklės pavaizduotos 15 paveiksle, o pilnas jų aprašymas pateikiamas 4 priede.



15 pav. ATL transformavimo taisyklių hierarchija

Kai taisyklės sukurtos, galime atlikti eksperimentinę transformaciją, o gautus XMI formatu rezultatus pamatyti tekstiniu ir grafiniu pavidalu.

PIM modelis	PSM modelis
	

16 pav. PIM ir PSM modeliai

16 paveiksle matomas pradinis sistemos PIM modelis, ir transformacijos rezultate gautas PSM modelis. Transformacijos rezultatai XMI formatu pateikti 5 priede.

5.5 Eksperimentinės PIM į PSM transformacijos MTL kalboje

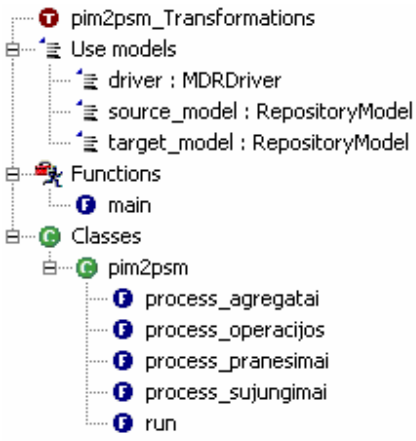
Eksperimentinių transformacijos užrašymo principai yra panašūs tiek ATL tiek ir MTL kalbose. Bet MTL kalba yra imperatyvi, todėl rašant transformavimo programą reikia pačiam pasirūpinti visų taisyklių iškvietimu. MTL sintaksė artima daugumai objektiškai orientuotų kalbų. BasicMTL transliatorius palaiko tik modelius užrašytus XMI1.1 standartu, tuo tarpu kai visi sukurti modeliai ir meta modeliai buvo specifikuoti modernesniu XMI2.0 formatu. Teko perdaryti eksperimentinius meta modelius ir modelius į XMI1.1 formatą. MTL sistemos architektūra yra pritaikyta naudoti įvairias modelių saugyklas (*repository*), bet standartiškai naudojamas MDR. Modelių ir meta

modelių pakeitimai buvo pakankamai sudėtingi, nes daugelį XMI kodo dalių teko rašyti rankiniu būdu. MTL.

MTL vartotojo sąsaja, kaip ir ATL, yra integruojama į universalaus redaktoriaus *Eclipse* aplinką. MTL *Eclipse* redaktoriuje galime rašyti transformavimo programas, tačiau sintaksės ir modeliavimo klaidų pranešimai pateikiami tik transliavimo arba programos vykdymo metu. Nuoseklus programos kodo vykdymas (*debug*), yra sudėtingesnis, nes programos paleidimas susideda iš 3 žingsnių:

1. iš MTL programos generuojama Java programa,
2. transliuojama Java programa,
3. vykdoma antrame žingsnyje gauta Java programa.

Dėl tokios vykdymo tvarkos programos klaidos galimos kiekviename iš ankščiau paminėtų žingsnių.



17 pav. MTL transformavimo funkcijų struktūra

Eksperimentinės MTL transformavimo programos struktūra pateikta 17 paveiksle. Gaila, bet MTL kalba neturi jokios grafinės transformacijos modelio atvaizdavimo sintaksės. Kadangi MTL modelių saugojimui naudoja XMI1.1 standartą, jos sukurtų modelių negalime pavaizduoti grafiniame EMF redaktoriuje, tačiau realiai gaunami modeliai yra tokie patys kaip ir 16 paveiksle, skiriasi tik jų užrašymo būdas.

5.6 Kiekybiniai eksperimentinių stebėjimų rezultatai

Atlikus bandymus buvo stebimi kokybiniai rezultatai. Grafinis jų atvaizdavimas pateiktas ankstesniuose skyriuose. Atsižvelgiant į tai, kad transformacijų metu gauti rezultatai buvo panašūs, nuspręsta padaryti apytikslį kiekybinį transformacijos įvertinimą.

Ta pati PIM modelio transformacija į PSM modelį užrašyta ATL kalba sudaro 50 kodo eilučių, o tuo tarpu kai MTL net dvigubai daugiau. PIM ir PSM modelių dydžiai skiriasi tik todėl, kad ATL kalboje jų užrašymui naudojame XMI2.0 formatą, o MTL – senesnį XMI1.1 formatą.

4 lent. Kiekybiniai eksperimentų rezultatai

Transformacijų specifikavimo kalba	**PIM modelio dydis (eil. sk.)**	**PSM modelio dydis (eil. sk.)**	**Transformavimo specifikacijos dydis (eil. sk.)**
ATL	22	21	50
MTL	32	42	109
Skirtumas (eil. sk.)	10	21	59
Skirtumas (%)	30%	50%	54%

Eilučių skaičius buvo pasirinktas kaip kiekybinio matavimo priemonė. Eksperimentų kiekybinių rezultatų suvestinė pateikta 4 lent..

6. Išvados

- Eksperimentinių modelių transformacijų realizavimui vykdyti buvo sukurtas: nuo platformos nepriklausantis agregatinių sistemų PIM meta modelis, nuo platformos priklausantis agregatinių sistemų PSM meta modelis, grafinis PIM modelių kūrimo įrankis Eclipse EMF programavimo aplinkoje ir bandomasis PIM modelis. Sukurtos eksperimentinės transformacijų realizacijos ATL ir MTL kalboms, sėkmingai įvykdytos bandomosios transformacijos

- Bandymų metu įvertinti kiekybiniai kriterijai parodė hibridinės ATL kalbos privalumus lyginant su imperatyvia MTL kalba. ATL kalbos deklaratyviomis taisyklėmis aprašyta transformacija buvo daug aiškesnė ir lakoniškesnė nei analogiška imperatyvi MTL programa. Žiūrint iš modeliuotojo pozicijos MTL kalba sunkiau įsisavinama ir diegiama. ATL kalba tyrimo metu leido naudoti naujesnius modelių specifikavimo formatus (XMI2,EMOF) ir žingsninę transformacijos analizę. Nors technologiniu aspektu MTL kalba turi daugiau pritaikymo ir paderinimo savybių, jai trūksta aiškumo ir išbaigtumo.

- MDA architektūrai labai svarbu standartizuoti modelių transformacijas. Modelių transformavimo kalboms siūloma daugybė galimų sprendimo būdų. Bet kol OMG nėra baigtas derinti ir kurti vieningas QVT-RFP standartas, šioje srityje yra daug erdvės vystyti moksliniams tyrimams.

7. Literatūra

1. OMG. Meta Object Facility (MOF) Specification. 2002 [žiūrėta 2005-02-20]. Prieiga per internetą: http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf.

2. OMG. Model Driven Architekture (MDA). 2001 [žiūrėta 2005-02-20]. Prieiga per internetą: http://www.omg.org/docs/ormsc/01-07-01.pdf.

3. Kleppe A., Warmer J., Bast W. MDA Explained. The Model Driven Architecture: Practice and Promise. Boston: Addison-Wesley, 2003.

4. OMG. MDA Guide Version 1.0.1. 2003 [žiūrėta 2004-10-11]. Prieiga per internetą: http://www.omg.org/docs/omg/03-06-01.pdf.

5. Bettin J. Model-Driven Architecture – Implementation and Metrics. Version 1.1. 2003 [žiūrėta 2004-01-29]. Prieiga per internetą : http://www.softmetaware.com/mdaimplementationandmetrics.pdf.

6. OMG. Meta Object Facility (MOF) 2.0 Core Proposal. 2003 [žiūrėta 2005-03-05]. Prieiga per internetą : http://www.omg.org/docs/ad/03-04-07.pdf.

7. OMG. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. 2002 [žiūrėta 2004-08-01]. Prieiga per internetą : http://www.omg.org/docs/ad/02-04-10.pdf.

8. Judson S. R., France R. B., Carver D. L.. Specifying Model Transformations at the Metamodel Level. [žiūrėta 2004-11-01] Prieiga per internetą: http://www.metamodel.com/wisme-2003/19.pdf.

9. OMG. XML Metadata Interchange (XMI) Specification. 2003 [žiūrėta 2005-01-16]. Prieiga per internetą: http://www.omg.org/cgi-bin/apps/doc?formal/03-05-02.pdf.

14. Didier Vojtisek BasicMTL realization guide. 2004. [žiūrėta 2005-02-25] Prieiga per internetą: http://www.carroll-research.org/docs/MOTOR-F3.2-BasicMTLRealisationGuide_V0.4.pdf

11. OMG. UML 2.0 OCL 2nd revised submission. 2003 [žiūrėta 2005-03-12]. Prieiga per internetą: http://www.omg.org/doc/ad/03-01-07.pdf.

12. Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, Jamal Eddine Raugui. First experiments with ATL model transformation language. 2003 [žiūrėta 2005-03-12]. Prieiga per internetą: http://www.softmetaware.com/oopsla2003/bezivin.pdf.

13. Freddy Allilaire, Tarik Idrissi. ADT:Elipse development tools for ATL. 2004. [žiūrėta 2005-04-05]. Prieiga per internetą: http://www.eclipse.org/gtm/adt/ADT.pdf

8. Terminų ir santrumpų žodynas

5 lent. Terminai

Lietuviškai	**Angliškai**	**Paaiškinimas**
Modelis	Model	Modelis yra sistemos dalies (funkcijos,struktūros ir/arba elgesio) atvaizdavimas kokioje nors sintaksiškai formalioje kalboje (pvz. UML, MOF).
Požiūris	View	Visos sistemos reprezentacija iš susijusių interesų perspektyvos. Kiekvienas atvaizdas papildo sistemos aprašymą tam tikros perspektyvos požiūriu.
Projekcija	Mapping	Taisyklių ir metodų aibė naudojama modelio modifikavimui norint gauti kitą modelį. Sukuriant modelių projekcijas į tam tikras platformas vėliau galima atlikti automatinį ar rankinį modelio transformavimą į tikslinę platformą.

6 lent. Santrumpos

Santrumpa	**Pilnas pavadinimas**	**Paaiškinimas / Vertimas**
ATL	ATLAS Transformation Language	Atlas transformavimo kalba
MTL	Model Transformation Language	Modelių transformavimo kalba
MDA	Model Driven Architecture	Modeliais pagrįsta architektūra.
MOF	Meta-Object Facility	Meta objektų infrastruktūra – meta modeliavimo ir metaduomenų saugyklų standartas.
OMG	Object Management Group	Kompanijos pavadinimas kuri sukūrė UML, MOF, CWM, XMI, CORBA ir daugybe kitų specifikacijų.
OCL	Object Constraint Language	OMG objektų apribojimų specifikavimo kalba.

9. Priedai

9.1 PIM meta modelis ecore XMI formatu

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="pim_metamodel"
    nsURI="http:///pim_metamodel.ecore" nsPrefix="pim_metamodel">
  <eClassifiers xsi:type="ecore:EClass" name="Modelis">
    <eStructuralFeatures xsi:type="ecore:EReference" name="agregatai" upperBound="-1"
        eType="#//Agregatas" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Agregatas">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pavadinimas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="aprasymas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="sujungimai" upperBound="-1"
        eType="#//Sujungimas" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="operacijos" upperBound="-1"
        eType="#//Operacija" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="pranesimai" upperBound="-1"
        eType="#//Pranesimas" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="eile" eType="#//Eile" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="busena" eType="#//Busena"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Operacija">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pavadinimas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="tipas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="kiekis" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Sujungimas">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pavadinimas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="aprasymas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="pabaiga" lowerBound="1"
        eType="#//Agregatas"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Pranesimas">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pavadinimas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="tipas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Eile">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pavadinimas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
```

```
   <eStructuralFeatures xsi:type="ecore:EReference" name="pranesimai" eType="#//Pranesimas"
      containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Busena">
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="parametrai" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EIntegerObject"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="pavadinimas" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
</ecore:EPackage>
```

9.2 PIM modelis XMI formatu

```
<?xml version="1.0" encoding="UTF-8"?>
<pim_metamodel:Modelis xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:pim_metamodel="http:///pim_metamodel.ecore">
  <agregatai pavadinimas="1" aprasymas="Pirmas agregatas turintis
2sujungimus ir 2pranesimus">
    <sujungimai pavadinimas="1-2" aprasymas="1"
pabaiga="//@agregatai.1"/>
    <sujungimai pavadinimas="1-3" aprasymas="2"
pabaiga="//@agregatai.2"/>
    <pranesimai pavadinimas="1"/>
    <pranesimai pavadinimas="2"/>
  </agregatai>
  <agregatai pavadinimas="2" aprasymas="Antras agregatas turintis
1operacija ir 1eile">
    <operacijos pavadinimas="1" tipas="1" kiekis="10"/>
    <eile pavadinimas="1"/>
  </agregatai>
  <agregatai pavadinimas="3" aprasymas="Trecias agregatas turintis
2sujungimus , 2pranesimus ir 2operacijas">
    <sujungimai pavadinimas="3-4" aprasymas="1"
pabaiga="//@agregatai.3"/>
    <sujungimai pavadinimas="3-1" aprasymas="1"
pabaiga="//@agregatai.0"/>
    <operacijos pavadinimas="2" tipas="1" kiekis="5"/>
    <operacijos pavadinimas="3" tipas="2" kiekis="6"/>
    <pranesimai pavadinimas="3"/>
    <pranesimai pavadinimas="4"/>
  </agregatai>
  <agregatai pavadinimas="4" aprasymas="Ketvirtas agregatas"/>
</pim_metamodel:Modelis>
```

9.3 PSM meta modelis ecore XMI formatu

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="psm_metamodel_fixed"
   nsURI="http:///psm_metamodel_fixed.ecore" nsPrefix="psm_metamodel_fixed">
 <eClassifiers xsi:type="ecore:EClass" name="Agregatas" eSuperTypes="#//AgModule">
   <eOperations name="internalEvent"/>
   <eOperations name="externalEvent"/>
```

```
    <eStructuralFeatures xsi:type="ecore:EReference" name="operExp" upperBound="-1"
        eType="#//AgOpExp" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="operFix" upperBound="-1"
        eType="#//AgOpFix" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="messages" upperBound="-1"
        eType="#//AgMessage" containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="ref" upperBound="-1" eType="#//AgItrPoint"
        containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AgModule">
    <eOperations name="addStat"/>
    <eOperations name="reportNotEmpty"/>
    <eOperations name="reportTable"/>
    <eOperations name="report"/>
    <eOperations name="internalEvent"/>
    <eOperations name="internalExEvent"/>
    <eOperations name="debug"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="statistics" eType="ecore:EDataType
MOF#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AgOpExp" eSuperTypes="#//AgOperation">
    <eOperations name="setFactory"/>
    <eOperations name="start"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="p1" eType="ecore:EDataType
MOF#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="eventFactory" eType="ecore:EDataType
MOF#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AgOperation">
    <eOperations name="startOperation"/>
    <eOperations name="out"/>
    <eOperations name="of"/>
    <eOperations name="on"/>
    <eOperations name="getModule"/>
    <eOperations name="getId"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="module" eType="ecore:EDataType
MOF#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="count" eType="ecore:EDataType
MOF#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="id" eType="ecore:EDataType
MOF#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AgOpFix" eSuperTypes="#//AgOperation">
    <eOperations name="setFactory"/>
    <eOperations name="start"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="p1" eType="ecore:EDataType
MOF#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="eventFactory" eType="ecore:EDataType
MOF#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AgMessage">
    <eOperations name="getMessageId"/>
    <eOperations name="getSender"/>
    <eOperations name="setSender"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="rfrom" eType="ecore:EDataType
MOF#//EString"/>
```

```
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="id" eType="ecore:EDataType
MOF#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AgItrPoint">
   <eOperations name="connect"/>
   <eOperations name="output"/>
   <eStructuralFeatures xsi:type="ecore:EReference" name="rto" lowerBound="1"
eType="#//Agregatas"/>
   <eStructuralFeatures xsi:type="ecore:EReference" name="rfrom" lowerBound="1"
eType="#//Agregatas"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Sistema" eSuperTypes="#//AgSystem">
   <eStructuralFeatures xsi:type="ecore:EReference" name="Aggregates" upperBound="-1"
      eType="#//Agregatas" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="AgSystem">
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="intEvQueue" eType="ecore:EDataType
MOF#//EString"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="cntEvList" eType="ecore:EDataType
MOF#//EString"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="theOutput" eType="ecore:EDataType
MOF#//EString"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="theDebugOutput" eType="ecore:EDataType
MOF#//EString"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="tsyst" eType="ecore:EDataType
MOF#//EString"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="tstop" eType="ecore:EDataType
MOF#//EString"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="filepatch" eType="ecore:EDataType
MOF#//EString"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="title" eType="ecore:EDataType
MOF#//EString"/>
  </eClassifiers>
</ecore:EPackage>
```

9.4 ATL programos tekstas

```
module PIM2PSM;
create OUT:psm_metamodel from IN:pim_metamodel;

-- ------------------------------------------------------------
-- Modelis -> Sistema
-- ------------------------------------------------------------
rule Modelis2Sistema{
  from e : pim_metamodel!Modelis
  to out : psm_metamodel!Sistema mapsTo e (Aggregates <-e.agregatai)
}

-- ------------------------------------------------------------
-- Agregatas -> Agregatas
-- ------------------------------------------------------------
rule Agregatas2Agregatas{
from e : pim_metamodel!Agregatas
to out : psm_metamodel!Agregatas mapsTo e
(
  ref <- e.sujungimai,
```

```
  messages<- e.pranesimai,
  operExp<-e.operacijos->select(x|x.tipas=1)->asSequence(),
  operFix<-e.operacijos->select(y|y.tipas=2)->asSequence()
)
}
-- ------------------------------------------------------------------
-- Sujungimas -> AgItrPoint
-- ------------------------------------------------------------------
rule Sujungimas2AgItrPoint{
  from e : pim_metamodel!Sujungimas
  to out : psm_metamodel!AgItrPoint mapsTo e ( rto <- e.pabaiga )
}

-- ------------------------------------------------------------------
-- Pranesimas -> AgMessage
-- ------------------------------------------------------------------
rule Pranesimas2AgMessage{
  from e : pim_metamodel!Pranesimas
  to out : psm_metamodel!AgMessage mapsTo e (id <- e.pavadinimas)
}

-- ------------------------------------------------------------------
-- Operacija -> AgOpExp
-- ------------------------------------------------------------------
rule Operacija2AgOpExp{
from e : pim_metamodel!Operacija (e.tipas = 1)
to out : psm_metamodel!AgOpExp mapsTo e (p1 <-'OpExp', count<-e.kiekis)
}

-- ------------------------------------------------------------------
-- Operacija -> AgOpFix
-- ------------------------------------------------------------------
rule Operacija2AgOpFix{
from e : pim_metamodel!Operacija (e.tipas = 2)
to out : psm_metamodel!AgOpFix mapsTo e (p1 <-'OpFix', count<-e.kiekis)
}
```

9.5 PSM modelis – ATL programos vykdymo rezultatas

```
<?xml version="1.0" encoding="ASCII"?>
<psm_metamodel_fixed:Sistema xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:psm_metamodel_fixed="http:///psm_metamodel_fixed.ecore">
  <Aggregates>
    <messages id="1"/>
    <messages id="2"/>
    <ref rto="//@Aggregates.1"/>
    <ref rto="//@Aggregates.2"/>
  </Aggregates>
  <Aggregates>
    <operExp p1="OpExp 1"/>
  </Aggregates>
  <Aggregates>
```

```
    <operExp p1="OpExp 2"/>
    <operFix p1="OpFix 3"/>
    <messages id="3"/>
    <messages id="4"/>
    <ref rto="//@Aggregates.3"/>
    <ref rto="//@Aggregates.0"/>
  </Aggregates>
  <Aggregates/>
</psm_metamodel_fixed:Sistema>
```

9.6 MTL programos tekstas

```
/*********************************************
***   $Id:$
***   File : pim2psm.mtl
***   Library : mtl
***   Version : 0.1
***   Author :Administrator
***   Date : May 16, 2005 12:04:26 AM CEST
*********************************************/

library pim2psm_Transformations;

model source_model : RepositoryModel;
model target_model : RepositoryModel;
model driver          : MDRDriver;

main() : Standard::Void
{
            //Load variables
            driver     :MDRDriver::MDRModelManager;
            aTransformation       :pim2psm;

            projectPath: Standard::String;
            PIMmetamodelFilename: Standard::String;
            PIMinputFilename: Standard::String;
            PIMoutputFilename: Standard::String;

            PSMmetamodelFilename: Standard::String;
            PSMinputFilename: Standard::String;
            PSMoutputFilename: Standard::String;

            //Define filenames
            PIMmetamodelFilename  := 'pim_metamodelis.xml';
            PIMinputFilename      := 'pim_modelis.xml';
            PIMoutputFilename     := 'pim.xml';

            PSMmetamodelFilename  := 'psm_metamodelis.xml';
            PSMinputFilename      := 'psm_modelis.xml';
            PSMoutputFilename     := 'psm.xml';



            '- Init MDR'.toOut();

            //Init MDR
            driver := new MDRDriver::MDRModelManager();
```

```
            driver.init();

            '- load MDR'.toOut();
            //Set output
            source_model := driver.getModelFromXMI( PIMmetamodelFilename,
            'PIM', 'PIM', PIMinputFilename, PIMoutputFilename );
            '- PIM Modeliai uzkrauti'.toOut();

            target_model := driver.getModelFromXMI( PSMmetamodelFilename,
            'PSM', 'PSM', PSMinputFilename, PSMoutputFilename );

            '- PSM Modeliai uzkrauti'.toOut();

            aTransformation := new pim2psm();
            aTransformation.run();

            '- Pabaiga'.toOut();
}

class pim2psm
{
            ////////////////////////////////////////////////////////
            run()
            {
                        pimModelis : source_model::Modelis;
                        pimIterator : Standard::Iterator;

                        'pim2psm BEGIN'.toOut();

                        pimIterator          :=
!source_model::Modelis!.allInstances().getNewIterator();
                        pimIterator.start();

                        while pimIterator.isOn()
                        {
                                    pimModelis:=
pimIterator.item().oclAsType(!source_model::Modelis!);

                                    process_modelis (pimModelis);
                                    pimIterator.next();
                        }

                        'pim2psm END'.toOut();
            }

            ////////////////////////////////////////////////////////
            process_modelis (theModelis : source_model::Modelis)
            {
                        newSistema : target_model::Sistema;

                        pimAgregatas : source_model::Agregatas;
                        pimIterator : Standard::Iterator;

                        'process_modelis BEGIN'.toOut();

                        newSistema := new target_model::Sistema();
                        newSistema.title := 'New Aggregate system';
```