KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Audrius Pranckevičius

Imitacinių modelių sudarymas naudojant UML

Magistro darbas

Darbo vadovas
Dr. V. Pilkauskas

Kaunas, 2006

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Audrius Pranckevičius

Imitacinių modelių sudarymas naudojant UML

Magistro darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
I. Mickienė
2006-05

Vadovas
dr. V. Pilkauskas
2006-05

Recenzentė
Doc. Dr. L. Nemuraitė
2006-05

Atliko
IFM-0/1 gr. stud.
Audrius Pranckevičius
2006-05-15

Kaunas, 2006

TURINYS

1. ĮVADAS 3

2. MODELIAIS PAGRĮSTOS INŽINERIJOS ANALIZĖ 5

2.1. MODELIAIS PAGRĮSTA INŽINERIJA 5
2.1.1. Modeliais pagrįsta architektūra 5
2.1.2. Modeliais pagrįsta inžinerija 8
2.2. MODELIŲ PROGRAMINĖS ĮRANGOS INŽINERIJOS SĄVOKA. 9
2.2.1. Modelio ir modeliavimo apibrėžimai 9
2.2.2. Apibrėžimų analizė 11
2.2.2.1. Modelis kaip gimininga idėja 11
2.2.2.2. Modeliai yra abstrakcijos 11
2.2.2.3. Ryšiai tarp modelio ir objekto sistemos 11
2.2.2.4. Modeliai kaip sistemos 13
2.2.3. Pasirenkant apibrėžimą 14
2.2.4. Modeliavimas kompiuterių moksle 14
2.2.4.1. Programinės įrangos sistemos kaip modeliai 14
2.2.4.2. Programinės įrangos sistemų modeliavimas 15
2.3. META-MODELIAI IR META-MODELIAVIMAS 15
2.3.1. Meta-modelio ir meta-modeliavimo apibrėžimai 16
2.3.1.1. Meta-modeliavimas kaip modeliavimo procesas 16
2.3.1.2. Meta-modeliavimas kaip kalbų modeliavimas 16
2.3.2. Meta modeliai, modeliai ir atskiras atvejis(kažko) ryšys 17
2.3.2.1. *Modelis (kažko)* ryšys tarp modelio ir meta-modelio 17
2.3.2.2. *Atskiras atvejis (kažko)* ryšys tarp modelio ir meta-modelio 18
2.4. MODELIŲ LYGIAI 21
2.4.1. Meta-modeliavimo architektūra 21
2.4.2. Meta-modeliavimo architektūros pavyzdžiai 22
2.4.2.1. Resursų apibrėžimo karkasas (RDF) 22
2.4.2.2. Išplėstinė pažymėjimo kalba (XML) 23
2.5. MODELIŲ TRANSFORMACIJOS 24
2.5.1. Apibrėžimai 24
2.5.2. Modelių transformacijos kalbos 27
2.5.2.1. Deklaratyvios ir imperatyvios transformacijos kalbos 27
2.5.2.2. Transformacijų kryptingumas 28
2.5.2.3. Įėjimo ir išėjimo kardinalumas transformacijų apibrėžimuose 28
2.6. TIKSLAI IR UŽDAVINIAI 29

3. AGREGATŲ IR SUJUNGIMŲ SCHEMŲ META-MODELIS 30

3.1. PLA METAMODELIS 30
3.1.1. Agregatų sujungimo schemos metamodelis 30
3.1.2. PLA agregatų specifikavimo metamodelis 31
3.2. NUO SKAIČIAVIMŲ NEPRIKLAUSOMAS MODELIS 34
3.2.1. Išėjimo aibės 34
3.2.2. Įėjimo aibės 35
3.2.3. Operatoriai 36
3.2.4. Išorinių įvykių aibė 37
3.2.5. Vidinių įvykių aibė 38
3.2.6. Diskretūs komponentai 40
3.2.7. Tolydžioji komponentė 41
3.2.8. Išėjimo taškai 43
3.2.9. Įėjimo taškai 44
3.2.10. Valdanti seka 45
3.2.11. Kanalas 46
3.2.12. CIM modelio specifikacija 48

4. TRANSFORMACIJOS 49

4.1. AGREGATŲ TRANSFORMACIJA 49
4.2. SUJUNGIMŲ TRANSFORMACIJOS 51
4.3. LIKUSIOS TRANSFORMACIJOS 51

5. NAFTOS TERMINALO IMITACINIO MODELIO SUDARYMAS 52

5.1. PRADINIAI DUOMENYS 52
5.1.1. Agregatas Railway *52*
5.1.1.1. Išėjimo signalų aibė 52
5.1.1.2. Tolydžiosios komponentės 53
5.1.2. Agregatas Station *53*
5.1.2.1. Išorinių įvykių aibė 53
5.1.2.2. Išėjimo signalų aibė 54
5.1.2.3. Tolydžiosios komponentės 55
5.1.2.4. Diskrečių komponentų aibė 55
5.1.3. Agregatas Train *56*
5.1.3.1. Išorinių įvykių aibė 56
5.1.4. Sujungimų modelis OilTerminal *57*
5.2. LAUKIAMI TRANSFORMACIJOS REZULTATAI 58
5.2.1. Agregatinė klasė Railway *58*
5.2.2. Agregatinė klasė Station *59*
5.2.3. Agregatinė klasė Train *60*
5.2.4. Sujungimų klasė OilTerminal *61*

6. IŠVADOS 62

7. CREATING SIMULATION MODELS USING UML 63

8. LITERATŪRA 64

9. TERMINŲ IR SANTRUMPŲ ŽODYNAS 66

10. PRIEDAI 67

10.1. TRANSFORMACIJOS FAILAS 67
10.2. DUOMENŲ MODELIS 73
10.3. CIM MODELIS 75
10.4. SUGENERUOTAS REZULTATŲ MODELIS 77
10.5. STRAIPSNIS 80

1. ĮVADAS

Šių dienų programinės įrangos sistemos yra labai sudėtingos. Tai yra dėl būdingo sudėtingumo tose srityse, kur programinės įrangos sprendimai naudojami. Dėl to programinės įrangos kūrimo procesas, tampa sudėtinga užduotimi. Be to, sričių sudėtingumas tik vienas veiksnys, lemiantis sunkumus, atsirandančius kuriant programinę įrangą. Kiti sunkumai susiję su metodologija, organizacija, ekonomija, kultūra ir kitais veiksniais.

Programinės įrangos sudėtingumas pramonėje, sukelia pastovų naujų technologijų srautą. Per pastarąjį dešimtmetį, mes buvome liudininkai keleto naujų kalbų atsiradimo (kaip C++, Java, C#), vidutinio dydžio technologijų (kaip CORBA, Web Services) ir duomenų atstovavimo standartų (kaip SGML ir XML). Šis nuolatinis perėjimas nuo vienos technologijos prie kitos atrodo neišvengiamas ir teikia ypatingą reikšmę, atitinkamų technologijų kopijavime, programinės įrangos kūrime.

Tam, kad būtų susidorota su šia problema, Object Management Group (OMG) pasiūlė naują programinės įrangos kūrimo požiūrį, pavadintą *Modeliais grįsta architektūra* (MDA). MDA pagrindinė idėja, naudoti modeliavimą ir modelius kaip pagrindinę veiklą, kuriant programinę įrangą. Kuriant programinę įrangą modelių pagalba, ji pasidaro daug atsparesnė įdiegiamų technologijų pokyčiams. Supratimas apie *Modeliais grįstą inžineriją* (MDE), pagamintą pagal MDA idėją, įneša didelę naudą programinės įrangos kūrimo procese.

MDE programinės įrangos kūrimo procese, sistema yra kuriama tobulinamais modeliais, pradedant nuo aukščiausių abstrakcijos lygių ir palaipsniui pereinant prie žemiausių, kol galiausiai sugeneruojamas sistemos kodas. Šis atsinaujinimas įgyvendinamas modelių transformacijos pagalba.

MDE apima didelį spektrą tyrinėjamų sričių, kai kurios iš jų jau sukurtos, kai kurios dar tik pradedamos kurti. Reikalinga daug tolimesnių pastangų norint pasiekti, kad jos būtų sklandžios ir suprantamos, pagrįstos atvirais standartais, palaikomos daugelio įrankių ir technologijų.

Šiame darbe, didžiausią dėmesį skirsiu vienai iš MDE veiklų: modelių transformavimui. Transformacijų apibrėžimai, parašyti transformavimo kalba, yra vienas iš programinės įrangos privalumų, kuris gali būti laikomas taip pat svarbiu, kaip modeliai ar sistemos. Transformavimo apibrėžimai gali būti projektavimo, įdiegimo, naudojimo ir evoliucijos subjektas. Be to, pakeitimų aplinka lemia transformaciją, kurioje ji egzistuoja.

Modelio transformacija yra procesas, paverčiantis vieną modelį kitu. Modelis gali būti paverstas daugeliu modelių, kurie funkcionaliai yra lygūs, bet skirtingi kokybės savybėmis, kurias jie teikia. Pavyzdžiui, vienas modelis gali būti labiau išplečiamas, tuo tarpu kitas, gali būti optimizuotas našumui. Programinės įrangos inžinieriai turi turėti galimybę atpažinti transformacijas, kurios pagamina modelius pagal norimas kokybės savybes. Jei programinės įrangos inžinieriai norėtų lyginti modelių funkcionalumą, tai turėtų būti sudarytos alternatyvios transformacijos.

Modeliai yra išreikšti įvairiomis modeliavimo kalbomis. Tačiau nėra idealios modeliavimo kalbos: modelių transformavimo kalbos, turi galėti pateikti transformavimo apibrėžimus modeliams, kurie parašyti įvairiomis modeliavimo kalbomis. Norint tai atlikti efektyviai, reikia atsakyti į keletą esminių klausimų. Kokios yra charakteristikos modeliavimo kalbų, kurios turi įtakos transformavimo kalbai? Kaip transformavimo kalba gali būti atskirta ir parametrituota, atsižvelgiant į besikeičiančias modeliavimo kalbų charakteristikas?

Tikimasi, jog transformacijų apibrėžimai susidoros su panaudojamumu, įdiegimu ir kompozicija taip, kaip susidoroja klasės ir bibliotekos. Papildomai transformavimo kalbos turėtų turėti galimybę būti plečiamos, nesugriaunant jau egzistuojančios struktūros.

Siekiant priversti transformavimo kalbas dirbti su modeliais, pateiktais skirtingomis modeliavimo kalbomis, reikia išnagrinėti pagrindines operacijas dabartinėse transformavimo kalbose ir, kaip jie bendrauja su modeliavimo kalbos naujovėmis. Tą ir stengsiuosi daryti šiame darbe.

2. MODELIAIS PAGRĮSTOS INŽINERIJOS ANALIZĖ

2.1. Modeliais pagrįsta inžinerija

Modeliais pagrįstos inžinerijos (MDE) idėja, iškilo kaip apibendrinimas Modeliais Pagrįstos Architektūros (MDA) požiūrio, programinės įrangos vystyme. Šioje dalyje, pirmiausiai supažindinsiu su pagrindinėmis MDA sąvokomis ir po to paaiškinsim, kaip iš pat pradžių MDE buvo pristatyta Kent'o [22].

2.1.1. Modeliais pagrįsta architektūra

MDA tai karkasas programinės įrangos vystymui, pritaikytas Object Management Group (OMG) 2001 ir aprašytas eilėje OMG dokumentų. Dokumentas MDA Guide [26] pateikia apžvalgą ir apibrėžimus sąvokų, naudojamų MDA.

Pagrindinis MDA tikslas – pateikti sprendimą problemai, nuolat pasirodančiai programinės įrangos technologijose, kuris priverčia kompanijas, kiekvieną kartą atsiradus naujai „karštai" technologijai, derinti savo programinės įrangos sistemas. Kaip pavyzdį galime pateikti vidutinio dydžio produktinių technologijų evoliuciją. CORBA [27] yra visuotinai priimtinas standartas, vidutinio dydžio produktams ir daugelis kompanijų naudoja jį realizuodamos savo sistemas. Galima pastebėti, kad Web Services [38] iš dalies pakeičia CORBA, kaip vidutinio dydžio produktų technologiją. Daugelis egzistuojančių sistemų gali būti nukreipiamos į šią naują technologiją, nors jų funkcionalumas gali likti nepakitęs. Kaip tik atsiranda tokia nauja technologija, ji kelia tokią pat problemą, reikalaujančia pertvarkyti egzistuojančias sistemas. Nuolatiniai keitimaisi technologijose, kelia problemą dėl pernešamumo, kuris gali reikalauti didžiulių pastangų. MDA skirta spręsti šią problemą.

Kitos problemos, kurias tikimasi išspręsti MDA pagalba yra aprašytos Kleppe el al [23]. Produktyvumo problema, kuriant dabartinę programinę įrangą, praktiškai yra sukeliama fakto, jog programinės įrangos kūrimo procesai yra paremti žemo lygio projektavimu ir programavimu. Didelių programinės įrangos sistemų kodo supratimas ir palaikymas –sudėtingas ir į klaidas linkęs procesas.

Operacinio suderinamumo problema yra nusakoma tuo, kad didelės sistemos yra ne vientisos, bet modulinės. Skirtingi moduliai yra sukurti technologijomis, tinkančiom spręsti dabartines problemas. Tačiau, programinės įrangos sistemos susideda iš komponentų, realizuotų skirtingomis technologijomis, kurios turi būti suderintos.

Siekdama susidoroti su pernešamumo problema, MDA sujungia du principus, kurie buvo naudojami kompiuterių ir kitose inžineriniuose moksliuose.

1. Naudoti modeliavimą ir modeliais vystyti programinės įrangos sistemas. Šis principas yra gerai įgyvendintas inžineriniuose moksluose. Pavyzdžiui namas, civilinėje inžinerijoje, yra apibūdinamas diagramų rinkiniu. Atlikus skaičiavimus pagal medžiagų savybes, gali prasidėti tikrasis statymo procesas. Kartais gali būti sukurti namo pakopiniai modeliai. Egzistuoja ryški riba tarp modeliavimo, modelių analizavimo ir realizavimo fazių.
2. Atskyrimas sistemos specifikacijos nuo detalių, kaip sistema bus realizuota konkrečiomis technologijomis. Abstrakti sistemos specifikacija tampa pagrindiniu dalyku kuriant programinę įrangą. Daugelis realizavimų, naudojant tam tikras technologijas, gali būti gaunama iš abstrakčių specifikacijų. Taip yra pasiekiami du tikslai: pernešamumas ir panaudojamumas.

OMG apibūdina MDA kaip sistemų vystymo suvokimą, pagrįstą modeliais. Sakoma *modeliais-pagrįsta* todėl, kad suteikia galimybes naudoti modelius, tiesioginiam krypties supratimui, projektavimui, konstravimui, išdėstymui, palaikymui ir modifikacijoms [26]..

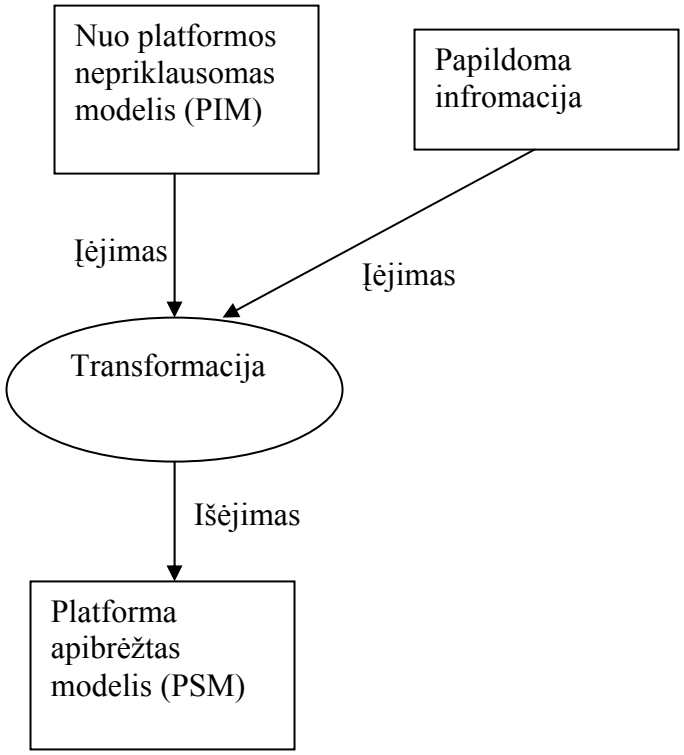
MDA klasifikuoja modelius į dvi klases: nuo platformų nepriklausomi modeliai (PIMs) ir tam tikrai platformai apibrėžti modeliai (PSMs). Šios klasės talpina modelius į skirtingus abstrakcijos lygius. MDA įvadas [26] nurodo tokį PIM apibrėžimą: „Nuo platformų nepriklausomi modeliai – tai žvilgsnis į sistemą iš platformai nepriklausomo taško“. PSM apibūdinami kaip: “Platformai apibrėžtas modelis – tai žvilgsnis į sistemą iš konkrečios platformos taško“. Šie apibūdinimai remiasi *platformos* idėja. MDA įvadas apibrėžia platformas kaip: „Platforma tai rinkinys posistemių ir technologijų, kuri pateikia aiškų rinkinį funkcionalumo per interfeisus ir apibūdina naudojimo šablonus, kuriuos bet kuri programa, palaikoma tos platformos, gali naudoti nesirūpindama detalėmis, kaip pateikiamas platformos funkcionalumas bus įgyvendintas“.

Sistemos vystymas, apibūdintas MDA, prasideda nuo tos sistemos PIM sudarymo. Tada PIM transformuojamas į viena ar daugiau PSM. PSM naudoja pasirinktos platformos pateiktas konstrukcijas. Galiausiai PSM yra transformuojami į kodą.

MDA pagrįstame programinės įrangos kūrime, pagrindinė operacija modeliams yra *modelių transformacijos*. Modelių transformacija yra procesas, paverčiantis vieną modelį į kitą

tos pačios sistemos modelį. MDA tikslas, kuo labiau automatizuoti modelių transformacijas. Transformacijos yra vykdomos įrankiais, naudojant transformavimo specifikacijas.

Paveikslas 2.1 parodo MDA transformavimo šabloną, kaip nurodyta MDA įvade. Transformacijos procesas priima PIM, kaip įėjimo duomenis ir sugeneruoja PSM, kaip išėjimo duomenis. Šablonas yra bendras, kadangi jis neapibrėžia, kaip tiksliai transformacijos yra apibrėžtos. Dažniausiai reikia papildomos informacijos apie platformas, kad būtų galima įvykdyti transformacijas.



2.1 pav.: MDA transformacijos šablonas

Dar vienas MDA tikslas – apjungti standartizuotas OMG egzistuojančias technologijas. Modeliai išreiškiami UML [30] ir UML profiliais. Jei naudojama kitokia modeliavimo kalba, ji turi būti apibūdinta naudojant MOF [28] meta-modeliavimo kalbą. MDA naudoja XML Meta Duomenų Apsikeitimą (XMI) [31], suspausti MOF modelius atskiroms dalims, XML formatu. OMG pradėjusi daryti standartizavimo kalbą, specifikuojančią transformacijas per MOF modelius, kaip atsaką į QVT reikalavimų pasiūlymą[29].

Kai kurios MDA idėjos yra apibrėžtos neaiškiai ir joms trūksta vientiso mokslinio pagrindimo. Bezivn [19] akcentuoja poreikį suprasti idėjas, tokias kaip *sistema, modelis*, *meta-modelis* ir ryšius tarp jų. Atkinson ir Kuhne [6] analizuoja modeliavimo karkasą, pateiktą OMG,

pateikdami reikalavimus, kurie nėra iš pat pradžios iškelti. Mes aptarsime šias idėjas vėlesniuose skyriuose.

2.1.2. Modeliais pagrįsta inžinerija

MDA supažindina su rinkiniu pagrindinių sąvokų, tokių kaip *modelis, meta-modelis, modeliavimo kalba* ir *transformacijos* ir siūlo modelių klasifikavimą: PIM ir PSM. Tačiau MDA trūksta supratimo apie programinės įrangos vystymo procesą. Modeliais pagrįsta inžinerija (MDE) labiau gilinasi į šią sritį, nei MDA.

Kent [22] apibrėžia MDE pagal MDA, pateikdamas supratimą apie programinės įrangos vystymo procesus ir modeliavimo aplinką, organizuojant modelius. Modeliavimo aplinka apibūdinama keletu dydžių. Kent taip pat apibrėžia reikalavimus įrankiams, kurie reikalingi norint atlikti operacijas su modeliais MDE.

MDA apibrėžia tik vieną modelių klasifikavimo dydį, paremtą dalijimuisi į PIM ir PSM. Tai dydis rodantis modelių abstrakcijos lygį. PIM galima laikyti aukštesniu abstrakcijos lygiu negu PSM. Tačiau vienas dydis, modeliavimo aplinkoje, atspindi tik modelių *abstraktumo* arba *konkretumo* laipsnį.

Kitas dydis kyla iš modelių išskyrimo pagal subjekto sritį. Skirtingi sistemos vartotojai gali įvairiai įsivaizduoti sistemą, sistemos galimybes. Šie vaizdai atsispindi skirtinguose tos pačios sistemos modeliuose. Subjekto sritys dažnai atitinka tikslus, apibrėžtus Aspect Oriented Software Development [17]. Susirūpinimą keliantys dalykai yra išvardyti kitos modeliavimo aplinkos dydyje. Šis dydis yra nominalus. Tarp interesų neegzistuoja jokia tvarka. Susirūpinimo pavyzdžiai – sutapimo kontrolė, apsauga, paskirstymas ir klaidų valdymas.

Trečias dydis yra atsiradęs dėl organizavimo problemų, pvz., versijavimas ir modelių autorystė. Dydžių skaičius modeliavimo aplinkoje nėra ribojamas ir priklauso nuo tam tikrų poreikių vystomame projekte.

Akivaizdu, jog pasiūlyta modeliavimo aplinka sugeba atpažinti modelį pagal daugiau kriterijų, negu paprastas PIM/PSM skaidymas. Vadinasi, ji gali būti laikoma labiau tinkama programinės įrangos vystymui.

Galimiems MDE procesams yra pateikti du požiūriai, kur pateiktos reikalingos idėjos, metodai ir įrankiai [9][2]. Favre [15][16] siūlo MDE viziją, kurioje MDA yra vienas iš galimų MDE atvejų, įgyvendintas komplektu technologijų, apibrėžtų OMG (MOF, UML, XMI). Modelio, meta-modelio ir transformacijų idėjos yra randamos ir kitose technologijose [24]. Pasak Favre, MDE turėtų talpinti įvairių technologijų sritis vienodai, prieš tai išanalizavusi jų reikšmę.

Aišku, jog tokia vizija kyla iš OMG standartų rinkinio ir žengia žingsnį į priekį, siūlydama daug atviresnį požiūrį.

Skyriuose 2.1.1 ir 2.1.2 supažindinama su modeliais pagrįsta architektūra ir modeliais pagrįsta inžinerija, kaip nauju požiūriu į programinės įrangos kūrimą. Jie remiasi rinkiniu bendrų idėjų, tokių kaip modelis, sistema, modeliavimo kalba, meta-modelis ir transformacija. Likusioje šio skyriaus dalyje bus aptartos šios idėjos.

2.2. Modelių programinės įrangos inžinerijos sąvoka.

Šiame skyriuje analizuosime modelio ir modeliavimo veiklos idėjas. Po trumpo modelio idėjos apibūdinimo, nukreipsime dėmesį į apibrėžimus, naudojamus kompiuterių moksle ir programinės įrangos inžinerijoje. Pasiūlysime savo pačių apibrėžimus modeliams, kurie bus naudojami šiame darbe.

2.2.1. Modelio ir modeliavimo apibrėžimai

Pradėsime nuo apibrėžimo randamo literatūroje, kuris apibrėžia bendrai modelius ir modeliavimą. Toliau pateiksime apibrėžimus, pritaikytus konkrečiai mokslo sričiai.

Žodis *modelis* gali būti kildinamas iš lotyniško žodžio *modulus*, kuris reiškia mažą matavimo vienetą. Merriam-Webster internetinis žodynas pateikia 13 žodžio modelis reikšmių. Pirmosios iš jų žodį *modelis* apibrėžia: miniatiūrinis kažko atvaizdavimas; pavyzdys imitavimo ir pamėgdžiojimo; panašumo apibrėžimas, naudojamas įsivaizduoti kažką, ko negalima tiesiogiai stebėti.

Kiti apibrėžimai yra pateikti filosofijos moksle. Apostel [5] apibūdina modelį taip: „bet koks dalykas naudojantis sistemą A, kuri nebendrauja tiesiogiai ar netiesiogiai su sistema B, kad išgautų informaciją iš sistemos B, naudojama A kaip *modelis* B“. Van Gigch [19] pateikia tokį veiklos modelio apibrėžimą: „tikras pasaulis modeliavime yra studijų objektas, kai stebėtojas-tyrėjas stengiasi aptikti pasikartojančių ryšių šablonus, kurie gali būti atvaizduoti sistematikai. Šitą šabloną ar ryšį galima atvaizduoti į *modelį* metodu, kuris gali privesti patį save prie formalių studijų“.

FRISCO ataskaitoje, informacinės sistemos karkasui [14], pateikiamas toks apibrėžimas: „*modelis* yra sąmoningai išskirtas, tuščias, tikslus ir nedviprasmiško suvokimo“. Šis apibrėžimas svarbus konceptualiai modelio prigimčiai. Tam, kad modeliai būtų materializuoti ir dalinami tarp žmonių, jie nurodomi naudojant kalbą. Ataskaita tęsiasi apibrėžimu: „*modelio denotacija* yra

tikslus ir nedviprasmiškas reprezentavimas *modelio,* atitinkama formalia arba pusiau formalia *kalba* “.

Kituose apibrėžimuose, modelio denotacija pateikiama kaip *modelis*. Starfield et al. [35] pateikia tokį apibrėžimą: „*modelis* yra idėjos reprezentavimas. Reprezentacija yra tikslinga: modelio tikslas naudojamas išgauti iš tikrovės svarbias detales“.

Skirtingi mokslai gali turėti ribotą modelio idėjos matymą. Daugelyje inžinerinių disciplinų yra naudojami matematiniai modeliai. Tokie modeliai reiškiami matematine kalba. Toks *matematinio modelio* ir *modeliavimo* apibrėžimas pateikiamas [25]: „matematinis modelis yra apibūdinimas eksperimentiškai įrodomo reiškinio, matematinės kalbos pagalba. Reiškinys, apibrėžime vadinamas *sistema* ir matematika savo sistemos interpretavimo kontekste yra vadinama *matematinio modeliu“*.

Tokios disciplinos kaip mechanika, civilinė inžinerija ir architektūra, dažnai naudoja pakopinius modelius, kurie yra tikro pasaulio objektų medžiagų kopijos (pvz. mašinos, pastatai, tiltai ir t.t.). Šie objektai gali būti pateikti diagramose, naudojamose norint išgauti produkciją iš modelio detalių. Visi šie pavyzdžiai yra modeliai, išreikšti tam tikru būdu.

Kompiuterių mokslas vartoja modelius skirtingose programinės įrangos kūrimo fazėse. Kuriant informacines sistemas, gali būti panaudotas supratimas, paremtas konceptualiu modeliavimu. Pasak Boman et al. [10], modelis yra „paprasta ir pažystama struktūra, arba mechanizmas, kuris gali būti panaudotas interpretuoti tam tikrą tikrovės dalį“.

MDA ir MDE remiasi modeliavimu ir modeliais kaip pagrindine idėja. Vis dėl to, nėra bendro priimto modelio apibūdinimo. Literatūroje buvo pateikti skirtingi apibrėžimai. Neseniai pasirodę straipsniai rodo, jog reikia svaraus ir tikslaus supratimo apie modeliavimo koncepciją, norint pradėt naudoti MDE. Šiam tikslui, keletas tyrėjų dirbančių šioje, modeliais pagrįstos inžinerijos srityje, pateikia apibrėžimus.

Seidewitz [34] apibūdina *modelį* kaip: „rinkinį sakinių apie sistemą po jos nagrinėjimo“. MDA įvadas [26] apibrėžia: „sistemos modelis yra sistemos specifikacijos ir jos aplinkos apibrėžimas, tam tikram tikslui. Modelis dažnai vaizduojamas kaip piešimo ir teksto kombinacija. Tekstas gali būti modeliavimo arba natūralioje kalboje“. Kleppe et al. [23] taip apibūdina *modelį*: „modelis tai apibrėžimas sistemos, parašytos tiksliai nusakyta kalba“. Bezivin ir Gerbe [8] apibrėžia: „modelis yra sistemos supaprastinimas, padaryta turint tam tikrą tikslą. Modelis turėtų galėti atsakyti į klausimus, vietoj pačios sistemos“.

Kitame skyriuje analizuojami apibrėžimai ir pateikiamos prielaidos, tinkančios šiam darbui.

2.2.2. Apibrėžimų analizė

Šiame skyriuje analizuojami apibrėžimai apie tai, kas yra bendra ir skirtinga tarp modelių ir modeliavimo.

2.2.2.1. Modelis kaip gimininga idėja

Labiausiai pastebimas bendrumas apibrėžimuose tai, kad jie apibrėžia modelį kaip giminingą idėją. Modeliai visuomet reikalaus modelių srities, kaip tikrojo pasaulio modelio dalelės. Tačiau, esybė neturėtų būti laikoma modeliu, jei jos sritis negali būti atpažinta. Aišku, kai kuriais atvejais, modelio sritis gali būti nesvarstoma ir nepaisoma jos. Taip pat sąvoka apie modelį turėtų būt suprantama kaip vaidmuo. Esybė gali būti modelis, atsižvelgiant į jos sritį ir ta pati esybė gali būti kito modelio sritis.

Modelio sritis vadinama skirtingai, skirtinguose apibrėžimuose: sistema, reiškinys, sistemos objektas, pokalbio visata, studijuojama sistema ir panašiai. Mes priimame terminą *sistemos objektas* arba tiesiog *sistema*, nurodant į tą dalį tikrumo, kuriai modelis yra sukurtas.

2.2.2.2. Modeliai yra abstrakcijos

Kita svarti modelių savybė yra ta, jog jie yra jų objektų sistemų abstrakcijos. Tai reiškia, jog kai kurios objekto sistemos savybės bus atstovaujamos modelyje, o kitos išmetamos. Todėl, modeliai yra tikrovės supaprastinimai, naudojami vietoj tikrovės.

Modeliai turėtų tarnauti duotajam tikslui. Tai išskirtinai pabrėžta kai kuriose, anksčiau minėtuose apibrėžimuose. Turėti tikslą yra kritiška procesui, suprantamam iš tikrovės. Tikrovė parodo potencialiai neapibrėžiamą skaičių naujų galimybių. Modelių tikslas yra vedantis principas, dalyje šių naujovių.

Van Gigch [19] išskiria dilemą, tarp modelių neapibrėžtumo ir specifiškumo, išankstiniam nustatymui iš abstrakčios prigimties. Bendras modelis turi savyje mažiau detalių, negu tam tikras modelis. Bendras modelis apibrėžia didelį rinkinį reiškinių, kai tuo tarpu tam tikras modelis yra nukreiptas į rinkinį reiškinių ir bendru atveju pateikia daugiau informacijos.

2.2.2.3. Ryšiai tarp modelio ir objekto sistemos

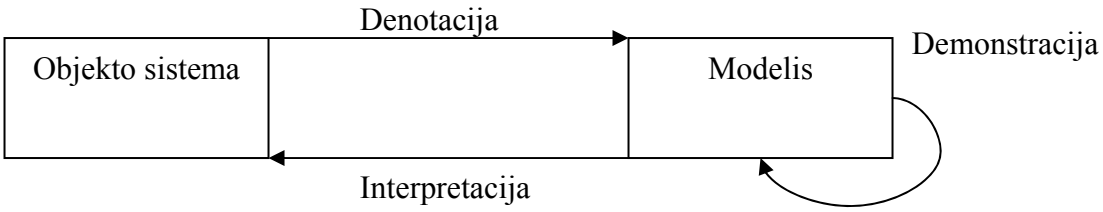
Ryšių svarbumas tarp modelio ir objekto sistemos yra pabrėžiamas keliuose apibrėžimuose. Šis ryšys vis dar tebėra diskusijų tema ir jo prigimtis vis dar nėra aiški bendrame filosofiniame kontekste. Modelis naudojamas netiesioginiam tikrovės (objekto sistemos) studijavimui. Netiesiogiškumas gali būti sukeliamas skirtingų kliūčių. Objekto sistema gali būti nepasiekiama, arba jos tiesioginis studijavimas yra per brangus, arba netgi objekto sistema gali

neegzistuoti ir modelis vaidins tik tam tikrą, objekto sistemos specifikacijos vaidmenį. Nepaisant kliūčių modelis turi būti teisinga, objekto sistemos reprezentacija. Žinios išgaunamos iš modelio turi atitikti objekto sistemą. Dažnai, šios žinios nėra tikslios, bet apytikslė tikrovė, tenkina tinkamą netikslumo laipsnį.

Šis labai svarbus ryšys, tarp modelio ir objekto sistemos yra detaliai aptartas apibūdinimuose, pateiktuose Apostel [5], Bezivin ir Gerbe [8] ir Boman et al. [10]

Be to žinios, išgautos iš modelio, yra nuo pat pradžių išreiškiamos modelio elementų išraiškomis. Šios žinios turi būti interpretuojamos ir išverstos į žinias, atitinkančias objekto sistemos išraiškas. Pavyzdžiui, matematiniame modelyje, kintamieji yra išgaunami iš jų tikrojo gyvenimo reikšmių ir gautos reikšmės, turi būti interpretuojamos objekto sistemos išraiškomis. Priklausomai nuo studijuojamo reiškinio, kintamųjų reikšmės gali atitikti temperatūrą, aukštį, laiką ir pan.

Ryšys tarp modelio ir objekto sistemos yra abipusis ir du atskiri ryšiai gali būti svarstomi, kaip rodo 2.2 paveiksle. Paveikslas pavadintas DDI sąskaita (DDI – denotacija, demonstracija, interpretacija), pristatyta Hughes.



2.2 pav.: Ryšys tarp objekto sistemos ir modelio

Objekto sistema yra *denotuota* (atstovaujama) modelyje. Ši denotacija turi išsaugoti kai kurias objekto sistemos charakteristikas tam, kad leistų išgauti žinias apie ją. Modelis naudojamas išgauti tvirtinimus apie modelio elementus. Šis procesas žinomas kaip *demonstracija*. Tai vyksta tik modelio kontekste. Galiausiai išgauti rezultatai yra sutapatinami su objekto sistema. Šis sutapatinimas vadinamas *interpretacija*. Žinios išgautos iš modelio, turi būti patikrinamos objekto sistemoje. Jei rezultatai, išgauti iš modelio, neatitinka empirinio įrodymo gauto iš tikrovės, tai modelis yra negaliojantis, atsižvelgiant į objekto sistemą.

Literatūros šaltiniai dažniausiai aprašo tik vieną ryšį tarp modelio ir jo objekto sistemos. Įvairūs vardai yra naudojami nusakyti ryšiui: *Modelis (kažko), Atvaizdavimas (kažko), Atvaizduota (į kažką), Modeliavimas (kažko),* ir panašiai. Likusioje darbo dalyje, mes naudosime

ryšį nukreiptą iš modelio į objekto sistemą, pavadintą *Modelis (kažko)*. Šio ryšio esmė yra atvaizduota DDI sąskaitos. Kaip pastebime, ryšio esmė turi du aspektus: atstovavimą (denotaciją), kuri gauna kai kurias objekto sistemos charakteristikas modelyje, ir interpretaciją, kuri pažymi galimybę atsekti žinias, išgautas iš modelio, atgal į objekto sistemą. Ryšys *Modelis (kažko)* turi du kitus ryšius: *Denotaciją* ir *Interpretaciją*.

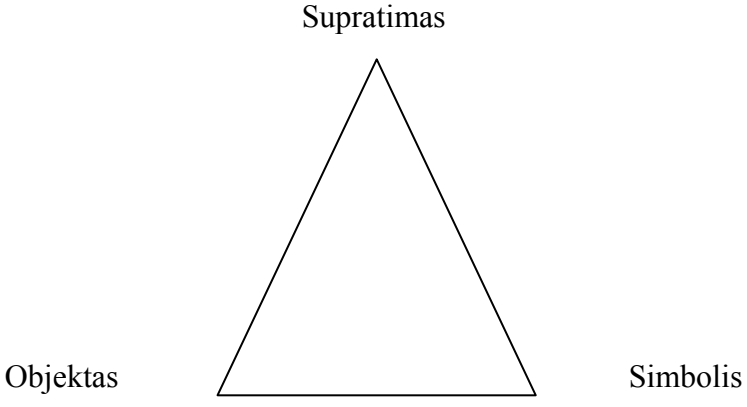
2.2.2.4. Modeliai kaip sistemos

Neskaitant panašumų, apibrėžimai taip pat pažymi ir skirtumus. Pats svarbiausias aspektas apibrėžimuose – kokios rūšies esybe yra modelis?

Daugelyje apibrėžimų modelis laikomas sistema. Tačiau, modelis gali būti klasifikuojamas pagal kai kurias sistemos klasifikacijas, kaip *konceptualus, konkretus* (arba fizinis) ir *simbolinis*. Daugelis apibrėžimų yra gana bendri, kad leisti trijų tipų modelius. Kiti apibrėžimai dėmesį nukreipia į tam tikrą modelio tipą.

Apibrėžimai FRISCO ataskaitoje [14] išskirtinai pabrėžia modelį, kaip konceptualią esybę ir jo modelio denotaciją, kaip simbolinę esybę. Kituose šaltiniuos [35] apibrėžimai išskiria faktą, jog modeliai yra idėjos atstovai. Matematinio modelio apibrėžimai [25] turi omenyje, jog modeliai yra simbolinės esybės, išreikštos matematikos kalba.

Bendru atveju, simbolinis modelis reiškia, jog egzistuoja konceptualus dublikatas modeliuotojo mintyse ir esybė tikrame pasaulyje, į kurią simbolis rodo. Tai faktų padarinys, jog simbolinį modelį galima laikyti kaip ženklą. Tačiau, jie yra semiotikos subjektas. Centrinis ryšys semiotikoje reiškia Ogden ir Richards trikampį. Šis trikampis pateikia ryšį tarp simbolio, to simbolio nurodyto objekto ir reikalingo supratimo, sujungti simbolį su objektu. Reikšmių trikampis pateiktas 2.3 paveiksle.



2.3 pav.: Reikšmių trikampis

Kompiuterių mokslas labai susirūpinęs simbolinių sistemų manipuliavimu. Pakopiniai modeliai, kurie yra realios esybės su išmatavimais erdvėje, ne dažnai panaudojamos kompiuterių moksle. Taip pat, konceptualios sistemos, kurios randasi tik žmogaus mintyse, nėra naudingos, kadangi ta sistema nėra atvaizduota ir dalinamasi tarp žmonių. Atvaizdavimas konceptualios sistemos reiškia, jog ji yra transformuojama į fizinę ar simbolinę sistemą. Todėl galime daryti prielaidą, jog modeliai su kurias turime reikalų kompiuterių moksle yra simbolinės sistemos. Dažniausiai simbolinės sistemos yra išreiškiamos kalba. Kalba naudojama išreikšti modelį yra *modeliavimo kalba*. Modeliavimo kalbos sąvoka labai svarbi MDE. Šio skyriaus sekančiose dalyse mes pamatysime, jog ji vaidina pagrindinį vaidmenį apibrėžiant kitą, labai svarbią idėją, pavadintą *meta-modeliu*.

2.2.3. Pasirenkant apibrėžimą

Pagal svarstymus skyriuose 2.2.1 ir 2.2.2 pasirenkam šiai disertacijai modelių apibrėžimą:

Modelis atvaizduoja dalį tikrovės, vadinamos objekto sistema ir yra išreiškiamas modeliavimo kalba. Modelis pateikia žinias, tam tikriems tikslams, kurios gali būti interpretuojamos, objekto sistemos išraiškomis.

Šis apibrėžimas yra kombinacija kai kurių apibrėžimų, pateiktų čia ir daro prielaidą, jog modeliai yra simbolinės sistemos, išreikštos kalba.

2.2.4. Modeliavimas kompiuterių moksle

Šiame skyriuje svarstysime dvi problemas: modelio vaidmenį, kurį programinė įranga gali vaidinti, atsižvelgdama į kitas sistemas ir modeliavimo vaidmenį programinės įrangos kūrime.

2.2.4.1. Programinės įrangos sistemos kaip modeliai

Dirbančios programinės įrangos sistemos, gali turėti tam tikros tikrovės dalies, tam tikrą modelį. Aptarsime informacinę sistemą, kuri susideda iš duomenų bazės su informacija apie studentus universitete ir susijusią programinę įrangą, kuri dirba duomenų baze. Ar galime laikyti sistemą kaip modelį? Jei atsakymas taip, tai kas tada yra objekto sistema tame modelyje? Sekantys klausimai yra lengvai gaunami iš 2.2 paveikslo: kaip objekto sistema atvaizduojama modelyje ir kaip žinios kurias pateikia modelis yra susiję su objekto sistema?

Mes galime laikyti informacinę sistemą kaip modelį. Šio modelio objekto sistema susidaro iš realių žmonių, kurie dabar (arba ateityje) yra to universiteto studentai. Kai kuri informacija apie šiuos žmones yra vaizduojama informacinėje sistemoje, atspindinčioje jų ryšius su universitetu. Be to, informacinė sistema yra naudojama išgauti žinias apie objekto sistemą. Vietoje to, kad klausinėtume kiekvieno studento asmeniškai apie jo universitetinę padėtį, mes galime išgauti šią informaciją iš sistemos. Akivaizdu, jog tai tenkina bendrą apibrėžimą apie modelį, pateiktą ankstesniame skyriuje.

Išgauta informacija yra simbolinės prigimties, bet informacinės sistemos vartotojai, informuoti apie jos reikšmes. Jie gali *interpretuoti* informaciją realių studentų išraiškomis, konkretaus universiteto fakultetais, ir realiais įvykiais, kurie jau įvyko: egzaminai, pabaigimai, ir pan.

Tai vienas iš aiškesnių pavyzdžių, kai programinės įrangos sistema yra tikrovės dalies modelis. Kiti pavyzdžiai yra programinės įrangos sistemos, kurios atlieka procesų imitavimus ir kompiuterių pagalbos projektavimo sistemos.

2.2.4.2. Programinės įrangos sistemų modeliavimas

Programinės įrangos sistemos gali būti modelių objektų sistemomis. Iš tikrųjų, skirtingose kompiuterių mokslų srityse, programinės įrangos sistemų modeliavimas vaidina svarbų vaidmenį.

Ankstesnėse diskusijose nusprendėme, jog modeliai yra objektinių sistemų simboliniai atstovai. Atstovavimas gali būti neoficialus, kaip piešimas eskizo, arba piešimas ant lentos, arba gali būti paremtas formalia modeliavimo kalba. Kompiuterių moksle, svarbiausia yra naujausi atvejai, kadangi numatoma, jog modeliai bus apibūdinami tiksliu ir nedviprasmišku būdu.

Viena tokia formali modeliavimo kalba yra paremta matematine kalba. Matematiniai modeliai naudoja matematinius užrašymus ir dažniausiai susideda iš kintamųjų, parametrų ir lygčių rinkinių. Šis formalus modelio atstovavimas leidžia programoms, žinančioms apie matematiką, savo ruožtu išgauti papildomų žinių iš modelio.

Kompiuterių moksle sudominti išreikšti modelius tikslia modeliavimo kalba. Tokie modeliai gali būti valdomi automatinių, arba pusiau automatinių įrankių, galiausiai pateikiančių pilnas programinės įrangos sistemas.

2.3. Meta-modeliai ir meta-modeliavimas

Šiame skyriuje analizuosime *meta-modelio* ir *meta-modeliavimo* idėją, kompiuterių mokslo kontekste. Kaip ir skyriuje 2.2, pasirinksime apibrėžimą, kuris bus naudojamas šiame

darbe. Analizuosime ryšius tarp modelių ir meta-modelių. Ryšys tarp modelio ir meta-modelio yra interpretuojamas plačiame kontekste, kurį bandysime išanalizuoti.

Kartais modelis laikomas meta-modelio atskiru atveju. Skyrius 2.3.2 supažindina su palyginimais tarp *Modelis (kažko)* ryšio ir *atskiras atvejis (kažko)* ryšio, meta-modelio kontekste. Šie ryšiai dažnai sutampa tarp duoto modelio ir meta-modelio ir turi tendenciją būti laikomais vienodais.

Meta-modelio, ištemptų ir *išplėstų* esybių, ir *atskiras atvejis (kažko)* ryšių idėja, bus naudojami tolimesniame, 2.5 skyriuje, kur pateikiamas supratimas apie modelio lygius ir meta-modeliavimo architektūrą.

2.3.1. Meta-modelio ir meta-modeliavimo apibrėžimai

Kaip pats pavadinimas sako, meta-modeliavimas yra modeliavimo veikla. Panašiai meta-modeliavimo produktas, vadinamas *meta-modeliu* ir yra taip pat modelis.

Jei esybė yra modelis, mes turime galėti aiškiai atpažinti jo objekto sistemą. Literatūroje randamos dvi meta-modelio objekto sistemos: modeliavimo procesas ir modeliavimo kalba.

2.3.1.1. Meta-modeliavimas kaip modeliavimo procesas

Van Gigch apibrėžia meta-modeliavimą kaip modeliavimo proceso modeliavimą [19]. Kitais žodžiais tariant, meta-modeliavimas moko, kaip modeliuoti. Hence apibūdina meta-modeliavimą, kaip modeliavimo proceso modelį. Brinkkemper [11] apibūdina meta-modeliavimą, kaip modeliavimo technologijos proceso suvokimą. Meta-modelis yra konceptualus modeliavimo technologijos modelis.

2.3.1.2. Meta-modeliavimas kaip kalbų modeliavimas

Kita meta-modelio objekto sistema yra modeliavimo kalba. Šiuo požiūriu ryšys tarp meta-modeliavimo ir modeliavimo kalbos yra būtinas. Šis požiūris yra priimtinas MDE. Pateikiame kai kuriuos apibrėžimus, randamus panašioje literatūroje.

Seidewitz [34] apibrėžia: „meta-modelis tai modelis modelių, išreikštų duotoje modeliavimo kalboje“. Dėl to meta-modelis apibūdina, ką galima išreikšti modeliais, toje kalboje. FRISCO ataskaita apibrėžia: „meta-modelis yra konceptualios kalbos prigimties modelis, susidedantis iš rinkinio pagrindinių idėjų ir rinkinio taisyklių, apibūdinančių rinkinį galimų modelių, nurodytų toje kalboje“ [14].

MDA įvadas apibrėžia meta-modelį kaip „modelių modelį“. Akivaizdžiai šiame apibrėžime trūksta ryšio į modeliavimo kalbą, kuri išreiškia modelius prieš tyrinėjimą.

Atrodo, jog du požiūriai į meta-modeliavimą, naudoja skirtingas objekto sistemas. Kai kuriais atvejais, pirmasis požiūris į meta-modeliavimą gali reikšti antrąjį. Tai priklauso nuo modeliavimo proceso ir modeliavimo technikos terminų supratimų. Jei modeliavimo procesas aiškiai apima produktus (modelius) ir modeliavimo kalbą, tada pirmasis požiūris yra labiau bendras.

Kaip minėta anksčiau, pirminis tyrimas nukreiptas į modelius, išreikštus duotoje kalboje. Procesas, gaminantis šiuos modelius (modeliavimo veikla) neįeina į šio darbo apimtį. Todėl naudosime antrąjį požiūrį objekto sistemai meta-modelyje: modeliavimo kalbą. Šiame darbe terminui meta-modelis naudosime tokį apibrėžimą:

Meta-modelis yra modeliuojamos kalbos modelis.

2.3.2. Meta modeliai, modeliai ir atskiras atvejis(kažko) ryšys

Bendru atveju egzistuoja *Modelis (kažko)* ryšys tarp meta-modelio ir objekto sistemos: modeliavimo kalbos. Daugeliu atvejų, *Modelis (kažko)* ryšys yra nesvarstomas ir pakeičiamas *atskiras atvejis (kažko)* ryšiu, tarp modelio ir meta-modelio. Šiame skyriuje analizuosime šių ryšių prigimtį, meta-modeliavimo kontekste. Nors jie sutampa tarp tų pačių esybių porų (modelio ir meta-modelio), jie yra skirtingos prigimties. *Atskiras atvejis (kažko)* ryšys gali padėti interpretuojant žinias, išgautas iš meta-modelio (2.2 paveikslas). Skaitysime, jog *atskiras atvejis (kažko)* ryšys, yra apibrėžtas duotos kalbos kontekste, kuri apibūdina ryšio semantiką. Padarėme išvadą, jog *atskiras atvejis (kažko)* ryšys ne visuomet pateikiamas tarp modelio ir meta-modelio.

Šiame skyriuje analizuosime *Modelis (kažko)* ryšį tarp modelių ir meta-modelių. Vėliau parodysime *atskiras atvejis (kažko)* ryšio pavyzdžius, tarp modelių ir meta modelių. Abu ryšiai yra lyginami pavyzdžių pagrindu.

2.3.2.1. *Modelis (kažko)* ryšys tarp modelio ir meta-modelio

Skyriuje 2.3.1 priėmėme apibrėžimą, kad meta-modelis yra modeliavimo kalbos modelis. Bendroje kalbų teorijoje, naudojamoje kompiuterių moksle, kalba yra rinkinys sakinių tarp rinkinio simbolių (alfabeto). Šie simboliai gali būti raidės, kaip natūralių kalbų, bet gali būti ženklai ar kitokie grafiniai simboliai, randami vaizdinėse kalbose.

Praktinio panaudojimo kalbos dažniausiai turi nesibaigiantį rinkinį sakinių. Neįmanoma sugeneruoti visus sakinius tokioje kalboje. Kompiuterių moksle mes naudojame rinkinį taisyklių, kurios gali būti naudojamos patikrinti, ar sakinys priklauso kalbai. Vienas iš tokių rinkinių taisyklių pavyzdžių yra specifikuotos kalbos gramatika. Išplėstoje Backus-Naur Formoje (EBNF) svarstome kalbą, kaip savo gramatikos apibrėžtą. Gramatikos taisyklės apima charakteristikas,

kurias turi visi sakiniai kalboje. Taigi galime įdėmiai pasižiūrėti į kalbos gramatiką, kaip į tos kalbos modelį. Modeliavimo kalbos atveju, bet koks kalbos modelis yra *meta-modelis*. Duotoji modeliavimo kalba gali turėti daugiau nei vieną meta-modelį.

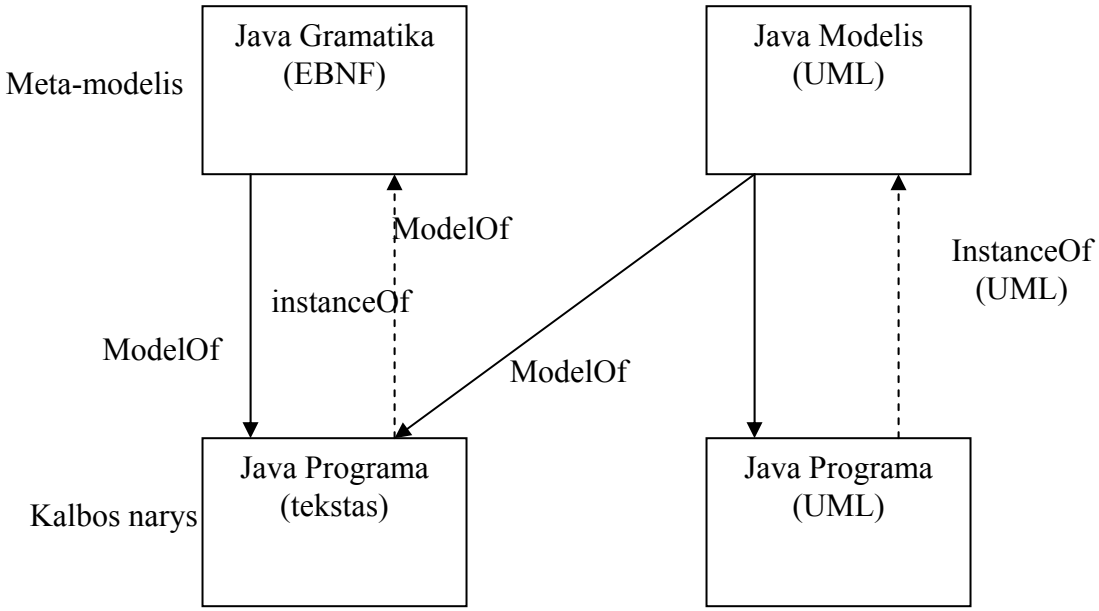
Teoriškai meta-modelio objekto sistema yra rinkinys modelių, išreikštų modeliavimo kalboje. Praktikoje domina išgauti žinias apie rinkinio narius, vietoj rinkinio kaip visumos. Kalbos meta-modelis apibrėžia apribojimus, kuriuos kiekvienas modelis, toje kalboje, turi tenkinti. Taigi dėmesys nukreiptas į ryšius tarp konkrečių modelių ir meta-modelių, parašytų tam tikra kalba. Ryšys vadinamas *atitinkantis (kažką)*. Detali prigimtinė šio ryšio elgsena yra pateikta Favre. Šiame darbe *atitinkantis (kažką)* yra apibrėžtas kaip dviejų ryšių sudėtis: *elementas (kažko)* pažymintis modelio narystę kalboje ir *atitikimas (kažko)* pažymintis ryšį tarp objekto sistemos ir modelio.

Dėl paprastumo gilinsimės į ryšį, tarp kalbos narių ir jų meta-modelių. *Modelis (kažko)* ryšys bus pavaizduotas tarp meta-modelio ir modelio, vietoje meta-modelio ir rinkinio modelių (kalbos).

2.3.2.2. *Atskiras atvejis (kažko)* ryšys tarp modelio ir meta-modelio

Atitinkantis (kažką) ryšys, aprašytas aukščiau, dažnai pakeičiamas ryšiu *atskiras atvejis (kažko)*. Daugelis OMG dokumentų ir straipsnių apie subjektą nurodoma, jog modelis yra meta-modelio atskiras atvejis. Šiuo klausimu gana dažnai gimsta diskusija.

Apsvarstykime 2.4 paveikslą, kuris parodo Java programą kaip tekstą ir gramatiką, apibrėžtą EBNF. Gramatika yra Java kalbos meta-modelis. Tai parodoma vientisa rodykle iš gramatikos į programą, pavadintą „Modelis (kažko)“. Turi būti galima interpretuoti žinias išgautas Javos programoje iš gramatikos, tekstinės formos terminais. Pavyzdžiui gali būti taisyklė, kuri apima ne galutinį žodį, pagal sintaksinę kategoriją, Java metodų apibrėžime. Šis žodis Java programoje, turėtų būti atsekamas iš frazės, kuri atspindi metodų aprašymą. Kaip šis interpretavimas palaikomas? Štai čia mums padės *atskiras atvejis (kažko)* ryšio sąvoka.



2.4 pav.: Meta-modelių, modelių ir ryšio atskiras atvejis(kažko) pavyzdys.

Bendra formalių kalbų teorija, pateikia gramatinio nagrinėjimo algoritmus, kurie sukonstruoja duotai programai ir gramatikai derivacijos medžius. Jei toks medis yra sukonstruojamas, sakome, jog programa yra gramatikos atskiras atvejis. Derivacijos medžiuose galime rasti atitikimus, tarp ne galutinių žodžių gramatikoje ir frazių programoje. Kitais žodžiais tariant, pagal derivacijos medį galime interpretuoti žinias gramatikoje, programos terminais.

Iš pirmo žvilgsnio ryšiai *Modelis (kažko)* ir *atskiras atvejis (kažko)* yra vienodi. Iš tiesų jie sutampa tarp tų pačių esybių, bet yra skirtingos prigimties.

Atskiras atvejis(kažko) ryšys pažymi narystę esybės su klase, kur klasė yra rinkinys. Klasė yra skirtinga pagal tos klasės apibrėžimą. Klasės apibrėžimas yra klasės stiprumas. Klasė yra klasės apibrėžimo papildymas. Iš šio taško pasakyti, jog modelis yra meta-modelio atskiras atvejis nėra teisinga, kadangi meta-modelis nėra rinkinys. Tačiau daugelyje programavimo ir modeliavimo kalbų, *atskiras atvejis (kažko)* ryšys yra nustatytas tarp klasės nario ir klasės apibrėžimo. Jei esybės yra paprašoma jos klasės, rezultatas bus klasės apibrėžimas, o ne klasė (rinkinys).

Pastebimas svarbus skirtumas tarp dviejų ryšių, kai ryšys *atskiras atvejis(kažko)* svarstomas kalbos prigimtyje. Tarkime, jog apibrėžėme kitą, Java kalbos meta-modelį išreikštą UML (2.4 paveikslas). UML modelis gali turėti klases, pavadintas *Metodais*. Žinios, kurias

išgauname sako, jog egzistuoja rinkinys metodų Javos programoje, kurie turi tam tikrą struktūrą. Turime galėti atpažinti metodus ir jų struktūrą programos tekste, pagal klasės *Metodo* apibrėžimą. Tai padės padaryti ryšys *Modelis (kažko),* kuris egzistuoja tarp Java meta-modelio ir Java programos. Tačiau negalime laikyti Java programos, kaip atskiro UML modelio atvejo, kaip padarėme su Java gramatika. Atskiras UML modelio atvejis yra apibrėžtas pagal UML semantiką ir yra objektų rinkinys. Šis atskiras atvejis yra Java programos atvaizdavimas ir yra skirtinga esybė. Java UML modelis yra taip pat Java programos modelis, atvaizduotos UML. Papildomai egzistuoja *atskiras atvejis (kažko)* ryšys tarp šių esybių, valdomas UML semantikos. Kaip ir ryšys tarp tekstinės programos ir jos gramatikos, šis *atskiras atvejis (kažko)* ryšys padeda interpretuoti UML modelio žinias, atvaizduotas UML, Java programos terminais. Šie *atskiras atvejis (kažko)* ryšiai yra skirtingi. Pirmasis yra apibūdintas nagrinėjant procesus. Antras remiasi UML semantika. Nėra tiesioginio kalbai būdingo *atskiras atvejis (kažko)* ryšio, tarp tekstinės Java programos ir jos UML modelio. Tačiau pastarasis yra modelis buvusio, nors mes negalime atsekti žinių iš modelio į objekto sistemą, per *atskiras atvejis (kažko)* ryšį.

Diskusijos esmė yra skirtumai tarp *Modelis (kažko)* ryšio ir *atskiras atvejis (kažko)* ryšio. Bendrai galime susieti predikatą klasei ir visoms jos esybėms, jog klasės nariai turėtų tenkinti predikatą. Iš braižymo perspektyvos, galime įsivaizduoti *atskiras atvejis (kažko)* ryšį tarp tekstinės Java programos formos ir Java kalbos UML modelio. Tai predikato tikslaus apibrėžimo esmė. Kai *atskiras atvejis (kažko)* naudojamas tam tikros kalbos kontekste, tada žinome kaip įvertinti predikatą. Pavyzdžiui turime algoritmą, patikrinti ar programa atitinka gramatiką. Panašiai turime įrankių, kurie patikrina, ar UML objektų diagrama atitinka klasės diagramą. Šiais atvejais naudojant *atskiras atvejis (kažko)* ryšį vietoje *Modelis (kažko)* ryšio, išgautume daugiau informacijos apie meta-modelio interpretaciją.

Padarius išvadas, *atskiras atvejis (kažko)* ryšys egzistuoja tarp klasės ir jos narių ir palaiko žinių interpretavimą, išgautą iš klasės apibrėžimo, klasės narių terminais. Šiuo atveju, taip pat turime *Modelis (kažko*) ryšį tarp klasės apibrėžimo ir klasės narių.

Šiame darbe laikysime *atskiras atvejis(kažko)* ryšį griežtai tam tikros kalbos kontekste, kuri apibūdina ryšio reikšmę. Todėl neįdedame *atskiras atvejis (kažko)* ryšio tarp Java programos tekstinėje formoje ir Java UML modelio. Tačiau *Modelis (kažko)* ryšys gali egzistuoti tarp šių esybių.

Subtilus skirtumas tarp *atskiras atvejis (kažko)* ir *Modelis (kažko)* ryšių taip pat akcentuojamas ir Seidewitz [34]. Šiame darbe *atskiras atvejis (kažko)* ryšys yra apibrėžtas pagal

modeliavimo kalbos semantiką. *Modelis (kažko)* ryšys tarp modelio ir jo objekto sistemos vadinamas *interpretacija*. Interpretacija tai sutapatinimas kalbos modelio ir kalbos elementų.

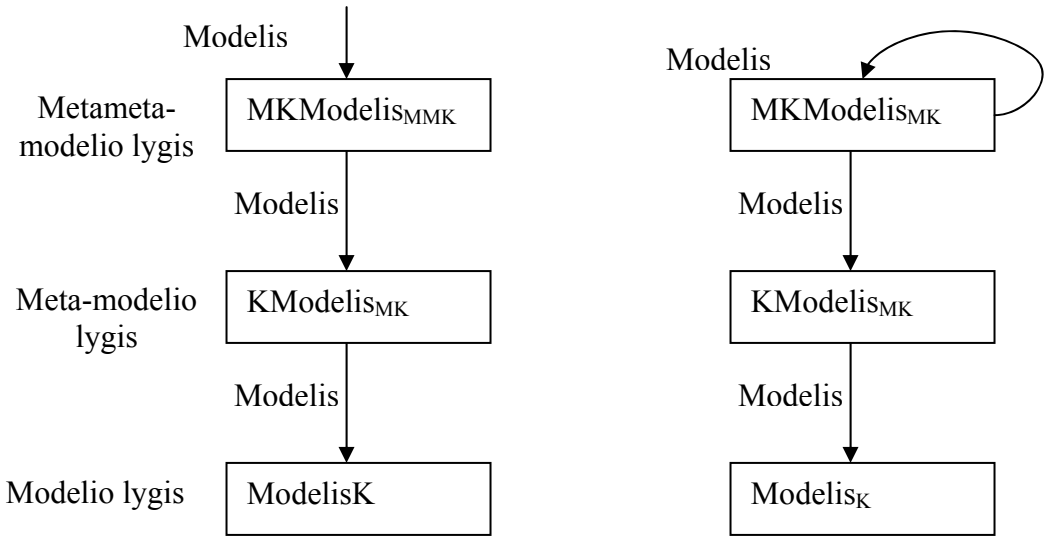
2.4. Modelių lygiai

Meta-modelių veikla gali būti taikoma pasikartojančiai, kad sukonstruotų modelių hierarchijas, kurios matuoja daugelį lygių. Lygių organizacija taip pat vadinama meta-modeliavimo krūva, meta-modeliavimo karkasu arba meta-modeliavimo architektūra. Naudosime terminą *meta-modeliavimo architektūra,* bet kiti terminai gali būti panaudoti kaip sinonimai.

2.4.1 skyriuje, nagrinėsime meta-modeliavimo architektūrą. Išanalizuosime 3 lygiuose, kadangi taip dažnai daroma įvairiose technologijose. Priimsime anksčiau apibrėžtą ryšio *atskiras atvejis(kažko)* sąvoką ir naudosime jį, meta-modeliavimo architektūros kontekste. 2.4.2 skyriuje pateiksime meta-modeliavimo pavyzdžių.

2.4.1. Meta-modeliavimo architektūra

2.5 paveikslas rodo meta-modeliavimo architektūrą. Apatiniame krūvos lygyje turime modelius, išreikštus įvairiose modeliavimo kalbose. Lygis vadinamas *modeliavimo lygiu*.



2.5 pav.: Meta-modeliavimo architektūra

Modelio lygyje pavyzdinis modelis yra $Modelis_K$, parašytas modeliavimo kalboje *K*. Laikysimės žymėjimo, kuriame modelio pažeminto šrifto dalis nurodo kalbą, kurioje modelis yra išreikštas. Galime pagaminti modelį *K* (tai yra meta-modelį) $KModelis_{MK}$, išreikštą kitoje kalboje, vadinamoje *Meta-kalba* (MK). Kalbų modeliai, naudojami modelio lygyje, suformuoja sekantį lygį krūvoje. Jis vadinamas *meta-modelio lygiu*. Egzistuoja *Modelis (kažko)* ryšys tarp meta-

modelio kalbos ir modelių, išreikštų toje kalboje. Galime pritaikyti tą patį požiūrį modeliams, meta-modelio lygyje. Kalbų modeliai, kurie išreiškia meta-modelį, suformuoja trečią lygį, vadinamą *metameta-modelio lygiu*. Kairioji paveikslo 2.5 dalis rodo *MK* modelį, vadinamą $MKModelis_{MMK}$, išreikštą trečia kalba, vadinama *Metameta-kalba* (MMK).

Šis požiūris gali būti pritaikytas begalę kartų, kaip siūlo kairioji paveikslo pusė. Vis dėl to praktikoje yra naudojami tiktai trys lygiai. Tiesiog galime praleisti modelį *MMK* tariant, kad jis yra duotas meta-modeliavimo krūvos išorėje. Kitas kelias yra išreikšti *MKModelis* modelį *MK* kalba. Tai parodyta 2.5 paveikslo dešinėje pusėje. Šiuo būdu viršutinis lygis turi savo pačio atspindėtą modelį. Jis išreikštas kalba, kuria modelis modeliuojamas. Meta-modelio lygyje turime modeliavimo kalbų modelius, išreikštus *MK*. Vis dėl to, *MK* yra modeliavimo kalba pati sau ir turėtų būti galima pritaikyti *MK* ant pačios savęs, kad išreikšti savo modelius.

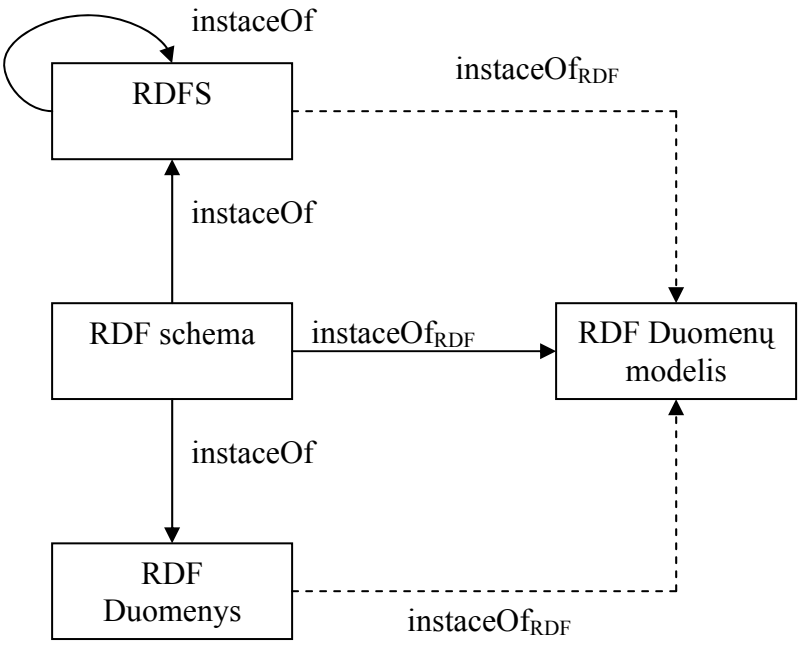
2.4.2. Meta-modeliavimo architektūros pavyzdžiai

Šis skyrius supažindina su dviem technologijų pavyzdžiais, kuriais remiasi meta-modeliavimo architektūra: Resursų apibrėžimo karkaso schema (RDFS) (Resource Description Framework Schema) ir ištęsiama žymėjimo kalba (XML)(Extensible Markup Language) ir XML-Schema.

2.4.2.1. Resursų apibrėžimo karkasas (RDF)

RDF [7] yra duomenų modelis, apibrėžtas pasaulio plataus tinklo konsorciumo (World Wide Web Consortium) (W3C), kad atvaizduoti duomenų apsikeitimą, meta-duomenų tinkle. RDF duomenų modelis apibrėžia tris tipus: *sakinys, resursas* ir *savybė*. RDF duomenys sukurti pagal šiuos tipus iš rodyklinių grafų, kur resursai yra grafo viršūnės ir savybės yra kraštinių pavadinimai.

RDF duomenys gali būti tipizuoti arba netipizuoti. Tipizuota informacija gali būti pridedama apibrėžiant RDF schemą. RDF Schema (RDFS) yra kalba, pasiūlyta W3C apibrėžti RDF schemas [24]. RDF duomenys, RDF schemos ir RDF Schemos kalba suformuoja lygių hierarchiją, pavaizduotą 2.10 paveiksle.

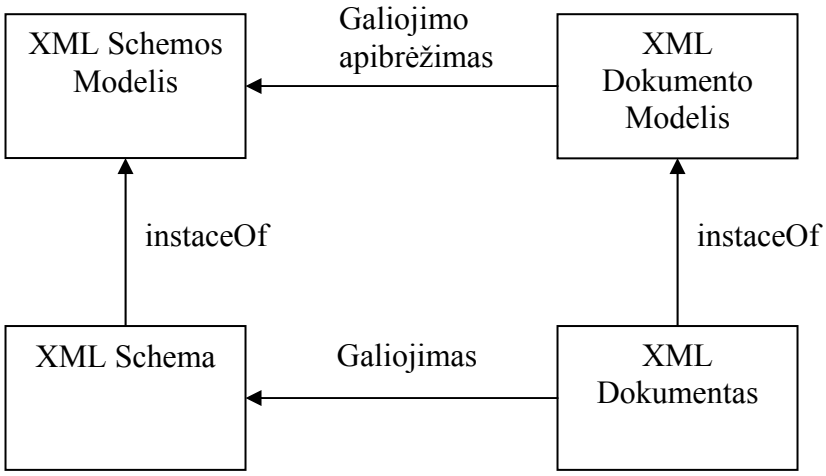


2.10 pav.: RDF lygiai

RDF duomenys gali atitikti RDF schemą, kuri apibrėžia klases ir klasių savybes. RDF schemos yra RDFS atskiri atvejai, kurios apibrėžia schemų kalbą. RDF yra atskiras atvejis pats savęs. Neskaičiuojant ryšių tarp RDF grafo ir jo schemos, visi lygiai atitinka bendrą RDF duomenų modelį. Dėl to, visa erdvė gali būti traktuojama vienodu keliu, pagal RDF duomenų modelį. Ji vaidina modelio išplėtimo(ištesimo) vaidmenį kiekviename lygyje. RDFS ir RDF schemos vaidina įtempimų vaidmenį.

2.4.2.2. Išplėstinė pažymėjimo kalba (XML)

XML tai technologija duomenų atvaizdavimui ir apsikeitimui, apibūdinta W3C [103]. Tai karkasas, naudojamas apibūdinti pažymėjimo kalbos sintaksę. Organizacijos lygis gali būti stebimas XML srityje. Jis pavaizduotas 2.11 paveiksle.



2.11 pav.: XML srities lygiai

XML specifikacija apibrėžia taisykles, sukurti XML dokumentus. Taisyklės išreikštos gramatiškai, tačiau praktiškai dažniausiai naudojamas XML dokumento modelis. XML dokumentai yra XML dokumento modelio atskiri atvejai. Dėl to, papildomas apribojimų rinkinys, gali būti priskirtas XML dokumentams. Jie yra žinomi kaip galiojimo apribojimai ir yra išreikšti XML schemoje. XML schemos yra parašytos XML schemos kalba [37] ir dėl to yra XML schemos modelio atskiri atvejai. XML schema yra įtempimas XML dokumentų rinkiniui. Kita vertus, XML dokumentas turi visada atitikti XML dokumento modelį. XML srityje, XML dokumentas, gali būti atskiras atvejis dviejų įtempimų, tuo pačiu metu. Abu yra svarbūs ir gali būti naudojami kartu. Pavyzdžiui, XKelias(XPath) 2.0 [39] leidžia išrinkimą XML viršūnių, XML dokumente, pagal pagrindinius tipus, apibrėžtus XML dokumento modelyje ir XML schemoje.

2.5. Modelių transformacijos

Skyrius 2.1.1 supažindino su pagrindinėmis MDA idėjomis. MDA modelių transformacijos yra nuosekliai pritaikomos modeliams, kol sugeneruojamas sistemos kodas. Šiame skyriuje detaliau apibrėšime modelių transformacijas ir transformavimo kalbas.

2.5.1. Apibrėžimai

Čia pakartosime modelio transformacijos apibrėžimus, pateiktus MDA įvade: „*modelių transformacija* yra procesas, paverčiantis vieną modelį į kitą, tos pačios sistemos modelį“ [26].

Dėl to, MDA įvadas apibrėžia *sutapatinimo* sąvoką kaip: „MDA sutapatinimas pateikia specifikacijas transformacijai, pereiti iš PIM į PSM modelį, tam tikroje platformoje“. Egzistuoja

keletas sutapatinimo tipų: modelio tipų sutapatinimas, meta-modelių sutapatinimas, modelių atskirų atvejų sutapatinimas, sujungtas sutapatinimas. Sutapatinimas yra specifikuojami *sutapatinimo kalboje*.

Kleppe et al. [23] apibrėžia: „*transformacija* yra automatinė, rezultatų modelio generacija iš šaltinio modelio, pagal *transformacijos apibrėžimą*“. Transformacijos apibrėžimas apibrėžiamas kaip: „*transformacijos apibrėžimas* yra rinkinys transformacijos taisyklių, kurios kartu nusako kaip modelis šaltinio kalboje, gali būti transformuotas į modelį rezultatų kalboje“.

Antrasis apibrėžimas pateiktas Kleppe et al. išskirtinai pažymi egzistavimą specifikacijos, kuris nurodo transformacijos procesą. Ši specifikacija MDA įvade [26] vadinama *sutapatinimu*. Dėl to, MDA įvado *sutapatinimo* apibrėžimas, apriboja šios sąvokos erdvę iki transformacijų, tarp PIM ir PSM. Kadangi mus domina platesnė MDE sąvoka, kur PIM/PSM yra tik vienas matavimas modeliavimo erdvėje, mes priimame antrąjį apibrėžimų rinkinį, pateiktą Kleppe et al. Sąvoka *transformacijos apibrėžimas* išskirtinai nurodo faktą, jog modeliai yra išreikšti duotoje kalboje. Tai tinka prielaidoms, kurias mes padarėme anksčiau, kur mūsų susidomėjimas buvo tik simboliniai modeliai, išreikšti modeliavimo kalba.

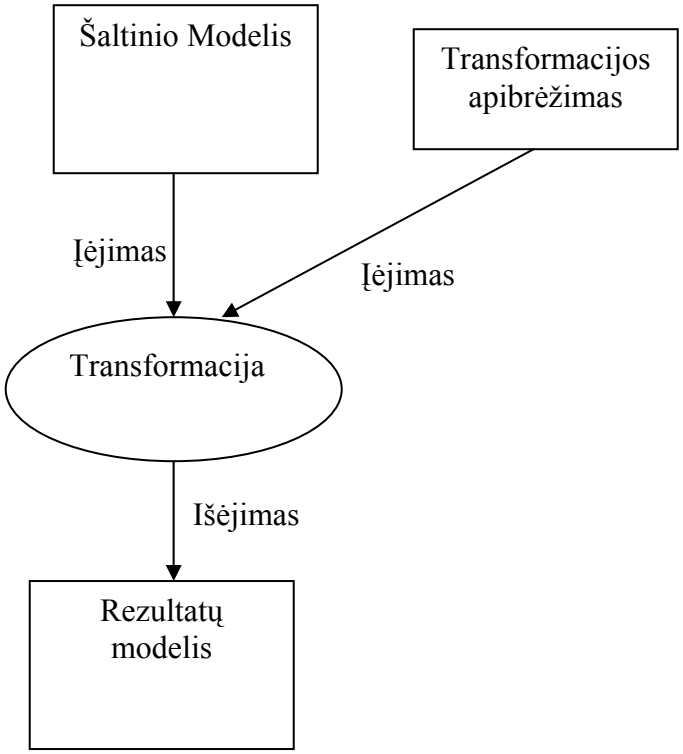
Pateikiame modelių transformavimo apibrėžimą, kuris naudojamas šiame darbe:

Modelių transformacija yra procesas, automatinio rezultatų modelio generavimo iš šaltinio modelio, pagal transformacijos apibrėžimą, kuris yra išreikštas modelio transformacijos kalba.

Šiuo metu, MDE bendruomenė neturi bendro susitarimo dėl sąvokos reikšmių *transformacija* ir *transformacijos apibrėžimas*. Galima panaudoti daugelį kalbų, kad apibrėžti transformacijos sąvoką. Pavyzdžiui, gali būti panaudota bendro tikslo kalba parašyti programą, kuri transformuoja vieną modelį į kitą. Vis dėl to, MDE bendruomenė krypsta į naudojimą sričiai-tikslios kalbos, aprašyti transformacijos apibrėžimus. Tai pateikta QVT reikalavimų pasiūlyme pagal OMG [29] ir skaičiumi modelio transformacijų kalbų pasiūlytų [3][12][21][32][33][36].

Svarbus reikalavimas, suformuotas OMG yra toks, jog transformavimo kalbos naudojamos MDA, būtų apibrėžtos kaip MOF meta-modeliai. Padariniai Transformacijų apibrėžimai, aprašyti tokia kalba tampa modeliais M1, lygyje MOF meta-modeliavimo krūvoje ir juos galima traktuoti kaip bet kokį kitą modelį.

Galime pakeisti MDA šabloną pavaizduotą paveiksle 2.1, į labiau bendrą šabloną, pavaizduotą 2.12 paveiksle.



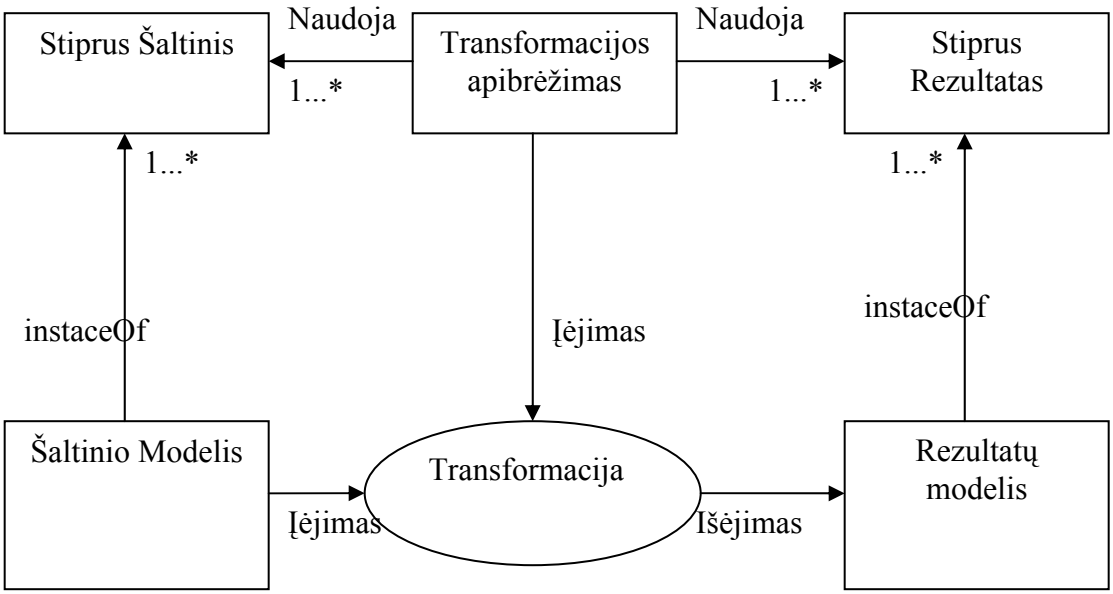
2.12 pav.: Bendras transformacijos šablonas

Transformacijos apibrėžimas dažniausiai sukurtas rinkiniui šaltinio modelių. Labai retas atvejis, kai transformacijos apibrėžimas sukurtas tiktai vienam modeliui. Tipiškas pavyzdys, kai transformacijos apibrėžimas sukurtas transformuoti modelius, parašytus duotoje kalboje.

Kad pasiektų bendrumą, transformacijos apibrėžimas yra sukurtas pagal stiprias žinias, apie šaltinio ir rezultatų modelį. Pavyzdžiui transformacija, kuri transformuoja modelius, išreikštus šaltinio kalba į modelius, išreikštus rezultatų kalba, naudoja meta-esybes, apibrėžtas meta-modeliuose. Šio tipo transformacijos apibrėžimas vadinamas *meta-modelių sutapatinimu,* MDA įvade [26].

Skyrius 2.4 parodė, jog duotas modelis gali būti susietas su daugiau nei vienu įtempimu. Įtempimai gali būti meta-modeliai ir srities modeliai. Bendru atveju, turi būti įmanoma panaudoti informaciją iš daugiau nei vieno įtempimo, transformacijos apibrėžime. Pavyzdys – XML technologija aptarta 2.4.2 skyriuje.

Tolimesnis transformacijos šablono tobulinimas, kuris parodo informaciją, naudojamą specifikuoti transformacijos apibrėžimą, pavaizduotas 2.13 paveiksle.



2.13 pav.: Transformacijos šablonas paremtas stipriomis žiniomis

Paveikslas rodo, kad šaltinio ir rezultatų modeliai gali būti atskiri atvejai daugiau nei vieno stipraus (nurodytas kaip *Stiprus Šaltinis* ir *Stiprus Rezultatas*), kiekvienas su savais, *atskiras atvejis(kažko)* ryšiais. Transformacijos apibrėžimas naudoja žinias iš stiprumo, kad apibrėžti taisykles. Kita informacija, kuri gali būti panaudota - modelio elementų savybių reikšmės, šaltinio modelyje.

2.5.2. Modelių transformacijos kalbos

Šiame skyriuje apibrėšime kai kurias, transformacijų kalbų klasifikacijas, pagal įvairius Czarnecki ir Helsen [27] ir Gardner et al. [42] apibrėžtus kriterijus. Papildomai informacijai, įdėtos nuorodos į pilnus straipsnius.

2.5.2.1. Deklaratyvios ir imperatyvios transformacijos kalbos

Transformacijos kalba yra *deklaratyvi*, jei transformacijos apibrėžimas, aprašytas toje kalboje, specifikuoja ryšius tarp elementų, šaltinio ir rezultatų modelyje, nesinaudojant vykdomuoju liepimu. Ryšiai gali būti specifikuoti funkcijų terminais, arba taisyklių išvadomis. Transformacijos varikliukas pritaiko algoritmą ryšiams, kad gautų rezultatą.

Imperatyvios transformacijų kalbos nurodo išskirtinę eilę žingsnių tam, kad gauti rezultatus.

Galima pakalbėti ir apie vidutinę kategoriją, žinomą kaip *hibridinės* transformacijos kalbos. Šios kalbos turi sumaišytą deklaratyvių ir imperatyvių kalbų konstrukcijų. Dažniausiai, transformacijos apibrėžimas parašytas hibridinėje kalboje, susidaro iš rinkinio taisyklių, kurios specifikuoja ryšius tarp elementų. Taisyklės gali turėti liepiamuosius kūnus, kurie specifikuoja žingsnius, kuriuos reikia įvykdyti, kad išgauti rezultatą.

2.5.2.2. Transformacijų kryptingumas

Kai kurios kalbos leidžia transformacijų specifikacijas, kurias galima pritaikyti tik viena kryptimi: iš šaltinio į rezultatų modelį. Šios transformacijos žinomos kaip *unikryptingos* (unidirectional) transformacijos.

Kitos kalbos (paprastai deklaratyvios) leidžia apibrėžimus, kurie gali būti įvykdyti abejom kryptim. Šios transformacijos žinomos kaip *abipusės* transformacijos. Ši galimybė naudojama kai abu modeliai turi būti sinchronizuoti. Tarkim, kad rezultatų modelis yra išvestas iš šaltinio modelio, naudojant abipusę transformaciją ir vėliau, tam tikri pakeitimai yra padaromi rezultatų modelyje. Transformacija gali būti pritaikyta kita kryptimi, kad įvesti atitinkamus pakeitimus šaltinio modelyje.

Galima pastebėti, jog transformacijų apibrėžimų kryptingumas priklauso ne tik nuo transformacijos kalbos. Kai kurie transformacijų apibrėžimai gali transformuoti daugelį šaltinio elementų į vieną rezultatų elementą, taip padarydamas atvirkštinę operaciją neapibrėžtą.

Jei transformacijos kalba nepalaiko abipusių transformacijų apibrėžimų, tada du skirtingi apibrėžimai gali būti naudojami: po vieną kiekvienai krypčiai.

2.5.2.3. Įėjimo ir išėjimo kardinalumas transformacijų apibrėžimuose

Tipiškas transformacijos scenarijus paima vieną modelį kaip įėjimą ir gamina vieną modelį kaip išėjimą(*1-su-1* transformacija). Bendrai, egzistuoja trys kiti atvejai: *1-su-N, N-su-1* ir *M-su-N*.

Daugeliu atvejų, daug modelių yra pagaminama iš vieno šaltinio modelio (*1-su-N* transformacija). Pavyzdžiui, vienas modelis gali būti panaudotas sugeneruoti Java kodą ir XML schemą, naudojamą duomenų apsikeitimui. Modelių sudėtis, kurioje keletas modelių yra apjungiama į vieną yra *N-su-1* transformacijos pavyzdys. Bendru atveju, *M-su-N* transformacijų palaikymas užtikrina kitų trijų atvejų galimybę.

2.6. Tikslai ir uždaviniai

Šio darbo tikslas – išnagrinėti imitacinių modelių, pateiktų UML diagramomis, automatizuoto transformavimo galimybes į PLA imitacinio modelio programinę realizaciją. Sprendžiant šią problemą reikia išspręsti šiuos uždavinius:

- Susikurti PLA modelio meta-modelį, kuris turėtų atitikti MDA rekomenduojamą PIM modelį.
- Susikurti savo CIM modelį, kuris turėtų atstovauti atliekamiems sutapatinimas tarp sukurto PIM modelio ir galutinio PSM modelio.
- Sukurti transformaciją, kuri pagal pateiktą PIM modelį sukuria imitacinio modelio programinę realizaciją.

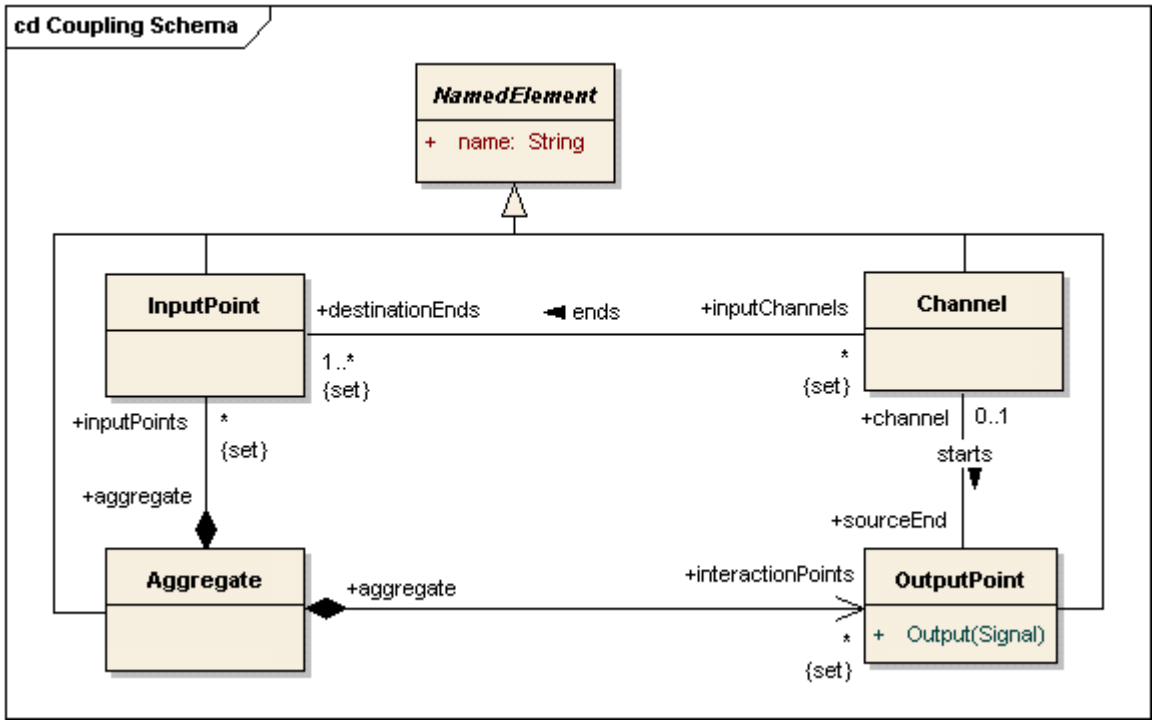
3. AGREGATŲ IR SUJUNGIMŲ SCHEMŲ META-MODELIS

3.1. PLA metamodelis

PLA specifikacija išskiria dvi bazines komponentes – agregatų sujungimo schemą bei sujungimo schemoje dalyvaujančių agregatų specifikacijas. Pateikdami PLA UML metamodelį taip pat remsimės šia tachionomija, todėl pateikdami bendrą PLA metamodelį išskirsime agregatų sujungimo schemos ir agregato specifikacijos (kuri detalizuoja sujungimo schemos elementus – agregatus) metamodelius.

3.1.1. Agregatų sujungimo schemos metamodelis

Agregatų sujungimo schemos struktūra ir elementai pateikti 3.1 paveiksle. Sujungimo schemą sudaro agregatai, kanalai, įėjimo ir išėjimo taškai. Agregatas savyje agreguoja išėjimo taškus, prie kurių jungiasi kanalai, t.y. išėjimo taškai yra salyginiai kanalų prieigos prie signalų šaltinio taškai. Savo ruožtu išėjimo taškas negali būti daugiau nei vieno kanalo signalų šaltiniu. Tiksliniais kanalo prieigos taškais yra įėjimo taškai, kuriais signalai patenka į agregatą. Kanalas vieną sąveikos tašką gali susieti su daugeliu įėjimo taškų skirtinguose agregatuose, todėl signalai iš vieno išėjimo taško gali pasiekti daug tikslinių įėjimo taškų. Agregatų, kanalų, sąveikos ir įėjimo taškų ontologija apibrėžiama abstrakčia NamedElement klase.

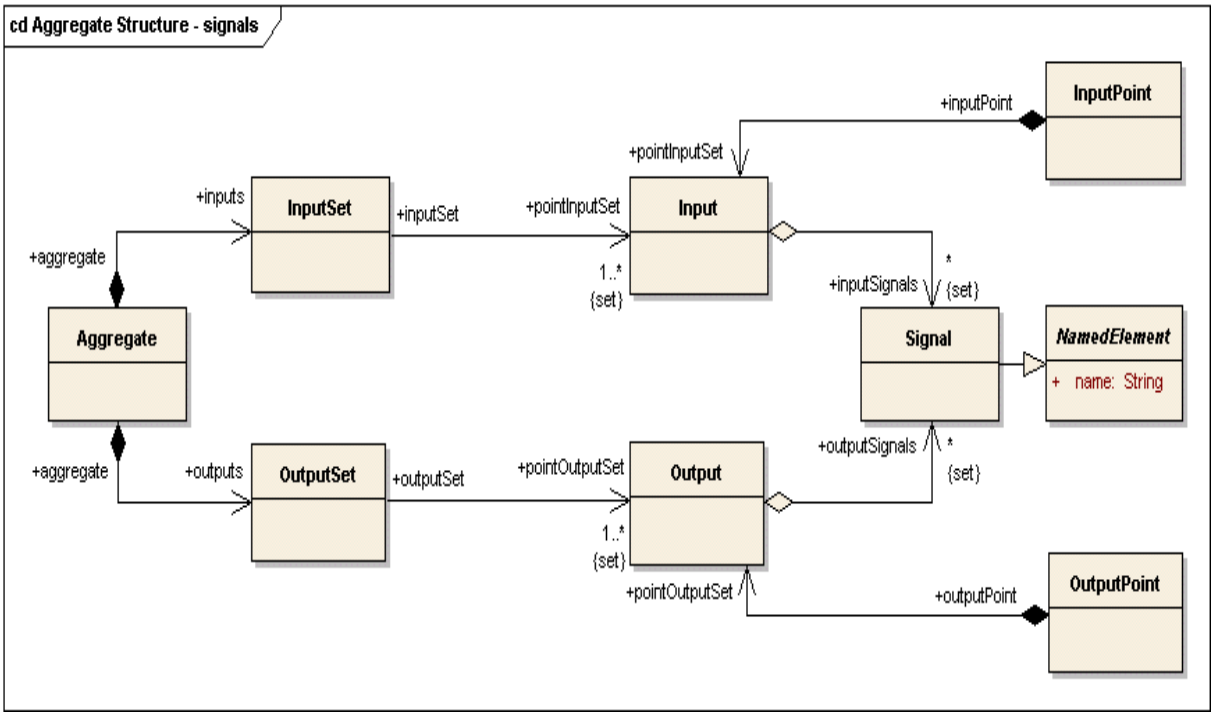


3.1 pav.: Sujungimo schemos metamodelis

3.1.2. PLA agregatų specifikavimo metamodelis

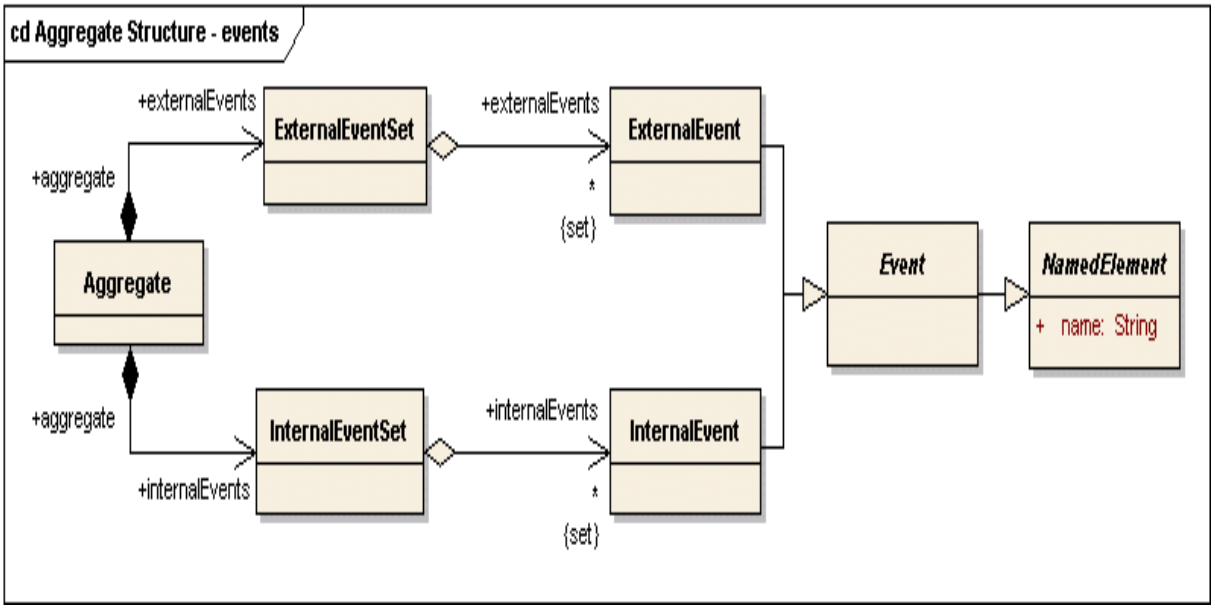
Agregatų specifikavimo metamodelį sudaro virš penkiolikos elementų, todėl vaizdumo ir patogumo dėlei metamodelis bus pateikiamas fragmentais.

Kiekviename agregate specifikuojamos įėjimo ir išėjimo signalų aibės, kurių ryšiai atvaizduoti 3.2 paveiksle. Įėjimo signalų aibę sudaro visi agregato įėjimo taškų signalų aibės. Savo ruožtu įėjimo taško signalų aibę sudaro įėjimo tašką kanalu pasiekiantys signalai. Analogiškai išėjimo signalų aibę sudaro sąveikos taškų išėjimo signalų aibės, kurios elementai yra signalai, perduodami konkrečiam sąveikos taškui. Signalų ontologija identifikuojama abstrakčia *NamedElement* klase.



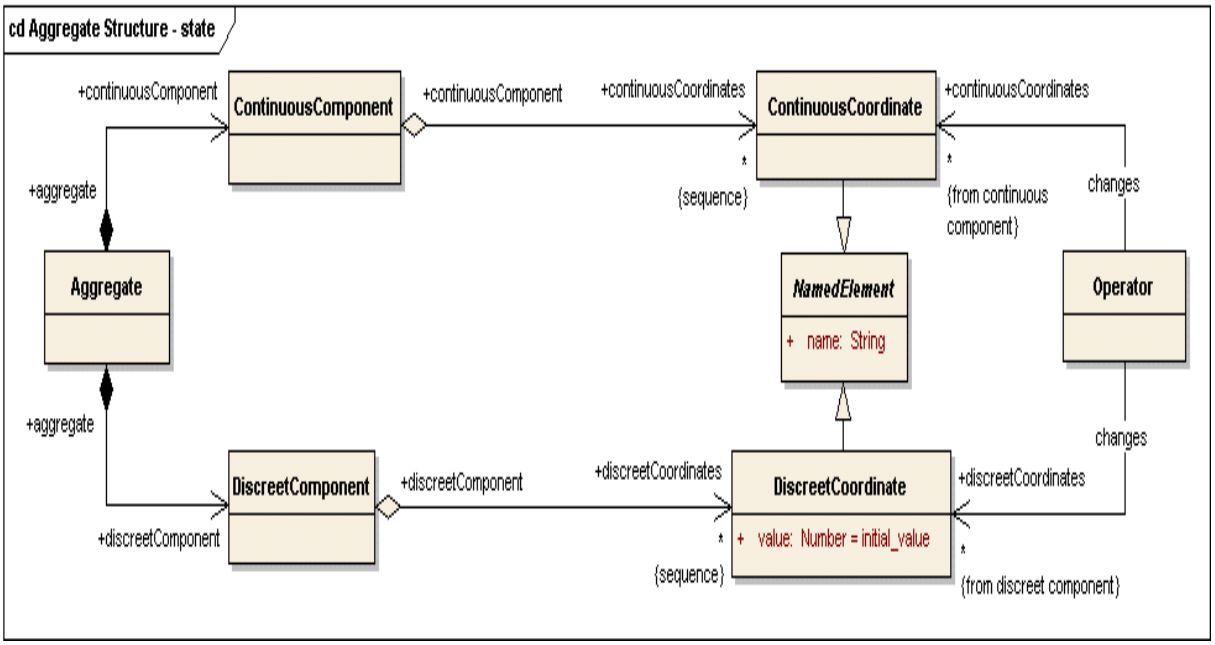
3.2 pav.: Įėjimo ir išėjimo signalų aibės

Įvykių tachionomija remiasi įvykį generuojančios priežasties lokalizacija agregato atžvilgiu, todėl išskiriami dveji įvykių tipai: vidiniai ir išoriniai. Vidinių ir išorinių įvykių diagrama pateikta 3.3 paveiksle. Vidinius ir išorinius įvykius abstrahuoja Event klasė, kurios ontologija identifikuojama NamedElement klase. Vidiniai įvykiai pateikiami agregato vidinių įvykių aibėje. Analogiškai išorinių įvykių aibėje išvardijami visi agregato išoriniai įvykiai.



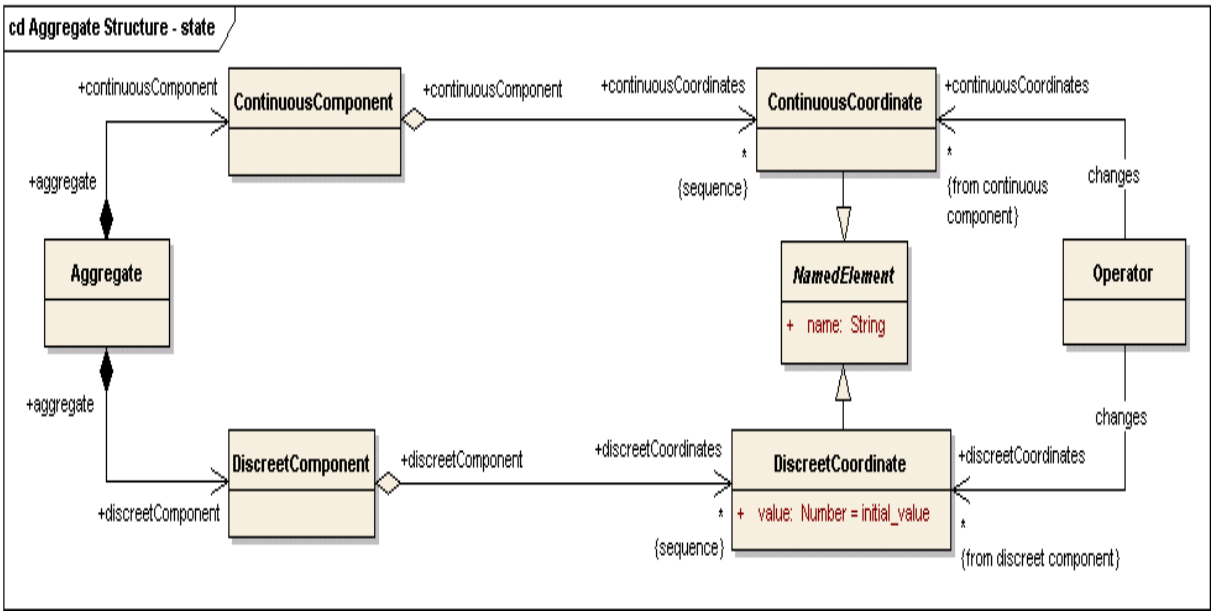
3.3 pav.: Vidiniai ir išoriniai įvykiai

Agregato būsenos struktūrinės komponentės pateiktos 3.4 paveiksle. Agregato būseną sudaro dvi komponentės – diskrečioji ir tolydžioji komponentės. Šios komponentės atitinkamai dar vadinamos diskrečiųjų ir tolydinių koordinačių aibėmis. Diskrečioje būsenos komponentėje saugomos diskrečiosios koordinatės, turinčios tam tikrą skaitinę vertę, kurios pradinė reikšmė specifikuojama papildomai. Diskrečiosios koordinatės ontologija apsprendžiama abstrakčia NamedElement klase. Tolydžiojoje būsenos komponentėje saugomos tolydžiosios koordinatės.



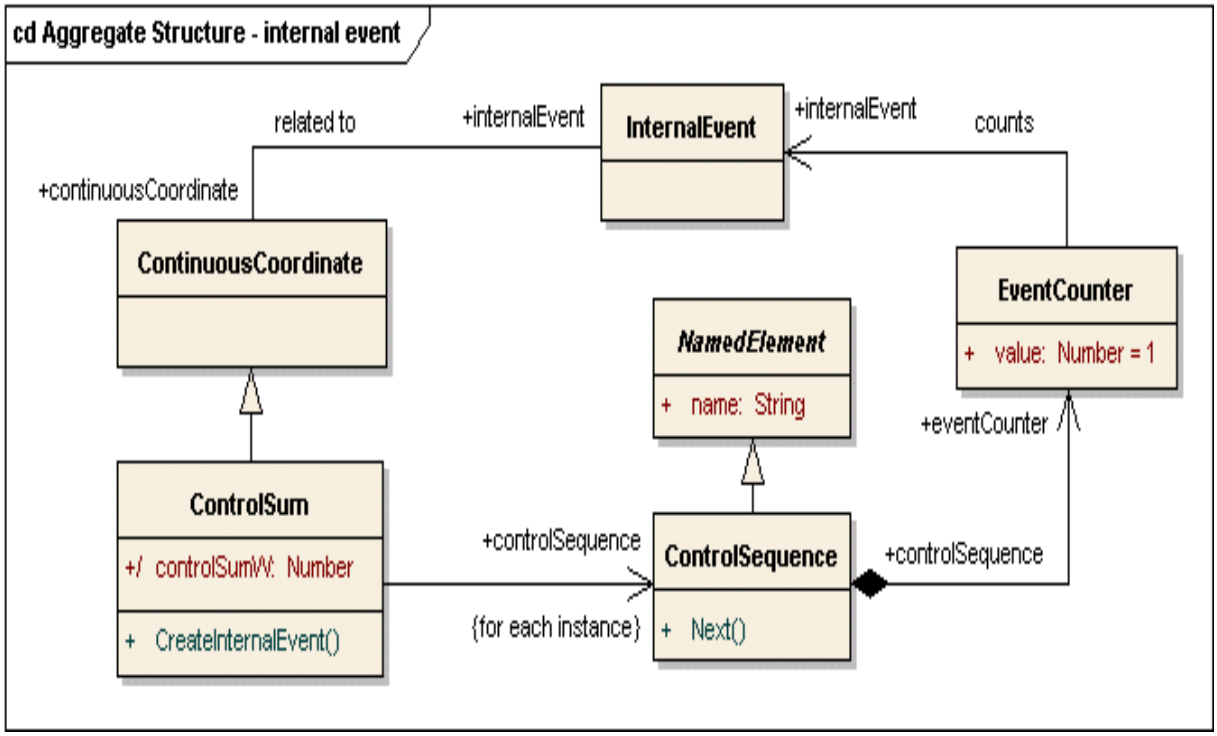
3.4 pav.: Agregato būsenos komponentės

Agregato specifikacijoje pateikiami perėjimo ir išėjimo operatorių aprašai, kurie detalizuoja išėjimo signalų generavimą ir agregato būsenos evoliuciją (3.4 paveikslas). Agregate visi įvykiai apdorojami perėjimo ir išėjimo operatoriuose (3.5 paveiksle). Apdorojant įvykius, perėjimo ir išėjimo operatoriai ne tik evoliucionuoja agregatų būseną, bet ir generuoja signalus iš agregato išėjimo signalų aibės.



3.5 pav.:Ryšiai su įvykiais

Vidinio įvykio struktūra pateikta 3.6 paveiksle. Kiekvienas vidinis įvykis susijęs su tolydžiąja koordinate, kuri apsprendžia laikinius įvykio parametrus. Kontrolinė suma yra atskiras tolydžiosios koordinatės atvejis, susijęs su tokiais parametrais kaip kontrolinės sumos S ir W. Pagrindinis kontrolinės sumos parametras yra kontrolinė suma W. Kontrolinė suma W yra išvestinis parametras, kurio reikšmė apskaičiuojama pagal kontrolinės sumos S ir kontrolinės sekos reikšmes. Jei nespecifikuojama papildomai, krypties koeficientu priimta laikyti –1 reikšmę. Kiekvienas vidinis įvykis susiejamas su kontrolinė seka, kurios reikšmės dažniausiai naudojamos identifikuoti operacijos trukmę. Kontrolinė seka savyje saugo įvykių skaitliuką, kurio reikšmė parodo sekančios vidiniam įvykiui priskiriamos kontrolinės sekos elemento indeksą.

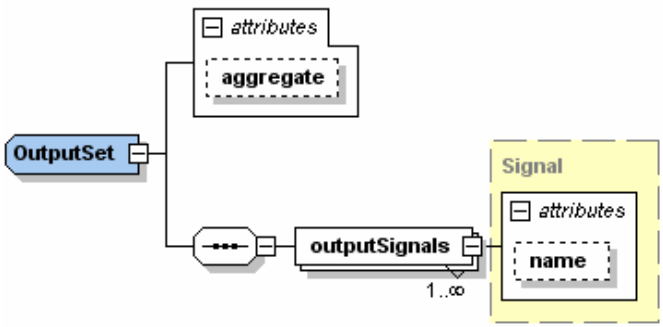


3.6 pav.: Vidinio įvykio sąryšiai

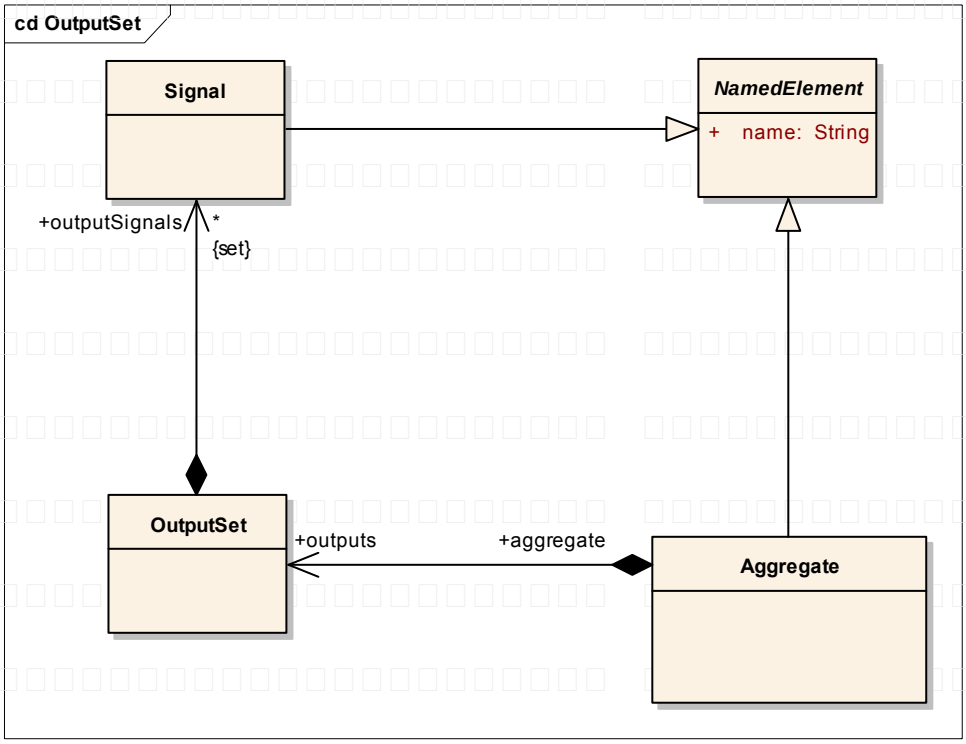
3.2. Nuo skaičiavimų nepriklausomas modelis

Šiame skyriuje bus pavaizduoti atliekami sutapatinimai tarp PIM modelio ir sukurto CIM (Computationally Independent Model) modelio. Kadangi PIM modelis yra aprašytas UML kalba, o CIM modelis yra aprašytas XSD kalba, visi sutapatinimai bus pavaizduoti grafiškai. Tikrojo CIM modelio tekstas bus pridėtas prie priedų.

3.2.1. Išėjimo aibės



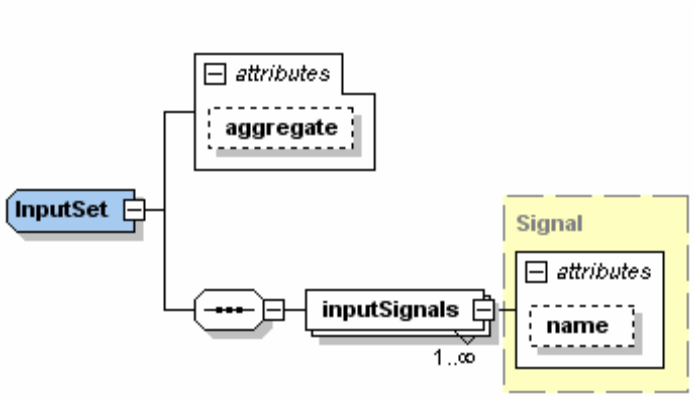
3.7a pav.: Išėjimo aibės CIM modelyje schema



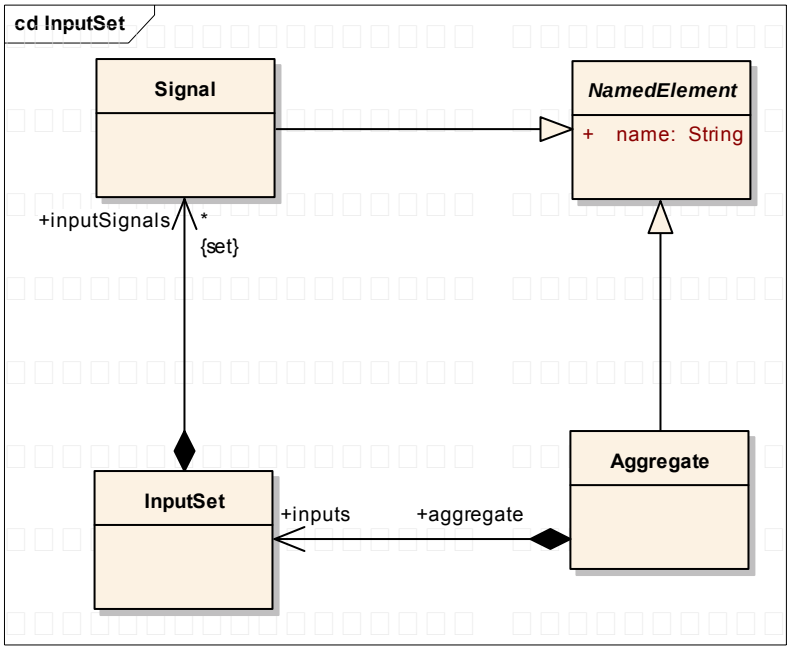
3.7b pav.: Išėjimo aibės PIM modelyje schema

Kiekviena išėjimo aibė turi po atributą *aggregate*, kuris nurodo kuriam *Aggregate* priklauso išėjimo aibė. Taip pat, kiekviena išėjimo aibė gali turėti daug išėjimo signalų (*outputSignals*), kurie 3.7b paveiksle atitinka *Signal* klasę. *OutputSignals* talpina savyje atributą *name*, kuris tiesiogiai atitinka paveldėtą klasę *NamedElement* 3.7b paveiksle.

3.2.2. Įėjimo aibės



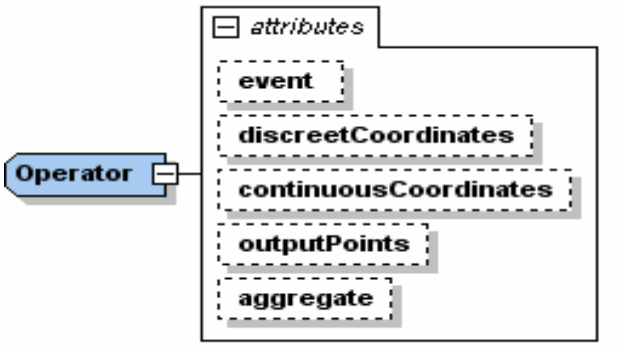
3.8a pav.: Įėjimo aibės CIM modelyje schema



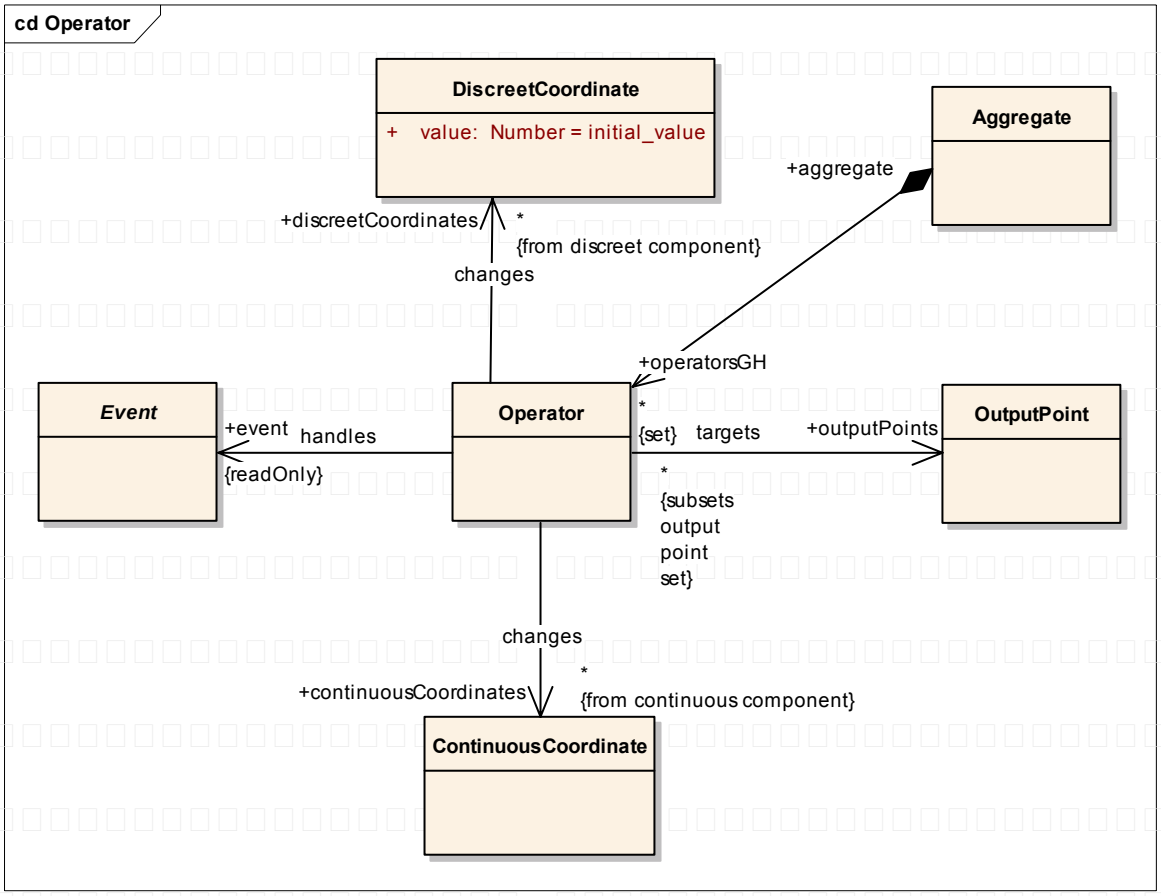
3.8b pav.: Įėjimo aibės PIM modelyje schema

Kaip ir išėjimo , taip ir įėjimo aibėje, kiekvienas elementas turi atributą, nurodantį kokiam *Aggregate* priklauso klasė. Taip pat, kiekviena įėjimo aibė gali turėti daug įėjimo signalų (*inputSignals*), kurie 3.8b paveiksle atitinka *Signal* klasę. *InputSignals* talpina savyje atributą, kuris tiesiogiai atitinka paveldėtą klasę *NamedElement* 3.8b paveiksle.

3.2.3. Operatoriai



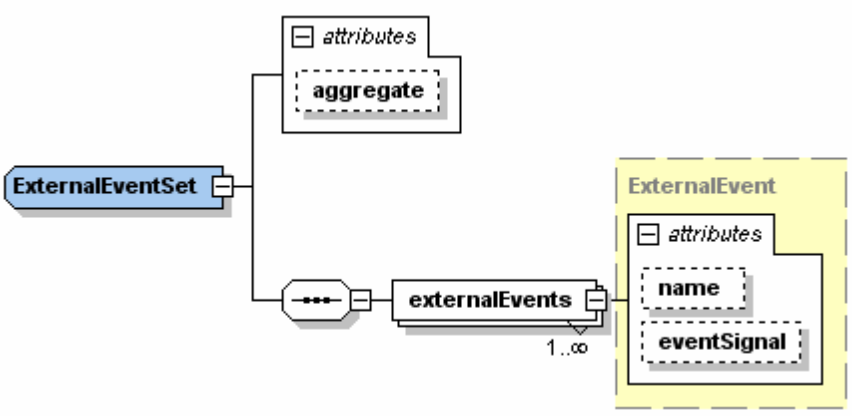
3.9a pav.: Operatorių aibės CIM modelyje schema



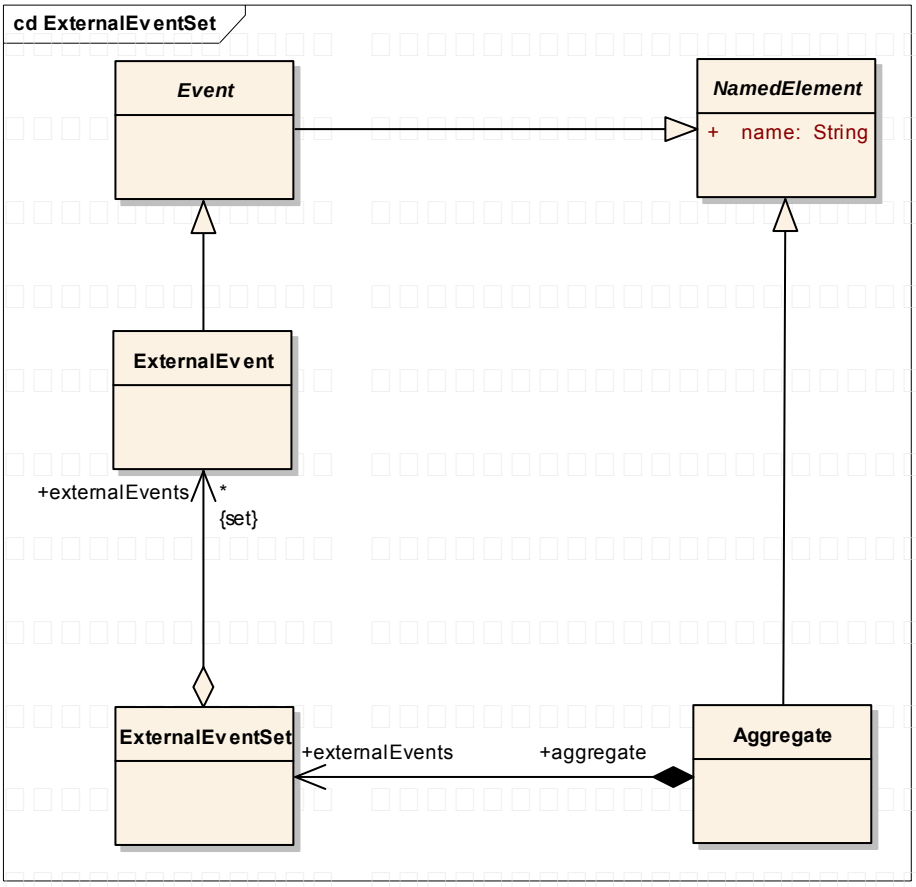
3.9b pav.: Operatorių aibės PIM modelyje schema

Operatoriaus atributas *aggregate* nusako kuriam agregatui priklauso operatorius. Atributai *event, discreetCoordinates, continuousCoordinates* ir *outputPoints* tiesiogiai atitinka 3.9b paveiksle atitinkamas klases: *Event, DiscreetCoordinate, ContinousCoordinate* ir *OutputPoint*.

3.2.4. Išorinių įvykių aibė



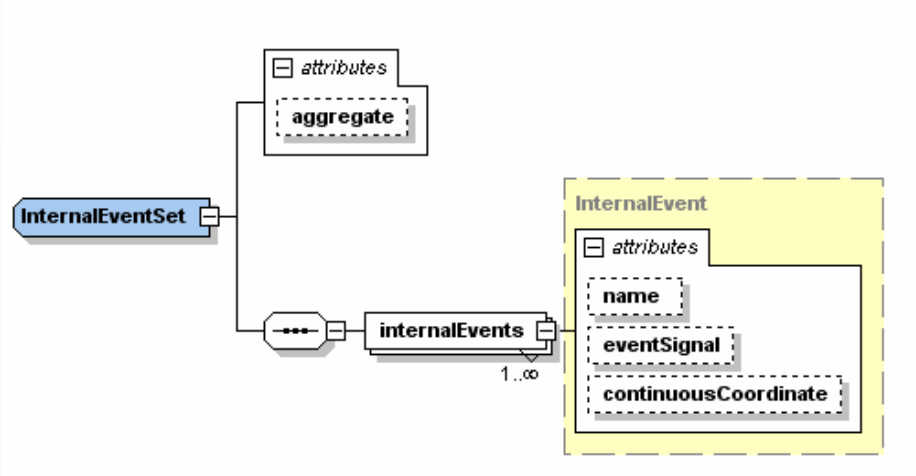
3.10a pav.: Išorinių įvykių aibės CIM modelyje schema



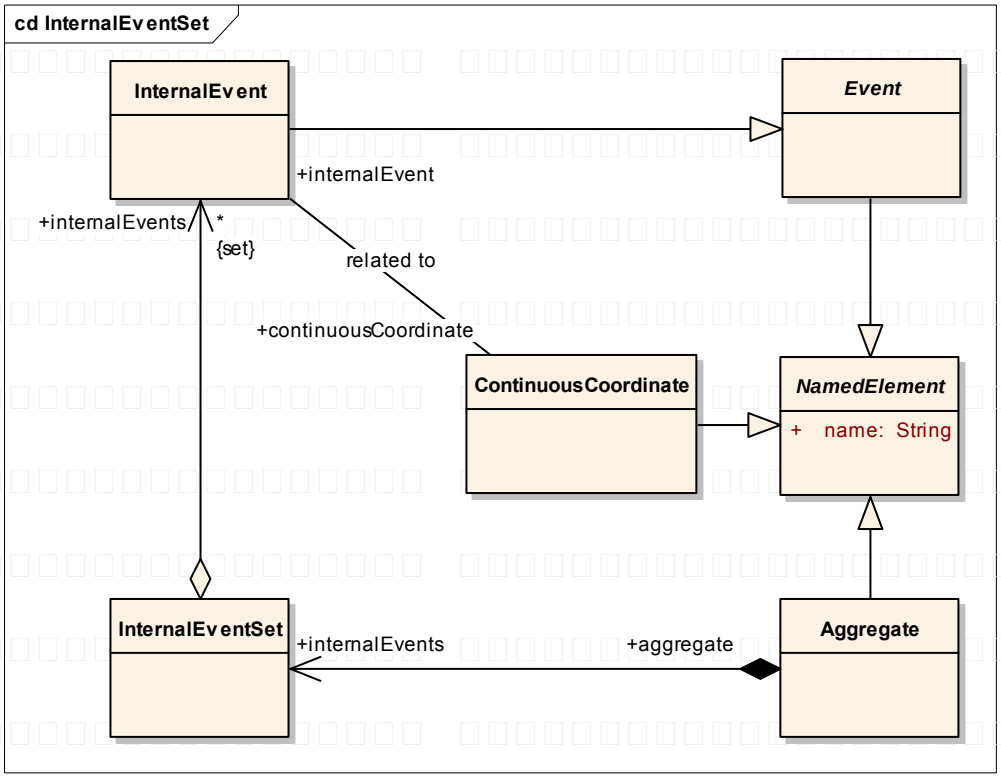
3.10b pav.: Išorinių įvykių aibės PIM modelyje schema

Išorinių įvykių aibės atributas *aggregate* nusako kuriam agregatui priklauso aibė. Aibė susideda iš daug išorinių įvykių (e*xternalEvents*), kurie tiesiogiai atitinka 3.10b paveiksle klasę *ExternalEvent*. Išorinio įvykio atributas *name* atitinka 3.10b paveiksle paveldėtą klasę *NamedElement*. Atributas *eventSignal* atitinka paveldėtą klasę *Event*.

3.2.5. Vidinių įvykių aibė



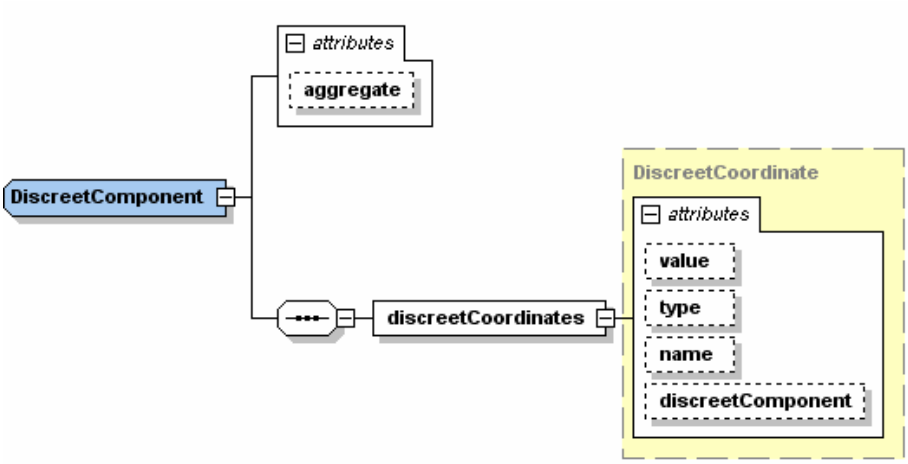
3.11a pav.: Vidinių įvykių aibės CIM modelyje schema



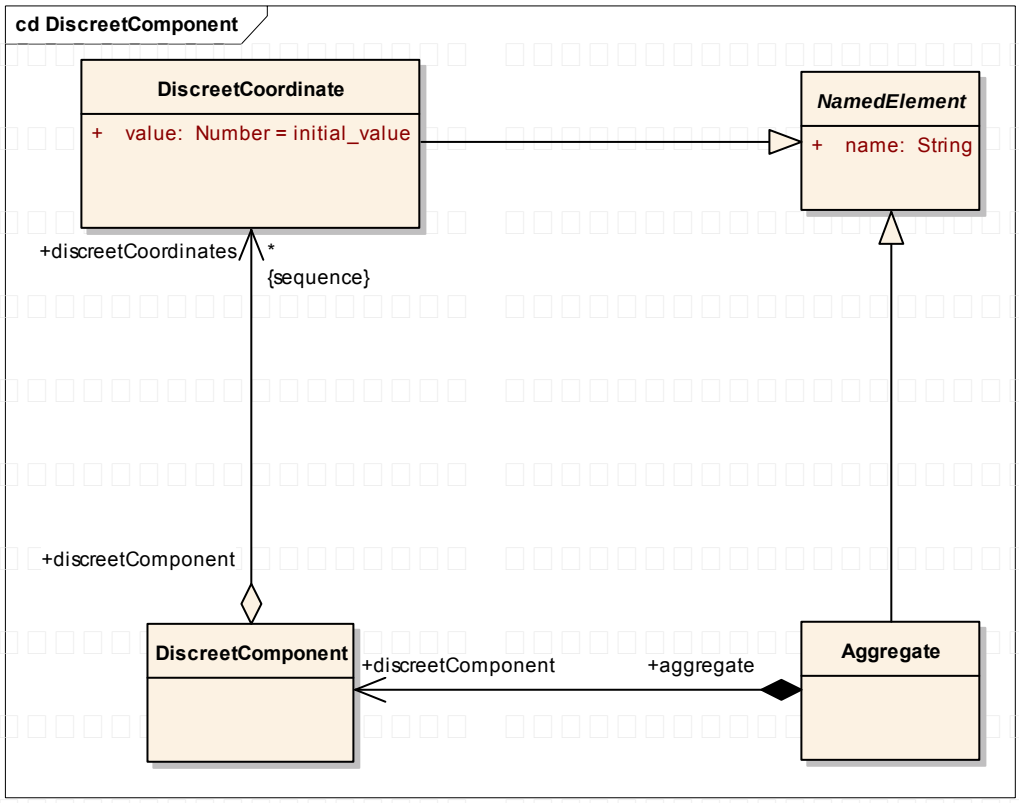
3.11b pav.: Vidinių įvykių aibės PIM modelyje schema

Vidinių įvykių aibės atributas *aggregate* nusako kuriam agregatui priklauso aibė. Aibė susideda iš daug vidinių įvykių (*internalEvents*), kurie tiesiogiai atitinka 3.11b paveiksle klasę *InternalEvent*. Išorinio įvykio atributas *name* atitinka 3.11b paveiksle paveldėtą klasę *NamedElement*. Atributas *eventSignal* atitinka paveldėtą klasę *Event*. Priešingai negu išorinių įvykių aibėje, šioje aibėje prisideda atributas *continousCoordinate* kuris tiesiogiai atitinka 3.11b paveiksle klasę *ContinousCoordinate*.

3.2.6. Diskretūs komponentai



3.12a pav.: Diskrečių komponentų CIM modelyje schema

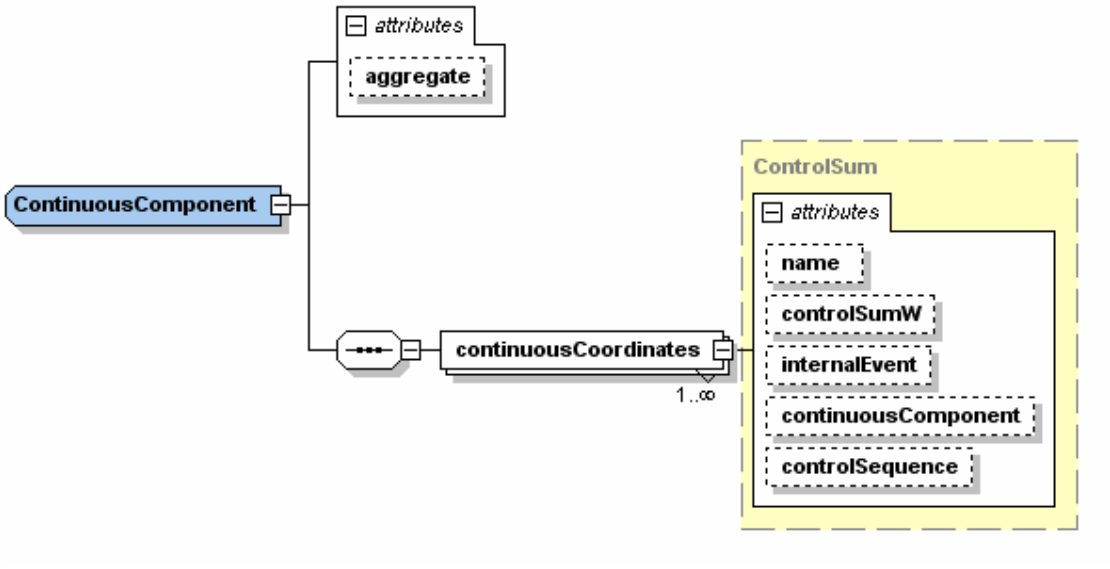


3.12b pav.: Diskrečių komponentų PIM modelyje schema

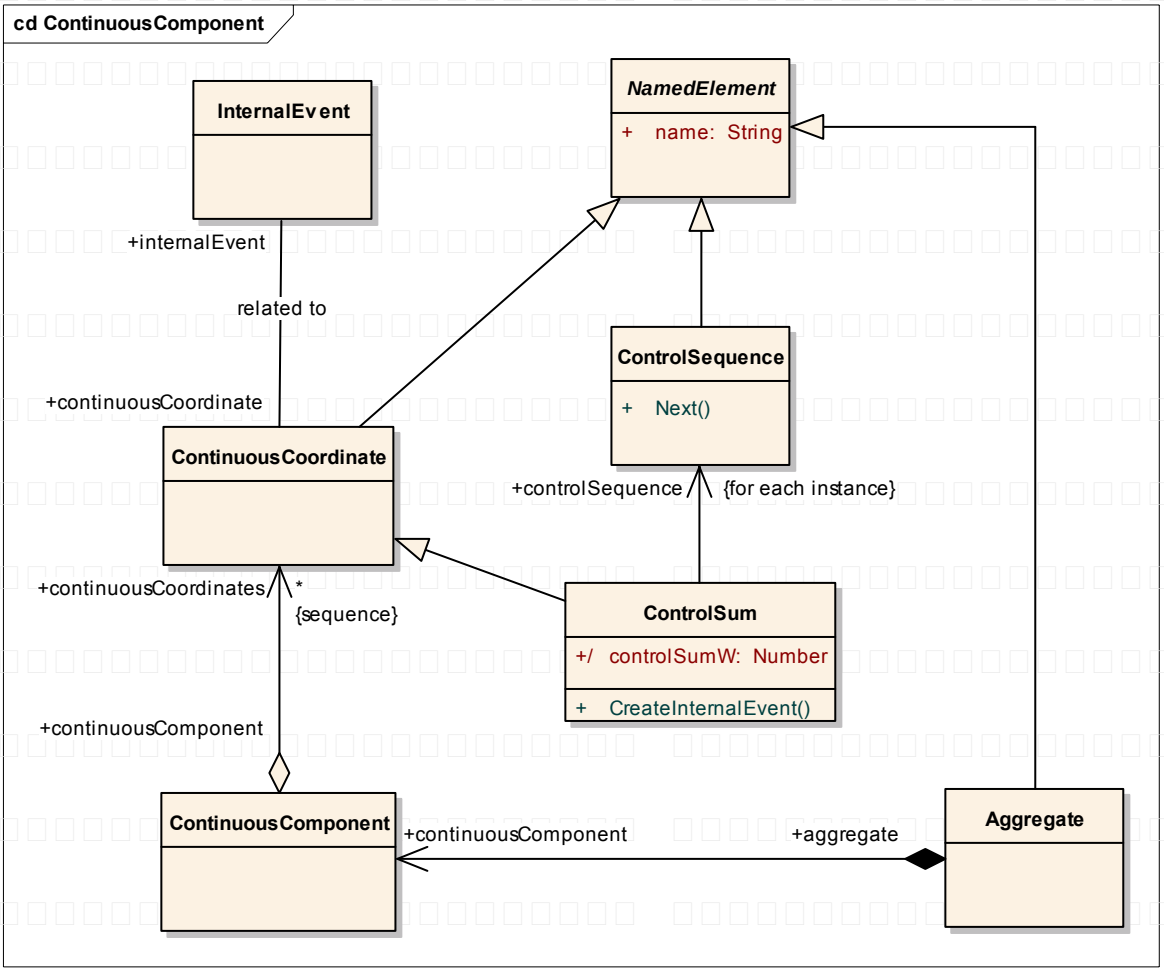
Diskretaus komponento atributas *aggregate* nusako kuriam agregatui priklauso komponentas. Komponentas susideda iš daug diskrečių koordinačių (*discreetCoordinates*), kurios tiesiogiai atitinka 3.12b paveiksle klasę *DiscreetCoordinate*. Diskrečios koordinatės atributas *name* atitinka 3.12b paveiksle paveldėtą klasę *NamedElement*. Atributai *type* ir *value* nusako tipą

ir reikšmę. Atributas *discreetComponent* atitinka 3.12b paveiksle pavaizduotą klasę *DiscreetComponent* ir parodo kuriam komponentui priklauso ši klasė.

3.2.7. Tolydžioji komponentė



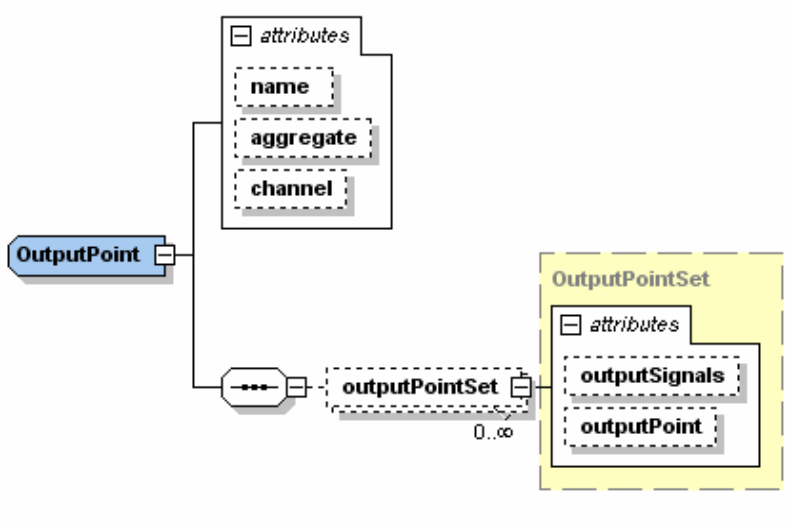
3.13a pav.: Tolydžių komponentų CIM modelyje schema



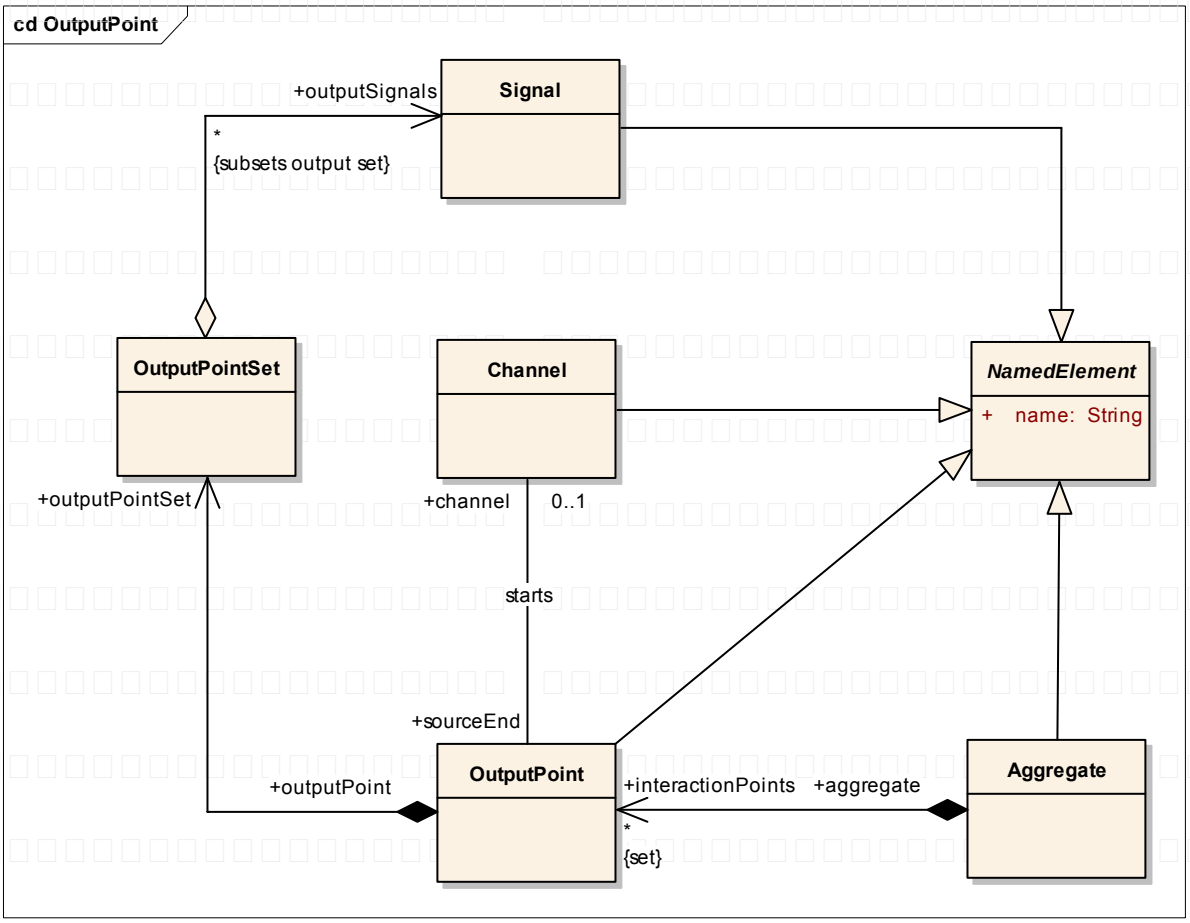
3.13b pav.: Tolydžiosios komponentės PIM modelyje schema

Tolydžiosios komponentės atributas *aggregate* nusako kuriam agregatui priklauso komponentas. Komponentas susideda iš daug tolydžių koordinačių (*continuousCoordinates*), kurios tiesiogiai atitinka 3.13b paveiksle klasę *ControlSum*. Diskrečios koordinatės atributas *name* atitinka 3.13b paveiksle paveldėtą klasę *NamedElement*. Atributas *controlSumW* nusako reikšmę. Atributas *internalEvent* atitinka 3.13b paveiksle pavaizduotą klasę *InternalEvent*. Atributas *continuousComponent* parodo kuriam komponentui priklauso ši klasė. Atributas *controlSequence* tiesiogiai atitinka 3.13b paveiksle klasę *ControlSequence*.

3.2.8. Išėjimo taškai



3.14a pav. Išėjimo taškų CIM modelyje schema

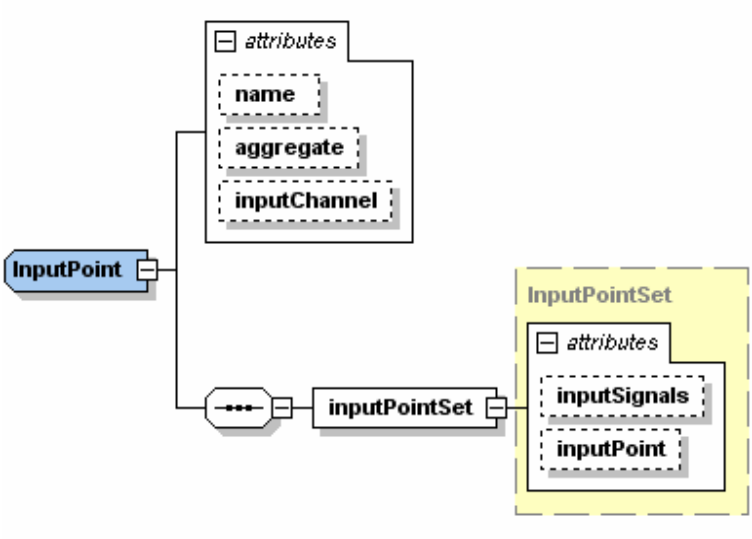


3.14b Išėjimo taškų PIM modelyje schema

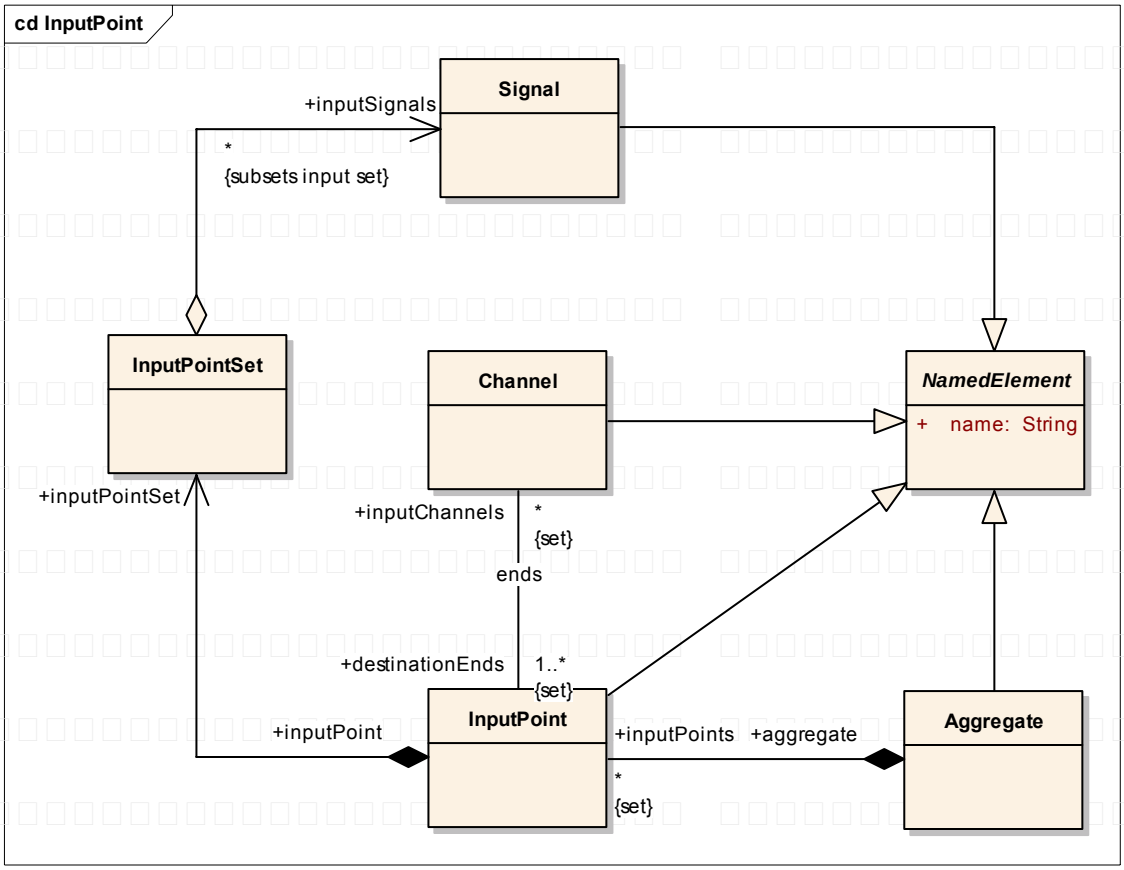
Išėjimo taško atributas *aggregate* nusako kuriam agregatui priklauso taškas. Atributai *name* ir *channel* tiesiogiai atitinka 3.14b paveiksle pavaizduotas klases *NamedElement* ir *Channel*

ir nurodo koks bus pavadinimas ir kokį kanalą naudos išėjimo taškas. Taškas susideda iš daug išėjimo taškų aibių (*outputPointSet*), kurios tiesiogiai atitinka 3.14b paveiksle klasę *OutputPointSet*. Išėjimo taškų aibės atributas *outputSignals* tiesiogiai atitinka 3.14b paveikslo *Signal* klasę. Atributas *outputPoint* nurodo kuriam išėjimo taškui priklauso aibė.

3.2.9. Įėjimo taškai



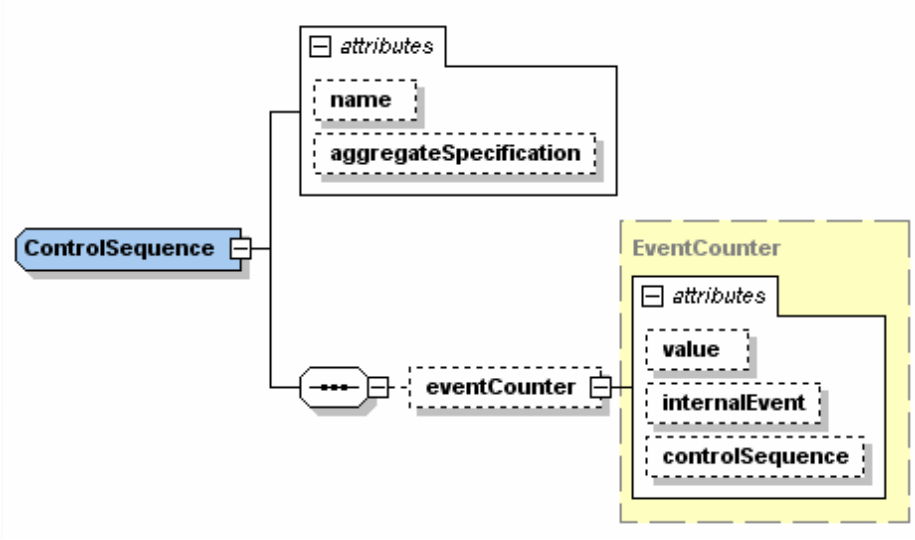
3.15a pav.: Įėjimo taškų CIM modelyje schema



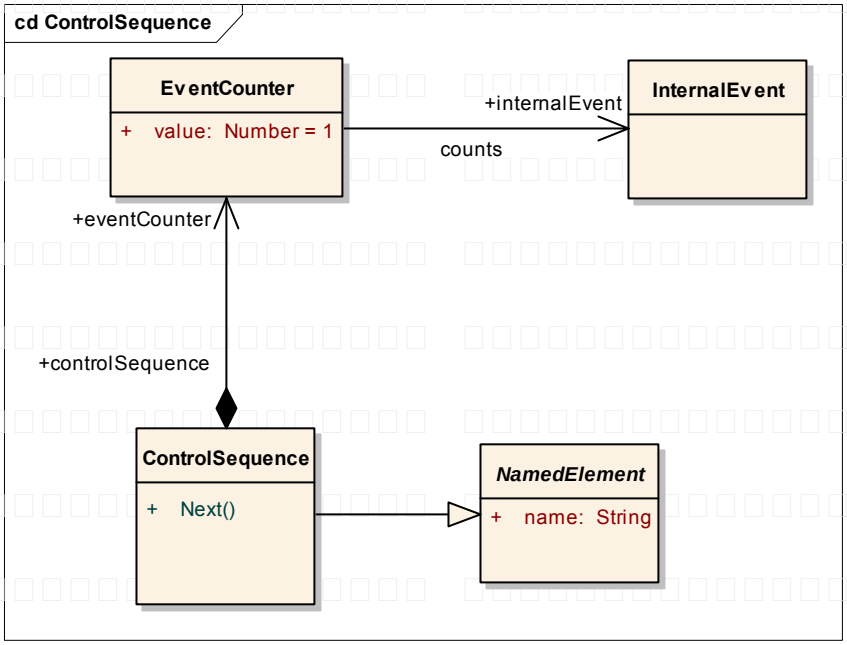
3.15b pav.: Įėjimo taškų PIM modelyje schema

Įėjimo taško atributas *aggregate* nusako kuriam agregatui priklauso taškas. Atributai *name* ir *channel* tiesiogiai atitinka 3.15b paveiksle pavaizduotas klases *NamedElement* ir *Channel* ir nurodo koks bus pavadinimas ir kokį kanalą naudos įėjimo taškas. Taškas susideda iš daug įėjimo taškų aibių (*inputPointSet*), kurios tiesiogiai atitinka 3.15b paveiksle klasę *InputPointSet*. Įėjimo taškų aibės atributas *inputSignals* tiesiogiai atitinka 3.15b paveikslo *Signal* klasę. Atributas *inputPoint* nurodo kuriam įėjimo taškui priklauso aibė.

3.2.10. Valdanti seka



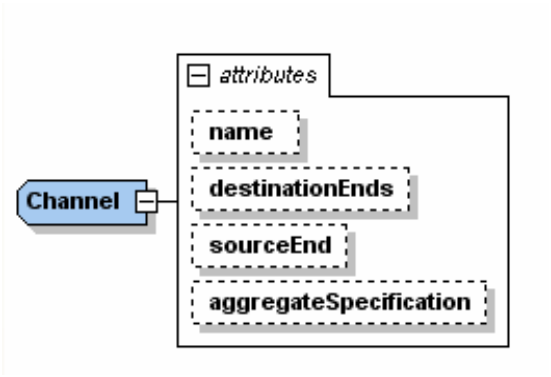
3.16a pav.: Kontroliuojamų sekų CIM modelyje schema



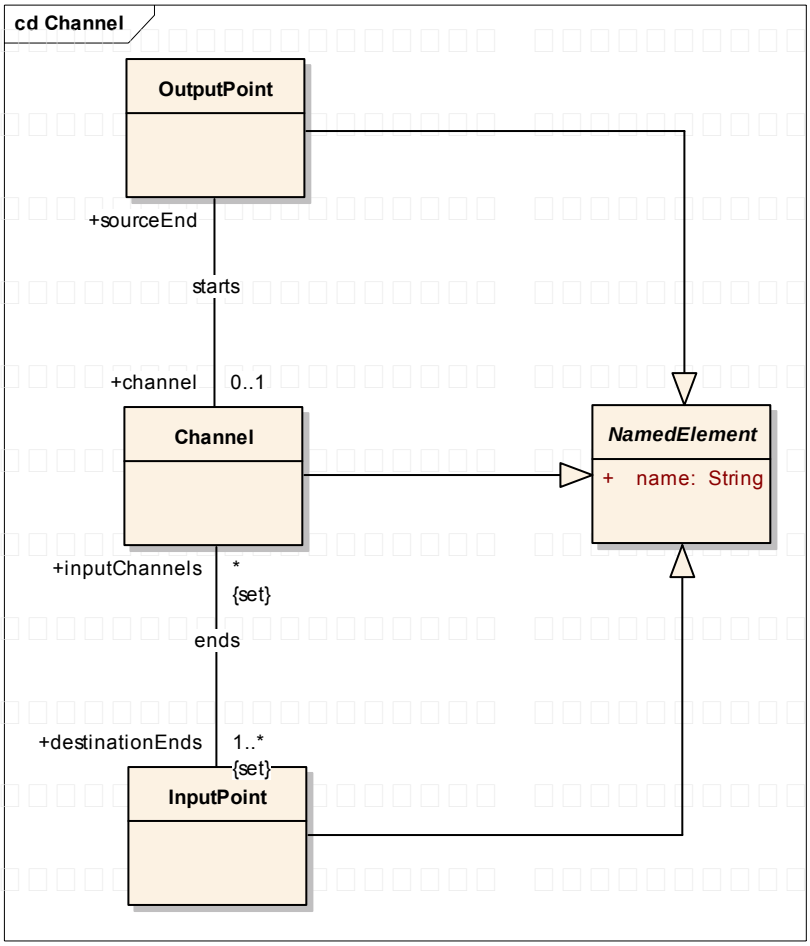
3.16b pav.: kontroliuojamų sekų PIM modelyje schema

Kontroliuojamos sekos atributas *aggregateSpecification* nusako kuriam tėviniam elementui priklauso seka. Atributas *name* tiesiogiai atitinka 3.16b paveiksle pavaizduotą klasę *NamedElement* ir nurodo, koks bus pavadinimas. Valdanti seka susideda iš daug *eventCounter* elementų, kurie 3.16b paveiksle atitinka *EventCounter* klasę. Įvykio skaitliuko atributas *value* nurodo reikšmę. Atributas *internalEvent* tiesiogiai atitinka 3.16b paveiksle *InternalEvent* klasę. Atributas *controlSequence* nurodo kuriai sekai priklauso šis elementas.

3.2.11. Kanalas



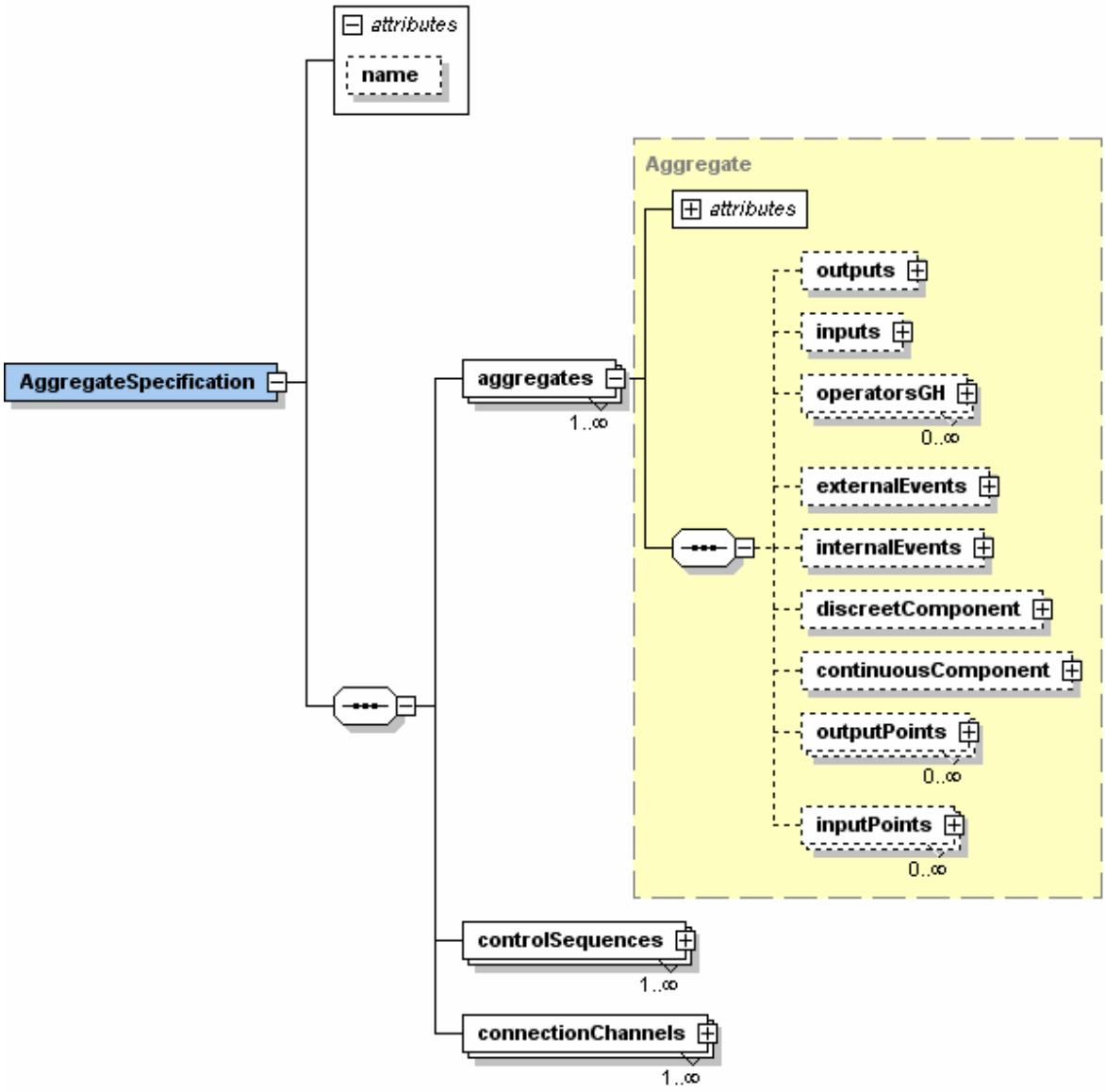
3.17a pav.: Kanalų CIM modelyje schema



3.17b pav.: Kanalų PIM modelyje schema

Atributas *name* tiesiogiai atitinka 3.17b paveiksle pavaizduotą klasę *NamedElement* ir nurodo, koks bus pavadinimas. Atributai *destinationEnds* ir *sourceEnds* atitinka 3.17b klases *InputPoint* ir *OutputPoint*, kuriose nurodo kas su kuo susijungia. Atributas *aggregateSpecification* nusako kuriam tėviniam elementui priklauso seka.

3.2.12. CIM modelio specifikacija



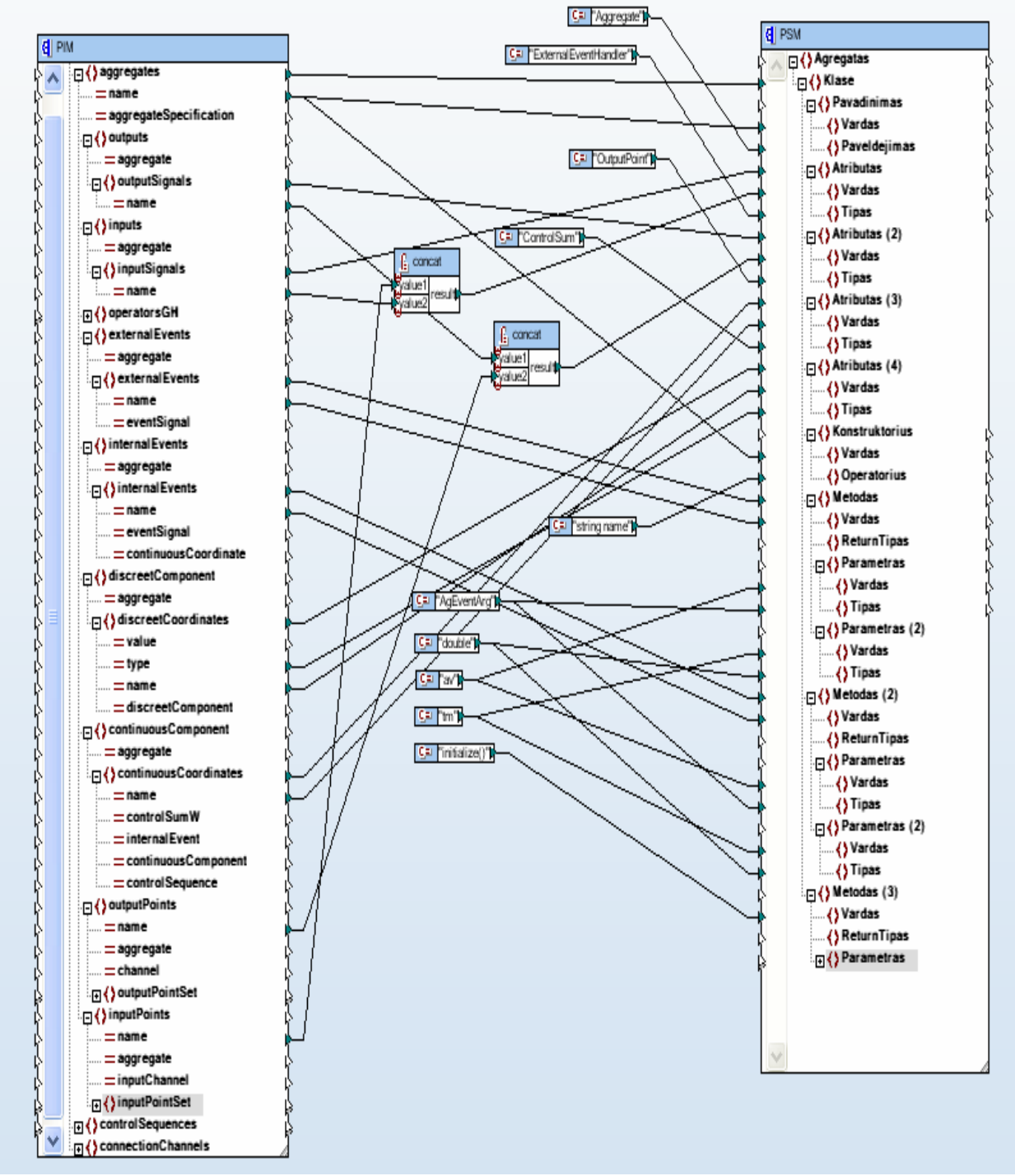
3.18 pav.: CIM modelio schema

CIM modelis susideda iš tėvinio elemento *AggregateSpecification*, bei daugelio *aggregates, controlSequences* ir *connectionChannels* elementų. Atributas *name* nurodo koks bus sujungimo klases vardas. Elementas *aggregates* tiesiogiai atitinka PIM modelio klasę *Aggregate*. Visi elementai, bei jų sutapatinimai yra aprašyti 3.2.1 - 3.2.11 skyriuose.

Sekančiame skyriuje aptarsime, bei pavaizduosime transformacijas tarp modelių. Bus panaudoti PIM ir CIM meta-modeliai tikintis gauti tam tikrą PSM modelį.

4. TRANSFORMACIJOS

4.1. Agregatų transformacija



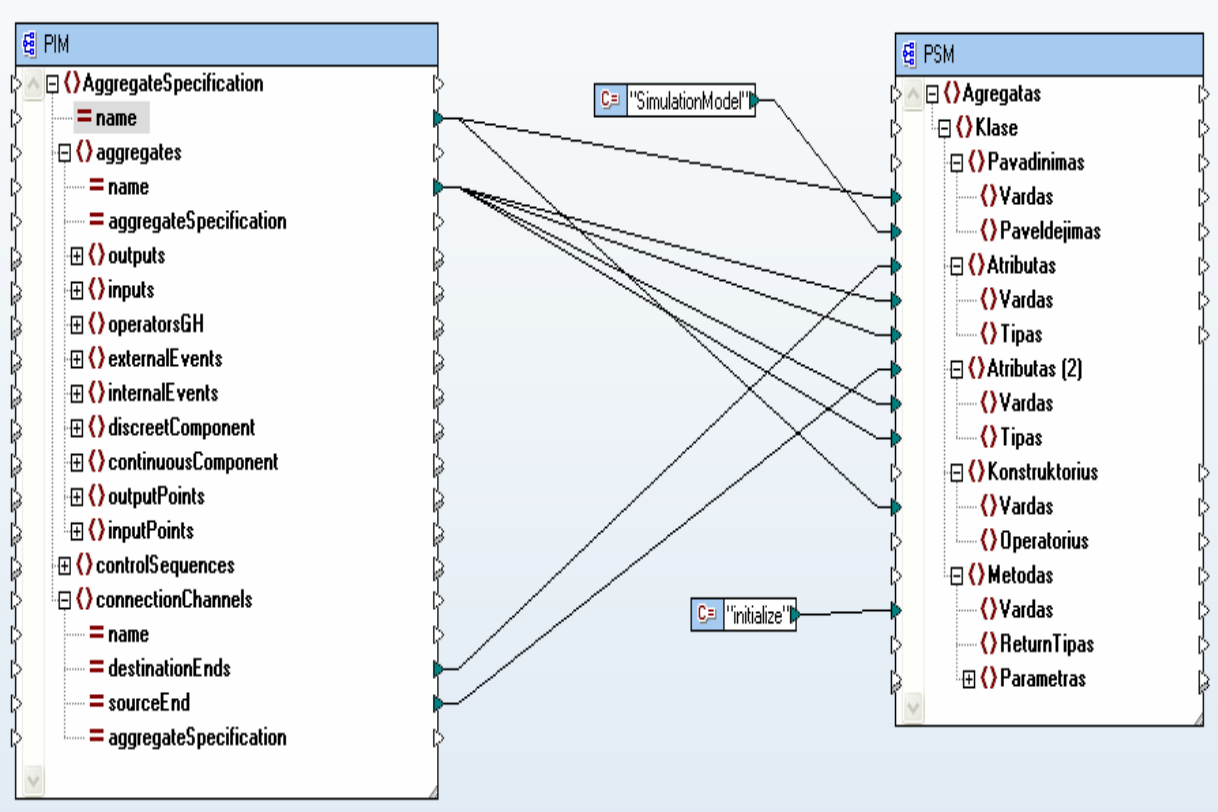
4.1 pav.: Transformacija tarp CIM ir PSM modelių norint išgauti agregatų klases

Kaip parodyta 3 skyriuje, XML failas gali turėti daug aprašytų agregatų. Pagal 4.1 paveikslą matome, jog kiekvienam agregatui bus sukuriama po atskirą klasę, kurios pavadinimas bus toks patas, kaip ir agregato atributo *name* pavadinimas. Klasė paveldės *Aggregate* meta-modelio klasę. Pagal klasės pavadinimą bus sukuriamas konstruktorius, kuris turės *string* tipo operatorių *name.*

Kiekviena sukurta klasė turės atitinkamą rinkinį atributų, kuriuos išgausime iš atitinkamų CIM modelio elementų. Kiekvienam *inputs/inputSignals* elementui bus sukuriamas atributas, kurio tipas bus *ExternalEventHandler*, o pavadinimas sudarytas pagal *inpuPoints* atributo *name* ir *inputs/inputSignals* atributo *name* sąjungą. Tą patį padarysime ir su *outputs/outputSignals* elementais. Tačiau šių elementų tipas bus *OutputPoint* ir pavadinimas susidės iš *outputPoints* ir *outputs/outputSignals* atributų *name* sąjungos. Kiekvienam elementui *continuousComponent/continuousCordinates* taip pat bus sukuriama po atributą su tipu *ControlSum* ir vardu paimtu iš elemento atributo *name* reikšme. Pagal tokį patį principą bus sukuriami atributai ir *discreetComponent/discreetCoordinate* elementui. Tačiau jo tipą ir vardą atitiks atitinkami elemento atributai *name* ir *type*.

Kiekvienam elementui *externalEvents/externalEvents* bus sukuriama po metodą, kurio pavadinimas atitiks atributo *name* reikšmę. Kiekvienas iš šių metodų turės po 2 parametrus: *AgEventArg* tipo kintamąjį *av* ir *double* tipo kintamąjį *tm*. Pagal tokį pat principą bus sukuriami ir elementų *internalEvents/internalEvents* metodai. Papildomai bus sukurtas vienas *initialize* metodas, kuriame bus nurodytos pradinės atributų reikšmės.

4.2. Sujungimų transformacijos



4.2 pav.: Transformacija tarp CIM ir PSM modelių norint išgauti sujungimo klasę

Pagal aprašytą meta-modelį, XML failas turi turėti sujungimų klasę. Kaip parodyta 4.2 paveiksle, bus sukurta klasė su *SimulationModel* paveldėjimu ir *AggregateSpecification* atributo *name* pavadinimu. Paveldėjimas atitiks meta-modelio klasę *SimulationModel*. Taip pat bus sukurtas paprastas konstruktorius.

Šiai klasei bus sukurtas rinkinys atributų, kurie aprašyti *connectionChannels* elemento *destinationEnds* ir *souceEnd* atributuose. Pavadinimai ir tipai tiesiogiai atitiks agregatų pavadinimus, kurie bus nurodyti *destinationEnds* ir *sourceEnd* atributuose.

Sujungimų klasei bus sukurtas *initialize* metodas, kuriame bus suteiktos pradinės reikšmės atributam, ir atliekami sujungimai.

4.3. Likusios transformacijos

Dėl sudėtingo kai kurių transformacijos dalių atvaizdavimo grafiškai, skyriuose 4.1 ir 4.2 atvaizduota nepilna transformacija. Pilna transformacija bus pateikta darbo prieduose XSL kalba. Taip pat bus pateikiami ir duomenų bei rezultatų failai, kurie bus atvaizduoti XML kalba.

5. NAFTOS TERMINALO IMITACINIO MODELIO SUDARYMAS

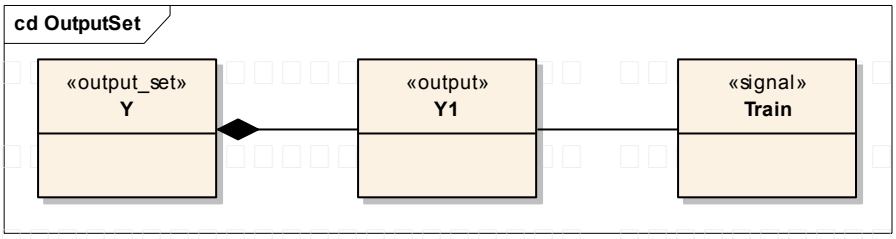
5.1. Pradiniai duomenys

Šiame skyriuje pateiksime transformavimo pavyzdį. Aprašysime pradinius duomenis, kurie bus pateikti UML kalba. Aptarsime kokių rezultatų tikimės pagal pateiktą duomenų pavyzdį. Sekančiame skyriuje pateiksime gautus rezultatus pateikdami juos UML kalba.

Pakrovus naftos produktus į vagonus-cisternas naftos perdirbimo gamykloje krovinys perduodamas geležinkelio operatoriui, kuris suformuoja sąstatus ir per eilę tarpinių geležinkelio stočių atgabena sąstatą į uosto geležinkelio stotį. Čia atvykus traukiniu sutvarkomi reikiami dokumentai ir sugrupavus į grupes, vagonai-cisternos paduodami į vieną iš dviejų išpylimo estakadų. Po to yra atliekamas naftos produktų perpylimas iš vagonų-cisternų į terminalo rezervuarus,. kad būtų galima sukaupti tanklaivio pakrovimui reikiamą naftos produkto kiekį.

5.1.1. Agregatas Railway

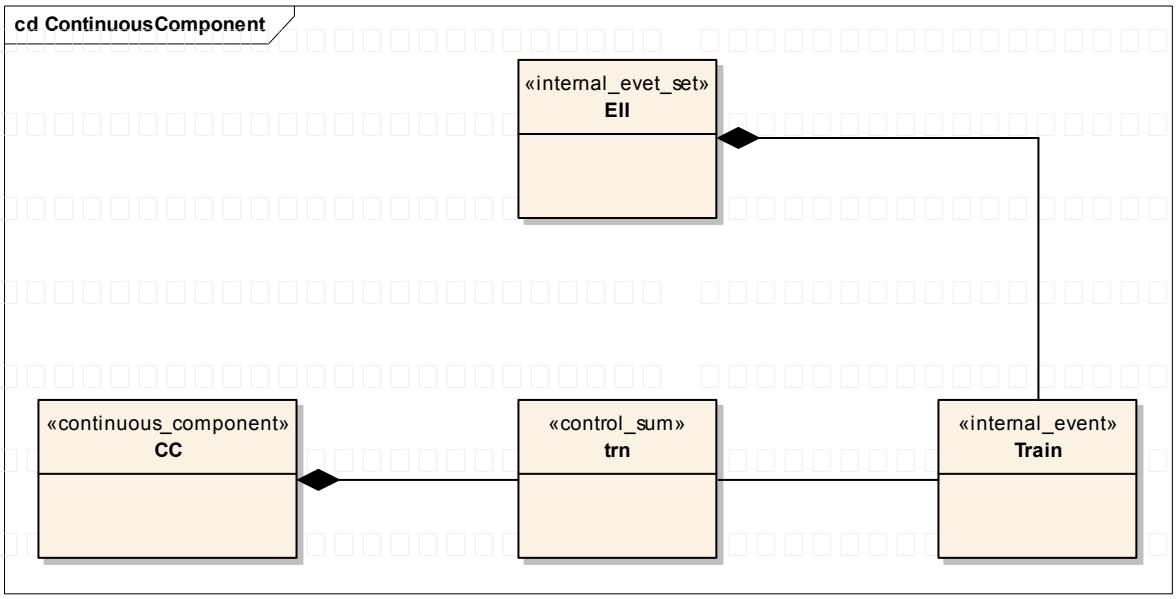
5.1.1.1. Išėjimo signalų aibė



5.1 pav.: Agregato Railway išėjimo signalų aibė

Agregatas *Railway* turi išėjimų aibę *Y*, kuri savyje agreguoja išėjimą *Y1*. Išėjimas tiesiogiai jungiasi su signalu *Train*.

5.1.1.2. Tolydžiosios komponentės

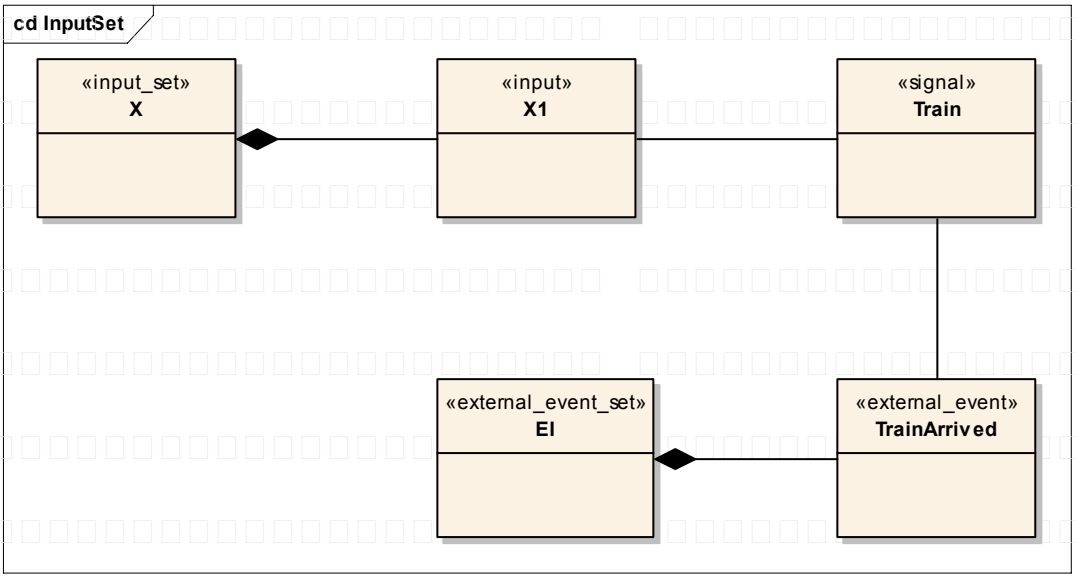


5.2 pav.: Agregato Railway tolydžiosios komponentės

Agregatas *Railway* turi tolydžiąsias komponentes *CC*, kurios agreguoja *ContorlSum* tipo klasę *trn*. Vidinių įvykių aibė *EII* agreguoja vidinį įvykį *Train*, kuris tiesiogiai jungiasi su *ControlSum* tipo klase *trn*.

5.1.2. Agregatas Station

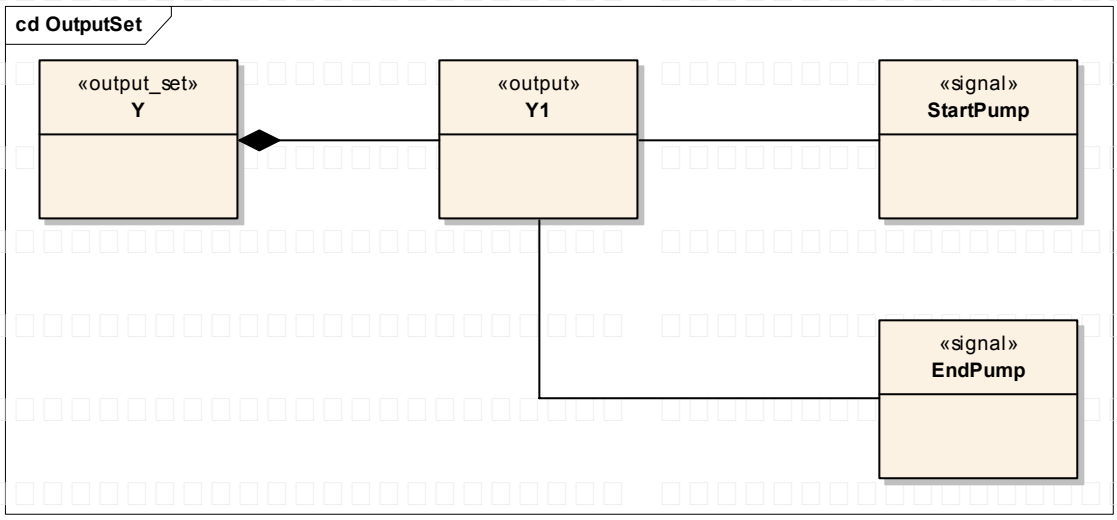
5.1.2.1. Išorinių įvykių aibė



5.3 pav.: Agregato Station išorinių įvykių aibė

Agregatas *Station* savyje turi įėjimo aibę *X*, kuri agreguoja išėjimą *X1*. Išorinių įvykių aibė *EI* agreguoja įvykį *TrainArrived* , kuris tiesiogiai jungiasi su signalu *Train*.

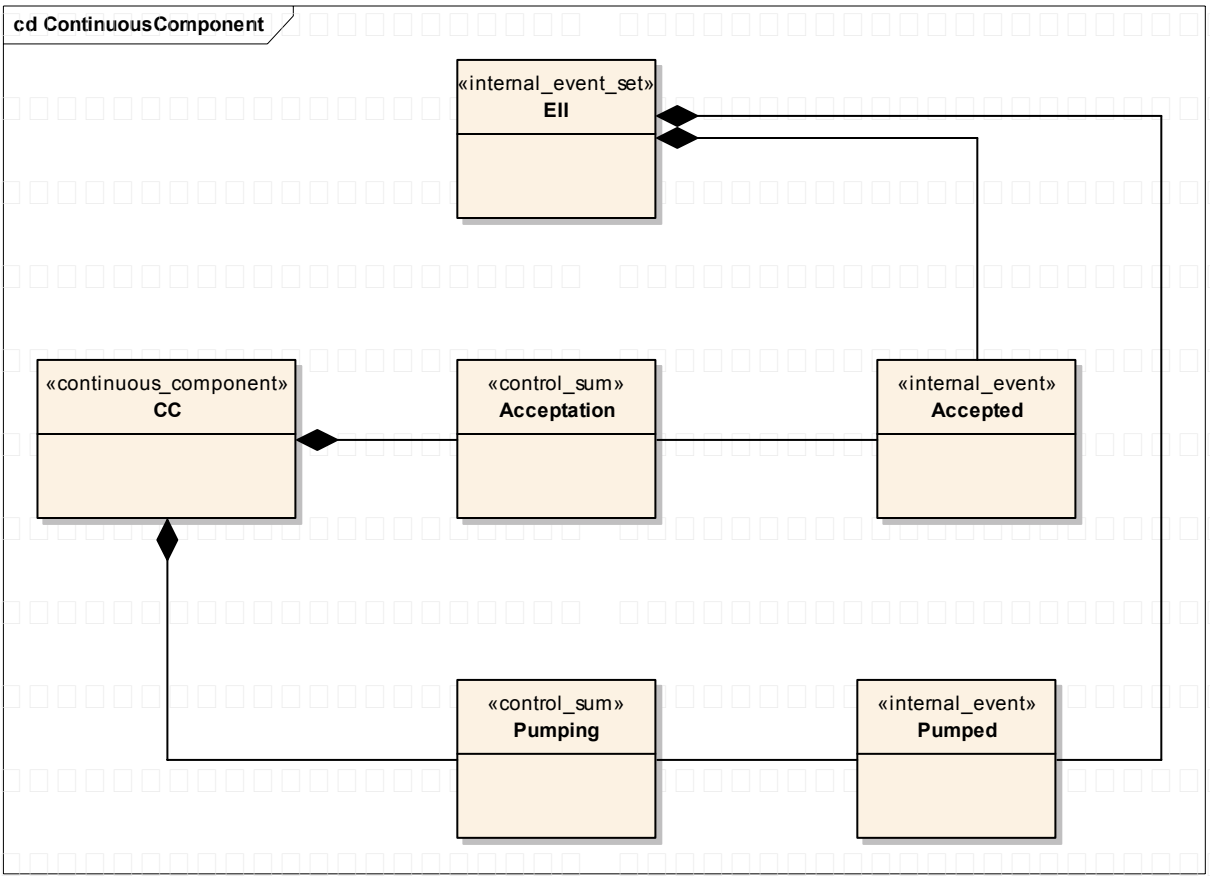
5.1.2.2. Išėjimo signalų aibė



5.4 pav.: Agregato Station išėjimo signalų aibė

Agregatas *Station* turi išėjimo aibę *Y*, kuri agreguoja išėjimą *Y1*. Išėjimas tiesiogiai siejasi su dviem signalais *StartPump* ir *EndPump*.

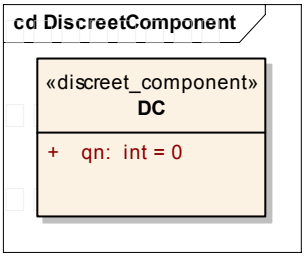
5.1.2.3. Tolydžiosios komponentės



5.5 pav.: Agregato Station tolydžios komponentės

Agregatas *Station* turi tolydžiąsias komponentes *CC*, kurios agreguoja *ContorlSum* tipo klases *Acceptation* ir *Pumping*. Vidinių įvykių aibė *EII* agreguoja vidinius įvykius *Accepted* ir *Pumped*, kurie tiesiogiai jungiasi su *ControlSum* tipo klasėmis *Acceptation* ir *Pumping*.

5.1.2.4. Diskrečių komponentų aibė

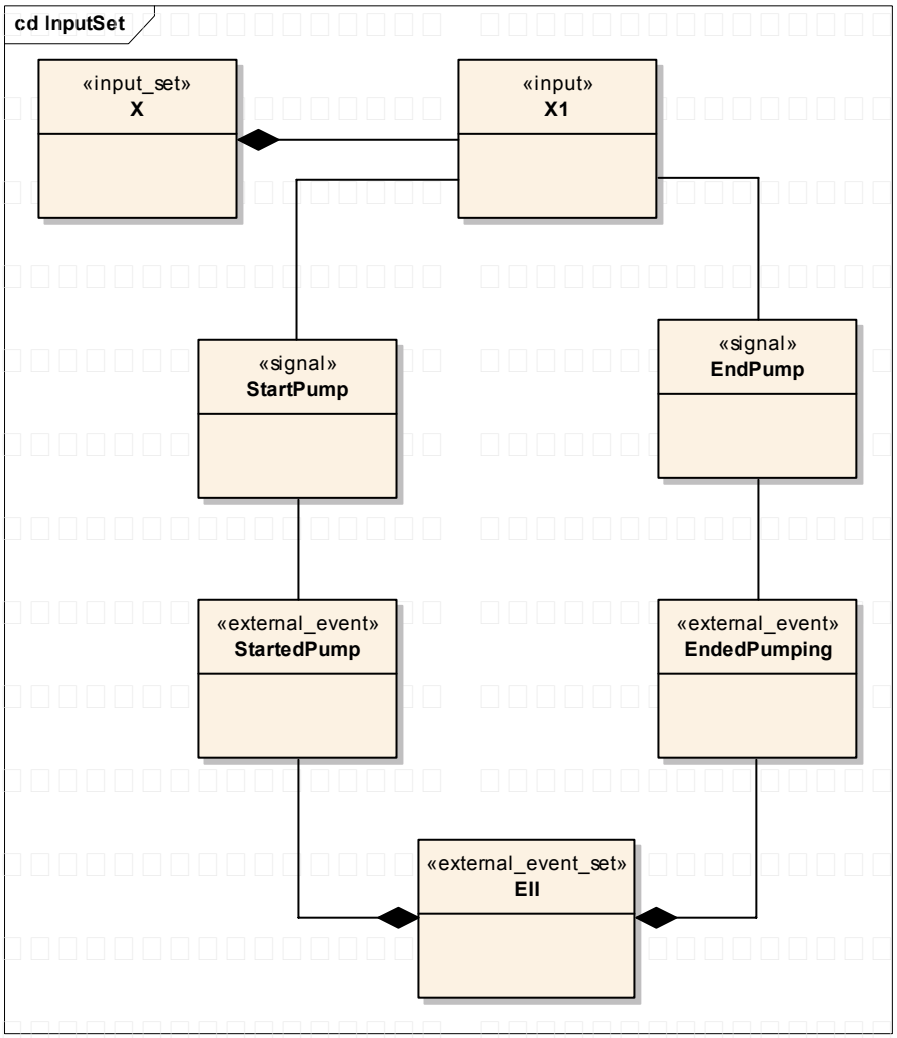


5.6 pav.: Agregato Station diskrečių komponentų aibė

Agregatas *Station* turi vieną diskretų komponentą *DC*, kuris savyje turi *int* tipo kintamąjį *qn* su pradine reikšme lygia *0.*

5.1.3. Agregatas Train

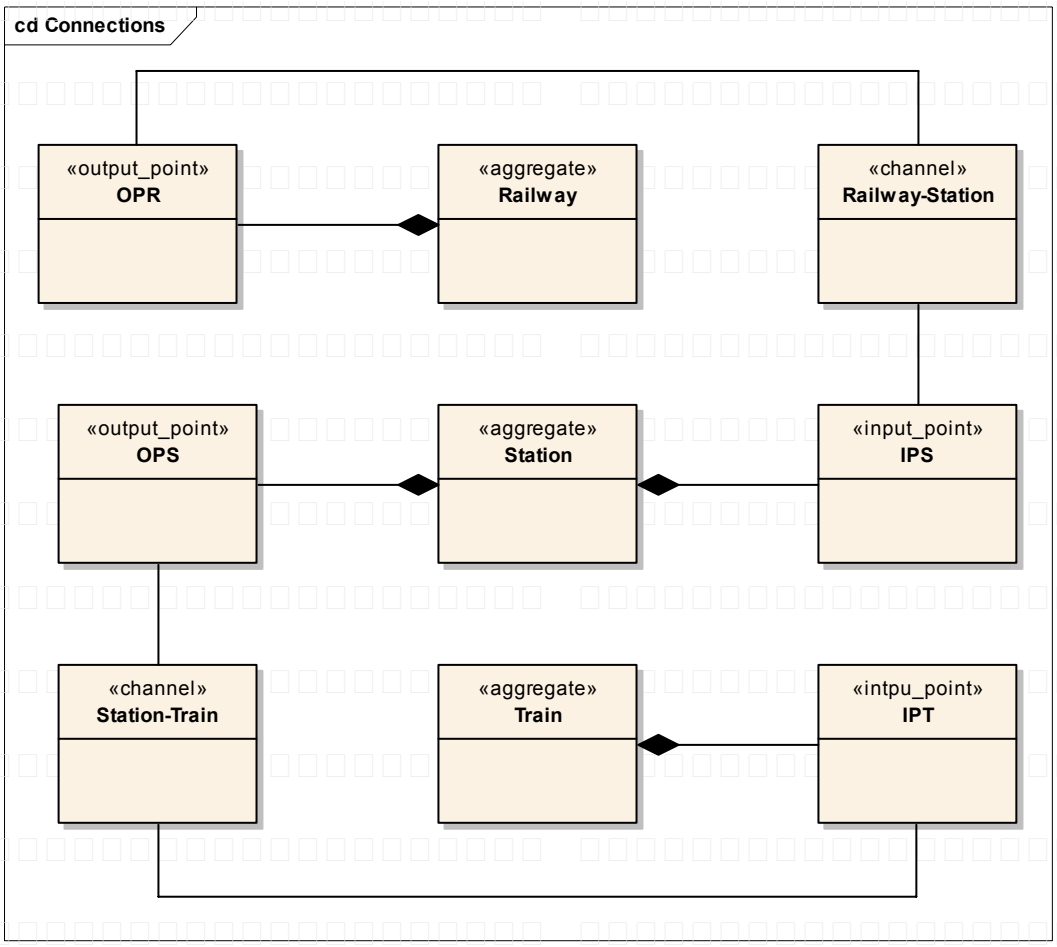
5.1.3.1. Išorinių įvykių aibė



5.7 pav.: Agregato Train išorinių įvykių aibė

Agregatas *Train* savyje turi įėjimo aibę *X*, kuri agreguoja išėjimą *X1*. Išorinių įvykių aibė *EII* agreguoja įvykius *StartedPump* ir *EndedPumping*, kurie tiesiogiai jungiasi su signalais *StartPump* ir *EndPump*.

5.1.4. Sujungimų modelis OilTerminal



5.8 pav.: Sujungimų modelis OilTerminal

Sujungimų modelis *OilTerminal* susideda iš trijų agregatų *Railway, Station* ir *Train*. Agregatas *Railway* per kanalą *Railway-Station* tiesiogiai jungiasi su agregatu *Station*. Taip pat agregatas *Station* per kanalą *Staion-Train* tiesiogiai jungiasi su agregatu *Train*.

Pradiniai duomenys pilnai atitinka PIM ir CIM modelius kurie buvo aprašyti 3.1ir 3.2 skyriuose. Pagal 4 skyriuje pateiktos transformacijos pavyzdį bandysime gauti rezultatus.

Kaip galime pastebėti iš pateiktų duomenų, jie susideda iš 3 agregatų. Pagal tai galime spręsti jog turėtų gautis 3 klasės su pavadinimais (*Railway, Station, Train*) ir *Aggregate* paveldėjimais. Kiekviena iš šių klasių pagal atitinkamas aibes turėtų sugeneruoti atitinkamus klasės atributus. Pavyzdžiui pirmasis agregatas turi vieną *ContinuousComponents* aibę, kuri nurodo, jog rezultate bus tikimąsi vieno klasės atributo su atitinkamu tipu ir pavadinimu (*ControlSum, trn*). Tuo tarpu antrasis agregatas turi vieną *DiscreetComponents* aibę, kuri nurodo, jog antroje klasėje turėtų būti atributas su atitinkamu tipu ir pavadinimu (*qn, int*). Trečiasis agregatas turi du aibės *InputSet* elementus, iš kurių gausime papildomą atributų rinkinį su

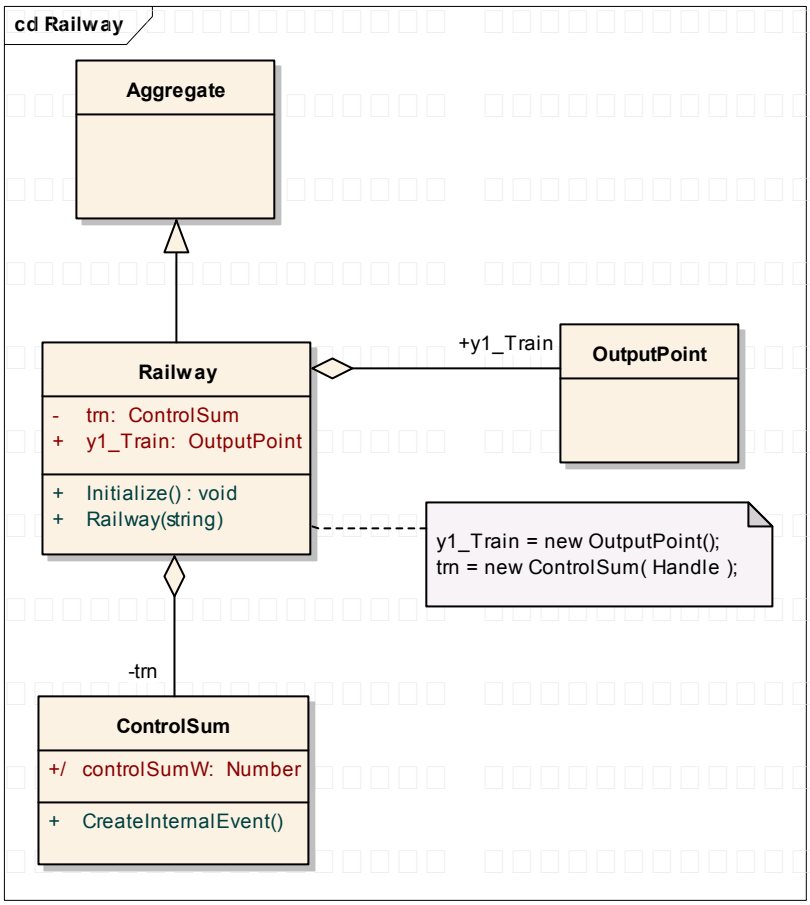
apjungtu pavadinimu ir nustatytu tipu (*x1_EndPump, x1_StartPump, ExtenalEventHandler*). Visuose agregatuose aptinkamos aibės *InputSet* ir *OutputSet*, kurios nurodo, jog turėtų būti sugeneruota atitinkama kombinacija metodų, atitinkančius šių aibių turinį. Papildomai visoms klasėms turėtų būti sugeneruoti konstruktoriai ir *initialize* metodai.

Automatiškai turėtų būti sukurta papildoma klasė, kuri atsakinga už sujungimus tarp agregatų. Klasės pavadinimas bus paimtas pagal sujungimų modulį. Klasės atributų sąrašą turėtų sudaryti agregatų sąrašas, kuris dalyvaus sujungimuose. Pagal sujungimus nurodytus 5.8 paveiksle, klasei turėtų būti sugeneruojami atitinkami sujungimai. Visi susijungimai turėtų būti sugeneruoti klasės konstruktoriuje. Papildomai turėtų būti sugeneruotas *initialize* metodas.

5.2. Laukiami transformacijos rezultatai

Šiame skyriuje bus aptarti gauti rezultatai. Jie bus palyginti su laukiamais, kurie buvo aprašyti ankstesniuose skyriuose. Taip pat bus aptartos klaidos, jų atsiradimo priežastys ir priemonės joms išvengti. Skyriaus pabaigoje bus pateiktos išvados ir galimi patobulinimai modelyje ir meta-modelyje.

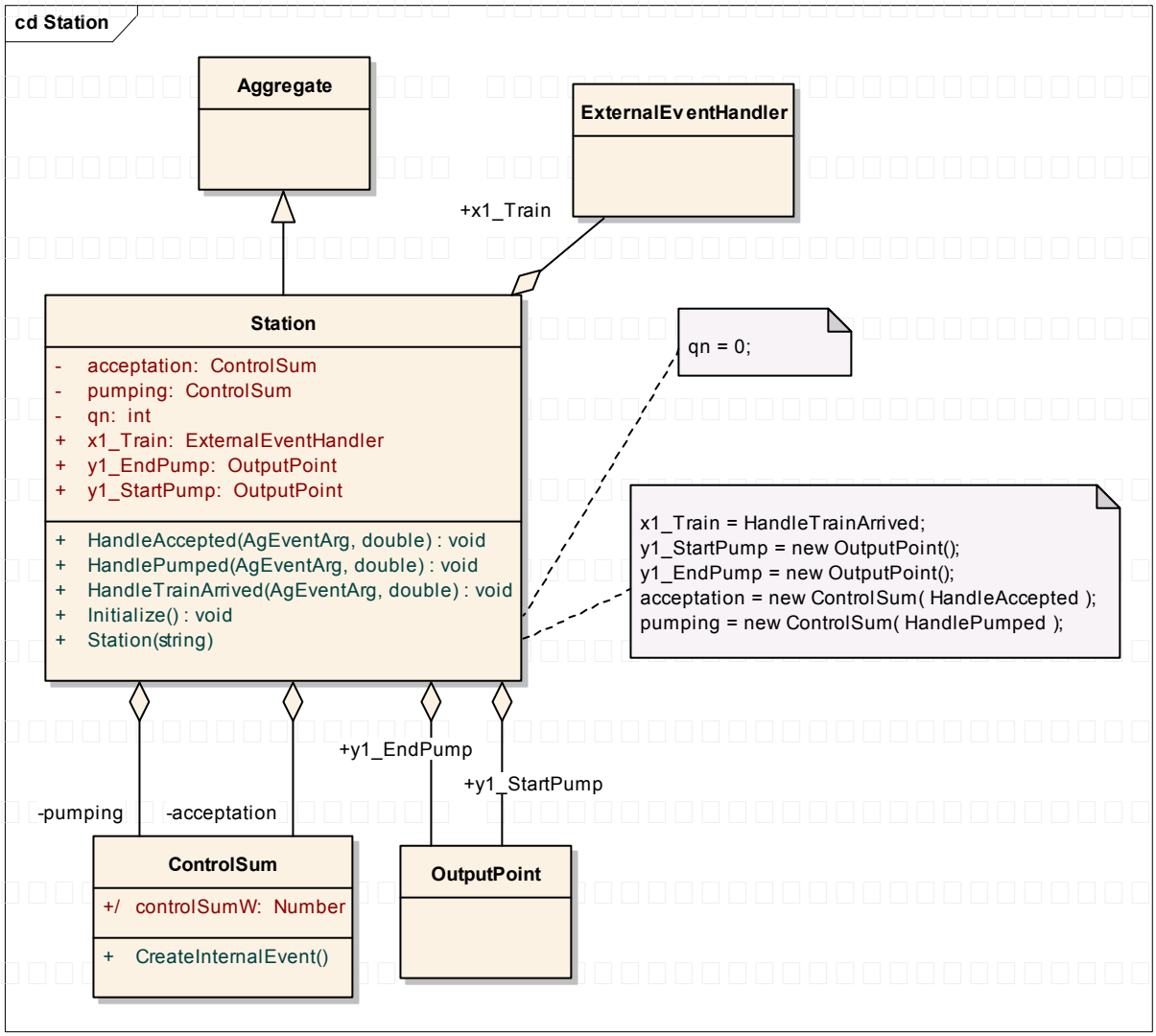
5.2.1. Agregatinė klasė Railway



5.9 pav.: Klasė Railway gauta sutransformavus duomenis

Gauti rezultatai 5.9 paveiksle rodo, jog transformacija sugeneravo klasę, kurios pavadinimas ir paveldėjimas pilnai atitinka laukiamus rezultatus. Papildomai klasės sugeneruoti atributai (*trn* ir *y1_Train*) ir jų tipai (*ControlSum* ir *OutputPoint*) atitinka šaltinio modelio duomenis. Sugeneruotas *initialize* metodas ir konstruktorius, kurie turėtų būti generuojami automatiškai kiekvienai klasei. Notacijoje pateikiama atributų įgyjamos reikšmės konstruktoriuje.

5.2.2. Agregatinė klasė Station

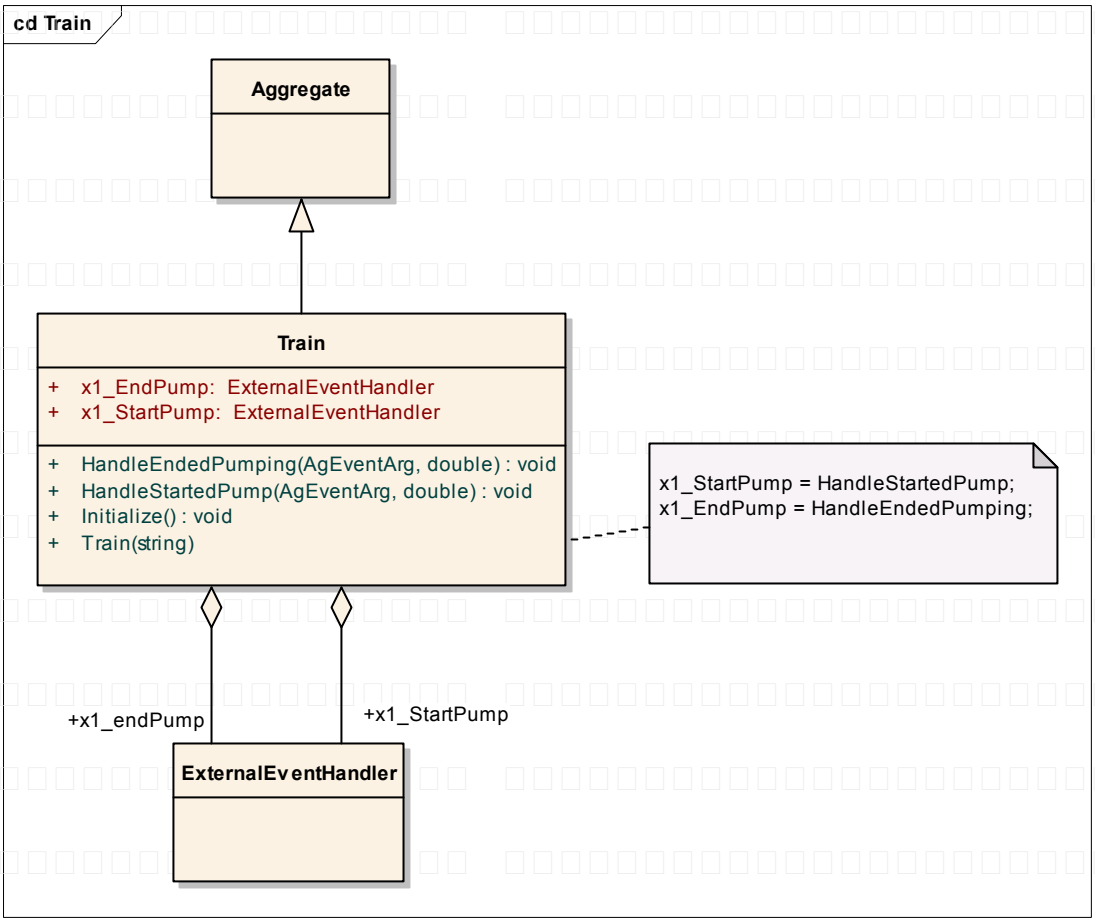


510 pav.: Klasė Station gauta sutransformavus duomenis

Gauti rezultatai 5.10 paveiksle rodo, jog transformacija sugeneravo laukiamus rezultatus. Kaip ir paveiksle 5.9, taip ir šiame klasės, atributų ir konstruktoriaus rezultatai pilnai atitinka pateiktus duomenis. Papildomai šioje klasėje buvo sugeneruotas *qn* atributas, kuris

pasitaikė tik vieną kartą šaltinio modelyje. Pagal teisingai sugeneruotus metodus (*HandleAccepted, HandlePumped* ir *HandelTrainArraived*) galime spręsti, jog transformacija teisingai sugeneruoja laukiamus metodus.

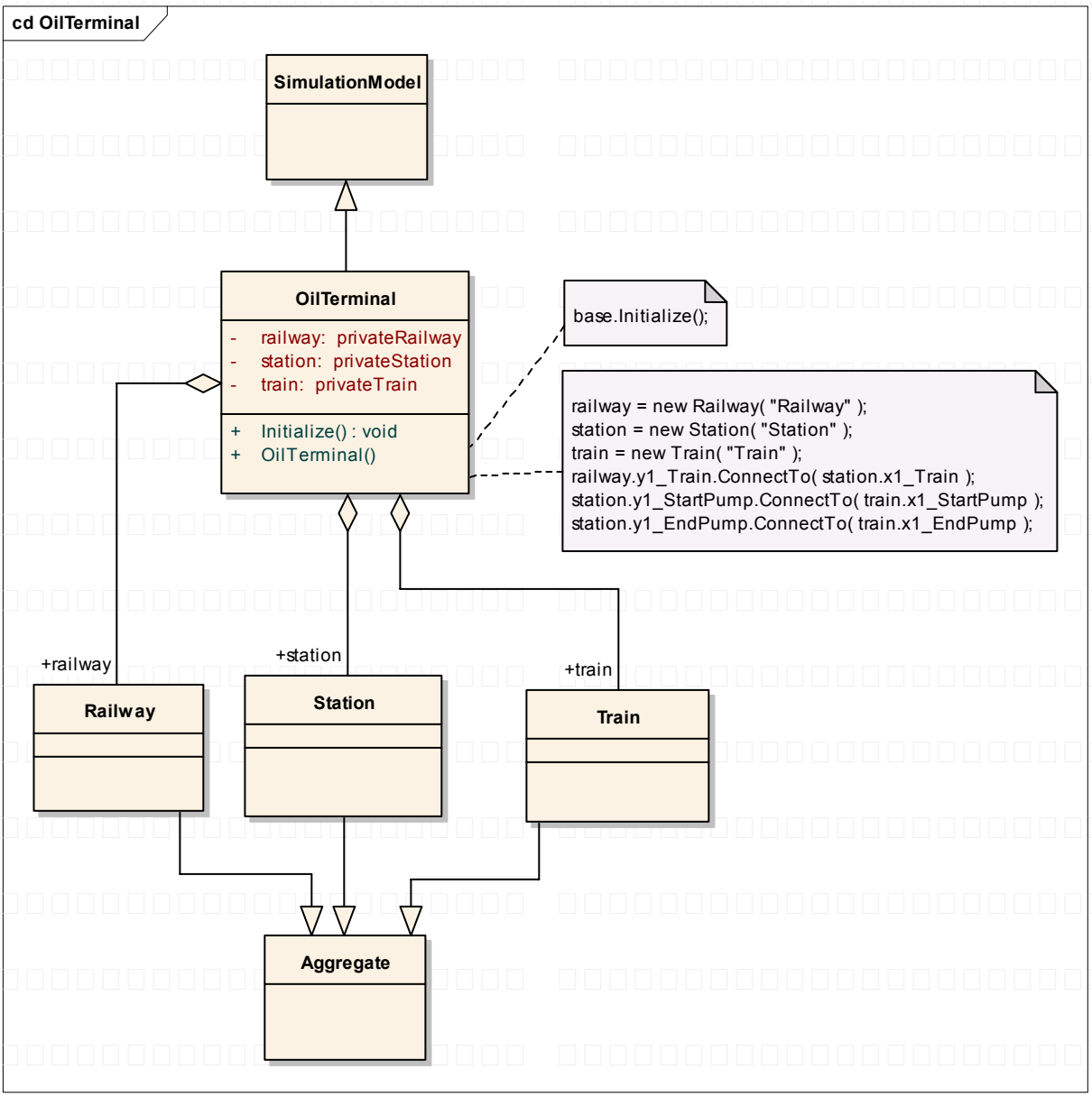
5.2.3. Agregatinė klasė Train



5.11 pav.: Klasė Train gauta sutransformavus duomenis

Kaip ir aukščiau aprašytuose skyriuose, transformacija pilnai pateisino laukiamus rezultatus. Klasė ir jos paveldėjimas atitinka duomenis, atributai ir metodų pavadinimai niekuo nesiskiria nuo laukiamų. Konstruktoriuje sugeneruota atributų reikšmių suteikimas atitinka aukščiau aprašytus rezultatus.

5.2.4. Sujungimų klasė OilTerminal



5.12 pav.: Klasė OilTerminal gauta sutransformavus duomenis

Paveikslas 5.12 rodo, jog transformacija sugeneravo klasę ir paveldėjimą pagal laukimą duomenų modelį. Klasės atributai pilnai atitinka agregatų klases aprašytas 5.2.1 – 5.2.3 skyriuose. Papildomai buvo sugeneruotas *initialiaze* metodas, kuris atitinka visų klasių *initialize* metodą. Klasės konstruktoriuje buvo suteiktos reikšmės klasės atributams (*railway, station* ir *train*). Papildomai pagal sujungimų aprašymus buvo atlikti sujungimai tarp agregatinių klasių, kurie pilnai atitinka laukiamus rezultatus.

6. IŠVADOS

UML, kaip modeliavimo kalbos visuotinis pripažinimas ir paplitimas, atrodo patrauklia priemones modelių vizualizavimui. Atsižvelgiant į specifinius reikalavimus, UML notacija sėkmingai galima specifikuoti ir imitacinius modelius.

Programinio imitacinio kodo kūrimo procesas, modeliais pagrįstos inžinerijos (MDE) kontekste, įgauna naują sampratą – pasitelkiant transformacijas galima automatizuoti perėjimą tarp įvairios abstrakcijos lygio modelių. Proceso automatizacija leidžia daugiau dėmesio skirti kitiems etapams.

Transformacijos MDE erdvėje glaudžiai susijusios su metamodeliais. Sukurtas PLA metamodelis suteikia priemones, PLA formalizmą panaudoti kaip transformacijos elementą. PLA metamodelis tuo pačiu išplečia MDE pritaikymo sritį.

Sukurtos transformacijos leidžia iš aukštesnės abstrakcijos modelių generuoti imitacinį kodą, adaptuotą konkrečioms taikymo platformoms.

7. CREATING SIMULATION MODELS USING UML

Summary

Software development process encounter productivity obstacles, which are affected by an arise of new technologies. Object Management Group (OMG) proposed model driven architecture (MDA) supposed to minimize effect of new technologies arrival into a process of software development. One of the most important aspects in MDA is meta-model, which is used to specify MDA style transformations. Theses present meta-model of PLA model and XSL transformations, used to retrieve PLA simulation model. Example of creating oil terminal simulation model is presented as well.

8. LITERATŪRA

[1] Akehurst, D., Kent, S. A relational approach to defining transformations in a metamodel. In proceedings of UML2002, Germany, 2002
[2] Alanen, M., Lilius, J., Porres, I., and Truscan, D. Realizing a Model Driven Engineering Process. TUCS Technical report No 565, 2003
[3] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Technologies Corp. Response to the MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2003-08-05, 2003
[4] Álvarez, J., Evans, A., Sammut, P. Mapping between levels in the metamodel architecture. In Proceedings of UML2001, Springer-Verlag Heidelberg, Toronto, Canada, 2001
[5] Apostel, L. Towards the formal study of models in the non-formal sciences. In H. Freudenthal (Ed.), The concept and the role of the model in mathematics and natural and social sciences. D. Reidel Publishing Company, Dordrecht, the Netherlands, 1960
[6] Atkinson, C., Kühne, T. Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. 12 (4), pp. 290-321, 2002
[7] Beckett, D. RDF/XML syntax specification. W3C Document, 2003
[8] Bézivin, J., Gerbe, O. Towards a precise definition of the OMG/MDA framework. In Proceedings of Automated Software Engineering, USA, 2001
[9] Bézivin, J., Farcet, N., Jezequel, J-M, Langlois, B., Pollet, D. Reflective model driven engineering. In Proceedings of UML2003, USA, 2003
[10] Boman, M., Bubenko Jr., J.A., Johannesson, P., Wangler, B. Conceptual Modelling. Prentice Hall, 1997
[11] Brinkkemper, S. Formalisation of information systems modeling. PhD Thesis, University of Nijmegen, 1990
[12] DSTC, IBM, CBOP. MOF Query/Views/Transformations. Second Revised Submission. OMG document ad/2004-01-06, 2004
[13] Duke, R., Rose, G., and Smith, G. Object-Z: a specification language advocated for the description of standards. Technical report 94-45, Software Verification Research Center, University of Queensland, Australia, 1994
[14] Falkenberg, E.D., Hesse, W., Lindgreen, P., Nilsson, B.E., Han Oei, J.E., Rolland, C., Stamper, R.K., van Assche, F.J.M., Verrijn-Stuart, A.A., Voss, K. A framework of information system concepts. The FRISCO report. 1998
[15] Favre, J-M. Towards a basic theory to model Model Driven Engineering. 3d Workshop in Software Model Engineering. In conjunction with UML2004. Portugal, 2004
[16] Favre, J-M., Nguyen, T. Towards a megamodel to model software evolution through transformations. Workshop on Software Evolution through Transformations (SETRA2004). In conjunction with 2nd Intl. Conference on Graph Transformations, Rome, Italy, 2004
[17] Filman, R., Elrad, T., Clarke, S., and Aksit, M. Aspect-Oriented Software Development. Addison-Wesley. 2004
[18] Geisler, R., Klar, M., Pons, C. Dimensions and dichotomy in metamodeling. In D.J. Duke and A.S.Evans (Eds.), 3rd BCS-FACS Northern Formal Methods Workshop. Springer-Verlag, New York, USA, 1998
[19] van Gigch, J.P. System design modeling and metamodeling. Plenum Press, New

York, 1991
[20] Hausmann, J., H. Metamodeling relations – relating metamodels. In 1st International Workshop on Metamodeling for MDA. York, UK, 2003
[21] Kalnins, A., Barzdins, J., Celms, E. Model transformation language MOLA. In U. Asmann (Ed.), Proceedings of Model Driven Architecture: Foundations and Applications 2004. Linkoping, Sweden, 2004
[22] Kent, S. Model Driven Engineering. In Proceedings of IFM2002, LNCS 2335, Springer, 2002
[23] Kleppe, A., Warmer, J., Bast, W. MDA Explained. The Model Driven Architecture: Practice and Promise. Addison-Wesley, 2003
[24] Kurtev, I., Bézivin, J., Aksit, M. Technological Spaces: an initial appraisal. CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine, CA, USA, 2002
[25] Molenaar, J. Mathematical modeling and dimensional analysis. In A. van den Burgh and J. Simonis (Eds.), Topics in Engineering Mathematics. Modeling and Methods, Kluwer Academic Publishers, 1992
[26] OMG. MDA Guide version 1.0.1. OMG document omg/2003-06-01, 2003
[27] OMG. Common Object Request Broker Architecture
[28] OMG. Meta Object Facility (MOF) Specification. OMG document formal/02-04-03, 2002
[29] OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10, 2002
[30] OMG. OMG Unified Modeling Language Specification v. 1.4. OMG Document, 2001
[31] OMG. XML Metadata Interchange (XMI) Specification. OMG document formal/ 03-05-02, 2003
[32] Patrascoiu, O. YATL: Yet Another Transformation Language. In M. van Sinderen and L. Pires (Eds.), 1st European MDA Workshop on Industrial Applications (MDA-IA), CTIT Technical report TR-CTIT-04-12, Enschede, the Netherlands, 2004
[33] QVT-Partners. Revised submission for MOF 2.0 Query/Views/Transformations RFP. 2003
[34] Seidewitz, E. What Models Mean. IEEE Software, 20(5), 2003.
[35] Starfield, M., Smith, K.A., and Bleloch, A.L. How to model it: Problem Solving for the Computer Age. McGraw-Hill, New York, 1990.
[36] Willink, E. UMLX: A graphical transformation language for MDA. In A. Rensink (Ed.), Model Driven Architecture: Foundations and Applications 2003, CTIT Technical Report TR-CTIT-03-27, University of Twente, the Netherlands, 2003
[37] W3C. XML Schema Parrt 1: Structures, 2001
[38] W3C. Web Services Activity. Available on http://www.w3.org/2002/ws/.
[39] W3C. XML Path Language (XPath) 2.0, 2005

9. TERMINŲ IR SANTRUMPŲ ŽODYNAS

PIM –Platform Independent Model – Nuo platformos nepriklausomas modelis.

PSM – Platform Specific Model – Tam tikros platformos modelis.

CIM – Computationally Independent Model – Nuo skaičiavimų nepriklausomas modelis.

MDA – Model Driven Architecture – Modeliais pagrįsta architektūra.

MDE – Model Driven Engeneering – Modeliais pagrįsta inžinerija.

PLA – Piece Linear Agregate – Atkarpomis tiesiniai agregatai.

UML – Unified Modeling Language – Suvienyta modeliavimo kalba.

XML – Extensible Markup Language – Paprastas ir lankstus teksto formatas.

XMI – XML Metadata Interchange – Standartas naudojamas apsikeisti modeliais per XML dokumentus.

XSD – XML Schema Definition – XML schemos apibrėžimas.

XSL – Extensible Stylesheet Language – Standartas apibrėžti XML dokumento transformacijoms.

OMG – Object Management Group – Organizacija leidžianti ir užtikrinanti kompiuterių mokslo standartus.

MOF – Meta Object Facility – OMG standartas, glaudžiai susijęs su UML, leidžiantis meta-duomenų valdymą ir modeliavimo kalbų apibrėžimą.

10. PRIEDAI

10.1. Transformacijos failas

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
           xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
           xmlns:PLAMM="PLAMM"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
           <xsl:output method="text" encoding="UTF-8"/>
           <xsl:variable
name="UpperCase">ABCDEFGHIKLMNOPQRSTVXYZ</xsl:variable>
           <xsl:variable
name="LowerCase">abcdefghiklmnopqrstvxyz</xsl:variable>
           <!-- Root template <aggregate ...>-->
           <xsl:template match="PLAMM:AggregateSpecification">
                      <!--create class header-->
                      <xsl:call-template name="CreateClassHeader"/>
                      <xsl:for-each select="PLAMM:aggregates">
                                 <xsl:variable name="AggregatePosition"
select="position() - 1" />
                                 <!--create class-->
                                 <xsl:call-template name="CreateClass">
                                            <xsl:with-param name="Name"
select="@name"/>
                                            <xsl:with-param name="Parent"
select ="'Aggregate'" />
                                 </xsl:call-template>
                                 <!--create ExternalEventHandler-->
                                 <xsl:call-template
name="CreateExternalEventHandlerAttributes" />
                                 <!--create OutputPoint-->
                                 <xsl:call-template
name="CreateOutputPointAttributes" />
                                 <!--create ContinuousComponents-->
                                 <xsl:call-template
name="CreateContinuousComponents" />
                                 <!--create discreateComponent-->
                                 <xsl:call-template
name="CreateDiscreateComponent" />
                                 <!--create constructor-->
                                 <xsl:call-template name="CreateConstructor">
                                            <xsl:with-param name="Name"
select="@name" />
                                            <xsl:with-param
name="AggregatePosition" select="$AggregatePosition"/>
                                 </xsl:call-template>
                                 <!--create  Initialize method-->
                                 <xsl:call-template name="CreateInitialize" />
                                 <!--create Handlers for internal and external
events-->
                                 <xsl:call-template name="CreateHandlers" />
                                 <!--close class-->
                                 <xsl:call-template name="CloseClass" />
                      </xsl:for-each>
                      <!--create class-->
                      <xsl:call-template name="CreateClass">
```

```
                                    <!--<xsl:with-param name="Name"
select="'OilTerminal'"/>-->
                                    <xsl:with-param name="Name" select="@name"/>
                                    <xsl:with-param name="Parent"
select="'SimulationModel'" />
                        </xsl:call-template>
                        <!-- create elements-->
                        <xsl:variable name="Connections"
select="PLAMM:connectionChannels"/>
                        <xsl:for-each select="PLAMM:aggregates">
                                    <xsl:variable name="AggregatePosition"
select="position() - 1" />
                                    <xsl:if test="count($Connections[substring-
before(substring-after(@destinationEnds, '#//@aggregates.'), '/') =
$AggregatePosition]) > 0 or count($Connections[substring-before(substring-
after(@sourceEnd, '#//@aggregates.'), '/') = $AggregatePosition]) > 0">
                                                <xsl:text>
            private</xsl:text>
                                                <xsl:value-of select="@name"/>
                                                <xsl:text> </xsl:text>
                                                <xsl:value-of
select="translate(@name, $UpperCase, $LowerCase)"/>
                                                <xsl:text>;&#10;</xsl:text>
                                    </xsl:if>
                        </xsl:for-each>
                        <!--create class Constructor derived from
SimulationModel-->
                        <xsl:call-template
name="CreateSimulationModelChildConstructor">
                                    <!--<xsl:with-param name="Name"
select="'OilTerminal'"/>-->
                                    <xsl:with-param name="Name" select="@name"/>
                        </xsl:call-template>
                        <!--create class initialization derived from
SimulationModel-->
                        <xsl:text>&#10;                 public override void
Initialize()&#10;</xsl:text>
                        <xsl:text>           {&#10;</xsl:text>
                        <xsl:text>
            base.Initialize();&#10;</xsl:text>
                        <xsl:text>           }&#10;</xsl:text>
                        <!--close class-->
                        <xsl:call-template name="CloseClass" />
                        <!--close Namespcse-->
                        <xsl:call-template name="CloseNameSpace"/>
            </xsl:template>
            <xsl:template name="CreateClassHeader">
                        <xsl:text>using System;&#10;</xsl:text>
                        <xsl:text>using
System.Collections.Generic;&#10;</xsl:text>
                        <xsl:text>using System.Text;&#10;</xsl:text>
                        <xsl:text>using PLASimulation;&#10;&#10;</xsl:text>
                        <xsl:text>namespace PLASimulationText&#10;</xsl:text>
                        <xsl:text>{&#10;</xsl:text>
            </xsl:template>

            <xsl:template name="CreateClass">
                        <xsl:param name="Name"/>
                        <xsl:param name="Parent"/>
```

```
                    <xsl:text> class </xsl:text>
                    <xsl:value-of select="$Name"/>
                    <xsl:text> : </xsl:text>
                    <xsl:value-of select="$Parent"/>
                    <xsl:text>&#10;</xsl:text>
                    <xsl:text> {&#10;</xsl:text>
          </xsl:template>

          <xsl:template name="CreateExternalEventHandlerAttributes">
                    <xsl:variable name="InputPointName"
select="PLAMM:inputPoints/@name"/>
                    <xsl:for-each select="PLAMM:inputs/PLAMM:inputSignals">
                              <xsl:text>           public
ExternalEventHandler </xsl:text>
                              <xsl:value-of select="$InputPointName"/>
                              <xsl:text>_</xsl:text>
                              <xsl:value-of select="@name" />
                              <xsl:text>;&#10;</xsl:text>
                    </xsl:for-each>
          </xsl:template>

          <xsl:template name="CreateOutputPointAttributes">
                    <xsl:variable name="OutputPointName"
select="PLAMM:outputPoints/@name"/>
                    <xsl:for-each
select="PLAMM:outputs/PLAMM:outputSignals">
                              <xsl:text>           public OutputPoint
</xsl:text>
                              <xsl:value-of select="$OutputPointName"/>
                              <xsl:text>_</xsl:text>
                              <xsl:value-of select="@name" />
                              <xsl:text>;&#10;</xsl:text>
                    </xsl:for-each>
          </xsl:template>

          <xsl:template name="CreateContinuousComponents">
                    <xsl:for-each
select="PLAMM:continuousComponent/PLAMM:continuousCoordinates">
                              <xsl:text>           private </xsl:text>
                              <xsl:value-of select="@xsi:type" />
                              <xsl:text> </xsl:text>
                              <xsl:value-of select="translate(@name,
$UpperCase, $LowerCase)" />
                              <xsl:text>;&#10;</xsl:text>
                    </xsl:for-each>
          </xsl:template>

          <xsl:template name="CreateDiscreateComponent">
                    <xsl:for-each
select="PLAMM:discreetComponent/PLAMM:discreetCoordinates">
                              <xsl:text>           private </xsl:text>
                              <xsl:value-of select="@type" />
                              <xsl:text> </xsl:text>
                              <xsl:value-of select="translate(@name,
$UpperCase, $LowerCase)" />
                              <xsl:text>;&#10;</xsl:text>
                    </xsl:for-each>
          </xsl:template>
```

```
        <xsl:template name="CreateConstructor">
                <xsl:param name="Name" />
                <xsl:param name="AggregatePosition"/>
                <xsl:text>&#10;                public </xsl:text>
                <xsl:value-of select="$Name" />
                <xsl:text>( string name ) : base( name )&#10;</xsl:text>
                <xsl:text>          {&#10;</xsl:text>

                <xsl:variable name="ExternalEvents"
select="PLAMM:externalEvents/PLAMM:externalEvents" />
                <xsl:variable name="InputPointName"
select="PLAMM:inputPoints/@name" />
                <xsl:for-each select="PLAMM:inputs/PLAMM:inputSignals">
                        <xsl:variable name="Position"
select="position() - 1" />
                        <xsl:text>                      </xsl:text>
                        <xsl:value-of select="concat($InputPointName,
'_', @name)" />
                        <xsl:text> = Handle</xsl:text>
                        <xsl:value-of
select="$ExternalEvents[@eventSignal = concat('#//@aggregates.',
$AggregatePosition,'/@inputs/@inputSignals.', $Position)]/@name" />
                        <xsl:text>;&#10;</xsl:text>
                </xsl:for-each>

                <xsl:variable name="OutputPointName"
select="PLAMM:outputPoints/@name" />
                <xsl:for-each
select="PLAMM:outputs/PLAMM:outputSignals">
                        <xsl:text>                      </xsl:text>
                        <xsl:value-of
select="concat($OutputPointName, '_', @name)" />
                        <xsl:text> = new
OutputPoint();&#10;</xsl:text>
                </xsl:for-each>

                <xsl:variable name="InternalEvents"
select="PLAMM:internalEvents/PLAMM:internalEvents" />
                <xsl:for-each
select="PLAMM:continuousComponent/PLAMM:continuousCoordinates">
                        <xsl:variable name="Position"
select="position() - 1" />
                        <xsl:text>                      </xsl:text>
                        <xsl:value-of select="translate(@name,
$UpperCase, $LowerCase)" />
                        <xsl:text> = new </xsl:text>
                        <xsl:value-of select="@xsi:type"/>
                        <xsl:text>( Handle</xsl:text>
                        <xsl:value-of
select="$InternalEvents[@continuousCoordinate =
concat('#//@aggregates.',$AggregatePosition,'/@continuousComponent/@continuous
Coordinates.', $Position)]/@name" />
                        <xsl:text> );&#10;</xsl:text>
                </xsl:for-each>

                <xsl:text>          }&#10;</xsl:text>
        </xsl:template>

        <xsl:template name="CreateInitialize">
```

```
                    <xsl:text>&#10;                public override void
Initialize()&#10;</xsl:text>
                    <xsl:text>           {&#10;</xsl:text>
                    <xsl:for-each
select="PLAMM:discreetComponent/PLAMM:discreetCoordinates">
                              <xsl:text>                      </xsl:text>
                              <xsl:value-of select="translate(@name,
$UpperCase, $LowerCase)" />
                              <xsl:text> = </xsl:text>
                              <xsl:value-of select="@value" />
                              <xsl:text>;&#10;</xsl:text>
                    </xsl:for-each>
                                                  <xsl:text>
          //TODO write code&#10;</xsl:text>
                    <xsl:text>           }&#10;</xsl:text>
          </xsl:template>

          <xsl:template name="CreateHandlers">
                    <xsl:for-each
select="PLAMM:externalEvents/PLAMM:externalEvents">
                              <xsl:text>&#10;                public void
Handle</xsl:text>
                              <xsl:value-of select="@name" />
                              <xsl:text>( AgEventArg ms, double tm
)&#10;</xsl:text>
                              <xsl:text>           {&#10;</xsl:text>
                              <xsl:text>                      //TODO write
code&#10;</xsl:text>
                              <xsl:text>           }&#10;</xsl:text>
                    </xsl:for-each>
                    <xsl:for-each
select="PLAMM:internalEvents/PLAMM:internalEvents">
                              <xsl:text>&#10;                public void
Handle</xsl:text>
                              <xsl:value-of select="@name" />
                              <xsl:text>( AgEventArg av, double tm
)&#10;</xsl:text>
                              <xsl:text>           {&#10;</xsl:text>
                              <xsl:text>                      //TODO write
code&#10;</xsl:text>
                              <xsl:text>           }&#10;</xsl:text>
                    </xsl:for-each>
          </xsl:template>

          <xsl:template name="CloseClass">
                    <xsl:text> }&#10;</xsl:text>
          </xsl:template>

          <xsl:template name="CloseNameSpace">
                    <xsl:text>}</xsl:text>
          </xsl:template>

          <xsl:template name="CreateSimulationModelChildConstructor">
                    <xsl:param name="Name"/>
                    <xsl:text>&#10;                public </xsl:text>
                    <xsl:value-of select="$Name"/>
                    <xsl:text>()&#10;</xsl:text>
                    <xsl:text>           {&#10;</xsl:text>
```

```
                        <xsl:variable name="Connections"
select="PLAMM:connectionChannels"/>
                        <xsl:for-each select="PLAMM:aggregates">
                                    <xsl:variable name="AggregatePosition"
select="position() - 1" />
                                    <xsl:if test="count($Connections[substring-
before(substring-after(@destinationEnds, '#//@aggregates.'), '/') =
$AggregatePosition]) > 0 or count($Connections[substring-before(substring-
after(@sourceEnd, '#//@aggregates.'), '/') = $AggregatePosition]) > 0">
                                                <xsl:text>
          </xsl:text>
                                                <xsl:value-of
select="translate(@name, $UpperCase, $LowerCase)"/>
                                                <xsl:text> = new </xsl:text>
                                                <xsl:value-of select="@name"/>
                                                <xsl:text>( "</xsl:text>
                                                <xsl:value-of select="@name"/>
                                                <xsl:text>" );&#10;</xsl:text>
                                    </xsl:if>
                        </xsl:for-each>
                        <xsl:variable name="Aggregates"
select="PLAMM:aggregates"/>
                        <xsl:for-each select="$Connections">
                                    <xsl:variable name="AggregateOutputNumber"
select="substring-before(substring-after(@sourceEnd, '#//@aggregates.'), '/')
+ 1"/>
                                    <xsl:variable name="OutputPointNumber"
select="substring-after(@sourceEnd, 'outputPoints.') + 1"/>
                                    <xsl:variable name="AggregateInputNumber"
select="substring-before(substring-after(@destinationEnds, '#//@aggregates.'),
'/') + 1"/>
                                    <xsl:variable name="InputPointNumber"
select="substring-after(@destinationEnds, 'inputPoints.') + 1"/>
                                    <xsl:for-each
select="$Aggregates[$AggregateOutputNumber]/PLAMM:outputs/PLAMM:outputSignals"
>
                                                <xsl:text>
          </xsl:text>
                                                <xsl:value-of
select="translate($Aggregates[$AggregateOutputNumber]/@name, $UpperCase,
$LowerCase)"/>
                                                <xsl:text>.</xsl:text>
                                                <xsl:value-of
select="$Aggregates[$AggregateOutputNumber]/PLAMM:outputPoints[$OutputPointNum
ber]/@name"/>
                                                <xsl:text>_</xsl:text>
                                                <xsl:value-of select="@name"/>
                                                <xsl:text>.ConnectTo( </xsl:text>
                                                <xsl:value-of
select="translate($Aggregates[$AggregateInputNumber]/@name, $UpperCase,
$LowerCase)"/>
                                                <xsl:text>.</xsl:text>
                                                <xsl:value-of
select="$Aggregates[$AggregateInputNumber]/PLAMM:inputPoints[$InputPointNumber
]/@name"/>
                                                <xsl:text>_</xsl:text>
                                                <xsl:variable
name="CurrentPosition" select="position()"/>
```

```
                                        <xsl:value-of
select="$Aggregates[$AggregateInputNumber]/PLAMM:inputs/PLAMM:inputSignals[$Cu
rrentPosition]/@name"/>
                                        <xsl:text> );&#10;</xsl:text>
                                </xsl:for-each>
                        </xsl:for-each>
                        <xsl:text>           }&#10;</xsl:text>
          </xsl:template>

</xsl:stylesheet>
```

10.2. Duomenų modelis

```
<?xml version="1.0" encoding="ASCII"?>
<AggregateSpecification
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="PLAMM" xsi:schemaLocation="PLAMM PIM.xsd" name="OilTerminal">
  <aggregates name="Railway">
    <outputs>
      <outputSignals name="Train"/>
    </outputs>
    <operatorsGH outputPoints="#//@aggregates.0/@outputPoints.0"/>
    <internalEvents>
      <internalEvents name="Train"
eventSignal="#//@aggregates.0/@inputs/@inputSignals.0"

continuousCoordinate="#//@aggregates.0/@continuousComponent/@continuousCoordin
ates.0"/>
    </internalEvents>
    <continuousComponent>
      <continuousCoordinates xsi:type="ControlSum" name="trn"/>
    </continuousComponent>
    <outputPoints name="y1" channel="#//@connectionChannels.0"/>
  </aggregates>
  <aggregates name="Station">
    <outputs>
      <outputSignals name="StartPump"/>
      <outputSignals name="EndPump"/>
    </outputs>
    <inputs>
      <inputSignals name="Train"/>
    </inputs>
    <operatorsGH event="#//@aggregates.1/@externalEvents/@externalEvents.0"
continuousCoordinates="#//@aggregates.1/@continuousComponent/@continuousCoordi
nates.0"/>
    <operatorsGH event="#//@aggregates.1/@internalEvents/@internalEvents.0"
discreetCoordinates="#//@aggregates.1/@discreetComponent/@discreetCoordinates.
0"

continuousCoordinates="#//@aggregates.1/@continuousComponent/@continuousCoordi
nates.1"
        outputPoints="#//@aggregates.1/@outputPoints.0"/>
    <operatorsGH event="#//@aggregates.1/@internalEvents/@internalEvents.1"
discreetCoordinates="#//@aggregates.1/@discreetComponent/@discreetCoordinates.
0"

continuousCoordinates="#//@aggregates.1/@continuousComponent/@continuousCoordi
nates.1"
        outputPoints="#//@aggregates.1/@outputPoints.0"/>
```

```
    <externalEvents>
      <externalEvents name="TrainArrived"
eventSignal="#//@aggregates.1/@inputs/@inputSignals.0"/>
    </externalEvents>
    <internalEvents>
      <internalEvents name="Accepted"
eventSignal="#//@aggregates.1/@inputs/@inputSignals.0"

continuousCoordinate="#//@aggregates.1/@continuousComponent/@continuousCoordin
ates.0"/>
      <internalEvents name="Pumped"
continuousCoordinate="#//@aggregates.1/@continuousComponent/@continuousCoordin
ates.1"/>
    </internalEvents>
    <discreetComponent>
      <discreetCoordinates name="Qn" value="0" type="int"/>
    </discreetComponent>
    <continuousComponent>
      <continuousCoordinates xsi:type="ControlSum" name="Acceptation"
internalEvent="#//@aggregates.1/@internalEvents/@internalEvents.0"
          controlSumW="0.0" controlSequence="#//@controlSequences.0"/>
      <continuousCoordinates xsi:type="ControlSum" name="Pumping"
internalEvent="#//@aggregates.1/@internalEvents/@internalEvents.1"
          controlSumW="0.0" controlSequence="#//@controlSequences.1"/>
    </continuousComponent>
    <outputPoints name="y1" channel="#//@connectionChannels.1">
      <outputPointSet
outputSignals="#//@aggregates.1/@outputs/@outputSignals.0
#//@aggregates.1/@outputs/@outputSignals.1"/>
    </outputPoints>
    <inputPoints name="x1" inputChannel="#//@connectionChannels.1">
      <inputPointSet inputSignals="#//@aggregates.1/@inputs/@inputSignals.0"/>
    </inputPoints>
  </aggregates>
  <aggregates name="Train">
    <inputs>
      <inputSignals name="StartPump"/>
      <inputSignals name="EndPump"/>
    </inputs>
    <operatorsGH/>
    <externalEvents>
      <externalEvents name="StartedPump"
eventSignal="#//@aggregates.2/@inputs/@inputSignals.0"/>
      <externalEvents name="EndedPumping"
eventSignal="#//@aggregates.2/@inputs/@inputSignals.1"/>
    </externalEvents>
    <inputPoints name="x1" inputChannel="#//@connectionChannels.0">
      <inputPointSet inputSignals="#//@aggregates.2/@inputs/@inputSignals.0
#//@aggregates.2/@inputs/@inputSignals.1"/>
    </inputPoints>
  </aggregates>
  <controlSequences name="Ksi"/>
  <controlSequences name="Niu"/>
  <connectionChannels name="Railway-Station"
destinationEnds="#//@aggregates.1/@inputPoints.0"
      sourceEnd="#//@aggregates.0/@outputPoints.0"/>
  <connectionChannels name="Station-Train"
destinationEnds="#//@aggregates.2/@inputPoints.0"
      sourceEnd="#//@aggregates.1/@outputPoints.0"/>
```

```
</AggregateSpecification>
```

10.3. CIM modelis

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2006 sp2 U (http://www.altova.com) by Audrius (502) -
->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="PLAMM"
targetNamespace="PLAMM" elementFormDefault="qualified">
           <xs:element name="AggregateSpecification">
                      <xs:complexType>
                                 <xs:sequence>
                                            <xs:element name="aggregates"
type="Aggregate" maxOccurs="unbounded"/>
                                            <xs:element name="controlSequences"
type="ControlSequence" maxOccurs="unbounded"/>
                                            <xs:element
name="connectionChannels" type="Channel" maxOccurs="unbounded"/>
                                 </xs:sequence>
                                 <xs:attribute name="name" type="xs:string"/>
                      </xs:complexType>
           </xs:element>
           <xs:complexType name="Signal">
                      <xs:attribute name="name" type="xs:string"/>
           </xs:complexType>
           <xs:complexType name="OutputSet">
                      <xs:sequence>
                                 <xs:element name="outputSignals"
type="Signal" maxOccurs="unbounded"/>
                      </xs:sequence>
                      <xs:attribute name="aggregate"/>
           </xs:complexType>
           <xs:complexType name="OutputPointSet">
                      <xs:attribute name="outputSignals"/>
                      <xs:attribute name="outputPoint"/>
           </xs:complexType>
           <xs:complexType name="OutputPoint">
                      <xs:sequence>
                                 <xs:element name="outputPointSet"
type="OutputPointSet" minOccurs="0" maxOccurs="unbounded"/>
                      </xs:sequence>
                      <xs:attribute name="name" type="xs:string"/>
                      <xs:attribute name="aggregate"/>
                      <xs:attribute name="channel"/>
           </xs:complexType>
           <xs:complexType name="InputPoint">
                      <xs:sequence>
                                 <xs:element name="inputPointSet"
type="InputPointSet"/>
                      </xs:sequence>
                      <xs:attribute name="name" type="xs:string"/>
                      <xs:attribute name="aggregate"/>
                      <xs:attribute name="inputChannel"/>
           </xs:complexType>
           <xs:complexType name="Channel">
                      <xs:attribute name="name" type="xs:string"/>
                      <xs:attribute name="destinationEnds"/>
                      <xs:attribute name="sourceEnd"/>
                      <xs:attribute name="aggregateSpecification"/>
           </xs:complexType>
```

```
            <xs:complexType name="InputSet">
                        <xs:sequence>
                                    <xs:element name="inputSignals" type="Signal"
maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="aggregate"/>
            </xs:complexType>
            <xs:complexType name="InputPointSet">
                        <xs:attribute name="inputSignals"/>
                        <xs:attribute name="inputPoint"/>
            </xs:complexType>
            <xs:complexType name="Operator">
                        <xs:attribute name="event"/>
                        <xs:attribute name="discreetCoordinates"/>
                        <xs:attribute name="continuousCoordinates"/>
                        <xs:attribute name="outputPoints"/>
                        <xs:attribute name="aggregate"/>
            </xs:complexType>
            <xs:complexType name="ExternalEvent">
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="eventSignal"/>
            </xs:complexType>
            <xs:complexType name="ExternalEventSet">
                        <xs:sequence>
                                    <xs:element name="externalEvents"
type="ExternalEvent" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="aggregate"/>
            </xs:complexType>
            <xs:complexType name="InternalEvent">
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="eventSignal"/>
                        <xs:attribute name="continuousCoordinate"/>
            </xs:complexType>
            <xs:complexType name="InternalEventSet">
                        <xs:sequence>
                                    <xs:element name="internalEvents"
type="InternalEvent" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="aggregate"/>
            </xs:complexType>
            <xs:complexType name="DiscreetComponent">
                        <xs:sequence>
                                    <xs:element name="discreetCoordinates"
type="DiscreetCoordinate"/>
                        </xs:sequence>
                        <xs:attribute name="aggregate"/>
            </xs:complexType>
            <xs:complexType name="DiscreetCoordinate">
                        <xs:attribute name="value" type="xs:string"/>
                        <xs:attribute name="type" type="xs:string"/>
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="discreetComponent"/>
            </xs:complexType>
            <xs:complexType name="ContinuousComponent">
                        <xs:sequence>
                                    <!--<xs:element name="continuousCoordinates"
type="ContinuousCoordinate" maxOccurs="unbounded"/>-->
```

```
                                    <xs:element name="continuousCoordinates"
type="ControlSum" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="aggregate"/>
            </xs:complexType>
            <xs:complexType name="ContinuousCoordinate">
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="internalEvent"/>
                        <xs:attribute name="continuousComponent"/>
            </xs:complexType>
            <xs:complexType name="ControlSum">
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="controlSumW" type="xs:double"/>
                        <xs:attribute name="internalEvent"/>
                        <xs:attribute name="continuousComponent"/>
                        <xs:attribute name="controlSequence"/>
            </xs:complexType>
            <xs:complexType name="ControlSequence">
                        <xs:sequence>
                                    <xs:element name="eventCounter"
type="EventCounter" minOccurs="0"/>
                        </xs:sequence>
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="aggregateSpecification"/>
            </xs:complexType>
            <xs:complexType name="EventCounter">
                        <xs:attribute name="value" type="xs:integer"/>
                        <xs:attribute name="internalEvent"/>
                        <xs:attribute name="controlSequence"/>
            </xs:complexType>
            <xs:complexType name="Aggregate">
                        <xs:sequence>
                                    <xs:element name="outputs" type="OutputSet"
minOccurs="0"/>
                                    <xs:element name="inputs" type="InputSet"
minOccurs="0"/>
                                    <xs:element name="operatorsGH"
type="Operator" minOccurs="0" maxOccurs="unbounded"/>
                                    <xs:element name="externalEvents"
type="ExternalEventSet" minOccurs="0"/>
                                    <xs:element name="internalEvents"
type="InternalEventSet" minOccurs="0"/>
                                    <xs:element name="discreetComponent"
type="DiscreetComponent" minOccurs="0"/>
                                    <xs:element name="continuousComponent"
type="ContinuousComponent" minOccurs="0"/>
                                    <xs:element name="outputPoints"
type="OutputPoint" minOccurs="0" maxOccurs="unbounded"/>
                                    <xs:element name="inputPoints"
type="InputPoint" minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="aggregateSpecification"/>
            </xs:complexType>
</xs:schema>
```

10.4. Sugeneruotas rezultatų modelis

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using PLASimulation;

namespace PLASimulationText
{
	class Railway : Aggregate
	{
		public OutputPoint y1_Train;
		private ControlSum trn;

		public Railway( string name ) : base( name )
		{
			y1_Train = new OutputPoint();
			trn = new ControlSum( HandleTrain );
		}

		public override void Initialize()
		{
			//TODO write code
		}

		public void HandleTrain( AgEventArg av, double tm )
		{
			//TODO write code
		}
	}
	class Station : Aggregate
	{
		public ExternalEventHandler x1_Train;
		public OutputPoint y1_StartPump;
		public OutputPoint y1_EndPump;
		private ControlSum acceptation;
		private ControlSum pumping;
		private int qn;

		public Station( string name ) : base( name )
		{
			x1_Train = HandleTrainArrived;
			y1_StartPump = new OutputPoint();
			y1_EndPump = new OutputPoint();
			acceptation = new ControlSum( HandleAccepted
);
			pumping = new ControlSum( HandlePumped );
		}

		public override void Initialize()
		{
			qn = 0;
			//TODO write code
		}

		public void HandleTrainArrived( AgEventArg ms, double tm
)
		{
			//TODO write code
		}

		public void HandleAccepted( AgEventArg av, double tm )
		{
```

```
                                        //TODO write code
                              }

                              public void HandlePumped( AgEventArg av, double tm )
                              {
                                        //TODO write code
                              }
                    }
                    class Train : Aggregate
                    {
                              public ExternalEventHandler x1_StartPump;
                              public ExternalEventHandler x1_EndPump;

                              public Train( string name ) : base( name )
                              {
                                        x1_StartPump = HandleStartedPump;
                                        x1_EndPump = HandleEndedPumping;
                              }

                              public override void Initialize()
                              {
                                        //TODO write code
                              }

                              public void HandleStartedPump( AgEventArg ms, double tm
)
                              {
                                        //TODO write code
                              }

                              public void HandleEndedPumping( AgEventArg ms, double tm
)
                              {
                                        //TODO write code
                              }
                    }
                    class OilTerminal : SimulationModel
                    {
                              privateRailway railway;
                              privateStation station;
                              privateTrain train;

                              public OilTerminal()
                              {
                                        railway = new Railway( "Railway" );
                                        station = new Station( "Station" );
                                        train = new Train( "Train" );
                                        railway.y1_Train.ConnectTo( station.x1_Train
);
                                        station.y1_StartPump.ConnectTo(
train.x1_StartPump );
                                        station.y1_EndPump.ConnectTo(
train.x1_EndPump );
                              }

                              public override void Initialize()
                              {
                                        base.Initialize();
                              }
```

```
        }
}
```

10.5. Straipsnis

IMITACINIO MODELIO AGREGATINĖS SPECIFIKACIJOS METAMODELIS

Gediminas Guginis, Audrius Pranckevičius

Kauno technologijos universitetas,

Programinės įrangos kūrimo procesas susiduria su produktyvumo sunkumais, kuriuos įtakoja naujų technologijų atsiradimas. OMG pateikta architektūra (MDA) leidžia sumažinti naujų technologijų įtaką programinės įrangos kūrimo procese. Vienu iš svarbiausių MDA esybių yra metamodelis, kuriuo remiantis galima specifikuoti MDA transformacijas. Straipsnyje pateikiamas imitacinio modelio agregatinės specifikacijos metamodelis, pagrįstas UML2 notacija.

1. Modeliais grįsta inžinerija

Nuolat besivystantis ir sudėtingėjantis programinės įrangos kūrimo procesas susiduria su naujomis kliūtimis ir uždaviniais. Tai, kas patikimai tarnavo prieš dešimt metų ir tikėtasi, jog tarnaus ir ateityje, nebepajėgia efektyviai prisiderinti prie technologijų pasirodymo, produktyvumo ir sistemų bendradarbiavimo pokyčių.

Nuolatiniai technologijų pokyčiai sukelia pernešamumo problemas, kurios dažniausiai pareikalauja žymių pastangų jas šalinant. Kaip pavyzdį būtų galima imti CORBA ir web servisų technologijas, kurios naudojamos tarpinės grandies sistemose. Web servisų technologijos paplitimas dalinai išstumia CORBA iš tarpinės grandiens sistemų nišos. CORBA technologiją naudojančių sistemų adaptavimas prie web servisų technologijos reikalauja papildomų pastangų. Tačiau sistemos funkcionalumas išlieka nepakitęs.

Produktyvumo problemas dabartiniame programinės įrangos kūrimo procese įtakoja ir tai, jog procesas remiasi žemo abstrakcijos lygmens projektavimu ir programavimu. Kodo palaikomumas stambių programinių sistemų atveju yra sudėtingas ir linkęs į klaidas procesas.

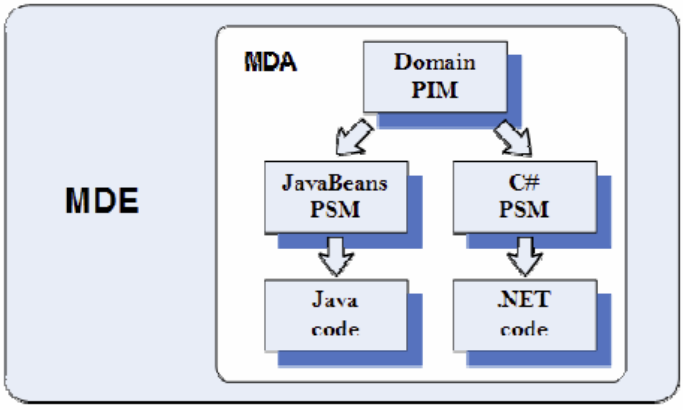
Stambių sistemų bendradarbiavimo problema kyla dėl jų nemonolitiškumo. Tokios sistemos paprastai būna modulinės. Skirtingi sistemos moduliai kuriami besiremiant tai probleminei sričiai labiausiai priimtina technologija.

Šios priežastys lėmė tai, jog esamas programinės įrangos sistemų kūrimo procesas turėjo būti peržiūrėtas ir pritaikytas naujoms realijoms. Iki šiol dominuojantis programinės įrangos

sistemų kūrimo procesas remiasi baziniu artefaktu – objektu, kaip geriausiai atspindinčiu esamos realybės ryšius ir charakteristikas. Pagrindiniu proceso peržiūrėjimo kriterijumi tapo proceso bazinio artefakto abstrakcijos lygmuo. Pasinaudojant laiko išbandytu metodu – abstrakcijos didinimu, naujuoju baziniu artefaktu deklaruojamas modelis. Abstrakcijos didinimo metodas pateisino save funkcinę paradigmą keičiant objektinę bei žemo lygio mašininį kodą funkcine.

Modelio kaip artefakto samprata nėra nauja – ji plačiai taikoma ir naudojama tiek teoriškai, tiek praktiškai. Modelio naudingumas bene geriausiai atsispindi inžinerinėse šakose. Retas kuris projektas pradedamas nagrinėti neturint projektinės dokumentacijos su apskaičiavimais, tam tikrų pasirinktų technologijų ar metodų pagrindimu. Sudėtingų projektų atveju gali būti atliekami bandymai su masteliniais modeliais, kurių modeliavimo rezultatai suteikia papildomos informacijos.

Modelis yra bazinis programinės įrangos kūrimo proceso, modeliais grįstos inžinerijos (toliau tekste nurodoma kaip MDE [1][2] – angl. model driven engineering) kontekste, artefaktas. Šis procesas MDE kontekste gali būti pateikiamas įvairiais pjūviais – reikalavimų surinkimo, analizės, projektavimo, realizavimo, palaikymo ir kitais aspektais. 2001 m. OMG (angl. Object Management Group) pateikta modeliais grįstos architektūros (toliau tekste nurodoma kaip MDA – angl. model driven architecture) [3][4] iniciatyva atvaizduoja MDE poaibį, susijusį su įvairaus abstrakcijos lygių modeliais, jų transformacijomis bei programinėmis platformomis. Žemiau esantis 1 paveikslas grafiškai anotuoja modeliais grįstą architektūrą MDE kontekste.



1 pav. *Modeliais grįsta architektūra (MDA) MDE kontekste*

Pagrindinis MDA tikslas yra pateikti architektūrą, kuri leistų neskausmingai adaptuotis prie naujų technologijų [5][6]. Tačiau MDA nenurodo, kokiomis priemonėmis ir kokiais aspektais remiantis pateikiamas dominamos probleminės srities modelis. Šis aspektas nagrinėjamas MDE erdvėje.

OMG pateikta MDA iniciatyva įveda dviejų abstrakcijos lygių modelius: nuo platformos nepriklausomus modelius (toliau tekste nurodomus kaip PIM – angl. platform independent model) ir nuo platformos priklausomus modelius (toliau tekste nurodomus kaip PSM – angl. platform specific model). PIM pateikia modelį, nesusietą su jokia konkrečia programine realizavimo platforma. PSM atvaizduoja su konkrečia programine platforma susietus modelius. Šio straipsnio tikslas yra pateikti agregatinių specifikacijų metamodelį, todel detaliau MDA nebus nagrinėjama. Plačiau apie tai galima susipažinti OMG MDA dokumente [7].

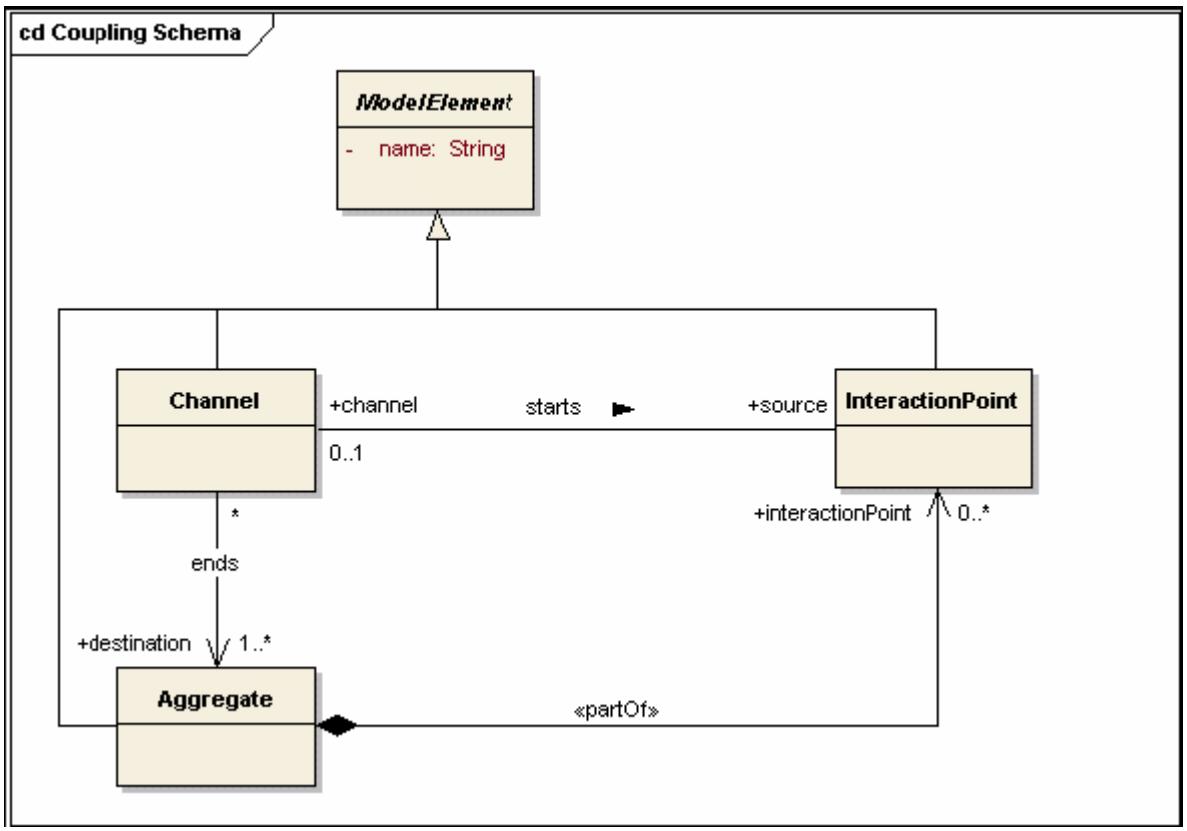
Tolimesnė straipsnio struktūra organizuota taip: 2 skyrius pateikia agregatinių specifikacijų metamodelį; 3 skyriuje supažindinama su programiniu imitacinio modelio agregatinės specifikacijos bibliotekos metamodeliu; modelio ir transformacijos pavyzdys aptariamas 4 skyriuje. Išvados ir tolimesnio darbo gairės pateiktos 5 skyriuje.

2. Agregatinės specifikacijos metamodelis

Išeities taškas imitacinio modelio generavimui yra modelio agregatinė specifikacija [11]. Toliau pateiksime imitacinio modelio agregatinės specifikacijos metamodelį, kuris yra būtinas transformacijoms specifikuoti. Metamodeliui specifikuoti naudosime UML2 [8][9][10] notaciją. Kai kurios metamodelio dalys tikslinamos viso skyriaus eigoje.

Modelio agregatinę specifikaciją sudaro dvi dalys [12]: agregatų sujungimo schema ir kiekvieno agregato specifikacija. Kiekviena iš šių dalių bus detaliau aptartos sekančiuose skyriuose.

Agregatų sujungimo schema aprašo struktūrinius sąryšius tarp modelio elementų. Ši schema detalizuojama CouplingSchema pakete, kurio sandara pateikiama žemiau esančiame 2 paveiksle.



2 pav. *Sujungimo schemos metamodelis*

Sujungimo schemos metamodelyje aprašytos šios klasės: agregatas, kanalas ir sąveikos taškas. Metamodelyje taip pat išskirta abstrakti klasė ModelElement. Sujungimo schemoje kanalo pradžia ir pabaiga papildomai neidenfikuojamos – sakoma, jog kanalas sujungia tam tikro agregato sąveikos tašką su tiksliniais agregatais. Kanalas jungia tik vieną sąveikos tašką su vienu ar daugiau agregatų. Sąveikos taškas su kanalu sujungtas su «partOf» stereotipu kompoziniu ryšiu. «partOf» stereotipas kompozicijos ryšyje reiškia esybės nuo kompozinių elementų nedalomumą ir visišką pastarųjų priklausomybę.

Agregato elementai aprašomi dviemis paketais: Aggregate Internals ir Aggregate Structure, kurie atitinkamai detalizuoja sudedamųjų vidinių komponenčių sąryšius ir agregato kompozitiškumą. Kiekvieną iš šių paketų aptarsime detaliau.

Agregato vidinę struktūrą aprašančių kompozicinių elementų metamodelis pateiktas pav. 3. Sudedamųjų elementų atvaizdavimui naudojamas kompozicinis ryšys, kuris detalizuojamas «partOf» stereotipu. Agregato kompoziciniai elementai:

− būsena;

− perėjimo operatorius H;

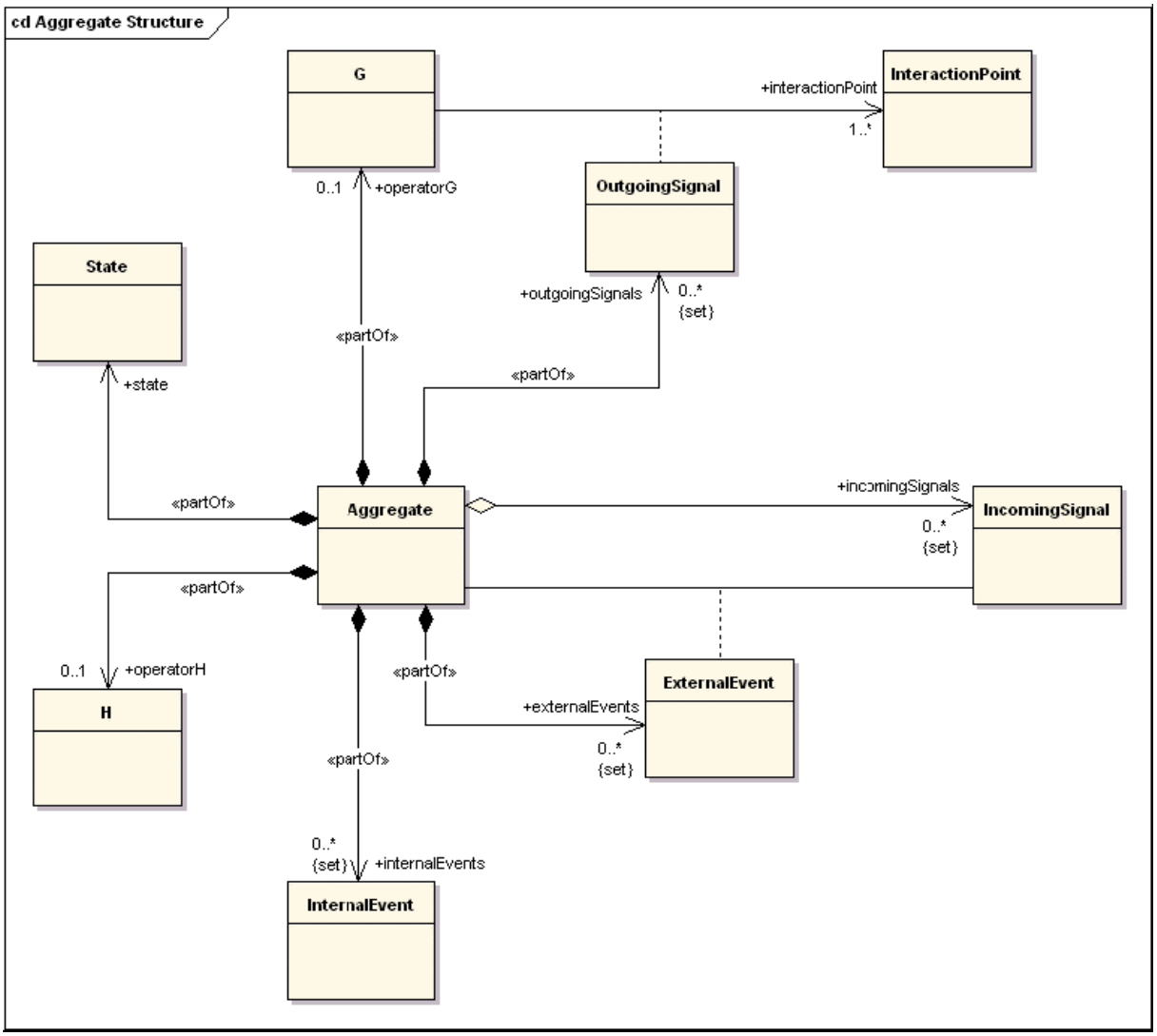
− išėjimo operatorius G;

− įėjimo signalų aibė;

− išėjimo signalų aibė;

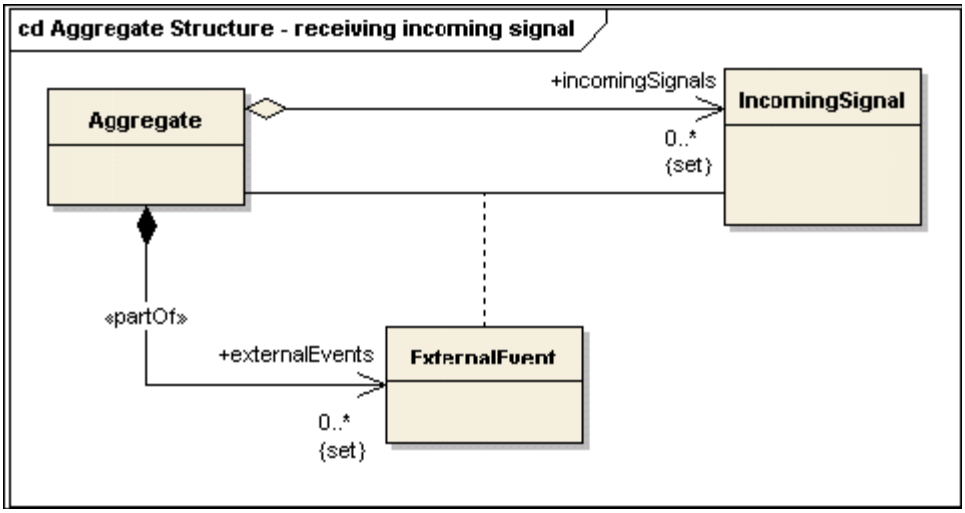
− išorinių įvykių aibė;

− vidinių įvykių aibė.



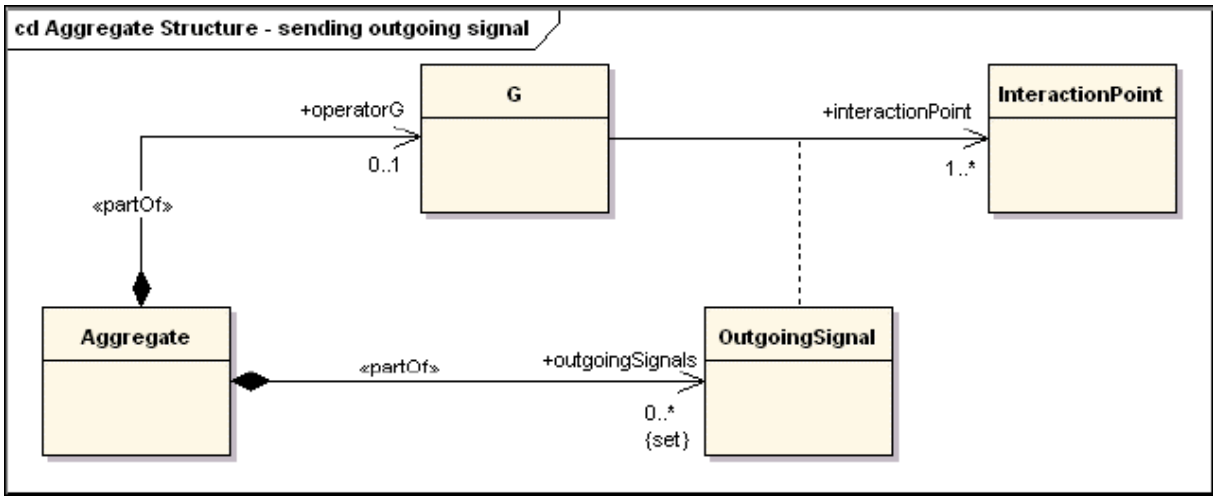
3 pav. Kompozicinių agregato elementų metamodelis

Įėjimo signalų aibė yra viena iš agregato kompozicinių elementų. Šioje aibėje išvardijami visi į agregatą patenkantys įėjimo signalai. Galimi variantai, kai agregatas neturi įėjimo signalų, t.y. įėjimo signalų aibė yra tuščia. Įėjimo signalo priėmimo faktas agregatate vadinamas išoriniu įvykiu, t.y. kiekvienam priimamo įėjimo signalo klasei egzistuoja išorinis įvykis. Įšorinių įvykių aibė irgi yra agregato kompozicinis elementas.



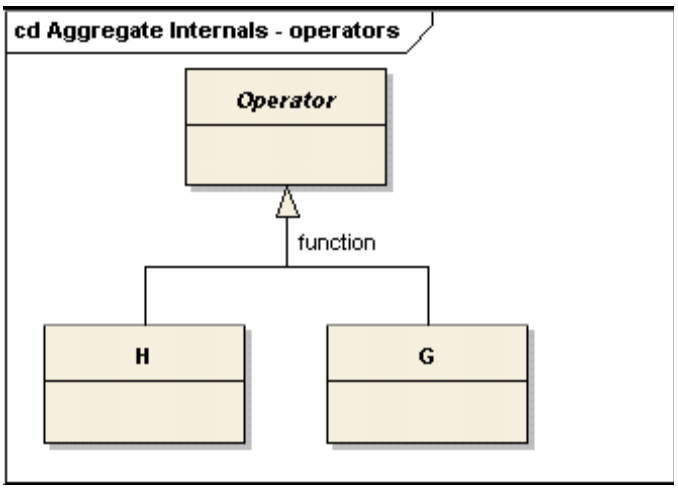
4 pav. *Įėjimo signalų ir išorinių įvykių aibės*

Išėjimo operatoriaus G sąryšį su sąveikos tašku charakterizuoja generuojamas išėjimo signalas. Agregato išėjimo signalų visuma sudaro išėjimo signalų aibę. Galimi variantai, jog neaprašomas išėjimo operatorius G, t.y. agregatas neišduoda jokių išėjimo signalų – tai implikuoja tuščia išėjimo signalų aibę (tokiu atveju išėjimo signalų aibė yra tuščia).



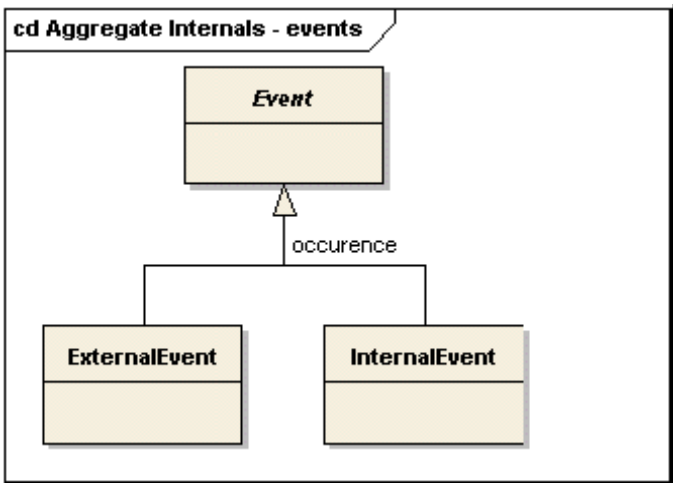
5 pav. *Išėjimo signalų aibė ir išėjimo operatorius G*

Agregate apdorojamų vidinių įvykių visuma sudaro vidinių įvykių aibę. Įvykiai gali būti apdorojami perėjimo (H) ir išėjimo (G) operatoriuose. Aprašyti šią savybę įvesta abstrakti klasė Operator, kuri inkapsuliuoja bendras G ir H operatorių savybes bei funkcionalumą.



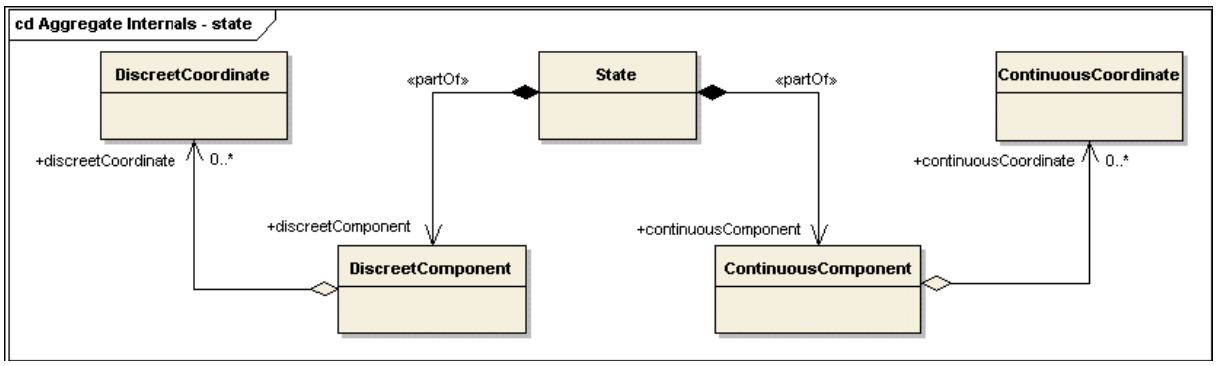
6 pav. *Operatorių klasifikacija*

Įvykiai agregatinėje specifikacijoje klasifikuojami į vidinius ir išorinius, priklausomai nuo stimulo atsiradimo vietos agregato atžvilgiu. Bendros vidinių ir išorinių įvykių charakteristikos ir funkcionalumas specifikuojamas abstrakčia Event klase.



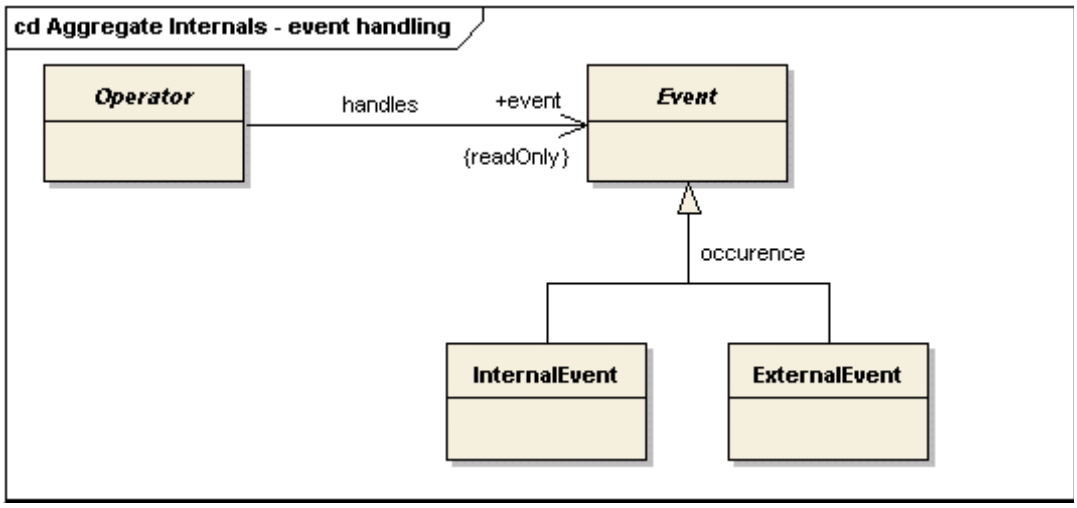
7 pav. *Įvykių klasifikacija*

Agregato būsena susideda iš dviejų komponenčių: diskrečiosios komponentės ir tolydžiosios komponentės. Tolydžioji ir diskrečioji komponentės agreguoja tolydžiąsias ir diskrečiąsias koordinates atitinkamai.



8 pav. *Agregato būsenos kompozicija*

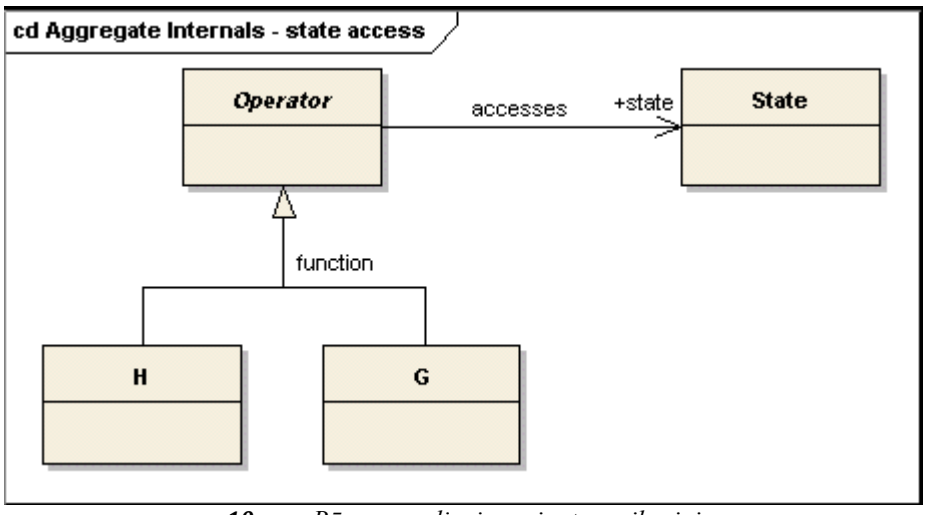
Agregato būsena gali kisti atėjus įėjimo signalui arba tolydžiajai koordinatei įgijus fiksuotą reikšmę. Įėjimo signalui atėjus į agregatą įvyksta išorinis įvykis (žr. įėjimo signalo aprašą), kuris gali būti apdorojamas perėjimo ir išėjimo operatoriuose. Analogiškai tolydžiajai koordinatei įgijus fiksuotą reikšmę, įvyksta vidinis įvykis, kuris bus apdorojamas perėjimo ir išėjimo operatoriuose.



9 pav. *Įvykių apdorojimas*

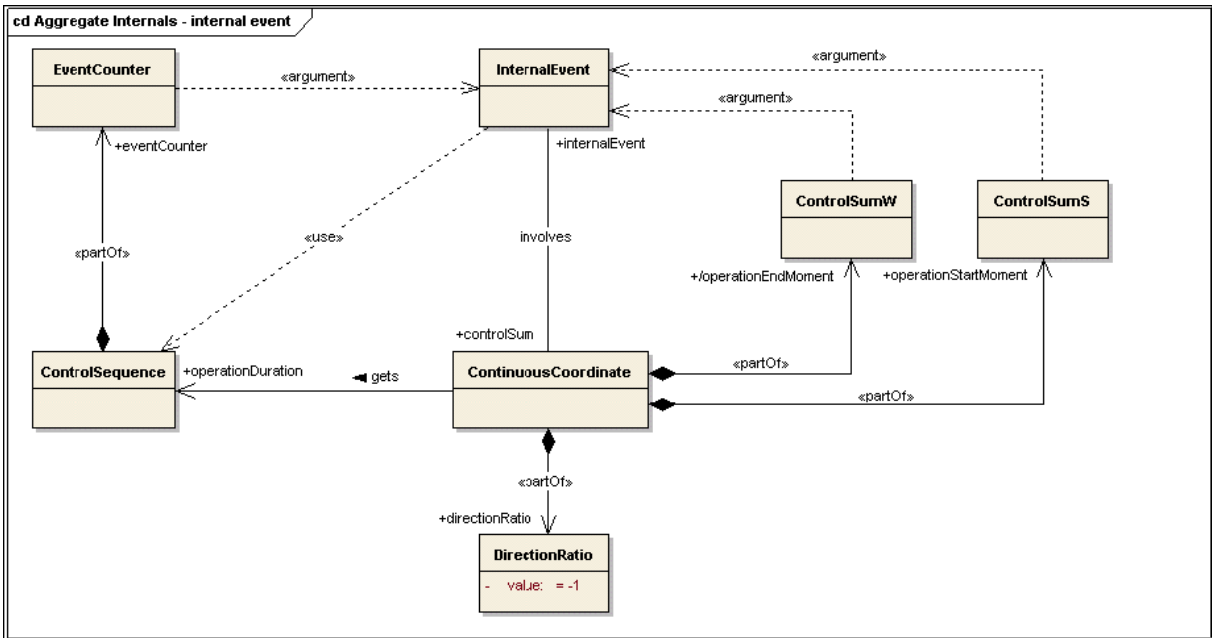
Pats įvykio apdorojimo procesas neįtakoja procesą sukėlusio įvykio. Todėl asociacija tarp Operator ir Event klasių nurodoma kaip readOnly, t.y. nekeičianti apdorojamo įvykio savybių.

Galime sakyti, jog agregato būseną evoliucionuojančiu faktoriumi gali būti bet kokios klasės įvykis. Įvykiai apdorojami operatoriuose, tad tik operatoriai gali evoliucionuoti agregato būseną. Šis faktas atvaizduotas žemiau pateiktame 10 paveiksle.



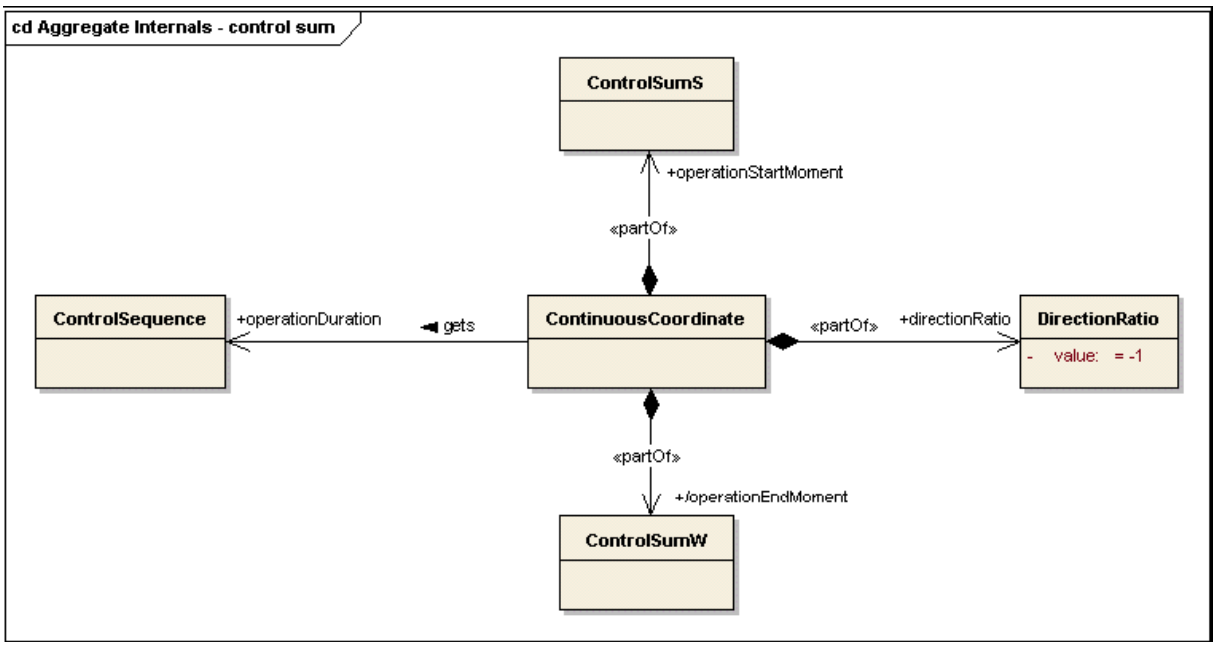
10 pav. *Būseną evoliucionuojantys veiksniai*

Taip pat galimi atvejai, jog įvykius apdorojančių operatorių algoritmas priklauso nuo būsenos momentinių parametrų, jų pačių nemodifikuojant.



11 pav. *Vidinis įvykis*

Su vidiniu įvykiu susijusi tolydinė koordinatė, kuri nurodo, kuriuo momentu baigsis įvykio apdorojimas ir pasikeis agregato būsena. Savo ruožtu tolydinė koordinatė kompoziciniu ryšiu susijusi su kontrolinėmis sumomis S ir W.



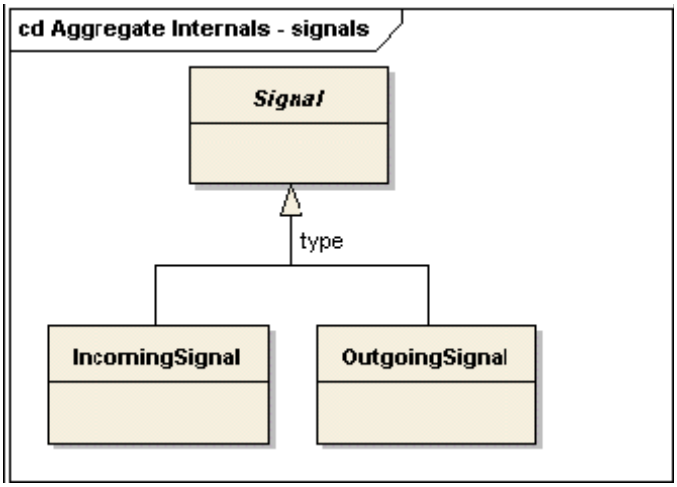
12 pav. *Tolydžioji koordinatė*

Kontrolinė suma S nurodo su vidiniu įvykiu susijusios operacijos (operacija asociuojama su vidinio įvykio apdorijimo operatoriuje procesu) pradžios momentą. Kontrolinė suma W nurodo su vidiniu įvykiu susijusios operacijos pabaigą, tačiau ji nėra apibrėžta, kol neprasideda su iššaukusiu vidiniu įvykiu susijusi operacija. Vidinį įvykį su ji charakterizuojančiomis kontrolinėmis sumomis S ir W jungia parametrinė priklausomybė, kuri atvaizduota priklausomybės ryšiu su «argument» stereotipu.

Vidiniu įvykiu iššauktos operacijos trukmę nusako atsitiktinio dydžio, kontrolinės sekos ControlSequence, reikšmė. Kiekvienas kreipinys į kontrolinę seką inkriminuoja kompozicinio sekos elemento - įvykio skaitliuko EventCounter, reikšmę. Ši reikšmė indeksuoja sekančio vidinio įvykio klasės kontrolinės sekos

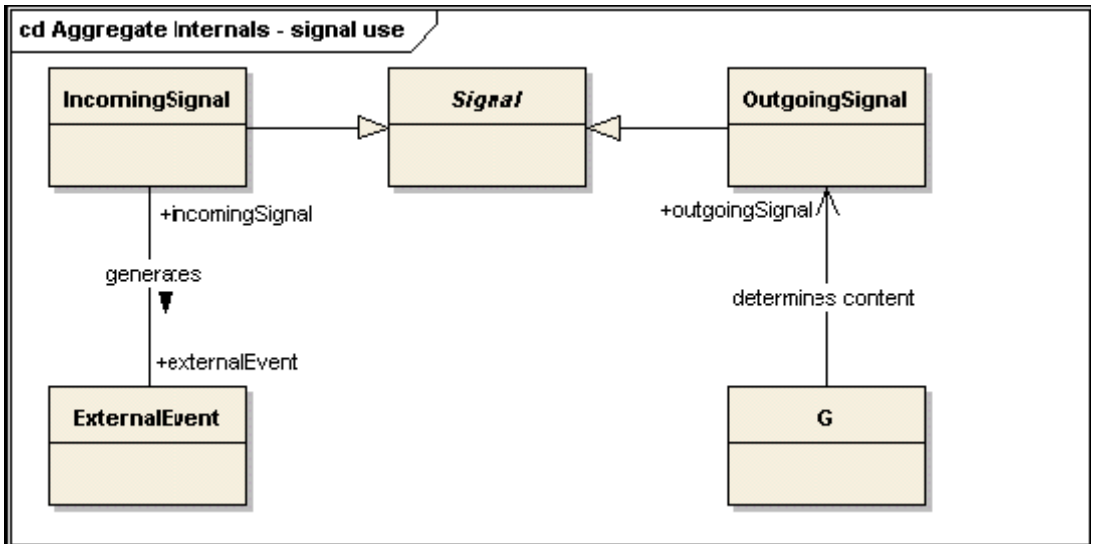
elemento reikšmę. Įvykių skaitliuką ir vidinį įvykį jungia parametrinės priklausomybės su «argument» stereotipu ryšys. Priklausomybės su «use» stereotipu ryšys tarp vidinio įvykio ir kontrolinės sekos yra formaliai apibrėžiamas agregatine specifikacija.

Tolydinės koordinatės krypties koeficientas nusako tolydinės koordinatės išvestinės laiko atžvilgiu reikšmę, t.y. koordinatės pokyčio greitį. Jei nespecifikuojama papildomai, taikoma -1 pagal nutylėjimą reikšmė.



13 pav. *Signalų klasifikacija*

Signalai agregatinėje specifikacijoje skirstomi į dvi rūšis: įėjimo ir išėjimo signalus. Įėjimo signalas yra toks signalas, kuris patenka į agregatą. Išėjimo signalu vadinamas signalas, kurio prasmingumas nusakoma išėjimo operatoriaus G. Išėjimo signalas savo gyvavimo kontekste atsivaizduoja į įėjimo signalą, nes kanalas kaip perdavimo terpė nemodifikuoja signalo prasmingumo. Kanalas išėjimo signalą gali perduoti daugeliui tikslinių agregatų, kur signalas aprašomas skirtingais, tačiau sinoniminiais pagal prasmingumą įėjimo signalais.



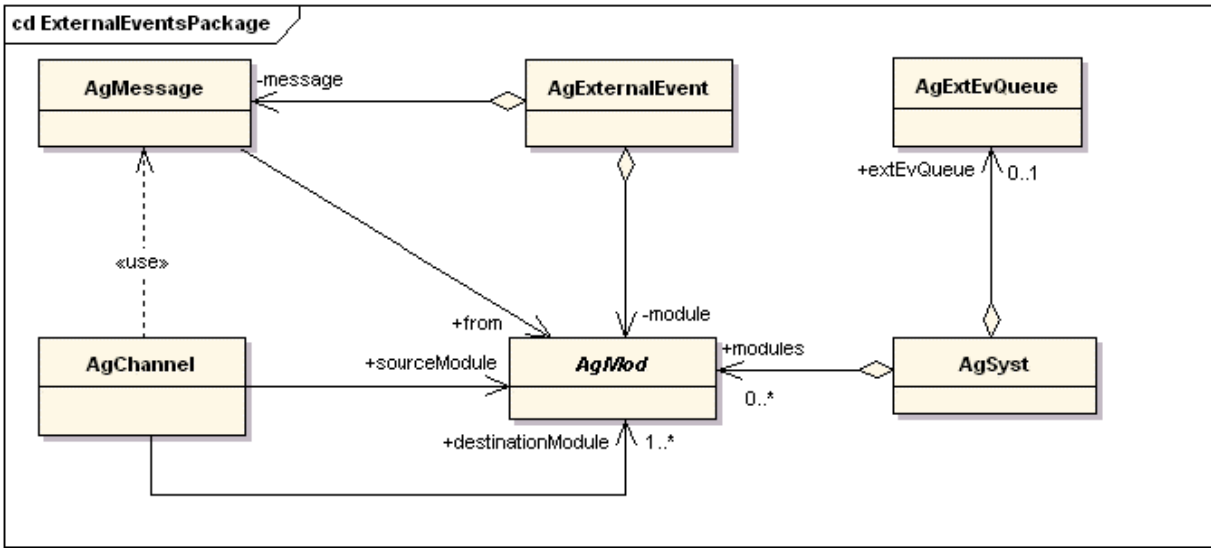
14 pav. *Signalų priėmimas ir siuntimas*

Kiekvienas įėjimo signalas iš agregato įėjimo signalų aibės atsivaizduoja į išorinį įvykį iš išorinių įvykių aibės ir atvirkščiai, t.y. egzistuoja tarpusavio 1:1 kardinalumo ryšys.

3. Bibliotekos metamodelis

Agregatinę sistemą sudaro tarpusavyje kanalais sujungti agregatai. Per šiuos kanalus agregatai perduoda pranešimus. Pranešimo priėmimas iššaukia agregato atitinkamą išorinį

įvykį. Agregato sujungimo į sistemą bei pranešimų perdavimo tarp agregatų funkcionalumo realizavimui bibliotekoje išskirtas išorinių įvykių paketas.



15 pav. *Išorinių įvykių paketas*

Paketą sudaro tokios klasės:

− AgExternalEvent – išorinio įvykio klasė;

− AgExtEvQueue – eilės, kurioje saugomi išoriniai įvykiai laukiantys savo apdorojimo, klasė;

− AgMessage – pranešimo perduodamo per kanalą bazinė klasė;

− AgMod – agregato bazinė klasė;

− AgSyst – agregatinės sistemos bazinė klasė;

− AgChannel – kanalo, kuriu perduodami pranešimai tarp agregatų, klasė.

Išorinių įvykių pakete realizuoti keturi agregatinės sistemos imitacinio modelio algoritmai:

− Agregatinių modulių sukūrimas ir sujungimas kanalais į uždarą agregatinę sistemą;

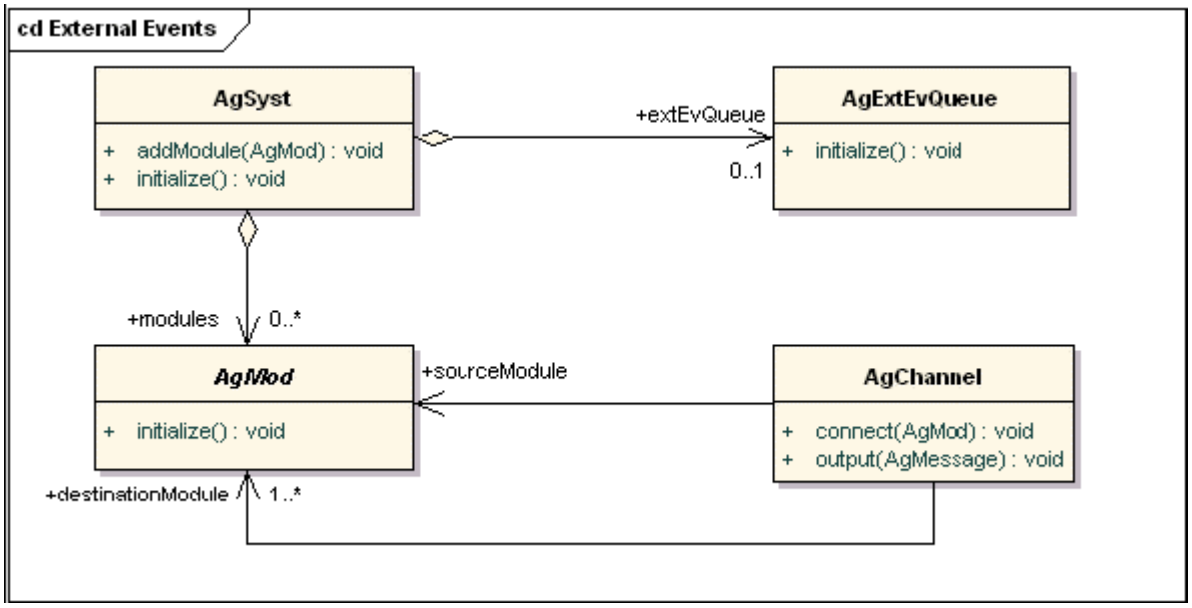
− Agregatinės sistemos pradinės būsenos nustatymas;

− Išorinio įvykio generavimas ir jo patalpinimas į agregatinės sistemos išorinių įvykių eilę;

− Išorinio įvykio išrinkimas ir jo perdavimas apdorojimui atitinkamame agregate.

3.1. Agregatinių modulių sukūrimas ir sujungimas kanalais

Agregatinės sistemos agregatų sujungimą realizuoja AgSyst klasė, talpinanti savyje agregatų sąrašą modules bei statinę išorinių įvykių eilię extEvQueue. Sąraše modules saugomi agregatinės sistemos agregatų objektai, kurių bazinė klasė yra AgMod. Sąrašas extEvQueue - tai statinė išorinių įvykių eilė, laukiančių savo apdorojimo atitinkamame agregato objekte.

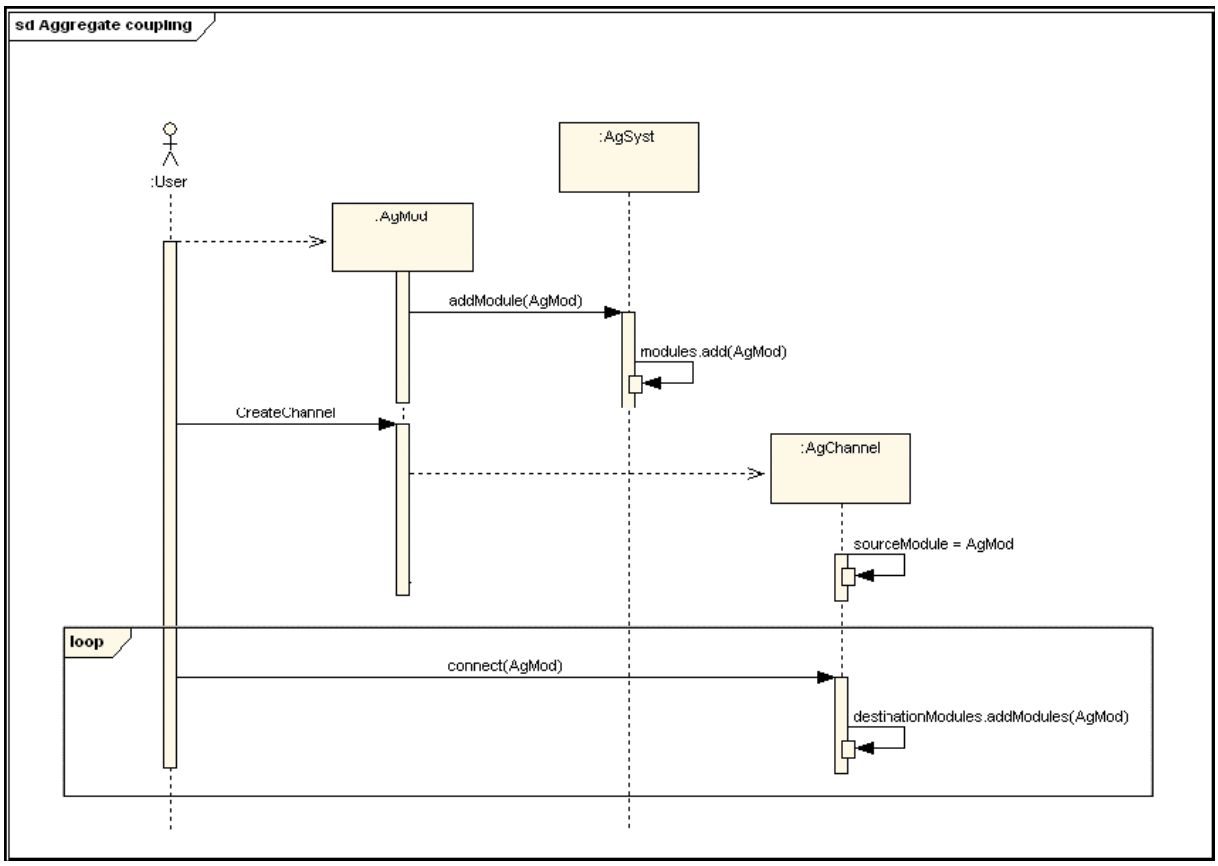


16 pav. *Išorinių įvykių klasių diagrama*

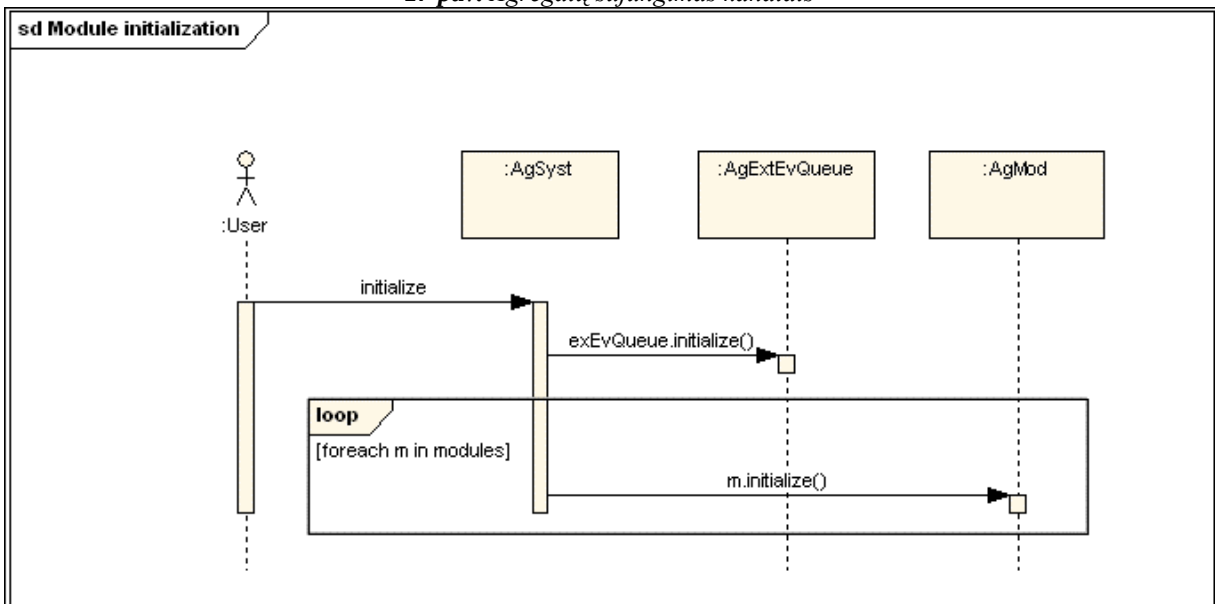
AgSyst klasės metodas addModule() leidžia prijungti į modules sąrašą naujai sukurtą agregatą (AgMod bazinės klasės objektą). Agregatų sąveika siunčiant pranešimus agregatų sujungimo kanalais realizuojama per klasės AgChannel objektus. Sukurdami kanalo objektą, konstruktoriui perduodame nuorodą į agregatinį modulį, kuris generuos pranešimus. Ši nuoroda išsaugoma kanalo klasės atribute sourceModule. Masyve destinationModule saugome nuorodas į modulius, kuriems bus perduodami pranešimai. Šis nustatymas atliekamas per klasės AgChannel metodą connect().

3.2. Agregatinės sistemos pradinės būsenos nustatymas

Agregatinės sistemos inicializacijai t.y. pradinės agregatinės sistemos būsenos nustatymui sukurtas AgSyst metodas initialize(). Šio metodo realizacija iškviečia visų sąrašo modules agregatų objektų metodus initialize() bei nustato išorinių įvykių eilę extEvQueue į pradinę būseną. Pagal nutylėjimą pradinė išorinių įvykių eilės būsena yra tuščia. Šis veiksmas atliekamas klasės AgExtEvQueue metodo initialize() pagalba.



17 pav. *Agregatų sujungimas kanalais*

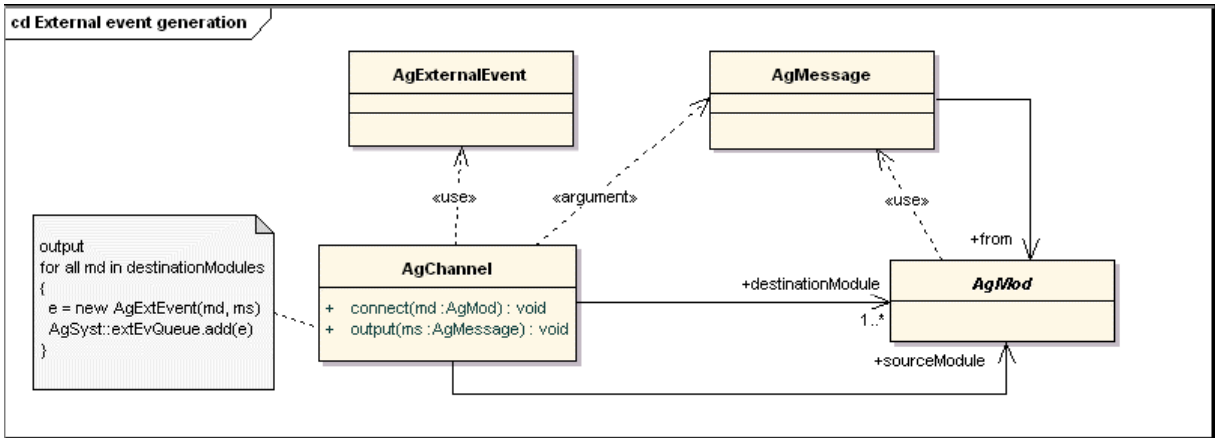


18 pav. *Agregatinių modulių inicializavimas*

3.3. Išorinio įvykio generavimas ir jo patalpinimas į agregatinės sistemos išorinių įvykių eilę

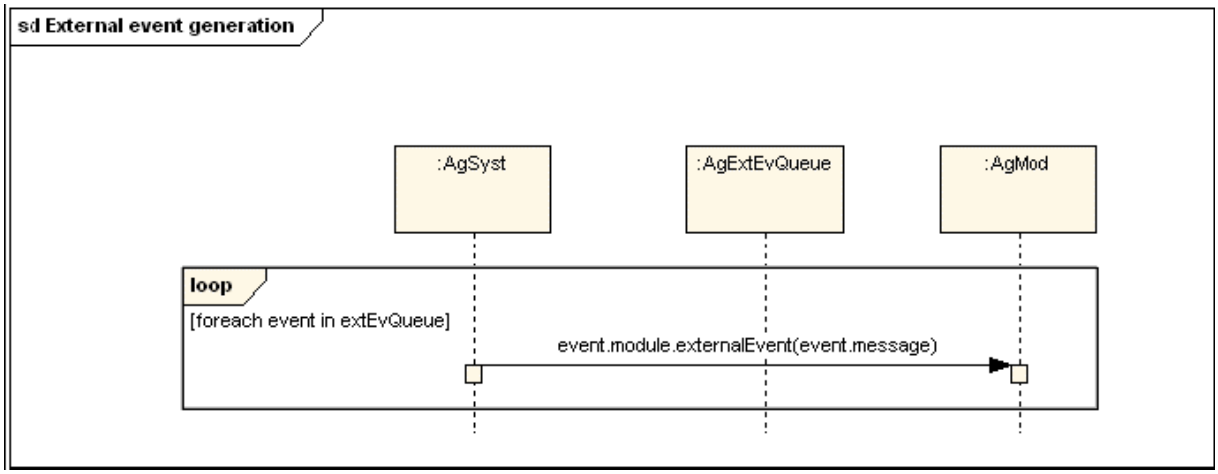
Išorinio įvykio generavimą inicijuoja agregatas-pranešimo siuntėjas, metodu output() kreipdamasis į atitinkamą kanalą. Siunčiamas pranešimas šiuo kanalu yra objektas, kurio bazinė klasė AgMessage. Kanalo AgChannel metode output() ši pranešimo struktūra yra įvelkama į naujai sugeneruotą išorinio įvykio AgExternalEvent tipo objektą. Tokių išorinių

įvykių sugeneruojama tiek, kiek yra saugomų nuorodų į agregatus klasės AgChannel masyve destinationModule. Visus tokius sugeneruotus išorinius įvykius sudedame ir saugome statinėje eilėje AgSyst::extEvQueue.



19 pav. *Išorinių įvykių generavimo klasių diagrama*

3.4. Išorinio įvykio išrinkimas ir jo perdavimas apdorojimui atitinkamame agregate

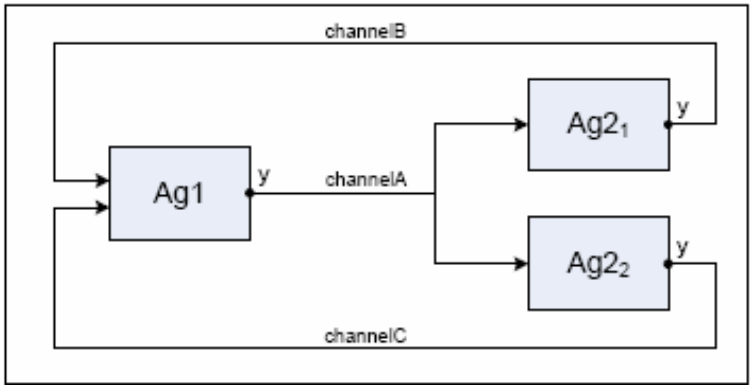


20 pav. *Išorinių įvykių generavimo sekų diagrama*

Eilinis išorinis įvykis išrenkamas išorinių įvykių eileje. Po to šis įvykis perduodamas apdoroti atitinkamame agregate, t.y. iškviečiamas atitinkamas išorinio įvykio perėjimo operatorius H. Tai kartojama tol, kol apdorojami visi išoriniai įvykiai.

4. Pavyzdys

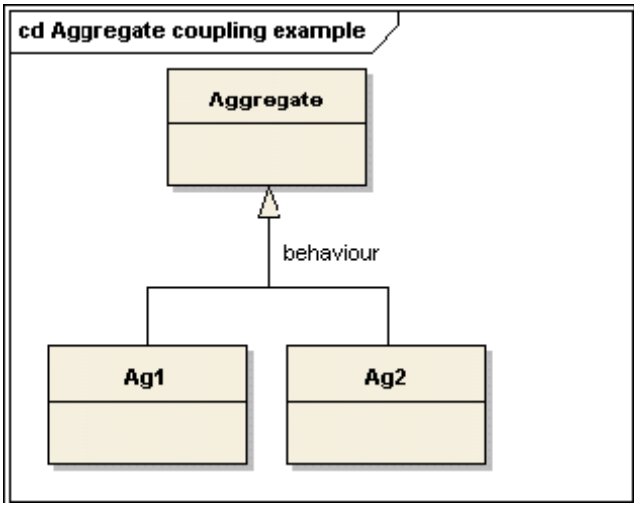
Šiame skyriuje pateiksime agregatų sujungimos schemos pavyzdį ir transformacijos ATL kalba [13][14] tekstą.



21 pav. *Agregatinio modelio sujungimo schemos pavyzdys*

Aukščiau pateiktame paveiksle pavaizduotas agregatinio modelio sujungimo schemos pavyzdys. Jį sudaro trys agregatai: vienas Ag1 tipo ir du Ag2 tipo ($Ag2_1$, $Ag2_2$). Ag1 su Ag2 agregatais jungia channelA kanalas, Ag2 su Ag1 jungia du kanalai: channelB ir channelC. Agregatai su kanalais sujungti per lokalius sąveikos taškus y – kiekvienas agregatas turi po vieną lokalų sąveikos tašką.

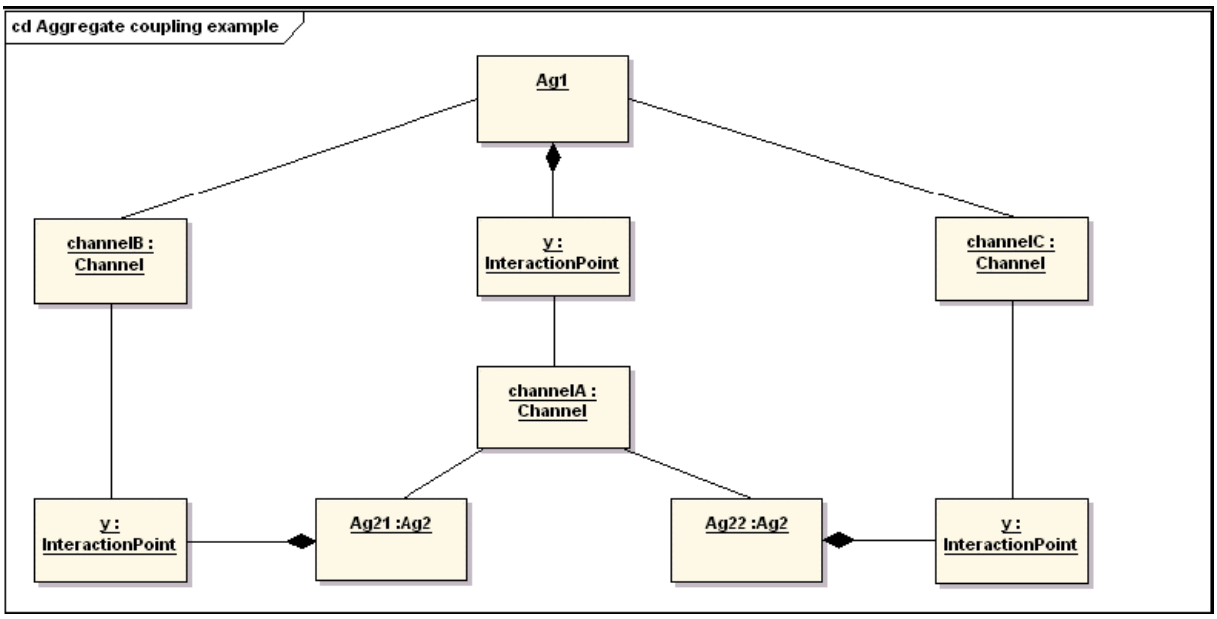
Pateiktą agregatinio modelio sujungimo schemą atitinka šios UML klasių ir objektų diagramos:



22 pav. *Sujungimo schemos agregatų tipai*

22 pav. ir 23 pav. atitinka agregatų sujungimo schemos PIM, nes neprisirišama prie jokios konkrečios platformos. PIM buvo sudarytas remiantis 2 skyriuje pristatytu imitacinio modelio agregatinės specifikacijos metamodeliu. Nuo platformos nepriklausomas modelis (PIM) buvo anotuotas UML2 klasių ir objektų diagramomis – tai sąlygota agregratinės specifikacija, kuria aprašomi agregatai (konkretūs objektai) arba agregatų struktūriniai sąryšiai.

Pateiksime ATL transformacijos fragmentą, kuris transformuoja agregatų bazinę klasę iš Aggregate (PIM) į AgMod (PSM).



23 pav. *Agregatų sujungimo schemos objektų diagrama*

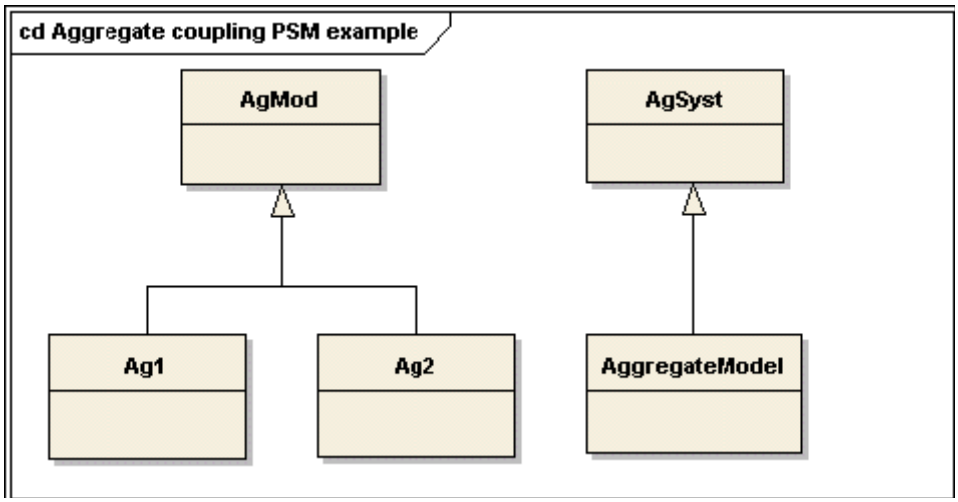
```
unique lazy rule AggregateOnceToAgMod(
     from
               source : AGMM!Aggregate (
                         source.name->isEqual( ‘Aggregate’ )
               )
     to
               target : AGLIBMM!AgMod (
               name = ‘AgMod’
               )
)
     rule AggregateToAgMod(
     from
               source : AGMM!Aggregate (
                         not source.name->isEqual( ‘Aggregate’ )
               )
     to
               target : AGLIBMM!AgMod (
                         name <- source.
                         name,parent <- source.parent
               )
```

)

24 pav. *Transformacijos fragmentas*

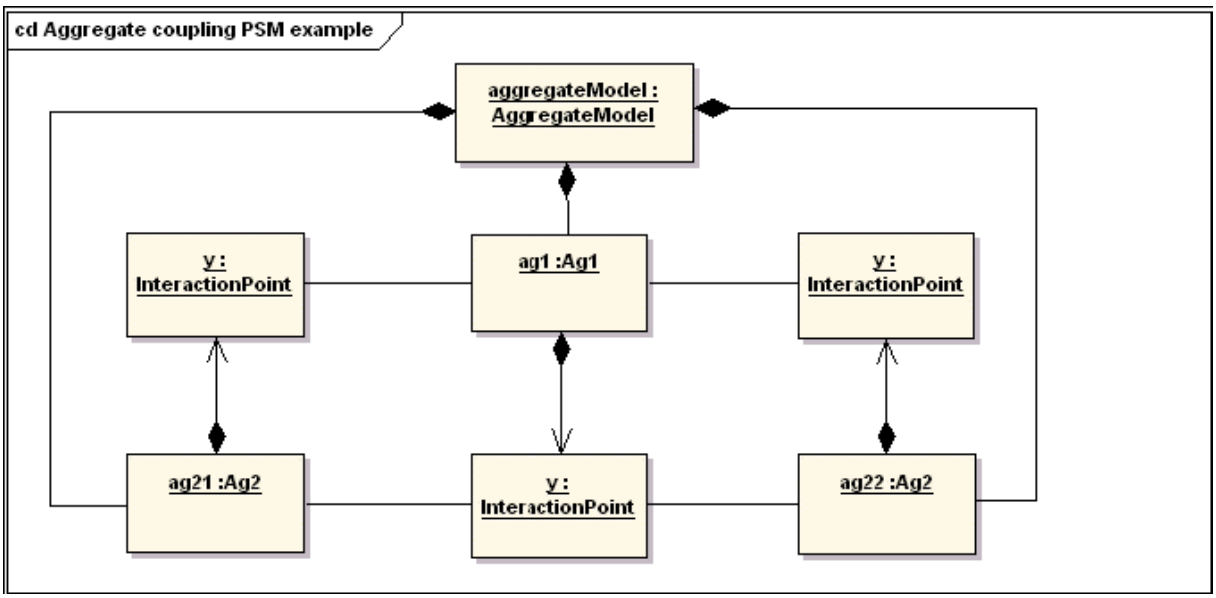
Transformacija susideda iš dviejų taisyklių: AggregateOnceToAgMod ir AggregateToAgMod. Pirmoji taisyklė skirta transformuoti Aggregate klasę į AgMod – pirmąkart vykdant taisyklę bus sukurta AgMod klasė, visais kitais vykdymo atvejais klasė nebebus kuriama, bus gražinama nuoroda į jau sukurtą AgMod klasę. T.y. ši taisyklė vykdoma tik kartą; ji neišreikštai iškviečiama iš AggregateToAgMod taisyklės, kai agregatų klasei priskiriama super klasė).

Transformacijos tekste AGMM atitinka imitacinio modelio agregatinę specifikacijos metamodelį (pristatytą 2 skyriuje), AGLIBMM – agregatinio modelio bibliotekos metamodelis (pristatytą 3 skyriuje). Po 24 pav. pateikto transformacijos fragmento gausime 25 pav. pateiktą vaizdą (



25 pav. *Sujungimo schemos agregatų tipai (PSM)*

Atlikus agregatų sujungimo schemos objektų diagramos pilną transformaciją (jos tekstas trumpumo dėlei nepateikiamas), gautume vaizdą, pateiktą 26 pav. 25 pav. ir 26 pav. atvaizduotas agregatų sujungimos schemos pavyzdžio PSM – t.y. nuo konkrečios programinės platformos (mūsų atveju nuo bibliotekos) priklausomas modelis.



26 pav. *Agregatų sujungimo schemos objektų diagrama (PSM)*

5. Išvados

Metamodelis, kuriuo remiantis sudaromas modelis yra viena iš būtinų aspektų, norint apibrėžti MDA transformacijos. Be metamodelio transformacijos, atitinkančios MDA viziją, nėra įmanomos.

Straipsnyje pateikiamas imitacinio modelio agregatinės specifikacijos metamodelis. Metamodeliui specifikuoti naudojama UML2 notacija (klasių diagramos). MDA transformacijai aprašyti yra naudojama ATL kalba, turinti tiek deklaratyvinės, tiek imperatyvinės kalbos bruožų.

Tolimesniame tiriamajame darbe didžiausias dėmesys bus telkiamas ties ATL transformacijomis. Taipogi bus detalizuojamas pateiktas agregatinės specifikacijos metamodelis, jį papildant OCL [15] detalėmis.

Literatūra

[1] F.Fondement, R.Silaghi, „Defining Model Driven Engineering Processes", Third International Workshop in Software Model Engineering (WiSME@UML 2004), Lisbon, Portugalija, 2004

[2] M.Alanen, J.Lilius, I.Porres, D.Truscan, „Model Driven Engineering: A Position Paper", proceedings of the 1st International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'04), Hamilton, Ontario, Kanada, 2004

[3] S.J. Mellor, K.Scott, A.Uhl, D.Weise, „MDA Distilled", Addison-Wesley, 2003, ISBN 0201788918, 176 psl.

[4] „Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach Productivity Analysis", prieiga per internetą http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf, žiūrėta 2005-12-16

[5] M.Staron, L.Kuzniarz, L.Wallin, „A Case Study on a Transformation Focused Industrial MDA Realization", Third International Workshop in Software Model Engineering (WiSME@UML 2004), Lisbon, Portugalija, 2004

[6] D.Simmonds, S.Ghosh, R.France, „Middleware Transparent Software Development & the MDA“, WiSME@UML 2003, San Francisko, JAV, 2003
[7] Object Management Group, „MDA Guide“, prieiga per internetą http://www.omg.org/cgi-bin/doc?omg/03-06-01, žiūrėta 2005-12-15
[8] Object Management Group, „UML 2.0 Infrastructure”, prieiga per internetą http://www.omg.org/docs/ptc/04-10-14.pdf, žiūrėta 2005-12-16
[9] Object Management Group, „UML 2.0 Superstructure”, prieiga per internetą http://www.omg.org/cgi-bin/doc?formal/05-07-04, žiūrėta 2005-12-16
[10] T. Pender, „UML Bible”, Wiley, 1-asis leidimas, 2003, ISBN 0764526049, 984 psl.
[11] H.Pranevičius, „Formal specification and analyzsis of distributed systems“, Lecturer Notes of the Nordic-Baltic Summer School on Application of AI to Production, Technologija, Kaunas, ISBN 9986-13-564-8, psl. 269-322
[12] G.Guginis, „Software simulation model creation technique based on PLA use“, proceedings of MOSIBUS 2003, Vilnius, Lietuva, 2003, psl. 185-189
[13] F.Jouault, I.Kurtev, „On the Architectural Alignment of ATL and QVT“, proceedings of ACM Symposium on Applied Computing (SAC 06), Dijon, Bourgogne, Prancūzija, 2006
[14] F.Jouault, I.Kurtev „Transforming Models with ATL“, proceedings of the Model Transformations in Practice, MoDELS 2005, Montego Bay, Jamaika, 2005
[15] J.Warmer, A.Kleppe, „The Object Constraint Language: Getting Your Models Ready for MDA“, Addison-Wesley, 2-asis leidimas, 2003, ISBN 0321179366, 240 psl.

Metamodel of Aggregate Specification for Simulation

Software development process encounter productivity obstacles, which are affected by an arise of new technologies. Object Management Group (OMG) proposed model driven architecture (MDA) supposed to minimize effect of new technologies arrival into a process of software development. One of the most important aspects in MDA is metamodel, which is used to specify MDA style transformations. Article presents metamodel of aggregate specification for simulation, based on UML2 notation. Example of agregate coupling schema is presented as well.