KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Darius Munčys

Agregatinių imitacinių modelių programinio kodo generavimas ir integravimas su duomenų baze

Magistro darbas

Darbo vadovas
doc. V.Pilkauskas

Kaunas, 2007

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
VERSLO INFORMATIKOS KATEDRA

Darius Munčys

Agregatinių imitacinių modelių programinio kodo generavimas ir integravimas su duomenų baze

Magistro darbas

Recenzentas
prof. dr. L. Nemuraitė
2007-05-28

Darbo vadovas
doc. V.Pilkauskas
2007-05-28

Atliko
IFM-1/1 gr. stud.
Darius Munčys
2007-05-24

Kaunas, 2007

Turinys

PAVEIKSLAI

LENTELĖS

Source code generation for aggregate simulation models and their integration with database

SUMMARY

This paper analyses MDE approach to software engineering as well as OMG's MDA and Microsoft DSL Tools.

MS DSL Tools are a powerful addition to MS Visual Studio 2005 for creating your own Domain Specific Language metamodels and generating IDEs for working with it. Then you can make a text template to transform your model directly into code.

The work describes developed aggregate modeling language metamodel. It also explains text template language for model transformations.

The research also includes analysis of the possibility to use model transformations to generate code for data acquisition form database.

1. Įvadas

Per keletą pastarųjų dešimtmečių kompiuterinė technika vystėsi gana greitai. Techninės įrangos vystymąsi yra gana lengva išreikšti skaičiais – tranzistorių dydis, operacijų per sekundę kiekis ir t.t. Sekant programinės įrangos raidą, skaičių rasti yra gerokai sunkiau. Taip yra todėl, kad programinės įrangos progresas dažnai pasižymi ne kiekybiniais o kokybiniais rodikliais. Atsiranda naujos programų kūrimo technologijos ir metodologijos. Pasirodo naujos vis aukštesnio lygio programavimo kalbos. Tobulinimas vyksta įvairiomis kryptimis: objektiškai orientuoti sprendimai leidžia sukurti vis sudėtingesnes sistemas, lygiagretūs sprendimai leidžia pasiekti vis didesnį našumą.

Kuriant sudėtingas sistemas atsiranda poreikis planuoti. Sistemos pradedamos projektuoti gerokai prieš realizacijos pradžią. Tai natūralu, nes pavyzdžiui, kalkuliatoriaus programą, atliekančią keturis pagrindinius aritmetinius veiksmus, sukurti yra žymiai lengviau, nei verslo valdymo sistemą.

Projektavimas dažnai vykdomas pasitelkiant kurią nors modeliavimo kalbą. Kuriami įvairių lygių modeliai. Kai kurie iš jų vaizduoja stambiaplanę sistemos struktūrą, kai tuo tarpu kiti gali vaizduoti smulkią tam tikro modelio veikimo schemą.

Paprastai modeliai naudojami tik ankstyvoje sistemos kūrimo stadijoje, kūrimui įsibėgėjus jie padedami į šalį ir užmirštami. Kompanijos OMG pasiūlyta architektūra, vadinama MDA (Model Driven Architecture), parodo kitokį požiūrį į modelių naudojimą. Modeliai gali būti naudojami ne tik kaip pradinė sistemos dokumentacija, bet ir kaip pagrindinis sistemos kūrimo artefaktas. Tai pasiekiama įvairaus abstraktumo modelius transformuojant į kitus modelius. Pavyzdžiui, modelis, vaizduojantis bendrą ir nuo platformos nepriklausomą architektūrą, gali būti transformuotas į tam tikrą platformą atitinkantį modelį, panaudojant konkrečią transformaciją. Taip pat galima modelį transformuoti į tam tikra kalba parašytą kodą, tuo gaunant baigtą ir paruoštą diegimui sistemą.

Šio darbo tikslas – ištirti agregatinio imitacinio modelio kodo generavimo galimybes.

2. Modeliais paremta inžinerija (MDE)

Modeliais paremta inžinerija – tai sistematiškas modelių, kaip pagrindinių inžinerijos artefaktų, naudojimas inžinerijos cikle.

Kai kalbama apie programų inžineriją, MDE reiškia aibę kūrimo metodų, naudojant programų modeliavimą, kaip pagrindinę išraiškos priemonę. Kartais modeliuojama iki tam tikro detalumo lygio, o vėlesniame etape kodas rašomas rankomis, o kartais kuriami pilni modeliai su vykdomaisiais veiksmais. Iš modelių gali būti generuojamas programų kodas – nuo sistemos griaučių iki užbaigto ir paruošto diegimui produkto.

MDE technologijos, labiau nukreiptos į architektūrą ir automatizavimą, programų inžinerijoje duoda didesnį abstrakcijos lygį. Toks abstrakcijos lygis skatina kurti paprastesnius modelius, skiriančius daugiau dėmesio probleminei sričiai. Kartu su vykdoma semantika, tai padidina galimą automatizavimo lygį.

Kompanija „Object Management Group“ (OMG) sukūrė standartą, pavadintą „Model Driven Architecture“ (MDA), tapusį pagrindu šiam į architektūrą nukreiptam metodui.

Kompanija „Microsoft“ taip pat kuria metodologijas ir įrankius, skirtus MDE panaudojimui.

2.1. Sričiai pritaikytos kalbos (DSL)

Naudojantis MDE dažnai naudojamos sričiai pritaikytos kalbos.

2.1.1. DSL kaip programavimo kalba

Sričiai pritaikytos kalbos gali būti laikomos programavimo kalbomis, dedikuotomis konkrečiai sričiai ar problemai. Ji savyje turi atitinkamas abstrakcijas ir žymėjimus, paprastai būna maža, labiau deklaratyvi nei imperatyvi, pigesnė už bendro naudojimo kalbas (GPL).

Pavyzdžiui, Unix aplinka (shell) gali būti laikoma DSL, tarp kurios srities abstrakcijų ir žymėjimų yra srovės (streams) kaip stdin ir stdout bei veiksmai su srovėmis – persiuntimas ir kanalas (redirection, pipe). Aplinkos paprastai suteikia paprastus interfeisus procesų paleidimui ir kontrolei. Jos turi paprastus kontrolės perdavimo ir manipuliavimo string tipo reikšmėmis mechanizmus, kurie patenkina daugelį dažniausių panaudojimo poreikių. Nors aplinkos ir yra išbaigtos Turingo

atžvilgiu, jos skiriasi nuo bendro naudojimo kalbų tuo, kad negali manipuliuoti sudėtingesniais duomenimis – pvz. struktūromis.

Sričiai pritaikytos kalbos dar vadinamos mikrokalbomis, pritaikymo kalbomis ir labai aukšto lygio kalbomis.

2.1.2. DSL kaip specifikavimo kalba

Kadangi DSL gali būti gana deklaratyvios ir slėpti didžiąją dalį realizacijos detalių, kai kurios iš jų gali būti laikomos labiau specifikavimo nei programavimo kalbomis. Tačiau tos specifikacijos dažnai gali būti vykdomosios. Pagrindinėmis charakteristikomis išlieka tam tikros abstrakcijos ir žymėjimai bei apribota (arba tiksliau sufokusuota) išreiškiamoji galia.

Pavyzdžiui Unix komanda *make* yra priemonė palaikyti programas: ji nusprendžia kurias didelės programos dalis reikia perkompiliuoti ir duoda atitinkamas komandas tai padaryti. *Makefile* kalba yra maža ir didžioji jos dalis yra deklaratyvi, nors yra ir imperatyvių konstrukcijų. Jos išreiškiamoji galia apsiriboja užduočių priklausomybių atnaujinimu – pačio perkompiliavimo veiksmai deleguojami aplinkai. Ji paslepia tokias realizacijos detales, kaip paskutinio modifikavimo laikas, ir suteikia tokias srities abstrakcijas, kaip failų priesagos ir kompiliavimo taisyklės. Tuo pasinaudodamas vartotojas gali glaustai ir tiksliai išreikšti atnaujinimo priklausomybes.

2.1.3. DSL privalumai

Daugumai panaudojimo sričių DSL yra patrauklesnės nei GPL (pavyzdžiui aplinkų ir makefile kalbos).

Paprastesnis programavimas:

Dėl atitinkamų abstrakcijų, žymėjimų ir deklaratyvių formuluočių, DSL programa yra mažesnė ir lengviau skaitoma nei jos GPL atitikmuo. Iš to seka trumpesnis kūrimo laikas ir geresnis palaikymas. Kadangi programavimas yra sukoncentruotas į tai, ką skaičiuoti, o ne kaip skaičiuoti, vartotojas neprivalo būti įgudęs programuotojas. Pavyzdžiui, perkompiliavimo atveju, programos, tiesiogiai tikrinančios visų failų modifikacijos laikus ir palaipsniui perkompiliuojančios sistemą, rašymas tikrai būtų ilgas, varginantis ir linkęs į klaidas, palyginus su makefile naudojimu.

Sistematiškas pakartotinis panaudojimas

Dauguma GPL programavimo aplinkų turi galimybę sugrupuoti panašias operacijas į bibliotekas. Nors kai kurios bibliotekos yra standartinės, pakartotinis panaudojimas paliekamas programuotojui. Kitą vertus DSL siūlo gaires ir ir integruotą funkcionalumą, kuris verčia pakartotinai panaudoti. Be to DSL saugo savyje informacija apie sritį: netiesiogiai, paslėpdama dažnai naudojamus šablonus DSL realizacijoje, arba tiesiogiai, duodama programuotojui priėjimą prie tam tikros parametrizacijos. Taigi bet kuris vartotojas privalės pakartotinai panaudoti bibliotekų komponentus ir srities žinias.

Lengvesnis patikrinimas

Dar vienas DSL‘ų privalumas yra tas, kad jos leidžia patikrinti daugiau programos savybių. Priešingai nei GPL, DSL semantika gali būti apribota pagal kritines sričiai savybes. Pavyzdžiui *make* praneša apie bet kokius ciklus priklausomybėse, taip visiškai išvengiant nesibaigiamumo (darant prielaidą kad atskiri veiksmai nėra cikle).

Nors visos išvardintos DSL savybės sprendžia svarbias programų inžinerijos problemas, jos mažai ką pasako apie programų, parašytų DSL, struktūrą. Tiesą sakant DSL primygtinai rekomenduoja konkrečias programines architektūras.

2.1.4. DSL programų architektūra

Programų architektūra išreiškia kaip sistemos turėtų būti sudarytos iš atskirų komponentų, ir kaip tie komponentai turėtų bendrauti. Iš programų architektūros pusės DSL gali būti matoma kaip parametrizacijos mechanizmas arba sąsajos modelis.

Parametrizavimo mechanizmas

Programa arba biblioteka gali būti daugiau arba mažiau bendra, priklausomai nuo jos sprendžiamų problemų. Pavyzdžiui mokslinė biblioteka gali būti laikoma labai bendra dėl didelio jos sprendžiamų problemų skaičiaus. Toliau plečiant bendrumo sąvoką gauname sudėtingus parametrus, kurie gali būti laikomi DSL. Pavyzdžiui funkcijos *printf* formato argumentas gali būti laikomas ir sudėtingu parametru, ir labai paprasta DSL. DSL programos laikymas sudėtingu argumentu labai parametrizuotam komponentui gali skambėti perdėtai, bet tai iš tikrųjų yra galutinis grandininio išreiškiamosios galios didėjimo naudojant parametrizaciją žingsnis. Ši situacija gali būti iliustruota Unix komandomis *grep, find, sort, sed, make, awk* ir t.t. bei progresavimu nuo paprastų komandinės eilutės parametrų iki programų failų. Galų gale

duomenų parametru tampa programa, kurią reikia vykdyti, taip pasiekiant dar didesnę parametrizavimo galią.

Bibliotekos sąsaja

Bibliotekai didėjant arba tampant bendresne, jos panaudojamumas mažėja dėl pateikiamų įėjimo taškų, parametrų ir parinkčių skaičiaus didėjimo. Dėl to programuotojai gali ignoruoti biblioteką – ji pasidaro per sudėtinga naudoti. Šioje situacijoje DSL gali pasiūlyti sričiai pritaikytą bibliotekos sąsają taip, kad programuotojas negali tiesiogiai manipuliuoti daugeliu iš labai parametrizuotų bibliotekos sudedamųjų dalių – sudėtingumas paslepiamas. Kita dažnai susidaranti situacija yra ta, kad kai kurie bibliotekos metodų rinkiniai kviečiami dažnai. Šiuo atveju DSL gali leisti tiesioginį priėjimą prie šių kombinacijų. Pavyzdžiui Unix aplinkos yra sąsajos standartinėms Unix bibliotekoms. Panašiai SQL paslepia žemo lygio kreipinius į duomenų bazę. Ši idėja taip pat naudojama skript kalbų, kurios sujungia galingų su tradicinėmis kalbomis parašytų komponentų grupę. Pavyzdžiui Tcl/Tk leidžia naudoti Tcl sąsają Tk grafiniame įrankių rinkinyje.

DSL pripažinimas parametrizavimo mechanizmu ir sąsaja įtakoja mastymą apie programinę įrangą ir jos struktūrą. DSL apibrėžia programos adaptyvumą. Tokia programinė įranga skaidoma į 2 dalis: DSL išreikštos parametrizacijos interpretavimą ir komponentų biblioteką.

Sudėtingi parametrai atsiranda, kai vietoj atskirų, bet susijusių įrankių grupės, sukuriama viena bet įvairiapusiška programa. Bibliotekos sukuriamos tam, kad būtų galima pakartotinai panaudoti duomenų tipus ir bazines operacijas susijusiose programose. Šis pastebėjimas veda prie kito, labiau koncepcinio DSL aspekto: programų šeimos.

Programų šeima

DSL programa yra programų šeimos narė. Programų šeima yra tokių programų rinkinys, kurios turi pakankamai bendro, kad vertėtų jas analizuoti kaip visumą. Programų šeimą galima apibrėžti ir kaip problemų šeimos (susijusių problemų) sprendimui skirtą programų grupę. Tam tikro įtaiso tipo draiveriai yra programų šeimos pavyzdys: neskaitant to, kad jie turi tą patį API (duotai OS), jie naudoja panašias operacijas, nors jos truputį ir skiriasi priklausomai nuo techninės įrangos.

2.1.5. Kada naudinga kurti DSL

Yra manoma, kad kai tik prireikia išspręsti problemų šeimą (t.y. sukurti programų šeimą), sukūrus architektūrą remiantis DSL, konfigūravimas (pvz. DSL programavimas) tampa paprastesnis. Bendru atveju kuriant naują programinę įrangą turėtų būti keliami šie klausimai: ar kuriama programa sprendžia pavienę izoliuotą problemą, ar ji vėliau gali tapti programų šeimos nare.

Dabartinės DSL yra naudojamos kurti programų šeimoms. Jos pritaikytos daugelyje sričių: grafikoje, finansiniuose produktuose, įrenginių draiveriuose, routerių tinkluose, robotų kalbose. Šis paplitimas, o taip pat su DSL susijusių renginių ir projektų skaičius parodo kylantį ir industrinės, ir mokslinės bendruomenės susidomėjimą DSL. Konkrečiai netrivialūs DSL yra kuriami ir naudojami tokiose didelėse kompanijose kaip Philips ir Motorola.

2.2. MDA (OMG)

Modeliais paremta architektūra (MDA) – tai 2001 metais kompanijos OMG pasiūlyta programų kūrimo metodologija, palaikanti modeliais paremtą programų sistemų inžineriją. MDA pateikia gaires specifikacijų, pateiktų modeliais, struktūrizacijai.

Taikant MDA metodą, iš pradžių sistemos funkcionalumas apibrėžiamas nuo platformos nepriklausančiu modeliu (PIM) naudojant atitinkamą taikymo srities kalbą. Tada pagal platformos apibrėžimo modelį (PDM), skirtą atitinkamai platformai (CORBA, .net ir t.t.), PIM transformuojamas į vieną ar daugiau nuo platformos priklausančių modelių (PSM). Šie modeliai gali būti paleisti kompiuteriuose naudojant kitą DSL, arba bendros paskirties kalbą (pvz.: Java, Python, C# ir t.t.). Paprastai transformacija atliekama automatinių įrankių.

2.2.1. MDA metodas

Vienas iš pagrindinių MDA tikslų yra dizaino ir architektūros atskyrimas. Dizaino ir architektūros realizacijos technologijos gali keistis nepriklausomai viena nuo kitos. Dizainas skirtas funkcinių reikalavimų realizacijai, o architektūra suteikia infrastruktūrą nefunkcinių reikalavimų (plečiamumas, patikimumas, našumas ir t.t.) įvykdymui. MDA daroma prielaida, kad PIM, kuris atvaizduoja koncepcinį dizainą, realizuojantį

funkcinius reikalavimus, liks nepaliestas realizacijos technologijų bei programinių architektūrų pokyčių.

Ypač svarbi modeliais paremtai architektūrai yra modelių transformacijos sąvoka. OGM sukūrė specialią standartinę modelių transformacijų kalbą, pavadintą QVT (užklausos, vaizdai, transformacijos).

2.2.2. MDA įrankiai

MDA įrankiai naudojami modelių ir metamodelių kūrimui, interpretavimui, lyginimui, transformavimui ir t.t. Kalbant apie MDA, modeliai skirstomi į dvi rūšis: žmonių sukurti pradinius modelius ir išvestinius modelius, automatiškai sugeneruotus programų. Pavyzdžiui, analitikas gali sukurti pradinį UML modelį, remdamasis verslo srities analizės rezultatais, o iš jo, pasinaudojant modelio transformacijos operacija, gali būti sugeneruotas Java modelis. MDA įrankis gali priklausyti vienai arba daugiau iš šių klasių:

- Kūrimo įrankis – tai įrankis, skirtas pradinių modelių kūrimui, arba išvestinių modelių redagavimui.
- Analizės įrankis – tai įrankis, skirtas modelio pilnumo patikrinimui, neatitikimų paieškai, klaidų ar įspėjimų sąlygų tikrinimui. Taip pat naudojamas modelio metrikų skaičiavimui.
- Transformacijos įrankis – tai įrankis, skirtas modelių transformacijai į kitus modelius, kodą, arba dokumentaciją.
- Kompozicijos įrankis – tai įrankis, skirtas sujungti (pvz. sulieti, naudojant tam tikrą sujungimo semantiką) keletą modelių. Pageidautina, kad sujungiami modeliai atitiktų tą patį metamodelį.
- Testavimo įrankis – tai įrankis, skirtas modelių testavimui pagal modeliais paremto testavimo metodiką.
- Imitavimo įrankis – tai įrankis, skirtas sistemos, atvaizduotos tam tikru modeliu, vykdymo imitavimui.
- Metaduomenų valdymo įrankis – tai įrankis, skirtas bendrųjų ryšių tarp modelių tvarkymui, o taip pat valdyti kiekvieno modelio metaduomenims (pvz.: autorius, paskutinio pakeitimo data, sukūrimo metodas (koks įrankis, kokia transformacija)) bei bendriesiems ryšiams tarp šių modelių (pvz.: vienas modelis yra kito versija, vienas modelis buvo sukurtas iš kito panaudojant transformaciją).
- Atgalinės inžinerijos įrankiai – tai įrankiai, skirti tam tikrų liktinių arba informacinių artefaktų rinkinių transformacijai į pilnus modelius.

Kai kurie įrankiai gali priklausyta keletai iš išvardintų klasių. Pavyzdžiui, kai kurie kūrimo įrankiai gali turėti transformacijos, arba testavimo galimybes. Egzistuoja kiti įrankiai, skirti tik kūrimui, tik grafiniam atvaizdavimui, tik transformacijai ir t.t.

Viena iš MDA įrankių savybių yra ta, kad dauguma jų naudoja modelius, kaip įeities duomenis, ir generuoja kitus modelius, kaip išeities duomenis. Tam tikrais atvejais gali būti naudojami parametrai, nepriklausantys MDA. Pavyzdžiui, transformacijos iš modelio į tekstą, arba iš teksto į modelį.

Paprastai MDA įrankiai painiojami su UML įrankiais (MDA įrankiai turi žymiai platesnį spektrą). Šie įrankiai gali būti vadinami atitinkamai kintamo ir pastovaus metamodelio įrankiais. UML CASE įrankiai paprastai tinka tik darbui su fiksuoto metamodelio modeliais, paprastai tam tikros UML metamodelio versijos (pvz. UML 2.1). MDA įrankiai paprastai turi vidinę genetinę galimybę adaptuotis prie tam tikrų metamodelių, arba tam tikros rūšies metamodelių.

2.2.3. Pagrindiniai MDA trūkumai

Nors nuo MDA sukūrimo 2001 metais, šis metodas labai išpopuliarėjo, bet vis dar egzistuoja tam tikros problemos, trukdančios MDA plėtrai:

- Nepilni standartai: MDA metodas remiasi įvairiais techniniais standartais, kurie dar nėra visiškai apibrėžti (pvz., veiksmų semantinė kalba,skirta xtUML), arba dar nėra realizuoti standartiniu būdu (pvz.: QVT transformacijų variklis, arba PIM, su virtualia vykdymo aplinka).
- Gamintojų užblokavimas: nors MDA buvo sukurtas, kaip metodas (techniniam) nepriklausomumui nuo platformos pasiekti, dabartiniai MDA įrankių gamintojai neskuba kurti įrankių, kuriuos būtų galima naudoti su skirtingomis platformomis.
- Idealistiškumas: MDA buvo sukurtas, kaip pirmyn nukreiptos inžinerijos metodas, t.y. modelis paverčiamas į realizacinį artefaktą (pvz.: kodas , arba duomenų bazės schema) viena kryptimi pilnai arba dalinai automatizuotu „generavimo“ žingsniu. Tai atitinka OMG viziją, kad MDA turėtų leisti sukurti visos probleminės srities procesų modelį UML kalba ir jo transformaciją į pilną (vykdomą) programą. Deja, iš to seka, kad realizacijos artefaktų keitimas (pvz. duomenų bazės schemos koregavimas) nėra palaikomas. Tai iškelia problemą srityse, kuriose šitoks potransformacinis modifikavimas yra būtinas.

- Specializuotas įgūdžių rinkinys: žmonės, naudojantys MDA paremtą programų inžineriją, privalo turėti gerus savo srities įgūdžius. Tokių žmonių yra žymiai mažiau nei paprastų programinės įrangos kūrėjų.
- OMG praeitis: OMG konsorciumas, kuris remia MDA metodo kūrimą, taip pat pasiūlė CORBA standartą, kuris nebuvo plačiai priimtas.

2.3. MS DSL įrankiai

2006 metais Microsoft korporacijos išleido savo sukurto „DSL Tools“ įrankio 1.0 versiją. Šis įrankis buvo išleistas kaip „Visual Studio 2005 SDK“ 3.0 versijos sudėtinė dalis.

Naudojant DSL Tools galima sukurti modeliavimo įrankius, pritaikytus tam tikram atvejui. Labai lengvai galima apibrėžti ir įgyvendinti modeliavimo kalbą. Pavyzdžiui, galima sukurti specializuotą kalbą, kuria aprašoma grafinė vertotojo sąsaja, verslo procesas, duomenų bazė, arba informacijos srautų judėjimas, o po to iš aprašo galima sugeneruoti kodą.

DSL Tools gali būti naudojamas sukurti grafinį redaktorių, pritaikytą tam tikrai probleminei sričiai. Pavyzdžiui, galimą sukurti įrankį, skirtą apibūdinti sąvokoms, kurios yra būdingos konkrečios organizacijos verslo procesų modeliavimo būdui. Jeigu kuriamas būsenų diagramų įrankis, galima apibrėžti, kas yra būsena, kokias savybes ji turi, kokios yra būsenų rūšys, kaip apibrėžiami perėjimai tarp būsenų ir t.t. būsenų diagrama, apibūdinanti kompanijos sutarčių būseną, iš pirmo žvilgsnio yra panaši į tą, kuri skirta apibūdinti vartotojo bendravimui su tinklo sąsaja, bet iš tikro jų vidus gerokai skiriasi. Sukuriant specialiai tai probleminei sričiai skirtą kalbą ir atitinkama grafinį įrankį, galima konkrečiai nurodyti, kurios būsenų diagramų savybės yra reikalingos.

2.3.1. DSL įrankių sandara

DSL Tools susideda iš:

- Projektų vediklio, kuris sukuria sukonfigūruotą sprendimą. Šiame sprendime galima aprašyti taikymo srities modelį, sudarytą iš grafinės kūrimo aplinkos ir išeities teksto generatoriaus. Paleidus pilną sprendimą iš Visual Studio, atidaromas naujo Visual Studio, su bandomuoju sprendimu, langas. Jame galima išbandyti grafinę aplinką ir išeities teksto generatorių.
- Grafinės aplinkos skirtos taikymo srities modelių kūrimui ir redagavimui.

- Grafinės aplinkos aprašymas XML kalba. Iš šio aprašo sugeneruojamas grafinės aplinkos kodas tam, kad pačiam nereikėtų rašyti jo rašyti, norint sukurti grafinę aplinką Visual Studio viduje.
- Kodo generatorių rinkinio, kuris, naudodamas taikymo srities modelio ir grafinės aplinkos aprašus kaip įvesties duomenis, generuoja kodą, kuris juos abu realizuoja. Kodo generatoriai taip pat validuoja srities modelio ir grafinės aplinkos aprašus ir, jeigu reikia, sugeneruoja atitinkamus pranešimus apie perspėjimus ir klaidas.
- Karkaso, skirto išeities teksto generatorių aprašymui.

Tam, kad sukurti naują kalbą, pirmiausia reikia panaudoti DSL kūrimo vediklį. Jis duoda pasirinkti vieną iš šių sprendimo šablonų:

- Darbo tėkmės diagramos – šablonas, kuriam būdingi plaukimo takeliai (swimlane) ir geometrinės figūros. Šis spendimo šablonas sukuria kalbą, panašią į UML veiklos diagramų kalbą. Jis naudojamas, kai reikia atvaizduoti darbų tėkmę, būsenas arba sekas. Pagrindiniai komponentai yra veiklos ir perėjimai tarp jų. Sprendimo šablone jau yra keletas pagalbinių elementų: objektai, pradinė būsena, galinė būsena, sinchronizacijos juosta.
- Klasių diagramos – šablonas, kuriam būdinga: į skyrius suskirstytos figūros, klasių paveldėjimas, ryšių paveldėjimas, figūrų paveldėjimas, ryšių savybės. Šis šablonas naudojamas, jei DSL naudoja objektus, esybes ir ryšius su parametrais. Šis spendimo šablonas sukuria kalbą, panašią į UML klasių diagramų kalbą. Pagrindiniai komponentai yra: klasės, interfeisai, keletas sąryšių tipų, generalizacijos ir realizacijos ryšiai. Klasė arba objektas atvaizduojami dėžutės formos figūra su išvardintai atributais.
- Minimali kalba - šablonas, kuriam būdinga: viena klasė arba figūra, vienas ryšys arba jungiamasis elementas. Šis šablonas naudojamas, kai kuriama DSL labai skiriasi nuo kitų sprendimo šablonų. Jis sukuria kalbą su viena klase ir vienu ryšiu, kurie atvaizduoti įrankių juostoje kaip stačiakampis ir linija. Klasė ir ryšys turi po pavyzdinį string tipo parametrą.
- Komponentų diagramos – šablonas, kuriam būdingi prievadai. Šis šablonas naudojamas, kai kuriama kalba naudoja prievadus. Prievadas – tai maža figūra, prijungta prie didesnės figūros krašto. Jis naudojamas modelio ryšių aprašymui. Ryšiai kuriami tarp objektų, atitinkančių prievadus, ir objektų, atitinkančių pagrindinius objektus.

Naudojimosi vedikliu pabaigoje nurodomos naujos kalbos savybės.

Pabaigus sprendimo kūrimo procesą, jame yra du projektai:

- **Dsl** – šiame projekte yra aprašyta kalba ir jos redagavimo bet apdirbimo įrankiai.
- **DslPackage** – šis projektas aprašo, kaip kalbos įrankiai apjungiami su Visual Studio.

Sukurtą pagal šabloną kalbą galima naudoti jos nepakeitus, arba galimą ją pakeisti taip, kad ji labiau atitiktų konkrečius poreikius. Beveik visas sprendimo kodas yra sugeneruojamas is kalbos aprašo bylos (paprastai DomainModel.tt). Taigi, didžioji dalis kabos pakeitimų gali būti padaryti modifikuojant šia bylą.

Išeities kodas yra generuojamas tekstinių šablonų (paprastai pavadintų *.tt), kurie skaito kalbos aprašą ir generuoja atitinkamą kodą. Taigi, kaskart paredagavus kalbos aprašą, reikėtų įvykdyti transformaciją su visais šablonais.

Taip pat galima parašyti papildomą kodą, skirtą modelio redaktoriaus elgesio detalizavimui ir papildomų apribojimų kalbai aprašymui. Jeigu reikia, galima padaryt didelius pakeitimus redaguojant tekstinius šablonus.

DSL testavimui galima sukompiliuoti ir atidaryti sprendimą atskirame Visual Studio redaktoriuje kuriame sukuriamas derinamasis projektas. Šį projektą galima naudoti kalbos testavimui. Dažniausiai iš taikymo srities kalbos yra generuojamas kodas, arba kiti artefaktai. Derinamajam projekte galima sukompiliuoti tekstinius šablonus, kurie perskaito modelį ir sugeneruoja atitinkamas bylas. Įsitikinus, kad viskas veikia teisingai, galima sukurti diegimo paketą (.msi bylą) ir platinti savo taikymo srities kalbą.

2.3.2. Artefaktų generavimas naudojant tekstinius šablonus

DSL Tools naudoja tekstinių šablonų transformacijos įrankių rinkinį, kuris leidžia apdirbti tekstinius šablonus. Tekstinis šablonas – tai byla, kurioje yra teksto blokų ir valdymo logikos mišinys. Transformuojant tekstinį šabloną, valdymo logika sujungia teksto blokus su duomenimis, esančiais modelyje, ir sugeneruoja išeities duomenis. Tekstiniai šablonai gali būti naudojami tekstinių artefaktų generavimui (kodo, HTML ataskaitų). Pavyzdžiui, modelis, apibrėžiantis valdymo perėjimą tarp atskirų vartotojo sąsajos puslapių, tokių kaip vediklio puslapiai, gali būti panaudotas, kaip įėjimo duomenys tekstiniam šablonui. Tekstinis šablonas gali sugeneruoti konfigūracijos bylą, realizuojančią šį perėjimą.

Visi DSL sprendimai turi derinamąjį sprendimą, kuris paleidžiamas sukompiliavus DSL sprendimą. Šiame derinamajame sprendime yra dvi pavyzdinės transformacijos: į C# ir Visual Basic kalbas.

Užduočių, kurias galima atlikti pasitelkus tekstinius šablonus, pavyzdžiai:

- Sukūrus modelius, galima pereiti per atmintyje esančias jų versijas (vietoje XML versijų naudojimo) ir sugeneruoti atitinkamą tekstą.
- Galima generuoti pasirinktą kodą, naudojant Visual C# arba Visual Basic judėjimui modeliu.
- Galima generuoti pasirinktas ataskaitas XML arba HTML kalba.

2.3.2.1. Tekstinių šablonų architektūra

Tekstinį šabloną sudaro trys dalys:

- Variklis – valdo tekstinio šablono transformacijos procesą.
- Šeimininkas – tai sąsaja tarp variklio ir vartotojo aplinkos. Visual Studio yra šeimininkas, DSL Tools turi komandinę eilutę, kuri irgi yra šeimininkas. Taip pat galima susikurti savo šeimininką.
- Viena arba kelios direktyvų doroklės – tai klasės, kurios tekstiniuose šablonuose tvarko direktyvas. Paprastai direktyvos naudojamos duomenų iš tam tikro šaltinio pateikimui šablonui (pavyzdžiui, iš modelio). Naudojant DSL Tools, kiekvienam DSL sugeneruojama po direktyvų doroklę tam, kad šablonai galėtų naudoti modelius, parašytus ta kalba, kaip įvesties duomenis. Taip pat galima susikurti savo direktyvų doroklę.

Tekstinio šablono transformacijos procesas sudarytas iš dviejų žingsnių:

- Pirmas žingsnis – variklis sukuria laikiną klasę, pavadintą „sugeneruota transformacijos klasė“. Ši klasė sudaryta iš tekstinio šablono kodo (t.y. tekstinių blokų, sakinių, išraiškų ir klasės savybių) ir direktyvų iškvietimo rezultatų.
- Antras žingsnis – variklis sukompiliuoja ir paleidžia sugeneruotą transformacijos klasę, kuri sugeneruoja išeities tekstą.

Variklis

Variklis pirmiausia gauna šabloną eilutės formatu iš šeimininko, kuris tvarko visas bylas. Tada variklis susisiekia su šeimininku ieškodamas direktyvų doroklių ir aplinkos kintamųjų. Po to jis sukompiliuoja ir paleidžia sugeneruotą transformacijos klasę.

Galiausiai variklis grąžina sugeneruotą tekstą šeimininkui, kuris savo ruožtu išsaugo jį byloje.

Šeimininkas

Šeimininkas yra atsakingas už viską, kas susiję su išorine aplinka, tame tarpe:

- Bylos arba rinkinio suradimas, kai variklis arba direktyvų doroklė jo paprašo. Šeimininkas gali peržiūrėti katalogus arba globalųjį rinkinių sandėlį, ieškodamas bylų ir rinkinių. Šeimininkas varikliui gali surasti direktyvų doroklės, pritaikytos tam tikroms reikmėms, kodą. Taip pat šeimininkas gali skaityti bylas ir grąžinti jų turinį eilutėmis.
- Standartinių rinkinių ir vardinių sričių sąrašo varikliui sudarymas. Variklis naudoja šiuos rinkinius ir vardines sritis kurdamas sugeneruotą transformacijos klasę tekstinio šablono transformacijos proceso metu.
- Terpės, reikalingos tam, kad variklis galėtu sukompiliuoti ir paleisti sugeneruotą transformacijos klasę, parūpinimas.
- Sugeneruotos išeities teksto bylos rašymas.
- Sugeneruotos išeities teksto bylos plėtinio, priskiriamo pagal nutylėjimą, nurodymas.
- Tekstinio šablono transformacijos klaidų gavimas iš variklio ir sprendimas ką su jomis daryti. Pavyzdžiui, šeimininkas gali jas išvesti vartotojo sąsajoje arba išsaugoti byloje.

Šeimininkai, pritaikyti tam tikroms reikmėms

Norint panaudoti tekstinių šablonų transformacijos funkcionalumą kitose programose, galima tam parašyti atitinkamą šeimininką. Keletas tokių šeimininkų pavyzdžių:

- Šeimininkas, integruotas į programą, kurioje pageidaujama naudoti tekstinių šablonų transformacijos funkcionalumą.
- Komandinės eilutės šeimininkas, optimizuotas tam tikrai direktyvų doroklei.

Direktyvos ir direktyvų doroklės

Direktyvos ir jų doroklės duoda tekstiniams šablonams papildomą funkcionalumą. Šis funkcionalumas gali varijuoti nuo tokio paprasto, kaip išeities bylos plėtinio nurodymas, iki sudėtingo, kaip speciali direktyvų doroklė skirta duomenų skaitymui iš duomenų bazės. Direktyvų doroklės turi po vieną ar daugiau direktyvų. Direktyvų ir

direktyvų doroklių paaiškinimui galima naudoti analogiją: direktyvų doroklės atitinka klases, o direktyvos – jų metodus. Direktyvos kviečiamos tiesiogiai iš tekstinių šablonų.

Direktyvos būna trijų tipų (Lentelė Nr. 1).

Lentelė Nr. 1 Direktyvų tipai

Direktyvos tipas	Apibūdinimas	Sudėtingumas	Naudojimo dažnumas
Integruota	Integruotos direktyvos įeina į tekstinių šablonų transformavimo įrankių rinkinį. Naudojant integruotas direktyvas, galima nurodyti tokias bendras parinktis, kaip tekstinio šablono kalba ir išeities bylos plėtinys. Yra penkios integruotos direktyvos: output, assembly, import, template ir include.	Mažas	Dažnas
Sugeneruota	Sugeneruotos direktyvos yra paremtos sukurta DSL. Iškvietus šias direktyvas, galima gauti priėjimą prie modelio iš tekstinio šablono sakinių ir išraiškų.	Vidutinis	Dažnas
Specialios paskirties	Specialios paskirties direktyvos rašomos, kad suteiktų tekstiniams šablonams papildomą funkcionalumą. Šias direktyvas bet kada galima naudoti tam, kad pasiekti išorinius duomenis.	Didelis	Retas

Direktyvos veikia papildydamos sugeneruotos transformacijos klasės kodą. Jos kviečiamos iš tekstinio šablono, o variklis apdoroja visus kreipinius ir tada sukuria sugeneruotą transformacijos klasę. Po sėkmingo direktyvos iškvietimo, likusi transformacijos šablone parašyto kodo dalis gali naudotis direktyvos suteikiamu funkcionalumu. Pavyzdžiui, galima iškviesti „import“ direktyvą eilute, pavaizduota 1 pav.

```
<#@ import namespace="System.Text" #>
```

1 pav. Direktyvos pavyzdys

Po šios direktyvos iškvietimo likusiame kode galima naudoti „StringBuilder“ klasę nesikreipiant į ją kaip „System.Text.StringBuilder“.

Sugeneruotos direktyvos

Panaudojus DSL Tools srities modelio sukūrimui, sugeneruojama direktyvų doroklė, skirta šiam srities modeliui. Tekstiniame šablone iškvietus šią doroklę, galima naudoti atitinkamame modelyje esančias srities klases. Po to tekstiniame šablone galima rašyti kodą, kuris kuria ataskaitas arba kodą, remiantis modeliu.

Pavyzdžiui, galima sukurti sprendimą, pasinaudojant „Minimalios Kalbos“ vedliu, sukompiliuoti ir paleisti jį, sukurti tekstinį šabloną ir į jį įrašyti eilutę, pavaizduotą 2 pav.

```
<#@ examplemodel processor="DSLMinimalTestDirectiveProcessor"
requires="fileName='Sample.min'"
provides="ExampleModel=ExampleModel" #>
```

2 pav. Direktyvos naudojimo pavyzdys

Šiame pavyzdyje „DSLMinimalTestDirectiveProcessor“ yra sugeneruota direktyvos doroklė, o „ExampleModel“ yra direktyva. Iškvietus šia direktyvą, galima iš tekstinio šablono pasiekti modelį (3 pav).

```
Model: <#=this.ExampleModel.Name#>
```

3 pav. Modelio pasiekimas iš tekstinio šablono

Specialios paskirties direktyvos

Galima parašyti specialios paskirties direktyvų dorokles, kad jos suteiktų tekstiniams šablonams tam tikrą papildomą funkcionalumą. Šias direktyvų dorokles galima naudoti, kai norima gauti priėjimą prie išorinių duomenų ar resursų iš tekstinio šablono.

Skirtingi tekstiniai šablonai gali naudoti funkcionalumą, kurį suteikia viena direktyvų doroklė, taigi, direktyvų doroklės gali būti naudojamos rašant kodą, skirtą pakartotiniam panaudojimui. Integruota „include“ direktyva yra panaši tuo, kad leidžia parašyti atskirą kodą, kuris gali būti naudojamas skirtinguose tekstiniuose šablonuose. Skirtumas toks, kad bet koks funkcionalumas, suteikiamas „include“ direktyvos, yra fiksuotas ir nepriima parametrų. Norint suteikti tekstiniams šablonams bendrą

funkcionalumą, kuris priima parametrus, reikia sukurti specialios paskirities direktyvų doroklę.

Keletas specialios paskirties direktyvų doroklių pavyzdžių:

- Direktyvų doroklė, skirta duomenų grąžinimui iš duomenų bazės ir priimanti vartotojo vardą ir slaptažodį, kaip parametrus.
- Direktyvų doroklė, skirta bylų atidarymui ir skaitymui bei priimanti bylos vardą kaip parametrą.

3. PLA imitacinio modelio sudarymo metodas

Aprašant sistemą, sistemos būsenų aibėje *S* yra išskiriama baigtinė pagrindinių būsenų aibė $I = \{0, 1, 2, \ldots, s\}$. Šios aibės elementai $\nu \in I$ yra pagrindinės agregato būsenos. Kiekvienai pagrindinei būsenai yra priskiriamas sveikas neneigiamas skaičius $\|\nu\|$, kuris vadinamas būsenos rangu, bei išgaubtas daugiakampis $Z^{(\nu)}$, nusakytas $\|\nu\|$ išmatavimų Euklido erdvėje. Sakoma, kad būsenų aibę $Z = \bigcup_{\nu \in I} Z^{(\nu)}$ sudaro poros $\left(\nu, z^{(\nu)}\right)$.

Pradiniu laiko momentu t_0 agregatas yra būsenoje $z(t_0) = \left(\nu, z^{(\nu)}(0)\right)$. Jei nėra įėjimo signalo, kai $t > t_0$, taškas $z^{(\nu)}(t)$ juda srityje $Z^{(\nu)}$ tol, kol pasieks šios srities kontūrą. Laiko momentas t_1, kai pasiekiamas kontūras, vadinamas *atraminiu*.

Daugiakampio *Z* kontūras yra aprašomas lygtimis:

$$\sum_{i=1}^{\|\nu\|} \gamma_{ji}^{(\nu)} z_i^{(\nu)} + \gamma_{j0}^{(\nu)} = 0, \quad j = 1, \ldots, m(\nu),$$

Atraminiu laiko momentu agregato pagrindinė būsena kinta iš ν į ν', o agregato papildomos koordinatės įgyja reikšmę $z^{(\nu')}(t_1) \in Z^{(\nu')}$.

Papildomos koordinatės kinta srityje $Z^{(\nu')}$ tol, kol nepatenka ant šios srities kontūro. Tam įvykus, agregatas vėl keičia būseną.

Atkarpomis-tiesinio agregato būsenai patekus į srities kontūrą, yra generuojamas išėjimo signalas $y \in Y$, čia *Y* – išėjimo signalų aibė, kuri yra analogiška aibei *Z*. Išėjimo signalo struktūra

$$y = \left(\lambda, y^{(\lambda)}\right),$$

$y^{(\lambda)}$ – išėjimo signalo papildomų koordinačių vektorius, priklausantis nuo λ:

$$y^{(\lambda)} = \left(y_1^{(\lambda)},\ldots,y_{r(\lambda)}^{(\lambda)}\right).$$

Jei laiko momentu t^* į agregatą yra paduodamas įėjimo signalas, tai agregato papildomos koordinatės nustoja kitusios ir agregato būsena akimirksniu pereina į kitą tos pačios ar kitos srities $Z^{(\nu'')}$ tašką.

Įėjimo signalo atėjimo momentu agregatas išduoda išėjimo signalą ir šis momentas taip pat yra atraminis. Toliau taškas $z^{(\nu'')}(t)$ srityje $Z^{(\nu'')}$ juda taip pat, kaip buvo aprašyta anksčiau.

Kai nėra įėjimo signalų, atkarpomis-tiesinio agregato būsenos kitimas aprašomas tokiomis lygtimis:

$$\nu(t) = \nu = const;\ \frac{dz^{(\nu)}}{dt} = -\alpha^{(\nu)},$$

Vektorinėje formoje diferencialinės lygties sprendinys $z^{(\nu)}(t)$ gali būti užrašytas

$$z^{(\nu)}(t) = z^{(\nu)}(0) + \alpha^{(\nu)}(t - t_0).$$

Spręsdami papildomų koordinačių kitimo ir sričių kontūrų lygtis, kartu galime apskaičiuoti ir laiko momentus, kai papildomų koordinačių reikšmės patenka į kontūrus.

Patekus į kontūrą, nauja pagrindinė būsena ν' apibrėžiama tikimybių skirstiniu P_1, priklausančiu tik nuo būsenos $z(t_1)$. Siekiant apibrėžti papildomų koordinačių vektorių $z^{(\nu')}$, sudaromas pagalbinis vektorius η, kurio skirstinys nepriklauso nuo proceso istorijos, o priklauso tik nuo ν ir $z^{(\nu)}$.

Papildomų koordinačių vektorius apskaičiuojamas

$$z^{(\nu')} = \left(z_*^{(\nu)}, \eta\right) \times L_z^{(\nu\nu')},$$

Kai ateina įėjimo signalas $\left(\mu, x^{(\mu)}\right)$, naują pagrindinę būseną ν'' nusako tikimybinis skirstinys P_2, kuris priklauso nuo ν, $z_*^{(\nu)}$ ir μ.

Papildomų koordinačių vektoriaus $z^{(\nu'')}$ apibrėžimui sudaromas papildomas vektorius ξ, kurio pasiskirstymas nepriklauso nuo proceso istorijos, o priklauso nuo ν, $z_*^{(\nu)}$ ir μ.

$$z^{(\nu'')} = \left(z_*^{(\nu)}, \xi, x^{(\mu)}\right) \times L_x^{(\nu,\nu'')},$$

Išėjimo signalas formuojamas, kai $z^{(\nu)}$ pasiekia kontūrą arba ateina įėjimo signalas (laiko momentu t').

Pirmu atveju:

$$\lambda = \lambda\left(\nu, \nu', z_*^{(\nu)}\right),$$

$$y^{(\lambda)} = \left(z_*^{(\nu)}, \eta\right) \times L_y^{(\nu\nu')}.$$

Matricos $L_y^{(\nu\nu')}$ elementai priklauso nuo ν, ν' ir $z_*^{(\nu)}$.

Antru atveju:

$$\lambda = \lambda\left(\nu, \nu'', z^{(\nu)}(t'), \mu\right),$$

$$y^{(\lambda)} = \left(z^{(\nu)}(t'), \xi, x^{(\mu)}\right) \times L_y^{(\nu\nu'')}.$$

Matricos $L_y^{(\nu\nu'')}$ elementai priklauso nuo ν, ν'', $z^{(\nu)}(t')$ ir μ.

Atkarpomis-tiesiniai agregatai priklauso automatų modelių klasei. Kaip ir automatas, atkarpomis-tiesinis agregatas aprašomas nurodant būsenų aibę *Z*, įėjimo signalų aibę *Y* bei perėjimo operatorių (atvaizdavimą) *H* ir išėjimo operatorių *G*. Tačiau agregatas turi nemažai ypatybių, skiriančių šį modelį nuo automatinių modelių

Agregato funkcionavimas stebimas aibėje laiko momentų $t \in T$, t.y. agregato būsena $z \in Z$ yra laiko funkcija $z(t)$. Atkarpomis-tiesinio agregato būsenos struktūra yra tokia pat, kaip ir atkarpomis-tiesinio Markovo proceso, t.y.:

$$z(t) = \left(\nu(t),\ z_\nu(t)\right),$$

Bendru atveju,

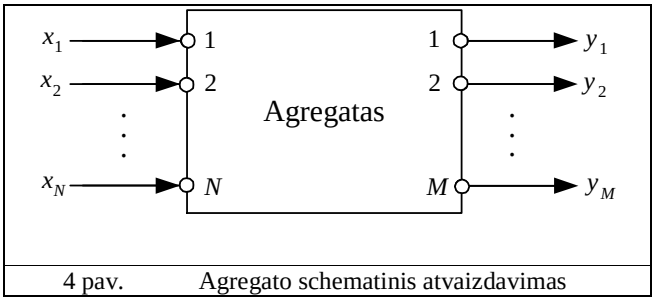
$$\nu(t) = \{\nu_1(t), \nu_2(t), \ldots, \nu_m(t)\}, \qquad z_\nu(t) = \{z_{\nu 1}(t), z_{\nu 2}(t), \ldots, z_{\nu k}(t)\},$$

Kai nėra įėjimo signalų, agregato būsenos kinta taip:

$$\nu(t) = \text{const}, \quad \frac{dz_\nu(t)}{dt} = -\alpha_\nu,$$

Agregato būsena gali pakisti tik dviem atvejais: kai į agregatą yra paduodamas įėjimo signalas arba kai viena iš tolydžios komponentės koordinačių įgyja tam tikrą reikšmę.

Tarkime, turime agregatą, su *N* įėjimo ir *M* išėjimo sąveikavimo taškų (*ST*) (4 pav).



4 pav. Agregato schematinis atvaizdavimas

Įėjimo signalai $x_1, x_2, ..., x_w \in X$ yra paduodami į įėjimo *ST*. Signalai $x_i \in X_i$, $i = \overline{1, N}$, yra vadinami elementariais signalais, o aibė X_i yra vadinama elementarių signalų aibe. Bendru atveju elementarus signalas yra vektorius, t.y. $x_i = \left(x_i^1, x_i^2, \ldots, x_i^{r_i}\right)$, be to, šio vektoriaus koordinačių reikšmės priklauso atitinkamai aibei, t.y. $x_i^j \in X_i^j$, $j = \overline{1, r_i}$.

Elementarūs signalai, kurie gali ateiti į *i*-jį *ST*, sudaro aibę:

$$X_i = X_i^1 \times X_i^2 \times \ldots \times X_i^{r_i},\ i = \overline{1, N}.$$

Agregatų įėjimo signalų aibė yra lygi aibių X_i sąjungai, t.y.

$$X = \bigcup_{i=1}^{N} X_i .$$

Analogiškai yra apibrėžiama išėjimo signalų aibė:

$$Y = \{y_1, y_2, \ldots, y_M\},\ \ y_1 = \left\{y_l^1, y_l^2, \ldots, y_l^{s_i}\right\} \in Y_l,$$

$$y_l^k \in Y_l^k,\ \ l = \overline{1, M},\ \ k = \overline{1, s_l}.$$

Elementarių signalų aibė išeinanti iš *l*-jo ST yra lygi:

$$Y_l = Y_l^1 \times Y_l^2 \times \ldots \times Y_l^{s_l},\ \ l = \overline{1, M}.$$

Agregato išėjimo signalų aibė:

$$Y = \bigcup_{l=1}^{M} Y_l .$$

Agregato funkcionavimas yra nagrinėjamas diskrečiais laiko momentais, kurie priklauso aibei $T = \{t_0, t_1, ..., t_m, ...\}$. Tais laiko momentais gali įvykti vienas ar keli įvykiai, kurie sukelia agregato būsenos pasikeitimą. Agregato įvykių aibę $E = E' \cup E''$ sudaro du nepersikertantys poaibiai $E' \cap E'' = \varnothing$. Aibę $E' = \{e_1', e_2', ..., e_N'\}$ sudaro įvykiai, kurie įvyksta dėl įėjimo signalų atėjimo. Tarp aibių X ir E' elementų yra funkcinis ryšys. Poaibis $E'' = \{e_1'', e_2'', ..., e_f''\}$ yra vadinamas vidinių įvykių poaibiu, čia $e_i'' = \{e_{ij}'',\ j = 1, 2, 3, \ldots\},\ i = \overline{1, f},$ yra agregato vidiniai įvykiai, *f* – operacijų, kurios gali vykti agregate, skaičius. Aibės E'' įvykiai fiksuoja operacijų pabaigą.

Laiko momentų aibė *T* susideda iš dviejų poaibių:

$$T = T' \cup T'',$$

Kiekvienam vidiniam įvykiui $e_i \in E''$ yra priskiriama valdanti seka, t.y.

$$e_i'' \mapsto \{\xi_j^{(i)}\},\ j = \overline{1, \infty},$$

Taip pat nurodoma skaitliukų aibė:

$$\{r(e_i'', t_m)\},\ i = \overline{1, f},$$

Tam, kad būtų galima apibrėžti operacijų pradžios ir pabaigos momentus, yra naudojamos aibės valdančiųjų sumų:

$$\{s(e_i'', t_m)\},\ \{w(e_i'', t_m)\},\ i = \overline{1, f},$$

Neprioritetinių operacijų atveju, valdanti suma $w(e_i'', t_m)$ yra apibrėžiama taip:

$$w(e_i'', t_m) = \begin{cases} s(e_i'', t_m) + \xi_{r(e_i'', t_m)+1}, & \text{jei laiko momentu } t_m \text{ vyksta operacija,} \\ & \text{kuriai pasibaigus, įvyks įvykis } e_i^{"}, \\ \infty, & \text{priešingu atveju.} \end{cases}$$

Begalybės simbolis (∞) yra naudojamas pažymėti, kad laiko momentas, kai pasibaigs operacija, yra nežinomas.

Pateiktas valdančios sumos apibrėžimas yra naudojamas sudarant imitacinius modelius. Kai agregatinis modelis yra naudojamas sistemų formalizavimui ir sistemos funkcionavimo korektiškumo analizei, valdanti suma apibrėžiama taip:

$$w(e_i'', t_m) = \begin{cases} < \infty, & \text{jei laiko momentu } t_m \text{ vyksta operacija, kuriai} \\ & \text{pasibaigus, įvyks įvykis } e_i^{"}, \\ \infty, & \text{priešingu atveju.} \end{cases}$$

Įvedus valdančias sumas, agregato būsenos tolydi komponentė įgauna pavidalą:

$$z_v(t_m) = \{w(e_1'', t_m), w(e_2'', t_m), \ldots, w(e_f'', t_m)\}.$$

Tolydžios komponentės koordinatės apibrėžia laiko momentus, kada agregate gali įvykti įvykiai. Be to, visada $w(e_i'', t_m) \geq t_m$.

Agregato būsena $z(t_m)$ kinta diskrečiais laiko momentais t_m, m = 1, 2, …, ir išlieka pastovi laiko intervalais $[t_m, t_{m+1})$, m = 0, 1, 2, …, čia t_0 – pradinis sistemos funkcionavimo momentas.

Kai yra žinoma agregato būsena $z(t_m)$, m = 0, 1, 2, …, laiko momentas t_{m+1}, kai įvyksta sekantis įvykis, paskaičiuojamas taip:

$$t_{m+1} = \min_i \{w(e_i'', t_m)\},\ 1 \leq i \leq f.$$

Operatorius H apibrėžia naują agregato būseną:

$$z(t_{m+1}) = H[z(t_m, e_i)], \quad e_i \in E' \cup E''.$$

Operatorius *G* apibrėžia išėjimo signalus:

$$y = G[z(t_m, e_i)], \quad e_i \in E' \cup E'', \quad y \in Y.$$

Aibė įvykių, kurie gali įvykti agregatinėje sistemoje:

$$E = \bigcup_{k=1}^{K} E_k'',$$

čia E_k'' – aibė įvykių, vykstančių *k*-me agregate.

Aibė laiko momentų, kurių metu įvyksta įvykiai iš aibės *E*

$$T = \bigcup_{k=1}^{K} T_k'',$$

čia T_k'' – aibė laiko momentų, kurių metu vyksta vidiniai įvykiai *k*-me agregate.

Kitas laiko momentas t_{m+1}, kurio metu įvyksta įvykis iš aibės *E*, aprašomas taip:

$$t_{m+1} = \min_{1 \le k \le K} \left(\min_{\substack{w_k(e_r'', t_m) \in z_{v_k}(t_m) \\ e_r'' \in E_k''}} w_k\left(e_{r,}'' t_m\right) \right). \qquad (1)$$

Išėjimo signalai gali būti formuojami tik įvykių iš aibės *E* įvykimo momentais ir todėl įėjimo signalų atsiradimo momentai apibrėžiami (1) išraiška.

Išėjimo signalų aibė

$$Y = \bigcup_{k=1}^{K} Y_k,$$

čia Y_k –*k*-jo agregato išėjimo signalų aibė. Naudojant matricą *H* kiekvienam išėjimo signalui iš aibės *Y* paskiriamas kanalas, kuriuo jis turi būti perduodamas. Matrica *R* apibrėžia agregatą ir įėjimo numerį, į kurį paskirtu kanalu perduodamas išėjimo signalas.

Žemiau yra pateikiamas agregatinės sistemos modeliavimo algoritmas.

1. Formuojamos agregatų pradinės būsenos:

$$z_k(t_0) = \{v_k(t_0); r_k(t_0); z_v^k(t_0)\}, \quad k = \overline{1, K}.$$

2. Naudojant (4.1) išraišką, nustatomas laiko momentas t_{m+1}, kada įvyks sekantis vidinis įvykis, ir numeris *k*-jo agregato, kuriame tas įvykis įvyks.
3. Apibrėžiama nauja *k*-jo agregato būsena $z_k(t_{m+1})$ pagal išraišką

$$z_k(t_{m+1}) = H[z_k(t_m), e_i], \quad e_i \in E''$$

ir formuojami išėjimo signalai, priklausantys aibei $\tilde{Y}$

$$y = G[z_k(t_m), e_i], \ e_i \in E'', \ y \in \tilde{Y}.$$

Visiems likusiems agregatams fiksuojama nauja būsena

$$z_i(t_{m+1}) = z_i(t_m), \ i = \overline{1,K}, \ i \neq K.$$

Agregatų, kurių numeris nelygus *k*, tiek diskrečių, tiek ir tolydžių būsenų dedamosios išlieka nepakitę.

4. Tikrinama aibė $\tilde{Y}$. Jei aibėje $\tilde{Y}$ yra bent vienas elementas $y_i \in Y_k$, $1 \leq i \leq M_k$, $1 \leq k \leq K$, tuomet, naudojantis matrica *H*, nustatomas kanalo numeris h_{ki}, kuriuo turi būti perduodamas išėjimo signalas. Naudojantis matrica *R* nustatomas agregato numeris $r_{1h_{k_i}}$ ir įėjimo poliaus $r_{2h_{k_i}}$, kur perduodamas įėjimo signalas y_i . Pereinama prie 5 žingsnio. Jei $\tilde{Y} = \varnothing$, tai pereinama prie 2 žingsnio.
5. Apibrėžiama nauja agregato $k = r_{1h_{k_i}}$ būsena, kai ateina įėjimo signalas x_i,

   čia $i = r_{2h_{k_i}}$ ir formuojamas išėjimo signalas, kuris fiksuojamas aibėje $\tilde{Y}$. Elementas y_i pašalinamas iš aibės $\tilde{Y}$. Pereinama prie 4 žingsnio.

4. Bibliotekos imitacinio modelio posistemės programinė realizacija.

Imitacinio modeliavimo posistemė realizuota Microsoft .NET 2.0 aplinkoje programavimo kalba C#. Posistemę sudaro:

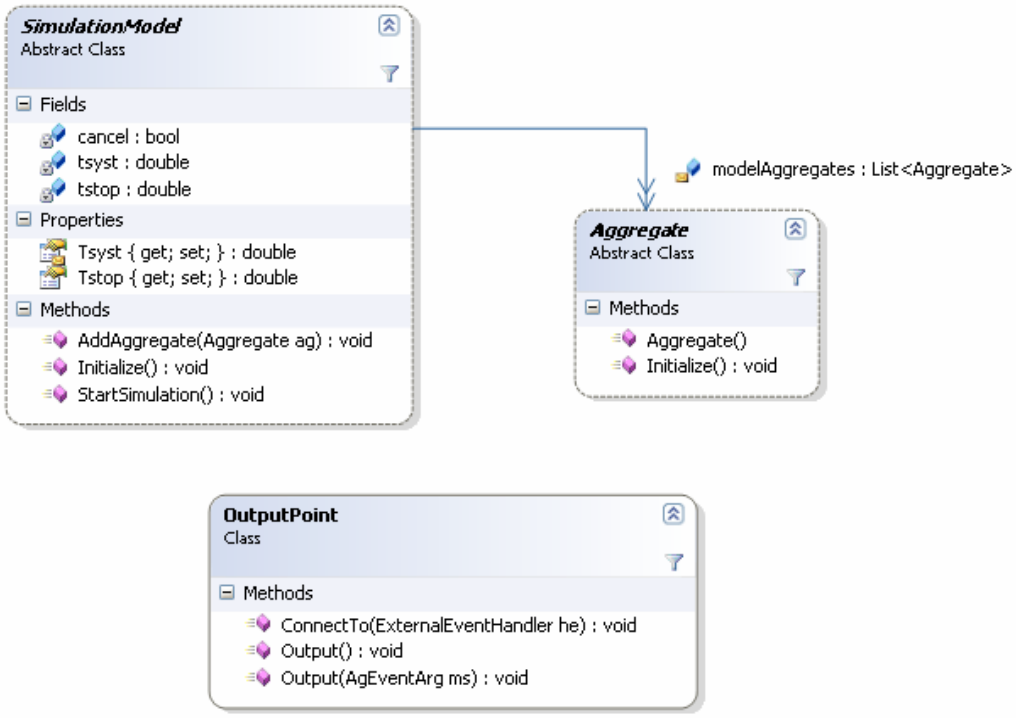
- Agregatinių imitacinių modelių bazinių klasių biblioteka;
- Programinė priemonė leidžianti automatiškai generuoti agregatinio imitacinio modelio programinį kodą iš modelio specifikacijos XML formoje gautos iš BPMN aprašymo.

Agregatinių imitacinių modelių bazinių klasių bibliotekoje agregatų sujungimo schemai realizuoti skirtos trys klasės (5 pav):

SimulationModel – agregatinio imitacinio modelio bazinė klasė;

Aggregete - agregato bazinė klasė;

OutputPoint – agregato sąveikos taško klasė.



5 pav. Agregatų sujungimo schemos realizavimo klasių diagrama

Agregatinės sistemos agregatų sujungimą realizuoja **SimulationModel** klasė, talpinanti savyje agregatų sąrašą **modelAggregates**. Šiame sąraše saugomi agregatinės sistemos agregatų objektai, kurių bazinė klasė yra **Aggregete**. Kuriant agregatų sujungimo schemą pirmiausiai sukuriami agregatų objektai ir klasės **SimulationModel** metodu **AddAggregate** patalpinami į **modelAggregates** sąrašą. Po to agregatų objektai su klasės **OutputPoint** t.y. agregato sąveikos taško objekto metodu **ConnectTo()** sujungiami sujungimo kanalais į uždarą sistemą. Agregatų sąveika siunčiant pranešimus agregatų sujungimo kanalais realizuojama sąveikos taško objekto metodu **Output().**

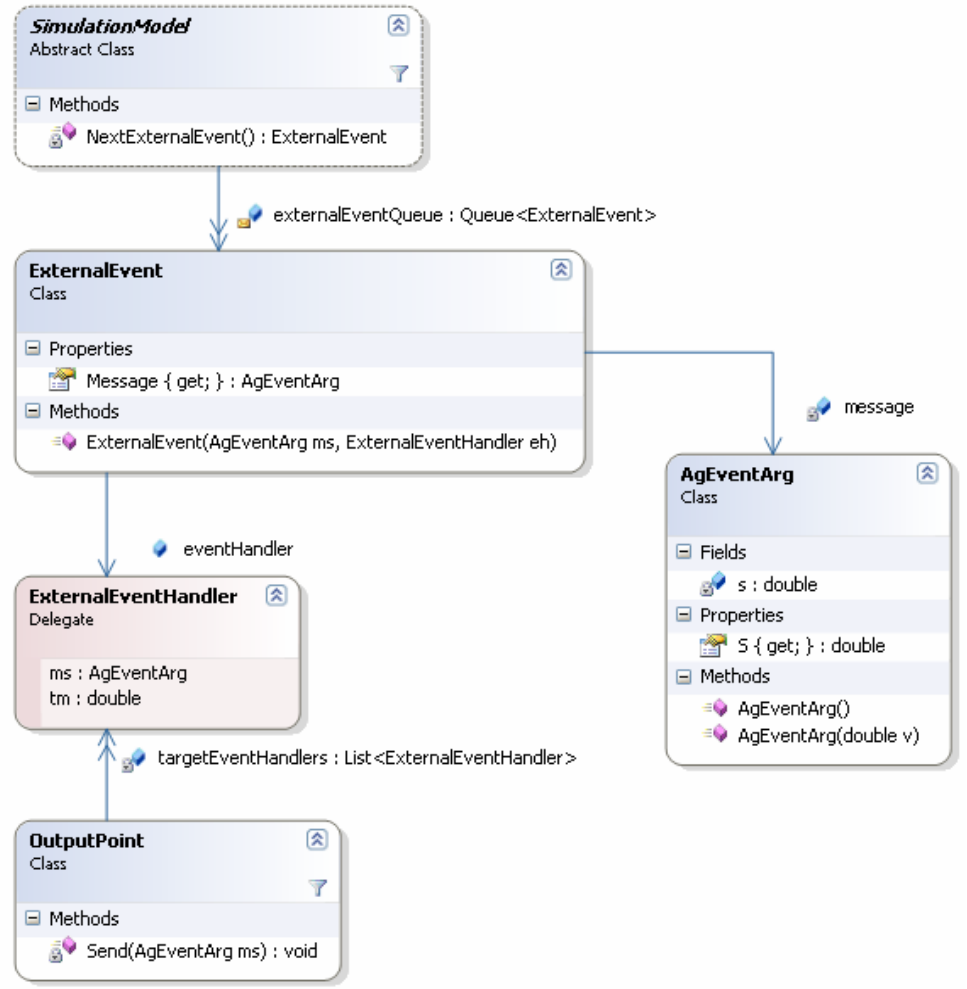
Taip pat agregatinės sistemos inicializacijai t.y. pradinės agregatinės sistemos būsenos nustatymui sukurtas **SimulationModel** klasės metodas initialize(). Šio metodo realizacija iškviečia visų sąrašo **modelAggregates** agregatų objektų metodus **initialize().**

Agregato išorinio įvykio generavimui ir jo apdorojimui bibliotekoje yra skirtos tokios klasės (6 pav):

ExternalEvent – išorinio įvykio klasė;

ExternalEventHandler - agregato išorinio įvykio apdorojimo metodo delegatas;

AgEventArg – pranešimo perduodamo tarp agregatų bazinė klasė.



6 pav. Išorinių įvykių realizavimo klasių diagrama

Agregato išorinio įvykio generavimą inicijuoja agregatas pranešimo siuntėjas metodu **Send()** kreipdamasis į atitinkamą sąveikos taško objektą. Siunčiamas pranešimas yra objektas, kurio bazinė klasė **AgEventArg**. Sąveikos taško objekto metode **output()** ši pranešimo struktūra yra įvelkama į naujai sugeneruotą išorinio įvykio **ExternalEvent** tipo objektą. Tokių išorinių įvykių sugeneruojama tiek kiek yra saugoma nuorodų sąveikos taško objekto atribute **targetEventHandlers**. Šiame atribute yra saugomos nuorodos į agregato atitinkamo išorinio įvykio apdorojimo metodo delegatą **ExternalEvenHandler**. Visi sugeneruoti išoriniai įvykiai patalpinami ir saugome eilėje **SimulationMoel.externalEventQueue** kol yra išrenkami apdorojimui klasės **SimulationModel** metodu **NextExternalEvent()**.

Agregato vidinio įvykio generavimui ir jo apdorojimui bibliotekoje yra skirtos tokios klasės (7 pav):

InternalEvent – vidinio įvykio klasė;

ContiousCoordinate – abstrakti tolydinės koordinatės klasė;

InternalEventHandler - agregato vidinio įvykio apdorojimo metodo delegatas;

ControlSum – valdančios sumos klasė;

ControlSequence - valdančios sekos klasė.

Agregato vidinio įvykio generavimą inicijuoja agregato valdančios sumos **ControlSum** metodas **CreateInternalEvent(w)**. Šio metodo parametru **w** nustatome laiko momentą kada yra numatomas vidinis įvykis. Parametro **w** nustatymui gali būti panaudotas valdančios sekos **ControlSequence** kasės objekto vienas iš metodų:

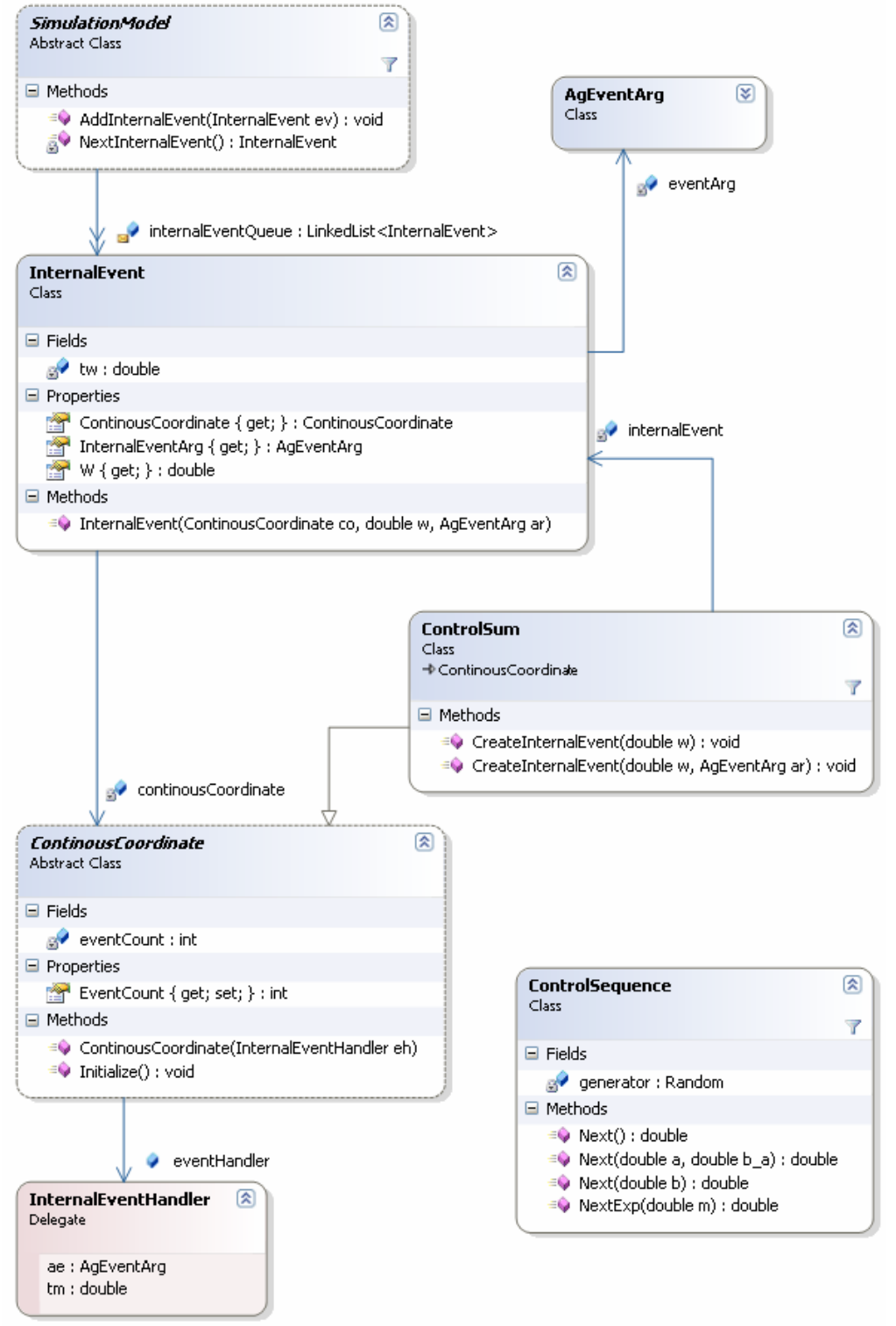
Next() – sugeneruoja atsitiktinį skaičių tolygiai pasiskirsčiusį intervale (**0, 1**);

Next(double a, double b) - sugeneruoja atsitiktinį skaičių tolygiai pasiskirsčiusį intervale (**a, b**);

Next(double b) - sugeneruoja atsitiktinį skaičių tolygiai pasiskirsčiusį intervale (**0, b**);

NextExp(duoble m) – sugeneruoja atsitiktinį skaičių pasiskirsčiusį pagal eksponentinį dėsnį su vidurkiu **m**.

Sugeneruotas vidinis įvykis patalpinamas į surūšiuotą vidinių įvykių sąrašą **internalEventQuque.** Rūšiavimas yra atliekamas pagal vidinio įvykio atributą **W**. Vidinis įvykis apdorojimui išrenkamas klasės **SimulationModel** metodu **NextInternalEvent()**.



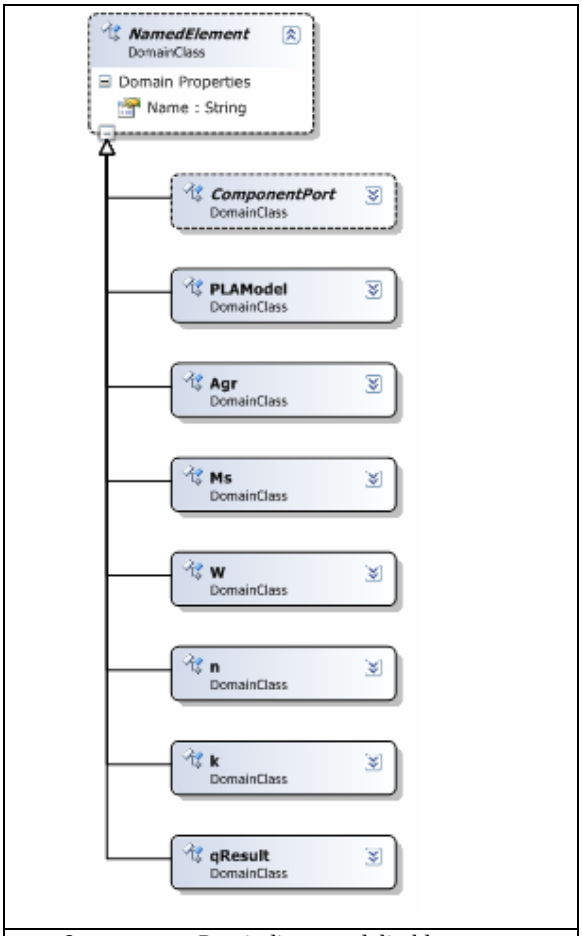
7 pav. Vidinių įvykių realizavimo klasių diagrama

5. PLA imitacinio modelio sudarymo kalba

5.1. Kalbos metamodelis

Imitacinio modelio realizavimo posistemės objektinis modelis

Imitacinio modelio PLA specifikaciją sudaro modeliavimui naudojamų agregatų specifikacija. Jos pagrindinės klasės pateiktos 8 pav.



8 pav. Pagrindinės modelio klasės

NamedElement – tai aukščiausio hierarchinio lygio abstrakti klasė. Ją paveldi visos kitos modelio klasės.

ComponentPort – abstrakti klasė, skirta agregatų sujungimo jungtims.

PLAModel – bazinė modelio klasė, kurioje aprašoma pagrindinė modelio struktūra.

Agr – agregatų klasė.

Ms – žinučių klasė.

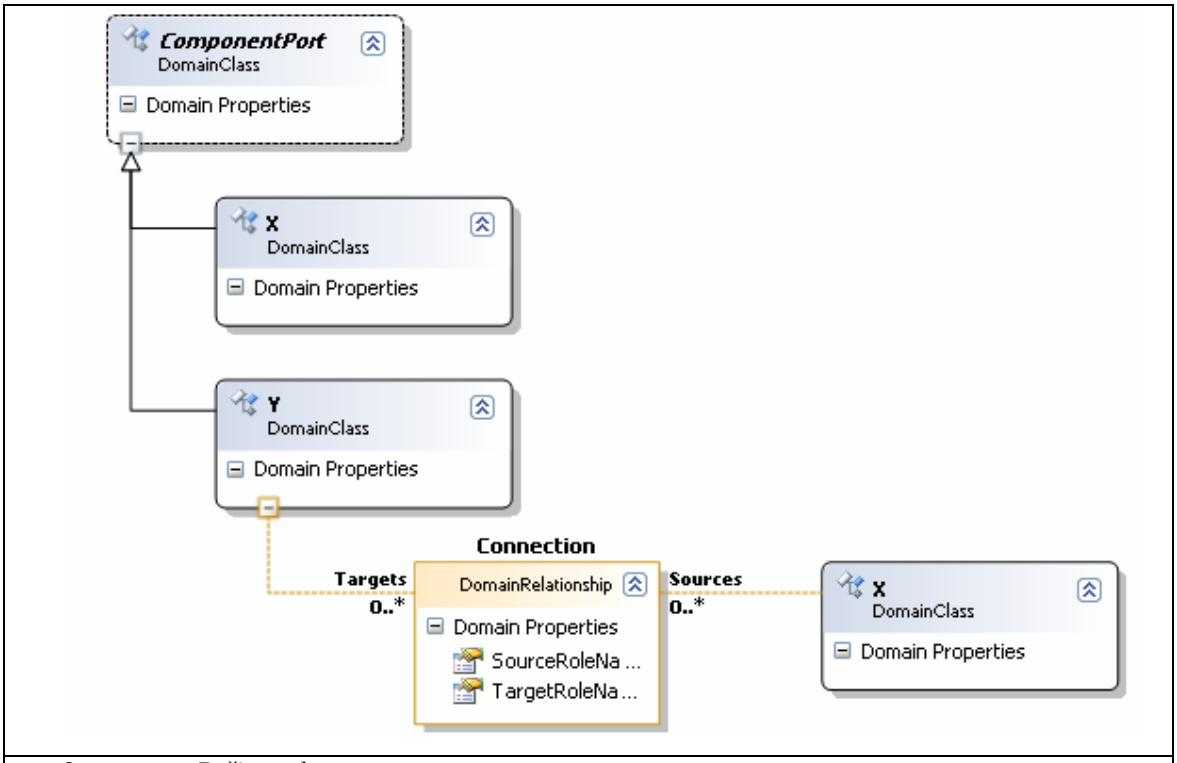
W – kontrolinių sumų klasė.

n, m – atributų klasės.

qResult – užklausų rezultatų klasė.

Šios klasės ir ryšiai tarp jų naudojami modelio aprašymui.

ComponentPort klasė pateikta 9 pav.



9 pav. Ryšių struktūra

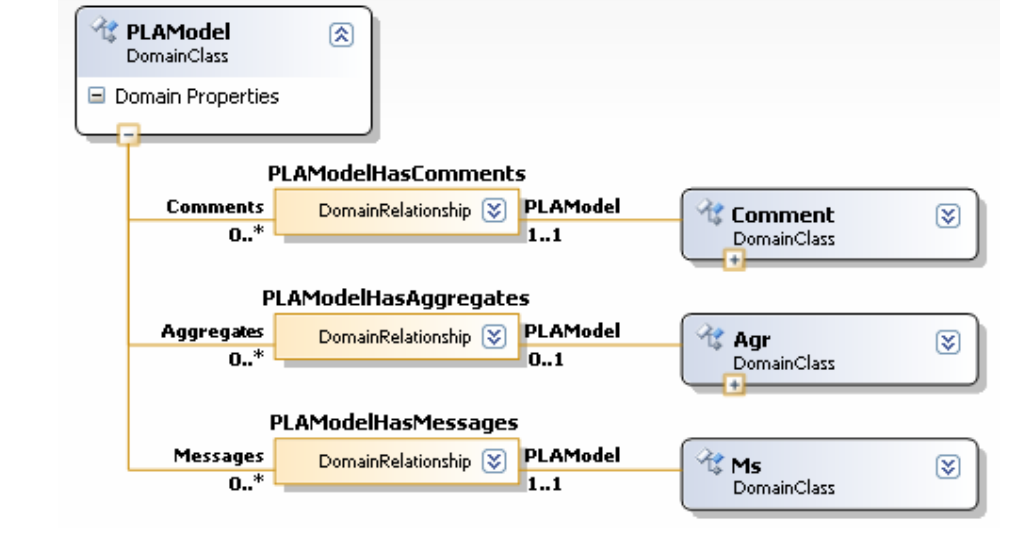
Dvi klasės – X ir Y, paveldi ComponentPort klasę.

X – tai įėjimo jungtis.

Y – tai išėjimo jungtis.

Tarp jų egzistuoja ryšys Connection, turintis du atributus: SourceRoleName ir TargetRoleName. Ryšio kardinalumas parodo, kad Connection turi vieną išėjimo jungtį Y ir vieną įėjimo jungtį X, t.y. jis jungia išėjimo jungtį su įėjimo jungtimi. Nors ryšys Connection gali jungti tik vieną išėjimo ir įėjimo porą, išėjimai ir įėjimai gali turėti bet kokį skaičių ryšių Connection.

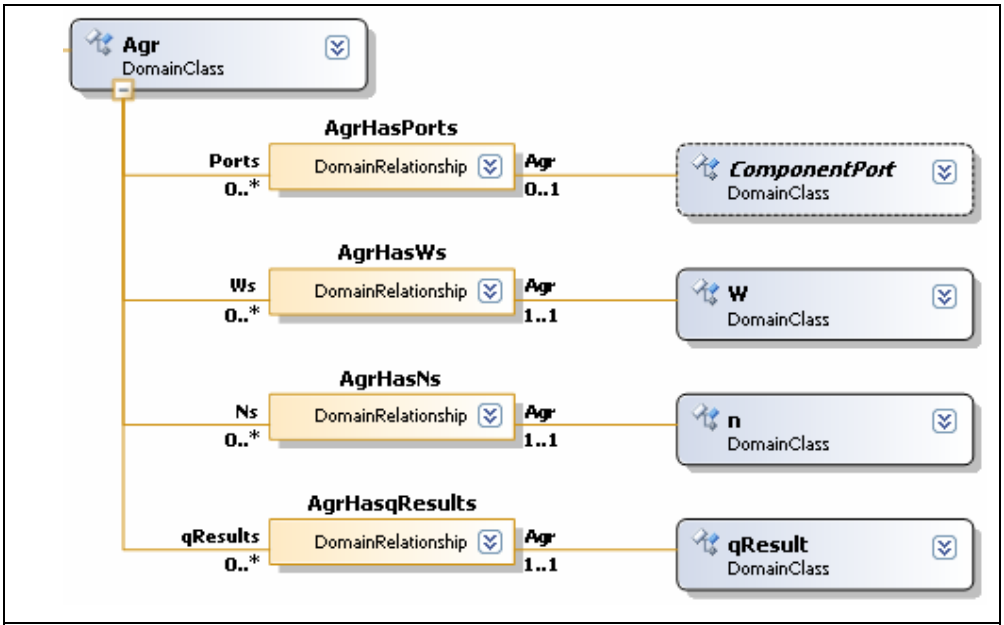
PLAModel klasė pateikta 10 pav.



10 pav. Modelio struktūra

PLA modelį sudaro komentarai, agregatai ir žinutės. Modelis gali turėti bet koki kiekį šių elementų, bet komentarai ir žinutės gali priklausyti tik vienam modeliui, o agregatai gali ir nepriklausyti jokiam.

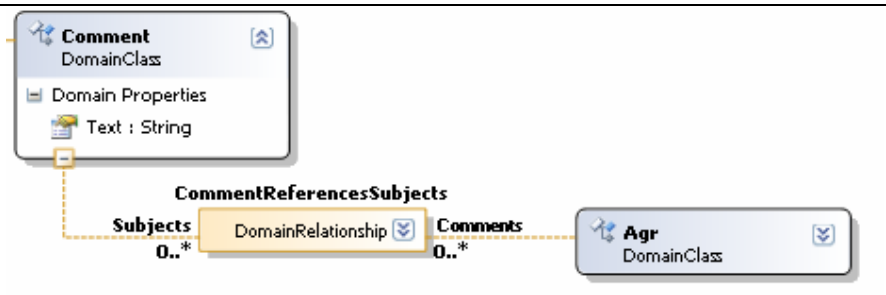
Agr klasė pateikta 11 pav.



11 pav. Agregato struktūra

Agregatas gali turėti bet kokį skaičių jungčių ComponentPort, kontrolinių sumų W, atributų n ir užklausų atsakymų qResult. Visi šie elementai negali priklausyti daugiau nei vienam agregatui, bet jungtis gali egzistuoti ir nepriklausydama jokiam.

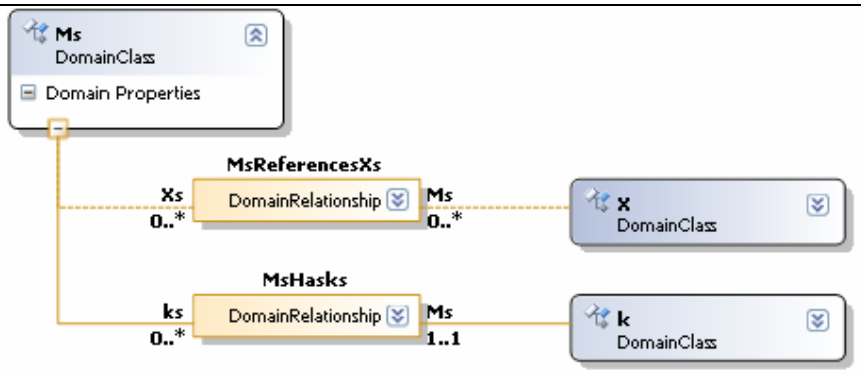
Comment klasė pateikta 12 pav.



12 pav. Komentaro struktūra

Komentaras turi tekstą ir gali turėti komentuojamų agregatų rodykles.

Ms klasė pateikta 13 pav.



13 pav. Žinutės struktūra

Žinutė gali bet koki skaičių k atributų ir rodyklių į įėjimo jungtis X. Nors atributas k ir turi būtinai priklausyti vienai žinutei, X įėjimo jungtis gali būti susieta su bet kokiu kiekiu žinučių.

5.2. Kodo generavimas

Kodas yra generuojamas tekstinio šablono, kuris naudoja modelį, kaip įeities duomenis ir transformuoja jį į kodą. Šablonas gali būti išskaidytas dalimis, kurių kiekviena apibrėžia tam tikrą transformacijos taisyklę.

Taisyklė nr. 1 pavaizduota 14 pav. Pagal ją sukuriamas klasės, kurios pavadinimas atitinka modelio pavadinimą, objektas.

```
<#=this.PLAModel.Name#> model = new <#=this.PLAModel.Name#>();
```

14 pav. Taisyklė nr. 1

Taisyklė nr. 2 pavaizduota 15 pav. Pagal ją kiekvienam modelio agregatui sukuriama po kintamąjį.

```
<# foreach(Agr ag in this.PLAModel.Aggregates) {
string name = ag.Name;
string className = ag.ClassName;
if(string.IsNullOrEmpty(className)) className = name + "Class";
#>
private <#=className#> <#=name#>;
<# } #>
```

15 pav. Taisyklė nr. 2

Taisyklė nr. 3 pavaizduota 16 pav. Pagal ją sukuriamas konstruktorius modelio klasei.

```
public <#=this.PLAModel.Name#>()
```

16 pav. Taisyklė nr. 3

Taisyklė nr. 4 pavaizduota 17 pav. Pagal ją remiantis modeliu sujungiami agregatų išėjimai su atitinkamų agregatų įėjimais.

```
<# foreach(Agr ag in this.PLAModel.Aggregates) {
foreach(ComponentPort port in ag.Ports) {
if(port is Y) {
Y yport = (Y)port;
foreach(X xport in yport.Targets) {
foreach(Ms mess in xport.Ms) {#>
<#=ag.Name#>.<#=yport.Name#>_<#=mess.Name#>.
ConnectTo(<#=xport.Agr.Name#>.<#=xport.Name#>_<#=mess.Name#>);
```

17 pav. Taisyklė nr. 4

Taisyklė nr. 5 pavaizduota 18 pav. Pagal ją kiekvienam agregato N rūšies atributui sugeneruojama po kintamąjį, kurio tipas ir pavadinimas paimami iš modelio.

```
<# foreach(n _n in ag.Ns) {#>
private <#=_n.Type#> <#=_n.Name#>;
<# } #>
```

18 pav. Taisyklė nr. 5

Taisyklė nr. 6 pavaizduota 19 pav. Pagal ją kiekvienam agregato qResult rūšies atributui sugeneruojama po kintamąjį, kurio tipas ir pavadinimas paimami iš modelio.

```
<# foreach(qResult _qResult in ag.qResults) {#>
private <#=_qResult.Type#> <#=_qResult.Name#>;
<# } #>
```

19 pav. Taisyklė nr. 6

Taisyklė nr. 7 pavaizduota 20 pav. Pagal ją kiekvienai agregato X rūšies (įėjimo) prieigos ir jos žinutės porai sugeneruojama po išorinių įvykių tvarkyklę.

```
<# foreach(ComponentPort port in ag.Ports) {
if(port is X) {
X xport = (X)port;
foreach(Ms mess in xport.Ms) {#>
private ExternalEventHandler <#=xport.Name#>_<#=mess.Name#>;
```

20 pav. Taisyklė nr. 7

Taisyklė nr. 8 pavaizduota 21 pav. Pagal ją kiekvienai agregato Y rūšies (išėjimo) prieigos ir jos žinutės porai sugeneruojama po kintamąjį.

```
<# foreach(ComponentPort port in ag.Ports) {
if(port is Y) {
Y yport = (Y)port;
foreach(X xport in yport.Targets) {
foreach(Ms mess in xport.Ms) {#>
public OutputPoint <#=yport.Name#>_<#=mess.Name#>;
```

21 pav. Taisyklė nr. 8

Taisyklė nr. 9 pavaizduota 22 pav. Pagal ją kiekvienam agregato W rūšies atributui sukuriama po ControlSum ir ControlSequence tipo kintamąjį, kurio pavadinimai paimami iš modelio.

```
<# foreach(W _w in ag.Ws) {#>
private ControlSum <#=_w.Name#>;
<# if(!string.IsNullOrEmpty(_w.ControlSequence)){#>
private ControlSequence <#=_w.ControlSequence#>;
<# }#>
```

22 pav. Taisyklė nr. 9

Taisyklė nr. 10 pavaizduota 23 pav. Pagal ją kiekviena agregato įėjimo prieiga sujungiama su atitinkamomis įvykių tvarkyklėmis

```
<# foreach(ComponentPort port in ag.Ports) {
if(port is X) {
X xport = (X)port;
foreach(Ms mess in xport.Ms) {#>
<#=xport.Name#>_<#=mess.Name#> = Handle<#=xport.Name#>_<#=mess.Name#>;
```

23 pav. Taisyklė nr. 10

Taisyklė nr. 11 pavaizduota 24 pav. Pagal ją kiekviena agregato išeinanti prieiga sujungiama su atitinkamais išėjimo taškais.

```
<# foreach(ComponentPort port in ag.Ports) {
if(port is Y) {
Y yport = (Y)port;
foreach(X xport in yport.Targets) {
foreach(Ms mess in xport.Ms) {#>
<#=yport.Name#>_<#=mess.Name#> = new OutputPoint();
```

24 pav. Taisyklė nr. 11

Taisyklė nr. 12 pavaizduota 25 pav. Pagal ją kiekvieną agregato W rūšies atributą atitinkantis kintamasis inicializuojamas su atitinkama funkcija.

```
<# foreach(W _w in ag.Ws) {#>
<#=_w.Name#> = new ControlSum(Handle<#=_w.Name#>);
<# } #>
```

25 pav. Taisyklė nr. 12

Taisyklė nr. 13 pavaizduota 26 pav. Pagal ją kiekvieną agregato W rūšies atributą atitinkantis kintamasis inicializuojamas su atitinkama reikšme.

```
<# foreach(n _n in ag.Ns) {#>
<#=_n.Name#> = <#=_n.InitialValue#>;
<# } #>
```

26 pav. Taisyklė nr. 13

Taisyklė nr. 14 pavaizduota 27 pav. Pagal ją kiekvienas kintamasis atitinkantis, agregato X rūšies prieigą, inicializuojamas atitinkama funkcija.

```
<# foreach(ComponentPort port in ag.Ports) {
if(port is X) {
X xport = (X)port;
foreach(Ms mess in xport.Ms) {#>
public void Handle<#=xport.Name#>_<#=mess.Name#>(AgEventArg av, double tm)
```

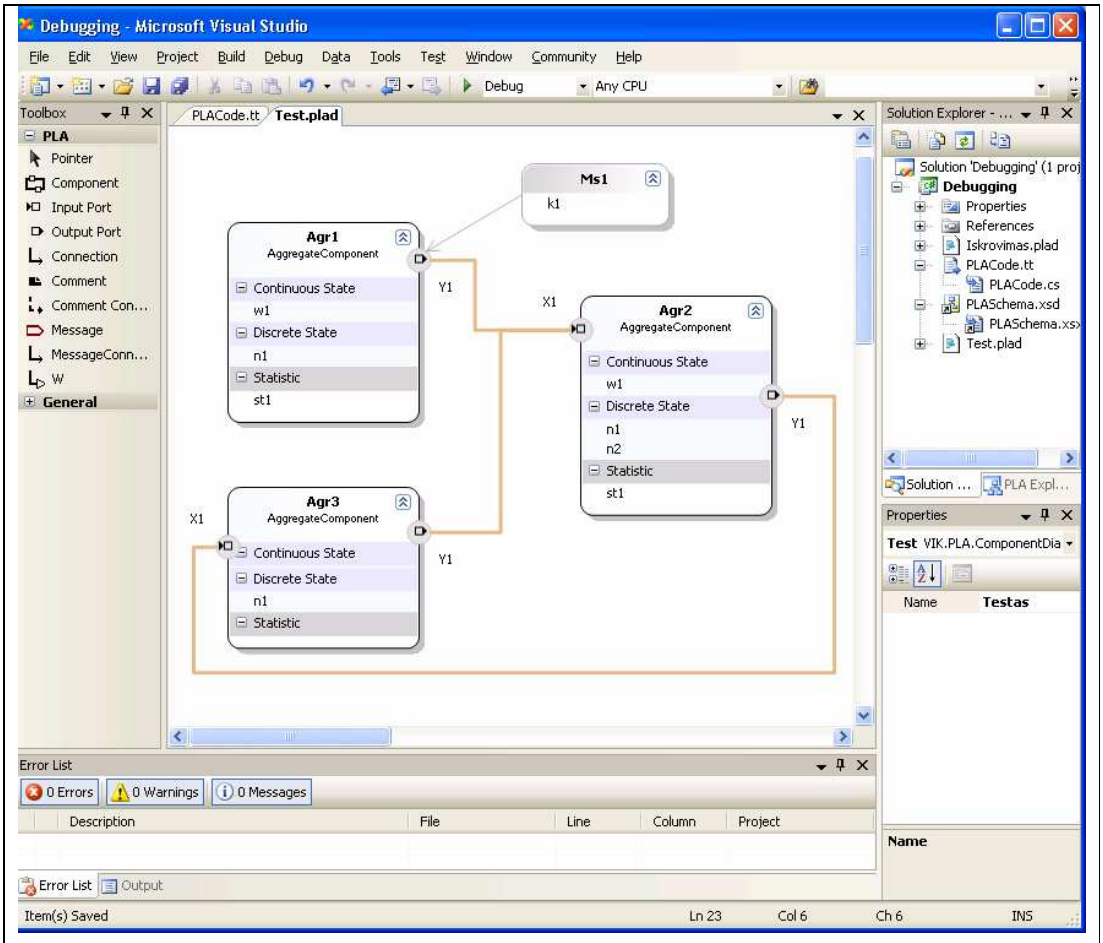
27 pav. Taisyklė nr. 14

Taisyklė nr. 15 pavaizduota 28 pav. Pagal ją Pagal ją kiekvienas kintamasis, atitinkantis agregato W rūšies atributą, inicializuojamas atitinkama funkcija.

```
<# foreach(W _w in ag.Ws) {#>
public void Handle<#=_w.Name#>(AgEventArg av, double tm)
```

28 pav. Taisyklė nr. 15

Žemiau pateiktas agregatinio imitacinio modelio grafinė specifikacija ir iš jos sugeneruotas imitacinio modelio programinis kodas.



29 pav. Agregatinio imitacinio modelio grafinė specifikacija

```
using System;
using System.Collections.Generic;
using System.Text;
using PLASimulation;

namespace PLASimulationTest
{
    class Program
    {
        static void Main(string[] args)
        {
            Testas model = new Testas();
            model.Initialize();
            model.StartSimulation();
        }
    }
```

```
    public class Testas : SimulationModel
    {
               private Agr1Class Agr1;
               private Agr2Class Agr2;
               private Agr3Class Agr3;

        public Testas()
        {
                       Agr1 = new Agr1Class();
                       Agr2 = new Agr2Class();
                       Agr3 = new Agr3Class();

                       Agr1.Y1_Ms1.ConnectTo(Agr2.X1_Ms1);
                       Agr2.Y1.ConnectTo(Agr3.X1);
                       Agr3.Y1.ConnectTo(Agr2.X1);
        }

        public override void Initialize()
        {
            base.Initialize();
        }
    }
    public class Ms1 : AgMessage
    {
               public int k1;
    }
        public class Agr1Class: Aggregate
        {
               private int n1;
               public OutputPoint Y1_Ms1;
               private ControlSum w1;
               private ControlSequence lamda;
               private StatD st1;

               public Agr1Class():base()
               {
                       Y1_Ms1 = new OutputPoint();
                       w1 = new ControlSum(Handlew1);
                       lamda = new ControlSequence();
                       st1 = new StatD("Testas");
               }
               public override void Initialize()
               {
                       n1 = 0;
                       w1.Initialize();
                       st1.Initialize();
               }
               public void Handlew1(AgEventArg av, double tm)
               {
               }
        }
        public class Agr2Class: Aggregate
        {
               private int n1;
               private int n2;
               private ExternalEventHandler X1_Ms1;
               private ExternalEventHandler X1;
               public OutputPoint Y1;
               private ControlSum w1;
               private StatD st1;

               public Agr2Class():base()
               {
                       X1_Ms1 = HandleX1_Ms1;
                       X1 = HandleX1;
                       Y1 = new OutputPoint();
                       w1 = new ControlSum(Handlew1);
                       st1 = new StatD("");
               }
               public override void Initialize()
               {
                       n1 = 0;
                       n2 = 0;
                       w1.Initialize();
                       st1.Initialize();
               }
               public void HandleX1_Ms1(AgEventArg av, double tm)
               {
               }

               public void HandleX1(AgEventArg av, double tm)
               {
               }

               public void Handlew1(AgEventArg av, double tm)
               {
               }
        }
        public class Agr3Class: Aggregate
        {
```

```
                private int n1;
                private ExternalEventHandler X1;
                public OutputPoint Y1;

                public Agr3Class():base()
                {
                        X1 = HandleX1;
                        Y1 = new OutputPoint();
                }
                public override void Initialize()
                {
                        n1 = 0;
                }
                public void HandleX1(AgEventArg av, double tm)
                {
                }

        }
}
```

30 pav. Sugeneruoto imitacinio modelio programinio kodo pavyzdys

6. Išvados

1. Analizė parodė, kad DSL Tools yra tinkamesnis įrankis agregatinių modelių transformacijai atlikti, nei įrankiai realizuojantys MDA architektūrą.
2. Sukurtas grafinės agregatinių imitacinių modelių specifikacijos kalbos metamodelis ir sugeneruota darbo su imitacinio modelio specifikacija aplinka turi užsibrėžtus funkcionalumus.
3. Sukurtos imitacinio modelio specifikacijos transformacijos į imitacinio modelio programinį kodą realizuoja pilną programinio kodo generaciją išskyrus agregatų perėjimo operatorių realizacijas.

7. Naudota literatūra

1. Creating Domain-Specific Languages. 2007 [žiūrėta 2007-05-01]
   Prieiga per internetą: http://msdn2.microsoft.com/en-us/library/bb126259(VS.80).aspx
2. Model-driven engineering. 2007 [žiūrėta 2007-03-14].
   Prieiga per internetą: http://en.wikipedia.org/wiki/Model_Driven_Engineering
3. Overview of Domain-Specific Language Tools. 2007 [žiūrėta 2007-03-08]
   Prieiga per internetą: http://msdn2.microsoft.com/en-us/library/bb126327(VS.80).aspx Unified Modeling Language. 2007 [žiūrėta 2007-04-24]
   Prieiga per internetą: http://en.wikipedia.org/wiki/Unified_Modeling_Language
4. Visual Studio 2005 Team System Modeling Strategy and FAQ. 2005 [žiūrėta 2007-01-20].
   Prieiga per internetą: http://msdn2.microsoft.com/en-us/library/ms379623(VS.80).aspx