KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
INFORMACIJOS SISTEMŲ KATEDRA

Linas Ablonskis

Programos kodo generavimas naudojant UML veiksmų semantiką

Magistro darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
I.Mickienė

Vadovė
doc. dr. Lina Nemuraitė

Recenzentas
doc. dr. V. Pilkauskas

Atliko
IFM-0/1 gr. stud.
Linas Ablonskis

Kaunas, 2006

TURINYS

1 ĮVADAS....1
2 GALIMYBIŲ GENERUOTI PROGRAMOS KODĄ IŠ UML VEIKSMŲ SEMANTIKOS ANALIZĖ....4
2.1 MDA ir programos kodo generavimas....4
2.1.1 Modeliavimo procesas, naudojant UML kalbą....4
2.1.2 Programos kodo generavimas pagal MDA, naudojant UML....5
2.1.3 Programos kodo generavimas be UML....5
2.2 Egzistuojančių programos kodo generatorių bei generavimo metodų analizė....6
2.2.1 Įrankiai, generuojantys programos kodą iš UML klasių diagramų....6
2.2.2 Įrankiai, generuojantys programos kodą iš UML būsenų diagramų....6
2.2.3 Įrankiai, generuojantys programos kodą iš UML veiklos diagramų....7
2.2.4 Kodo generavimas iš kitų rūšių UML diagramų....8
2.3 UML veiksmų ir veiklų tinkamumo programos kodui generuoti analizė....9
2.3.1 UML standarto evoliucija....9
2.3.2 UML veiksmų ir veiklų ryšys....9
2.3.3 UML veiksmų tipai....10
2.3.4 UML veiklų specifikacijoje aprašyti elementai....18
2.4 Programos kodo generavimo iš UML veiksmų semantikos ir kitų UML elementų palyginimas....19
2.5 Analizės išvados....20
3 METODO, PROGRAMOS KODUI IŠ UML VEIKSMŲ SEMANTIKOS GENERUOTI, APRAŠYMAS....22
3.1 Individualaus UML veiksmo išraiška programos kodu....22
3.2 Lygiagretumas UML veiklose....23
3.3 Apribojimai nuosekliu programinių kodu išreiškiamoms UML veikloms....24
3.4 Lygiagrečios UML veiklos išreiškimo nuosekliu programiniu kodu metodas....24
3.5 Rekursyvios išraiškos turinio išreiškimo nuosekliu kodu metodas....26
3.6 ConditionalNode bei Clause ypatybės....26
3.7 Apibendrintas metodą realizuojantis algoritmas....26
4 APRAŠYTO METODO ILIUSTRACIJA PRAKTINIU PAVYZDŽIU....28
4.1 Modeliuojamos sistemos klasių diagrama....28
4.2 Veiklos, realizuojančios modeliuojamos sistemos metodus....29
4.3 UML veiklų vykdymo grafų sudarymas ir būtino veiksmų vykdymo eiliškumo nustatymas....32
4.4 UML elementų kontekstų išskyrimo procesas....33
4.5 UML elemento šablonas ir jo parinkimas....34
4.6 UML elementų šablonų panaudojimas programos kodui surinkti....34
4.7 Programos kodo generavimas Java kalba....35
4.7.1 Kodo šablonai....35
4.7.2 Programos kodo surinkimas....39
4.8 Galimybės generuoti pavyzdžio kodą kitomis kalbomis....41
5 EKSPERIMENTINĖS METODO REALIZACIJOS APRAŠYMAS....42
5.1 Naudojamos priemonės....42
5.2 Duomenys, procesas, rezultatai....42
6 IŠVADOS....44
7 LITERATŪRA....45
8 GENERATION OF PROGRAM CODE USING UML ACTION SEMANTICS....47
9 SANTRUMPŲ IR TERMINŲ ŽODYNAS....48
10 I PRIEDAS....49
11 II PRIEDAS....50

PAVEIKSLŲ SĄRAŠAS

1 pav. PIM modelio specializacija.....4
2 pav. UML2 veiklos modelio pavyzdys.....9
3 pav. Ta pati UML2 veikla standartinėje notacijoje. Atkreipkite dėmesį, kad dalis informacijos tampa paslėpta.....9
4 pav. Iškvietimo veiksmai.....11
5 pav. Signalų siuntimo veiksmai.....11
6 pav. Objektų veiksmai.....11
7 pav. Papildomi objektų veiksmai.....12
8 pav. Veiksmai darbui su struktūrinėmis savybėmis.....13
9 pav. Ryšių objektų nuskaitymo veiksmai.....14
10 pav. Ryšio sukūrimo veiksmas.....14
11 pav. Ryšio nuskaitymo veiksmas.....14
12 pav. Veiksmai darbui su ryšiais.....15
13 pav. Tekstinės išraiškos reikšmės nuskaitymo veiksmas.....16
14 pav. Įvykių priėmimo veiksmai.....16
15 pav. Veiksmai darbui su vietiniais kintamaisiais.....17
16 pav. Veiksmas išimčiai sukelti.....17
17 pav. Užbaigto paieškos algoritmo iliustracija.....25
18 Pav. UML veiklos diagrama, iliustruojanti metodą realizuojantį algoritmą.....27
19 pav. Modeliuojamos sistemos klasių diagrama.....28
20 pav. Metodą „MainClass::start“ specifikuojanti veikla.....29
21 pav. Metodą „MainClass::sayHello“ specifikuojanti veikla.....30
22 pav. Metodą „SpeakingClass::sayToWorld“ specifikuojanti veikla.....31
23 pav. Metodą „MainClass::start“ specifikuojančios veiklos vykdymo grafas.....32
24 pav. Metodą „MainClass::sayHello“ specifikuojančios veiklos vykdymo grafas.....32
25 pav. Metodą „SpeakingClass:sayToWorld“ specifikuojančios veiklos vykdymo grafas.....33
26 pav. Duomenų srautų diagrama atspindinti programos kodo generavimo procesą.....42

LENTELIŲ SĄRAŠAS

1 lentelė. Šaltinių kodo generavimui palyginimas.....19
2 lentelė. Elemento šablonui aprašyti naudojamos lentelės forma.....34
3 lentelė. Programos kodo šablonas elementui Class.....35
4 lentelė. Programos kodo šablonas elementui Class.....36
5 lentelė. Programos kodo šablonas elementui Class.....36
6 lentelė. Programos kodo šablonas elementui InitialNode.....36
7 lentelė. Programos kodo šablonas elementui ActivityFinalNode.....36
8 lentelė. Programos kodo šablonas veiksmui CreateObjectAction.....37
9 lentelė. Programos kodo šablonas veiksmui ValueSpecificationAction.....37
10 lentelė. Programos kodo šablonas veiksmui CallOperationAction.....37
11 lentelė. Programos kodo šablonas veiksmui CallOperationAction.....37
12 lentelė. Programos kodo šablonas įeinančiam ActivityParameterNode.....37
13 lentelė. Programos kodo šablonas išeinančiam ActivityParameterNode.....38
14 lentelė. Programos kodo šablonas elementui ObjectFlow.....38
15 lentelė. Programos kodo šablonas veiksmui ReadSelfAction.....38
16 lentelė. Specialus programos kodo šablonas, įvadiniam programos taškui.....38

1 ĮVADAS

Šio darbo **tyrimo sritis** yra programos kodo generavimo iš UML (angl. *Unified Modeling Language*) modelių metodai ir įrankiai, **tyrimo objektas** – kodo generavimo metodas, grindžiamas UML veiksmų semantika (angl. *Action Semantics*).

UML yra kalba ir standartas, skirtas programinės įrangos sistemoms modeliuoti. UML yra glaudžiai susijusi su MDA (angl. *Model Driven Architecture*). MDA yra programinės įrangos sistemų kūrimo paradigma, kurioje kertinį vaidmenį atlieka kuriamos sistemos modelis [6]. Pagal MDA ideologiją, programų kūrimas vyksta aukštesniame abstrakcijos lygmenyje, kuris trumpai apibūdinamas taip: „modelis yra kodas“. Taupant laiką bei siekiant, kad kuriamos programinės įrangos kodas kiek įmanoma labiau atitiktų jos modelį, yra bandomą tą kodą sugeneruoti tiesiai iš modelio [6].

MDA privalumas – modeliai tampa daugkartinio panaudojimo elementais, nes iš vieno modelio galima sugeneruoti keletą sistemos funkcionavimui reikalingų schemų ir specifikacijų įvairiose platformose. Šiuo metu modeliai yra vienintelė priemonė, kuri gali įveikti augantį ne tik pasikartojančio rankinio darbo kiekį, bet ir programavimo technologijų bei kalbų skaičių, kurias turi žinoti kiekvienas programuotojas net ir nedideliam uždaviniui kompiuterizuoti.

Programų sistemos modelį galima aprašyti įvairiais būdais, tačiau šiame darbe koncentruojamasi į programos kodo generavimą iš UML modelių.

Iš modelio sugeneruojamo programinio kodo kiekis skiriasi priklausomai nuo modeliavimo kalbos bei kodo generatoriaus galimybių. Didinant iš modelio sugeneruojamo ir ranka parašomo programinės įrangos kodo santykį, sutaupoma laiko, išvengiama dalies klaidų, didinamas programinės įrangos atitikimas modeliui.

Dauguma šiuolaikinių UML modeliavimo įrankių („*Rational Rose*“, „*MagicDraw UML*“) leidžia sugeneruoti programos klasių skeletus iš jos klasių diagramos. Šiek tiek tobulesni, kodo generavimo prasme, modeliavimo įrankiai („*ArcStyler*“), naudodami būsenų diagramas, leidžia sugeneruoti ir loginius išsišakojimus klasių metodų viduje. Dar kiti įrankiai („*Fujaba*“, „*ArgoUML*“), naudojantys UML1.5 arba UML2 standarto modelius bei veiklos diagramas, leidžia visiškai užpildyti generuojamų metodų vidų. Egzistuoja kodo generatorius „*iCCG*“, kurio gamintojai naudoja specializuotą UML profilį *xUML* (angl. *Executable UML*), leidžiantį sugeneruoti iki 100% modeliuojamos sistemos programinio kodo. Visi minėti įrankiai turi savų trūkumų, kurie analizuojami šiame darbe.

UML veiksmų semantika yra ta UML modelio semantikos dalis, kuri perteikiama

naudojant UML veiksmus (angl. *Action*) ir veiklas (angl. *Activity*). Generuojant programos kodą, UML modelio semantika suprantama, kaip tame modelyje esančių esybių ir jų ryšių įtaka, sugeneruotam programos kodui bei išraiška jame. Terminas „UML veiksmų semantika“ anglų kalboje turi ir papildomą vaidmenį: jis reiškia UML specifikacijos dalį, kuri apibrėžia UML veiksmus bei UML veiklas.

Šio **darbo tikslas** - išanalizuoti galimybes generuoti kaip įmanoma daugiau galutinio programinės įrangos kodo, naudojant UML veiksmų semantiką, bei sukurti metodą tam atlikti. Bandoma sukurti ne universalų programos kodo generatorių, o gerai veikiantį metodą programos kodui iš UML veiksmų semantikos generuoti, tam kad šį metodą būtų galima panaudoti kuriant konkretų programos kodo generatorių. Kaip bus parodyta vėliau, šio darbo rašymo metu egzistavę kodo generavimo iš UML modelių įrankiai arba nenaudojo UML veiksmų semantikos, arba naudojo tik labai nedidelę jos suteikiamų galimybių dalį.

Yra keletas UML standarto versijų, pradedant versija 1.0 ir baigiant versija 2.0 (tarpinės versijos yra 1.3 , 1.4 ir 1.5). Šiame darbe remiamasi UML versija 2.0.

Norint sukurti metodą, kuris leistų generuoti bet kokios programavimo kalbos kodą iš UML veiklų (angl. *Activity*) specifikacijų, teko išnagrinėti tiek programavimo kalbų, tiek UML veiksmų semantikos ypatybes. Pagrindinė problema buvo, kad dauguma programavimo kalbų leidžia aprašyti tik nuosekliai vykdomą programos kodą. Tuo tarpu UML veiklose naudojami lygiagrečiai veikiantys elementai. Todėl pagrindinę metodo dalį sudaro algoritmas, leidžiantis nustatyti būtiną UML veiklų elementų vykdymo eiliškumą nuosekliame vykdymo scenarijuje. Algoritmo veikimas yra paremtas UML veiklų vykdymą atvaizduojančių grafų sudarymu ir analize.

Visas kodo generavimo metodas apima UML modelio klasių metodų specifikavimą veiklomis, veiklų elementų vykdymo eiliškumo nustatymą nuoseklaus vykdymo atveju, modelio elementų kontekstų nustatymą, programos kodo UML veikloms generavimą, klasių skeletų generavimą ir visko sujungimą į vientisą kodą, kuris toliau pasiunčiamas standartiniams apdorojimo įrankiams (kompiliatoriui ir t.t.). Metodą realizuojančio įrankio prototipas buvo sukurtas *Eclipse* platformoje.

Toliau šiame darbe bus nagrinėjamas programos kodo generavimas bendrai bei programos kodo generavimas iš įvairių UML modelio dalių. Bus analizuojami egzistuojantys programos kodą iš UML modelių generuojantys įrankiai bei nurodomos jų silpnosios pusės. Bus pateiktas įvairių UML modelio dalių tinkamumo programos kodui generuoti palyginimas bei aprašytas metodas programos kodui iš UML veiksmų semantikos generuoti. Bus pateiktas iliustracinis to metodo veikimo pavyzdys.

Darbo struktūra:

- Dalyje „*MDA ir programos kodo generavimas*“ yra nagrinėjami su MDA susiję modeliavimo ir programos kodo generavimo aspektai bei programos kodo generavimas bendrai.
- Dalyje „*Egzistuojančių programos kodo generatorių bei generavimo metodų analizė*“ nagrinėjami egzistuojantys programos kodo generavimo iš UML modelių būdai, juos naudojantys įrankiai. Nurodomi tų įrankių ir būdų trūkumai.
- Dalyje „*UML veiklų ir veiksmų tinkamumo programos kodui generuoti analizė*“ nagrinėjamos UML veiklos (angl. *Activity*) ir UML veiksmai (angl. *Action*), jų galimybės perteikti modeliuojamos sistemos elementų elgseną bei tų galimybių ribos. Išskiriami veiksmų ir UML veiklų elementų tipai, tinkami nuoseklaus programinio kodo generavimui.
- Dalyje „*Programos kodo generavimo iš UML veiksmų semantikos ir kitų UML elementų palyginimas*“ yra pateikiamas UML veiksmų semantikos ir kitų UML elementų palyginimas programos kodo generavimo atžvilgiu, parodoma, kad UML veiksmų semantika yra turtingiausias (iš lyginamų) šaltinis programos kodui generuoti.
- Dalyje „*Metodo, programos kodui iš UML veiksmų semantikos generuoti, aprašymas*“ yra pateikiamas šiame darbe sukurtas metodas, programos kodui iš UML veiksmų semantikos generuoti.
- Dalyje „*Aprašyto metodo iliustracija praktiniu pavyzdžiu*“ yra pateikiamas praktinis metodo panaudojimo pavyzdys ir papildomai paaiškinamos kai kurios to metodo vietos.
- Dalyje „*Eksperimentinės metodo realizacijos aprašymas*“ yra pateikiamas trumpas eksperimentinės metodo realizacijos aprašymas bei paaiškinimas.

Dalis šiame darbe sukurto metodo, programos kodui iš UML veiksmų semantikos generuoti, buvo pristatyta konferencijoje, kurios leidinyje buvo išspausdintas atitinkamas straipsnis. Kitas straipsnis, anglų kalba, buvo priimtas į tarptautinę konferenciją „DB&IS 2006“. Straipsnių kopijos pateikiamos prieduose.

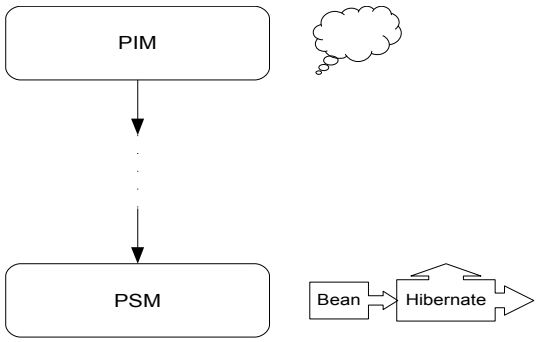
2 GALIMYBIŲ GENERUOTI PROGRAMOS KODĄ IŠ UML VEIKSMŲ SEMANTIKOS ANALIZĖ

2.1 MDA ir programos kodo generavimas

2.1.1 Modeliavimo procesas, naudojant UML kalbą

Pagal MDA, programinės įrangos kūrimo metu sudaromus UML modelius galima suskirstyti į tris grupes [6]:

- CIM (angl. *Computation Independent Model*). Tai neformalus modelis, kuris apibūdina modeliuojamos sistemos aplinką bei jai keliamus reikalavimus. Šis modelis programos kodo generavimo procese nenaudojamas.
- PIM (angl. *Platform Independent Model*). Tai formalus modelis, kuriame sistema apibūdinama nepriklausomai nuo realizacijos technologijos. Šiame modelyje apibrėžiama sistemos architektūra, loginė struktūra, komponentai bei jų sąveika manipuliacijų duomenimis lygyje.
- PSM (angl. *Platform Specific Model*). Tai PIM modelis, papildytas arba transformuotas taip, kad jame būtų atspindima tam tikrai realizacijos platformai arba jų aibei būdinga informacija.



1 pav. PIM modelio specializacija

UML modelių klasifikacija į PIM ir PSM modelius dažnai daroma ir šiek tiek kitokiu principu nei minėtas anksčiau. PIM modeliu vadinamas abstraktesnis formalus sistemos modelis, o PSM modeliu vadinamas labiau specializuotas formalus sistemos modelis, gautas iš PIM modelio [6]. Todėl modelis, kuris yra PSM vieno modelio atžvilgiu, gali būti PIM kito modelio atžvilgiu.

2.1.2 Programos kodo generavimas pagal MDA, naudojant UML

Teoriškai, siekiant generuoti programos kodą, PIM modelis yra transformuojamas į PSM modelį, o PSM modelis transformuojamas į programos kodą [6]. PSM modeliai gali būti transformuojami ir specializuojami keliais žingsniais. Kiekvieno žingsnio metu atitinkamas UML modelis tampa vis labiau susietas su konkrečia realizavimo technologija arba jų aibe.

Realizavimo technologija yra sudaryta iš tikslinių (angl. *Target*) programavimo kalbų, kuriomis bus išreikštas programos kodas, aibės bei naudojamų technologijų, bibliotekų ir platformų aibės [6].

Viena iš MDA keliamų idėjų yra, kad UML modelis gali tarnauti kaip abstrakti programinės sistemos specifikacija, iš kurios būtų automatiškai arba pusiau automatiškai sugeneruojamas tos programinės sistemos kodas. Tai leistų išvengti dalies problemų, kylančių norint suderinti paveldėtas (angl. *Legacy*) programas ir naujas technologijas, ar kai reikia tą pačią programą perkelti į kitą platformą. Norint atlikti šiuos darbus, pakaktų paimti pakankamai abstraktų paveldėtos arba pernešamos programinės sistemos UML modelį, padaryti iš jo PSM modelį tikslinei platformai (angl. *Target platform*), sukonfigūruoti generatorių ir gauti programos kodą, veikiantį norimoje aplinkoje.

Norint tai pasiekti reikia, jog UML modelis galėtų pernešti ne tik struktūrinę, bet ir modeliuojamos sistemos elgseną apibūdinančią informaciją nuo platformos nepriklausomu būdu. UML2 specifikaciją buvo stengiamasi pritaikyti šiam poreikiui [4] [5]. UML2 veiksmų semantika yra iš dalies skirta modeliuojamos sistemos elgsenos aprašymui nuo platformos nepriklausomu būdu [13].

2.1.3 Programos kodo generavimas be UML

Nors MDA dažnai siejama su UML, tačiau UML nėra būtina MDA dalis. Programos kodas gali būti generuojamas iš modelio, padaryto bet kokia konkretaus projekto modeliavimo reikmes tenkinančia kalba, kad ir XML (angl. *Extensible Markup Language*) [10].

Pagrindinė pamoka, kurią galima išmokti iš praktinių programos kodo generavimo projektų patirties [10] yra ta, kad bendru atveju kiekvienas projektas (arba programinių produktų linija) yra unikalus, naudoja unikalią realizacijos technologijų kombinaciją ir turi unikalius poreikius. Tai reiškia, kad praktiškai neįmanoma sukurti statiško programos kodo generatoriaus, kuris tenkintų visų projektų poreikius.

Kai kurie projektai naudoja tik struktūrinio programos kodo bei pagalbinių priemonių generavimą, kiti generuoja dalį elgseną nusakančio kodo [10]. Vieniems projektams visapusiškos

UML specifikacijos naudojimas modeliavime būtų bereikalingas sudėtingumo padidinimas, kiti projektai tiesiog negali išsiversti be tam tikros UML dalies naudojimo [10]. Dažnai dalis programos kodo yra sugeneruojama, o dalis parašoma ranka [10].

Yra laikoma, kad programos kodo generatorius turėtų būti pernešamas tarp skirtingų projektų [10], nes jo sukūrimas sueikvoja nemažai laiko. Tai reiškia, kad programos kodo generatorius turėtų pateikti karkasą (angl. *Framework*), kurį būtų galima pritaikyti konkretaus projekto reikmėms. Karkasai būna sudaryti iš aibės komponentų, kurių kiekvienas specializuotas tam tikram darbui atlikti. Todėl šiame darbe buvo bandoma sukurti ne universalų programos kodo generatorių, o gerai veikiantį metodą programos kodui iš UML veiksmų semantikos generuoti. Tą metodą galima naudoti kuriant konkretų programos kodo generatorių.

2.2 Egzistuojančių programos kodo generatorių bei generavimo metodų analizė

2.2.1 Įrankiai, generuojantys programos kodą iš UML klasių diagramų

Šiai kategorijai galima priskirti beveik visus UML modeliavimo įrankius.

Paprasčiausiu atveju, UML klasių diagrama tiesiogiai paverčiama į atitinkamos programinės kalbos klases, perkeliant UML klasių pavadinimus, atributus bei metodus.

Dauguma UML modeliavimo įrankių (*„Rational Rose“*, *„MagicDraw UML“*), atsižvelgia į klasių stereotipus bei sritį, kurioje veiks sugeneruotas kodas. Tada generuojamas klasių kodas papildomas tam tikrai sričiai, pavyzdžiui, EJB (angl. *Enterprise Java Beans*), reikalingais išplėtimais, metodais, pagalbinėmis klasėmis ir t.t.

Tarkime, jeigu klasės stereotipas nurodo, kad UML klasė bus išsaugoma ir įkraunama iš duomenų bazės, o įrankiui nurodyta, kad bus naudojamas konkretus ORP (angl. *Object Relational Persistence*) paketas, tai sugeneruotas klasės kodas papildomas metodais, kurie reikalingi, naudojant tą konkretų ORP paketą, arba sugeneruojamos reikiamos pagalbinės klasės.

Generuojant kodą iš klasių diagramos, gaunamas programos skeletas su pagalbiniais metodais. Programos elgseną aprašantį kodą tenka įrašyti ranka.

Programos kodo generavimas iš UML klasių diagramų duoda mažiausią sugeneruoto ir parašyto ranka kodo santykį lyginant su visais kitais.

2.2.2 Įrankiai, generuojantys programos kodą iš UML būsenų diagramų

Šios kategorijos atstovas yra UML modeliavimo įrankis „*ArcStyler*“ [9]. „*ArcStyler*“

leidžia aprašyti modelio esybių veiklą būsenų diagramomis, kurios atsispindi sugeneruoto programos kodo metoduose kaip loginiai išsišakojimai. Visą kitą tenka užpildyti rankiniu būdu.

Bendru atveju, smulkiausias iš UML būsenų diagramos sugeneruojamo kodo vienetas yra kreipinys į metodą. Yra daug būdų generuoti programos kodą iš būsenų diagramos [1][2][3], tačiau dauguma jų arba naudoja nepilną UML specifikaciją, arba duoda sunkiai skaitomą ir aptarnaujamą kodą [1].

Vienas iš pagrindinių būsenų diagramų trūkumų generuojant kodą yra tas, kad jos leidžia naudoti bet kokią kalbą perėjimų sąlygoms (angl. *Trigger*), saugams (angl. *Guard*), bei įvykiams (angl. *Event*) aprašyti. Tai automatiškai susieja UML modelį su konkrečia programavimo kalba, kuri turi būti suprantama kodo generatoriui arba sutapti su tiksline programavimo kalba. Tai mažina modelio pakartotinio naudojimo ir pernešimo tarp įvairių generavimo įrankių galimybes.

„*ArcStyler*“ įrankis būsenų diagramose naudoja tokią pačią programavimo kalbą, kuria bus išreikštas sugeneruotas programos kodas.

Būsenų diagramos neleidžia sugeneruoti visapusiškai programos veikimą aprašančio kodo, kadangi mažiausias sugeneruojamas vienetas yra kreipinys į metodą. Siekiant tai ištaisyti, kartais UML modelis papildomas tikslinės programavimo kalbos kodo gabalais, kurie įterpiami į sugeneruotą kodą. Tačiau tai UML modelį visiškai susieja su tiksline programavimo kalba ir nelabai kuo skiriasi nuo rankinio programavimo.

Programos kodo generavimas iš UML būsenų diagramų leidžia pasiekti nemažą sugeneruoto ir ranka parašyto kodo santykį, tačiau padaro UML modelius daugiau ar mažiau priklausomus nuo tikslinės platformos.

2.2.3 Įrankiai, generuojantys programos kodą iš UML veiklos diagramų

Šiai kategorijai atstovauja įrankiai „*Fujaba*“ [7], „*ArgoUML*“ [8] bei kodo generatorius „*iCCG“* [14].

UML modeliavimo įrankis „*Fujaba*“ leidžia generuoti visapusiškai programos veikimą aprašantį kodą iš UML veiklos diagramų. Tačiau šiame įrankyje tai pasiekiama, surašius programos kodą atitinkamuose UML veiksmuose. Generavimo metu UML veiksme įrašytas programos kodas tiesiog perkeliamas į UML veiksmą atitinkančią vietą, o veiksmai sujungiami tokiu eiliškumu, kokiu jie nurodyti UML veikloje. Faktiškai tai nelabai kuo skiriasi nuo programos kodo užrašymo rankiniu būdu, tik jis įrašomas UML modelyje.

„*ArgoUML*“ kodo generavimo ir veiklų modeliavimo aspektais nelabai skiriasi nuo „*Fujaba*“, tačiau jis papildomai leidžia naudoti *CreateObjectAction* bei *DestroyObjectAction*

veiksmus. Kadangi likusi kodo dalis įrašoma rankiniu būdu, tai iš papildomos objektų sukūrimo bei sunaikinimo operacijų abstrakcijos naudos beveik nėra.

„*iCCG*“ leidžia generuoti visapusiškai programos veikimą aprašantį kodą, įvairiose tikslinėse kalbose. Tačiau jis supranta tik modelius, parašytus xUML profilyje. Šiame profilyje naudojama „*iCCG*“ kūrėjų nuosava ASL (angl. *Action Semantics Language*) kalba, kuri, jų teigimu, yra paverčiama į UML veiksmus. Informacijos apie naudojamą UML versiją bei konkrečius metodus „*iCCG*“ kūrėjai nepateikia. Pagal savo aprašymą, „*iCCG*“ yra arčiausiai to, ką bandoma daryti šiame darbe.

Programos kodą iš UML veiklos diagramų generuojantys įrankiai arba naudoja tik nežymią UML veiksmų ir veiklų specifikacijos dalį, arba „*iCCG*“ atveju, yra uždari ir detalesnės informacijos apie naudojamus kodo generavimo metodus nepateikia. Todėl yra prasminga sukurti nuosavą programos kodo generavimo iš UML2 veiklų metodą, kuris atliktų daugiau nei metodai, naudojami „*Fujaba*“ bei „*ArgoUML*“.

Programos kodo generavimas iš UML veiklų leidžia nesusieti UML modelio su konkrečia realizacijos technologija [4] bei sugeneruoti visapusiškai programinės įrangos veikimą aprašantį kodą. UML2 veiklų ir veiksmų specifikacijoje apibrėžtos operacijos yra smulkesnės, negu naudojamos būsenų bei sekų diagramose, todėl teoriškai programos kodo generavimas iš UML veiksmų semantikos turėtų duoti didžiausią sugeneruoto ir ranka parašyto kodo santykį.

2.2.4 Kodo generavimas iš kitų rūšių UML diagramų

Specialiais atvejais programos kodą galima generuoti ir iš anksčiau nepaminėtų UML diagramų.

Sekų diagrama yra tinkamas kandidatas programos veikimą aprašančio kodo generavimui dėl savo trivialios struktūros ir nuoseklios natūros. Iš sekos diagramos sugeneruojamas kodas nėra smulkesnis už kreipinį į metodą.

Norint aprašyti detalesnes sekų diagramas, faktiškai į jas tenka surašyti visą programą, kodo generatoriui paliekant tik atitinkamų kodo gabalų sustatymą į reikiamas vietas. Tačiau toks modeliavimo būdas nelabai kuo skiriasi nuo programos rašymo rankiniu būdu.

Iš sekų diagramos sugeneruojamo bei ranka parašomo kodo santykis turėtų būti panašus į sugeneruojamo bei ranka parašomo kodo santykį, gaunamą generuojant kodą iš būsenų diagramų.

2.3 UML veiksmų ir veiklų tinkamumo programos kodui generuoti analizė

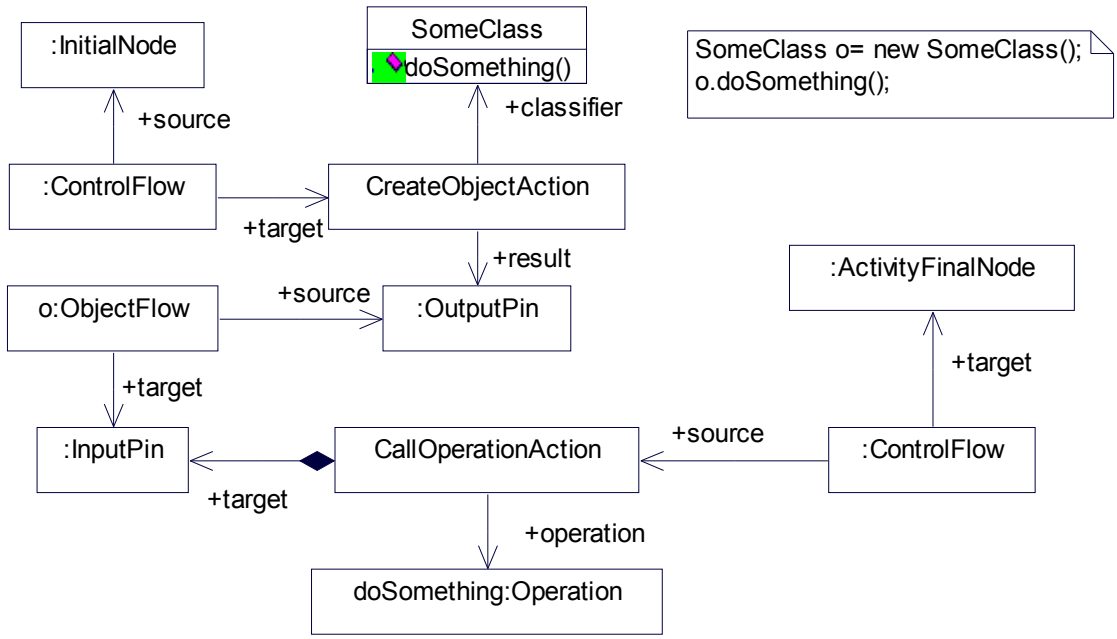
2.3.1 UML standarto evoliucija

Yra keletas UML standarto versijų: UML1 , UML1.3 , UML1.4, UML1.5, UML2.

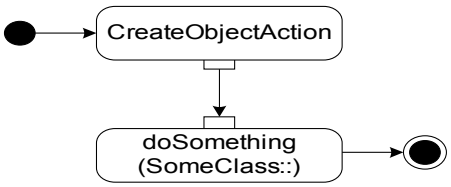
UML1 standarte, UML veiklos buvo tik atskiras būsenų mašinų variantas. UML1.4 ir UML1.5 standartuose atsirado UML veiksmai. UML2 veiklos buvo atskirtos nuo būsenų mašinų, veiksmai buvo išplėsti ir pertvarkyti, siekiant padaryti juos tinkamesnius modelio esybių veikimui aprašyti [13].

2.3.2 UML veiksmų ir veiklų ryšys

UML veiksmas yra mažiausias veikimą aprašantis UML specifikacijos vienetas. UML veikla yra konteineris, kuriame grupuojami ir laikomi UML veiksmai, bei kiti veiklų specifikacijoje aprašyti elementai. UML2 veikla nebūtinai turi turėti grafinę išraišką, tačiau ją galima aprašyti UML2 metamodelio esybėmis bei jų ryšiais.



2 pav. UML2 veiklos modelio pavyzdys



3 pav. Ta pati UML2 veikla standartinėje notacijoje. Atkreipkite dėmesį, kad dalis informacijos tampa paslėpta

UML veiksmas duomenis gauna bei rezultatus pateikia per *Pin* objektus (aliuzija į lusto

kojelę). *Pin* objektai skirstomi į *InputPin* – duomenų įvedimui, *OutputPin* – rezultatų pateikimui bei *ActionInputPin*, kuris yra skirtas rekursyvių išraiškų (angl. *Nested expression*) vykdymui. Kiekvieno *Pin* vaidmenį parodo jo ryšio su UML veiksmu vaidmuo (angl. *Role*). Duomenys tarp *Pin* objektų keliauja per *ObjectFlow* tipo esybes, kurios yra skirtos duomenų fragmentų perdavimui. Valdymo žymes galima perduoti per *ControlFlow* tipo esybes.

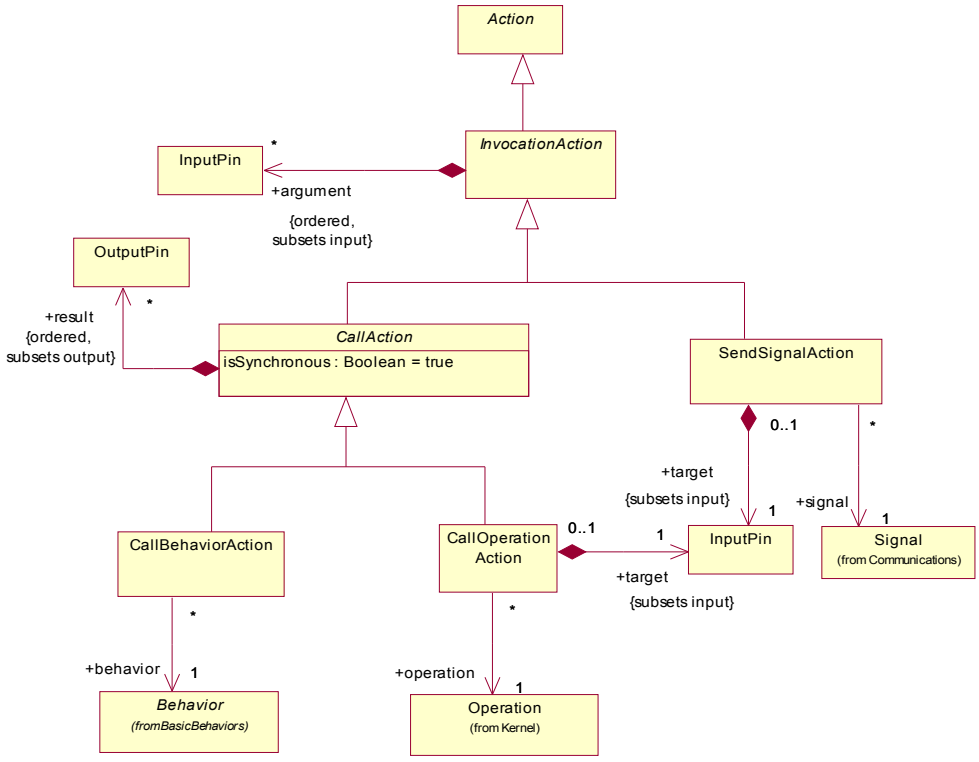
Į UML veiklą iš dalies galima žiūrėti kaip į Petri tinklą, nes joje viskas vyksta Petri tinklui analogiškų žymių pagalba. Žymė gali pernešti duomenų objektą arba valdymo srautą. UML veiksmo vykdymas prasideda tada, kai visuose su juo sujungtuose *InputPin* ir įeinančiuose *ControlFlow* yra po reikiamą žymių kiekį. Baigęs darbą, UML veiksmas sugeneruoja po vieną žymę kiekviename su juo sujungtame *OutputPin* bei išeinančiame *ControlFlow*.

UML veikla yra parametrizuotas konteineris. Tai reiškia, kad UML veikla gali turėti įėjimo bei išėjimo parametrus. Tai padaro ją tinkamą klasių metodų kūnams aprašyti, bei leidžia pakartotinai panaudoti vieną ir tą pačią UML veiklą keliose vietose.

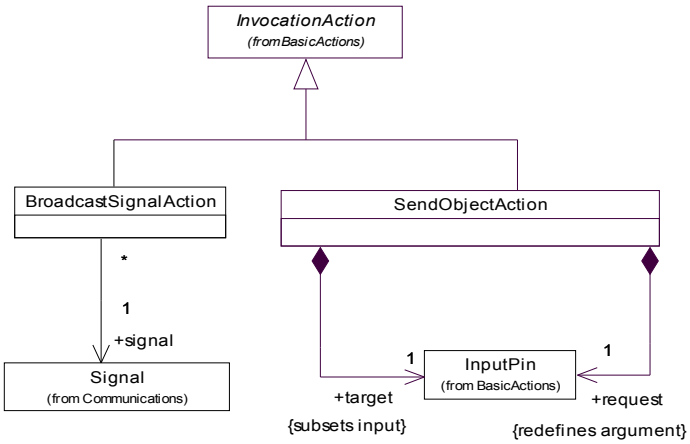
2.3.3 UML veiksmų tipai

UML2 standartas aprašo įvairius UML veiksmų tipus. Skirtingų tipų veiksmai yra skirti skirtingoms programos veikimo aprašyme reikalingoms operacijoms išreikšti:

- Iškvietimo veiksmai – skirti veikimą aprašančioms esybėms (angl. *Behavior*) iškviesti:



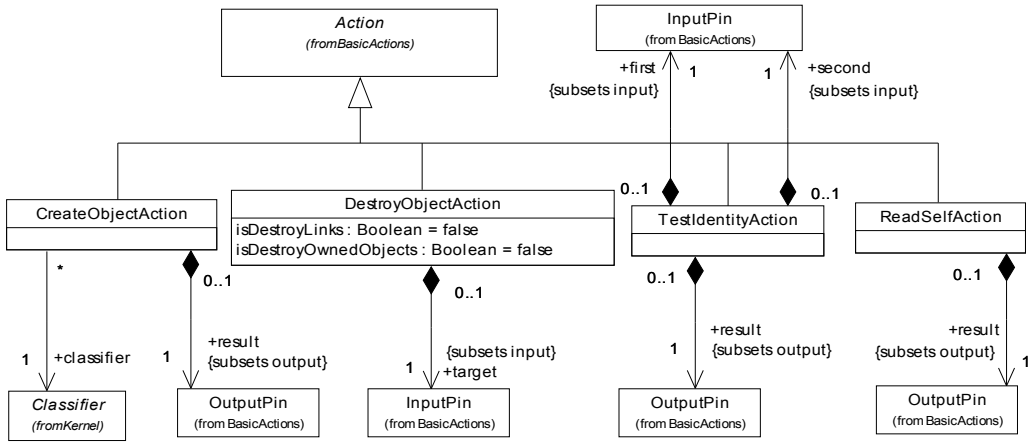
4 pav. Iškvietimo veiksmai



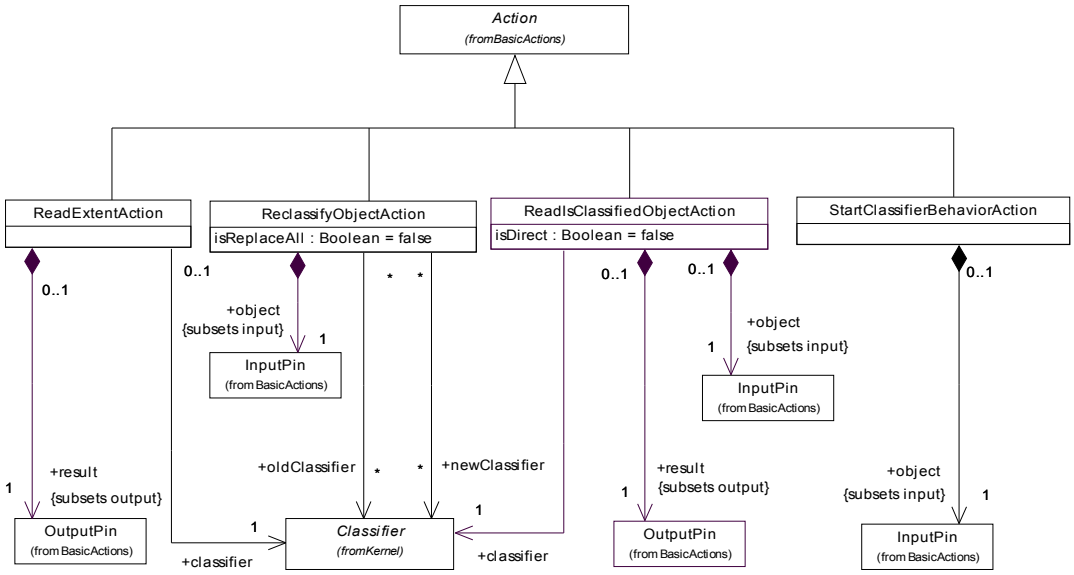
5 pav. Signalų siuntimo veiksmai

- *CallBehaviorAction* – skirtas *Behavior* tipo esybėms aktyvuoti. UML veikla yra *Behavior* potipis.
- *CallOperationAction* – skirtas klasių metodams iškviesti.
- *SendSignalAction* – skirtas signalams siųsti.
- *BroadcastSignalAction* – skirtas signalams transliuoti.
- *SendObjectAction* – skirtas siųsti signalui, kurio turinys yra objektas.

- Veiksmai darbui su OOP (angl. *Object Oriented Programming*) stiliaus objektais:



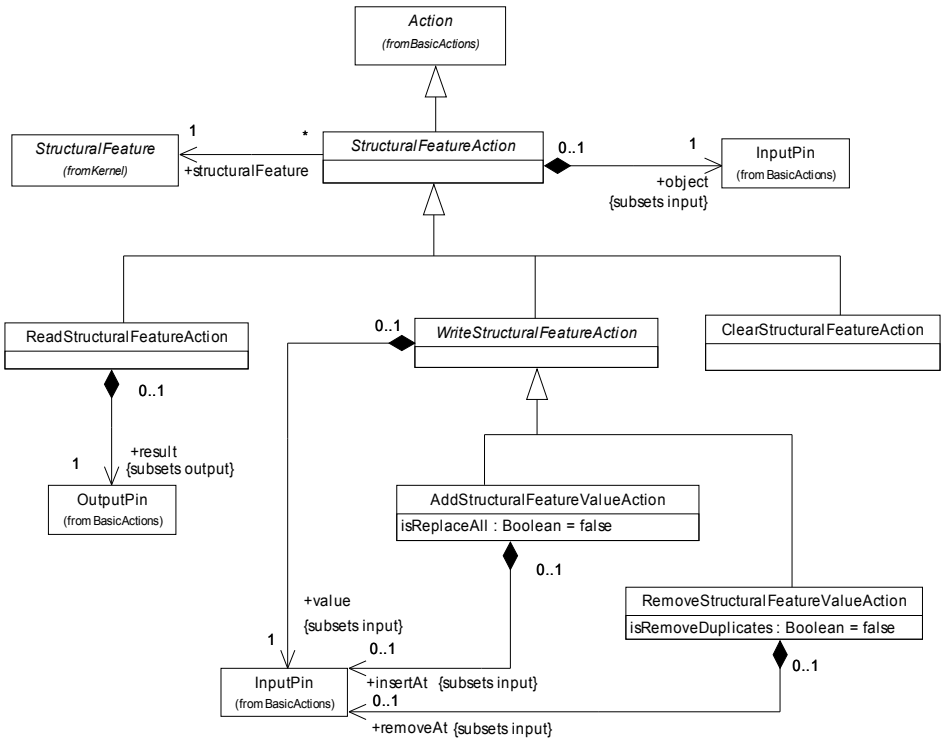
6 pav. Objektų veiksmai



7 pav. Papildomi objektų veiksmai

- *CreateObjectAction* – skirtas naujam klasifikatoriaus egzemplioriui sukurti.
- *DestroyObjectAction* – skirtas objektui sunaikinti.
- *TestIndentityAction* – skirtas nustatyti, ar du objektai yra identiški.
- *ReadSelfAction* – skirtas gauti nuorodai į objektą, kurio kontekste vykdoma veikla.
- *ReadExtentAction* – skirtas gauti visiems klasifikatoriaus egzemplioriams kokiame nors kontekste.
- *ReclassifyObjectAction* – skirtas pakeisti (angl. *To cast*) objekto tipą į kokį nors suderinamą tipą.
- *ReadIsClassifiedObjectAction* – skirtas nustatyti, ar objektas yra kokio nors klasifikatoriaus egzempliorius.
- *StartClassifierBehaviorAction* – skirtas objekto veikimą aprašančiam *Behavior* iššaukti. Objekto veikimą galima aprašyti naudojant *Behavior* tipo esybę, kuri sujungta su klasifikatoriumi naudojant „*classifier*“ vaidmenį (angl. *Role*). Tuo pačiu tokia *Behavior* tipo esybė atliktų klasės konstruktoriaus vaidmenį, nes pagal UML specifikaciją ji būtų iššaukiama kiekvieną kartą, sukūrus naują klasės egzempliorių [4].

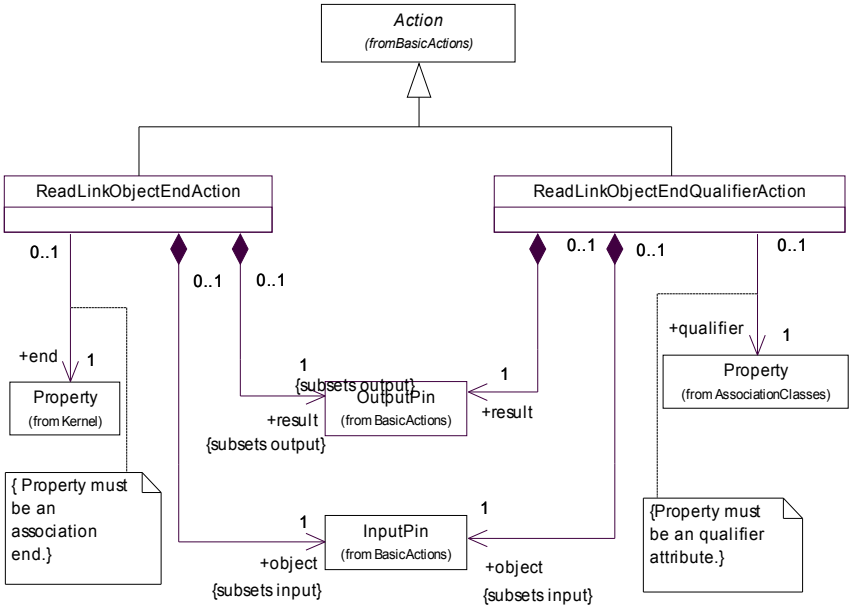
- Veiksmai, skirti darbui su klasių atributais:



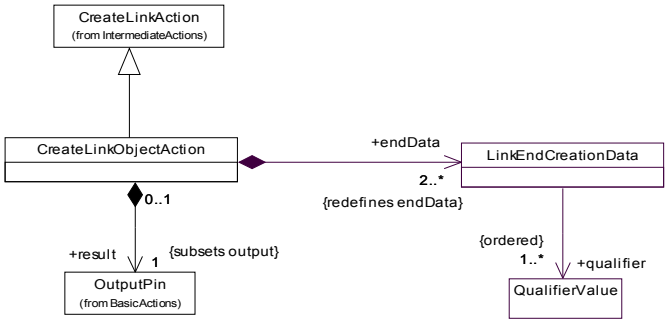
8 pav. Veiksmai darbui su struktūrinėmis savybėmis

- *ReadStructuralFeatureAction* – skirtas nuskaityti klasės atributo reikšmei.
- *AddStructuralFeatureValueAction* – skirtas pridėti naujai reikšmei į daugianarį klasės atributą arba pakeisti senai reikšmei.
- *RemoveStructuralFeatureValueAction* – skirtas pašalinti reikšmei iš daugianario klasės atributo.
- *ClearStructuralFeatureAction* – skirtas pašalinti visoms klasės atributo reikšmėms.

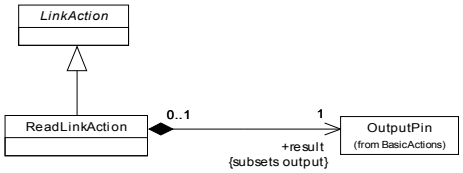
- Veiksmai, skirti darbui su ryšiais (angl. *Link*). Ryšys yra asociacijos (angl. *Association*) egzempliorius. Tie veiksmai yra:



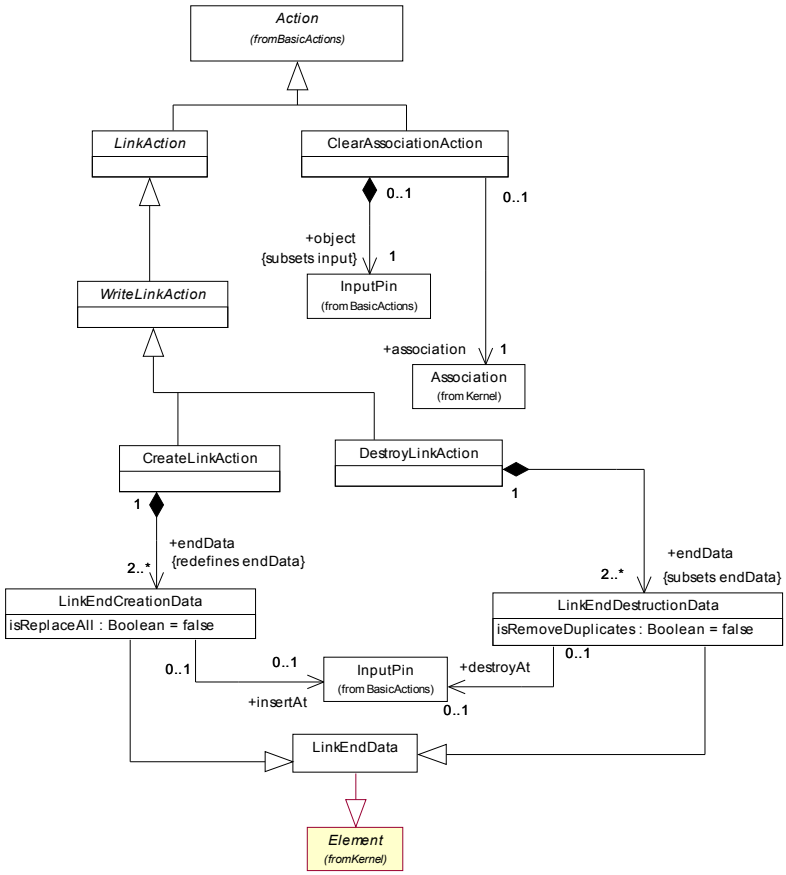
9 pav. Ryšių objektų nuskaitymo veiksmai



10 pav. Ryšio sukūrimo veiksmas



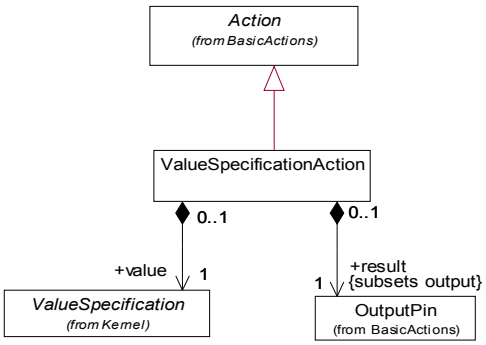
11 pav. Ryšio nuskaitymo veiksmas



12 pav. Veiksmai darbui su ryšiais

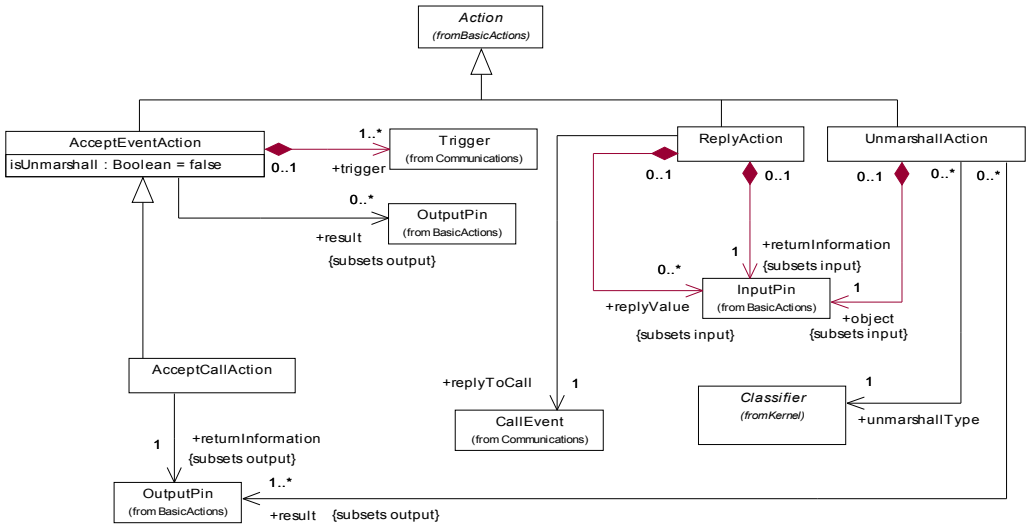
- *ReadLinkAction* – skirtas nuskaityti ryšio reikšmei.
- *ClearAssociationAction* – skirtas sunaikinti visiems asociacijos ryšiams.
- *CreateLinkAction* – skirtas sukurti naujam ryšiui arba senam perrašyti.
- *DestroyLinkAction* – skirtas ryšiui sunaikinti.
- *ReadLinkObjectEndAction* – skirtas kuriame nors sąsajos gale esančiam objektui gauti.
- *ReadLinkObjectEndQualifierAction* – skirtas nustatyti kuriame nors ryšio gale esančio objekto tipui.
- *CreateLinkObjectAction* – skirtas naujam asociacijos klasės egzemplioriui sukurti.

- *ValueSpecificationAction*, kuris yra skirtas tekstinėms išraiškoms interpretuoti ir rezultatui nuskaityti.



13 pav. Tekstinės išraiškos reikšmės nuskaitymo veiksmas

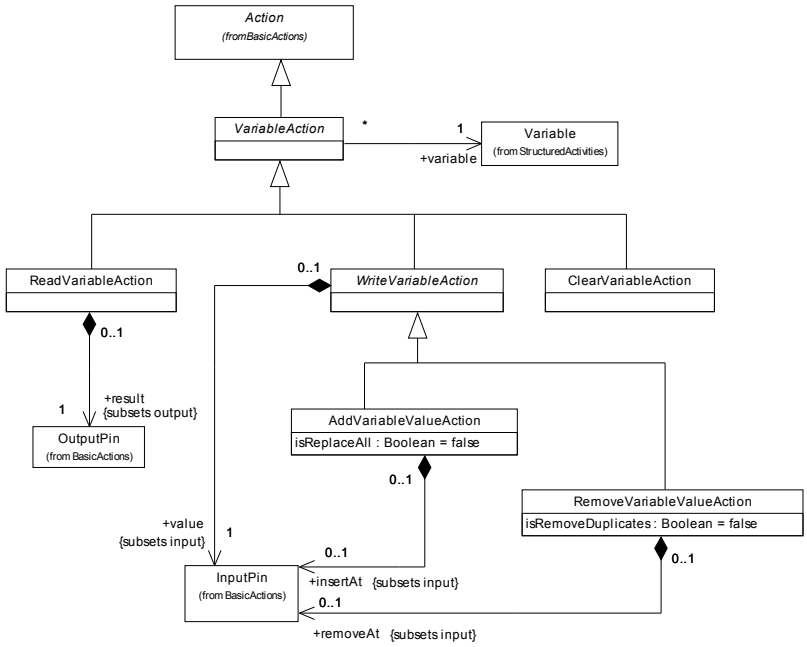
- Veiksmai, skirti įvykiams priimti:



14 pav. Įvykių priėmimo veiksmai

  - *AcceptEventAction* – skirtas įvykiui (angl. *Event*) arba signalui (angl. *Signal*) priimti.
  - *AcceptCallAction* – skirtas asinchroniniam iškvietimui priimti.
  - *ReplyAction* – skirtas asinchroninio iškvietimo rezultatui siųsti.
  - *UnmarshalAction* – skirtas signalui išskleisti į kokio nors klasifikatoriaus atributus.

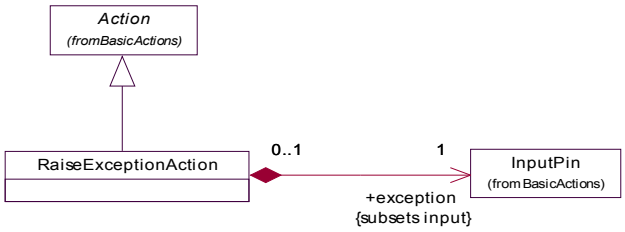
- Veiksmai, skirti darbui su vietiniais veiklos kintamaisiais (angl. *Variable*):



15 pav. Veiksmai darbui su vietiniais kintamaisiais

- *ReadVariableAction* – skirtas kintamojo reikšmei nuskaityti.
- *AddVariableValueAction* – skirtas pridėti naujai daugianario kintamojo reikšmei, arba senai reikšmei pakeisti.
- *RemoveVariableValueAction* – skirtas pašalinti daugianario kintamojo reikšmei.
- *ClearVariableAction* – skirtas pašalinti visoms kintamojo reikšmėms.

- *RaiseExceptionAction* skirtas išimčiai (angl. *Exception*) sukelti. Išimtis yra pagaunama pirmos veiklos arba *StructuredActivityNode*, kuri supa ją sukėlusį *RaiseExceptionAction* ir yra susieta su išimties tipą atitinkančia *ExceptionHandler* tipo esybe.



16 pav. Veiksmas išimčiai sukelti

Iš UML veiksmų tipų ir apibūdinimų galima pamatyti, kad UML2 standarte yra visi veiksmai, reikalingi darbui su klasėmis, objektais bei kintamaisiais. Galima sukurti naujus objektus, pašalinti senus, skaityti ir rašyti kintamuosius bei atributus, kviesti metodus, siųsti ir priimti signalus bei sukelti išimtis [4]. Signalų siuntimo ir priėmimo veiksmai leidžia aprašyti nutolusių procesų sąveiką.

UML veiksmai gali būti panaudoti modeliuojamų programinės įrangos esybių veikimui aprašyti, nepriklausomai nuo tų esybių realizacijos technologijos.

UML standarte nėra veiksmų, skirtų aritmetinėms operacijoms vykdyti, teksto eilutėms, datoms manipuliuoti ir pan. [4] Tai reiškia, kad šių operacijų išraiška bei atlikimo būdas yra palikti įrankių kūrėjų nuožiūrai.

2.3.4 UML veiklų specifikacijoje aprašyti elementai

UML2 veiklų specifikacija buvo kuriama siekiant patenkinti dviejų sričių poreikius – programų inžinerijos bei veiklos procesų inžinerijos [12].

Programų inžinerijai skirti dariniai leidžia aprašyti algoritmus. Dauguma jų atitinka darinius, kuriuos galima surasti kiekvienoje modernioje programavimo kalboje. Programų inžinerijai skirti dariniai yra:

- *ObjectFlow* – skirtas duomenų srauto žymėms perduoti.
- *ControlFlow* – skirtas valdymo srauto žymėms perduoti.
- *ActivityParameterNod*e – skirtas veiklos parametrams paversti duomenų žymėmis.
- *InitialNode* – skirtas generuoti pradinėms žymėms, kurios pradeda veiklos vykdymą.
- *ActivityFinalNode* – skirtas veiklai arba esamai *StructuredActivityNode* užbaigti.
- *StructuredActivityNode* – atlieka struktūrinių skliaustų paskirtį.
- *ConditionalNode* ir *Clause* – skirti loginiams išsišakojimams aprašyti.
- *LoopNode* – skirtas ciklams aprašyti.
- *SequenceNode* – skirtas vykdymo sekai nurodyti
- *ExceptionHandler* – skirtas išimtims sugauti ir apdoroti.
- *ExpansionRegion* ir *ExpansionNode* – skirti perėjimui per kolekcijos elementus.

Veiklos procesų inžinerijai skirti dariniai leidžia modeliuoti veiklos procesus. Modeliuojami veiklos procesai nebūtinai turi būti tiesiogiai susiję su programomis. UML2 specifikacijoje naudojamuose pavyzdžiuose yra modeliuojamas fabriko veikimas [4]. Procesų inžinerijai skirti dariniai yra paremti lygiagrečiai keliaujančiais žymių srautais [4]. Tai:

- *ForkNode* – skirtas išskleisti vienam žymių srautui į keletą lygiagrečių srautų.
- *JoinNode* – skirtas sinchronizuoti lygiagretiems žymių srautams.

- *MergeNode* – skirtas praleisti bet kuriam iš įeinančių žymių srautų.
- *DecisionNode* – skirtas loginiam išsišakojimui sukurti.
- *DataStoreNode* – skirtas duomenų saugyklos ekvivalentui nurodyti.
- *CentralBufferNode* – skirtas laikinam žymių kaupimui.
- *InterruptibleActivityRegion* – skirtas sukurti regionui, kurio viduje esanti veikla nutraukiama, jeigu iš to regiono per tam tikrą srautą iškeliauja žymė.
- *ObjectNode* su išplėtimais, skirtais žymėms kaupti ir išrinkti.
- *FlowFinalNode* – skirtas atskirų srautų žymėms sunaikinti.
- Anksčiau paminėti programų inžinerijos dariniai, su atitinkamais išplėtimais: *ActivityEdge* ant kurių uždėti saugai (angl. *Guard*) bei svoriai; *ObjectFlow* esybės su žymių kvotomis bei jų išrinkimo elgesiu; veiksmai su pridėta žymių srautų (angl. *Stream*) apdorojimui skirta semantika ir t.t.
- *AcceptEventAction* veiksmas su semantiniu išplėtimu, kuris jam leidžia savaime aktyvuotis, jeigu nėra į jį įeinančių *ControlFlow* esybių. Toks veiksmas sukuria lygiagrečiai su visa kita veiklos dalimi veikiantį žymių srautą.
- Anksčiau paminėti programų inžinerijos dariniai, kurie tinka ir procesų inžinerijai. Tai *InitialNode* ir *ActivityFinalNode*. Šiuos darinius naudoja abiejų sričių veiklos.

2.4 Programos kodo generavimo iš UML veiksmų semantikos ir kitų UML elementų palyginimas

1 lentelė. Šaltinių kodo generavimui palyginimas

Šaltinis kodo generavimui	***UML veiksmų semantika***	***UML klasių diagramos***	***UML būsenų diagramos***	***UML sekų diagramos***
Sugeneruojamas galutinio programos kodo kiekis	Didelis	Mažas	Vidutinis	Vidutinis
Leidžia aprašyti modelio dalių elgseną nepriklausomai nuo realizacijos technologijos	Taip	Ne	Ne	Iš dalies
Leidžia aprašyti lygiagrečiai veikiančius procesus	Taip	Ne	Taip	Taip

Pažvelgus į palyginimo lentelę bei anksčiau išsakytus teiginius, galima pastebėti, kad:

- UML veiksmų semantikos naudojimas leidžia sugeneruoti didesnį galutinio programos

kodo kiekį, nei kitų šaltinių naudojimas.

- UML veiksmų semantikos naudojimas leidžia aprašyti modelio dalių veikimą nepriklausomai nuo realizacijos technologijos, o tai leidžia keisti kodo generavimo proceso rezultatą nekeičiant modelio; arba pakeitimai turi būti mažesni, nei generuojant iš kitų šaltinių.
- UML veiksmų semantika yra geresnis šaltinis programos kodo generavimui už kitus paminėtus.

Palyginime buvo laikoma, kad kodo generavime naudojamose klasių, būsenų ir sekų diagramose nėra įrašyti galutinio programos kodo gabalai, nes tai beveik nesiskiria nuo rankinio programavimo.

UML veiksmų semantiką galima interpretuoti kaip standartizuotą tarpinę kalba (angl. *Intermediate Language*). Ji leidžia modeliuoti, naudojant daug srities kalbų (angl. *Domain Language*), kurių kodo generatoriui nereikia žinoti. Jeigu pažvelgtume į būsenų arba sekų diagramas, tai pamatytume, kad jose naudojama kalba priklauso nuo susitarimo (dažniausiai tai būna ta pati programavimo kalba į kurią bus paverstas UML modelis) ir kodo generatorius ją žinoti privalo.

UML2 būsenų diagramose, būsenų veiksmams „*entry*“, „*do*“, „*exit*“ aprašyti galima naudoti UML veiksmų semantiką. Kodo generavimo iš tokių būsenų diagramų sudėtinė dalis būtų kodo generavimas iš UML veiksmų semantikos. Šio darbo rašymo metu nebuvo rasta tai atliekančių sprendimų.

2.5 Analizės išvados

1. Darbe atlikta kodo generavimo galimybių iš įvairių UML diagramų analizė parodė, kad UML veiksmų semantika yra turtingesnis duomenų šaltinis programos kodo generavimui, negu UML klasių, būsenų ar sekų diagramos.
2. UML veiksmų semantikos analizė parodė, kad ją naudojant klasių metodų specifikavimui galima sugeneruoti visiškai veikiantį galutinį programos kodą, ne tik skeletą.
3. UML veiksmų semantika neapibrėžia operacijų su duomenų tipais (išskyrus objekto tipo nustatymą ir dviejų klasės egzempliorių ekvivalentumo nustatymą), todėl šių operacijų išraiška UML modelyje gali būti pasirenkama laisvai.
4. Atlikta kodą generuojančių CASE įrankių analizė parodė, kad egzistuojantys įrankiai arba nesinaudoja UML veiksmų semantika, arba („*iCCG*“ atveju) neskelbia kodo generavime

naudojamų metodų. Todėl yra tikslinga sukurti nuosavą metodą programos kodui iš UML veiksmų semantikos generuoti.

5. UML veiklų specifikacija yra pritaikyta dviem sritims – programų inžinerijai bei veiklos procesų inžinerijai. Veiklos procesų inžinerijai skirtų darinių veikimas yra paremtas lygiagrečiais žymių srautais, o programų inžinerijos dariniai beveik visiškai atitinka modernioje programavimo kalboje pasitaikančius darinius. Norint iš UML veiklų generuoti nuoseklų programos kodą, reikėtų išvengti procesų inžinerijai skirtų darinių.

3 METODO, PROGRAMOS KODUI IŠ UML VEIKSMŲ SEMANTIKOS GENERUOTI, APRAŠYMAS

Šiame skyriuje pateikiamas sukurtas apibendrintas metodas, kurį galima pritaikyti kodo generatoriuose.

3.1 Individualaus UML veiksmo išraiška programos kodu

Kiekvienas UML veiksmas apibūdina kokią nors abstrakčią primityvią operaciją. UML specifikacijoje yra nurodyta, ką kiekvieno tipo UML veiksmas turi atlikti bei kokius jis gali naudoti duomenis ir veikti objektus, tačiau nenurodyta, kaip UML veiksmai turėtų atsispindėti programos kode.

Konkrečiame programinės įrangos projekte UML veiksmo išraiška programiniame kode priklausys nuo tame projekte naudojamų programavimo kalbų ir realizacijos technologijų. Todėl į UML veiksmą galima žiūrėti kaip į parametrizuotą kodo gabalą, kuriame su tuo UML veiksmu susieti *Pin* tipo objektai atitinka parametrizuotus kintamuosius, o veiksmo tipas apibrėžia reikalavimus to kodo gabalo veikimui.

Vieno tipo UML veiksmas gali veikti keliuose skirtinguose kontekstuose (aplinkose) ir veikti objektus, esančius keliuose skirtinguose kontekstuose. Elemento kontekstas priklauso nuo jo konteinerio konteksto bei paties elemento savybių. Kontekstai statiškai apibrėžiami modeliuojant. Generuojant programos kodą, reikės skirtingų kodo gabalų kiekvienai UML veiksmo konteksto ir jo veikiamo konteksto kombinacijai.

Skirtingus kontekstus gerai iliustruoja metodo iškvietimas – tokio pat tipo UML veiksmas (*CallOperationAction*) gali kviesti tiek metodą esantį tame pačiame procese, tiek nutolusį metodą. Nutolusio metodo kvietimas gali būti atliekamas naudojant „*Java RMI*“ arba naudojant CORBA. Kiekvienu atveju reikės vis kitokio programinio kodo metodo iškvietimo veiksmui realizuoti.

Tai reiškia, kad programos kodo generatoriui turės būti pateikti UML veiksmų aprašai, atspindintys jų išraišką kodu kiekvienai UML veiksmo tipo ir to veiksmo konteksto kombinacijai, kuri pasitaiko modelyje. Taip pat programos kodo generatoriui turės būti pateiktos taisyklės, leidžiančios nustatyti bei įvardinti skirtingus kontekstus. Matematiškai, naudojant aibes, tai būtų išreiškiama taip:

A - UML veiksmas iš UML veiksmų aibės S_A , $A \in S_A$
C - visų UML modelyje esančių kontekstų aibė, $C \neq \{\emptyset\}$

C_h - kontekstas, kuriame yra UML veiksmas, $C_h \in C$

C_t - kontekstas, kuriame yra UML veiksmo veikiami objektai, $C_t \in C$

Tada:

„UML veiksmo išraiška kodu“ V yra $V = V(A, C_h, C_t)$.

3.2 Lygiagretumas UML veiklose

Pagal UML specifikaciją, bet kokia UML veiklos išraiška programiniu kodu yra laikoma priimtina, jeigu pasireiškia visi efektai, kurie pasireikštų modeliuojant tos veiklos veikimą [4]. Šis reikalavimas užtikrina, kad modeliuotojo intencijos bus perkeltos į programinį kodą.

Kaip jau buvo minėta anksčiau, UML veiklų specifikacija yra skirta patenkinti dviejų sričių – programų inžinerijos ir veiklos procesų inžinerijos modeliavimo poreikiams.

Procesų modeliavimui skirtų darinių veikimas yra paremtas lygiagrečiais žymių srautais. Tačiau dauguma šiuolaikinių programavimo kalbų yra vykdomos nuosekliai. Lygiagrečiai veikiančių žymių srautų sąveikoje yra svarbus laiko faktorius. Bendrai paėmus yra neįmanoma išreikšti lygiagrečiai veikiančių žymių srautų nuosekliu programiniu kodu ir išsaugoti visus efektus, kurie buvo numatyti modeliuojant atitinkamos UML veiklos veikimą.

Bendru atveju, tinkamas būdas išreikšti UML veiklą, kurioje yra procesų modeliavimui skirtu darinių, programiniu kodu yra arba vykdant kiekvieną tos veiklos esybę kaip atskirą giją (angl. *Thread*), arba vykdant UML veiklą virtualioje lygiagretumą imituojančioje mašinoje.

Kita vertus, programų inžinerijai skirti UML veiklų dariniai beveik tiesiogiai atitinka darinius, pasitaikančius modernioje nuoseklioje programavimo kalboje. Juos galima sujungti lygiagrečiai, tačiau jų veikimą galima išreikšti ir nuosekliu programos kodu, neiškraipant modeliavimo metu numatytų efektų.

Kai modeliuotojas UML veikloje tyčia arba netyčia sukelia lenktyniavimo sąlygas (angl. *Race condition*), iš UML veiklos sugeneruotame nuosekliame kode būtų gaunamas konkretus tų lenktyniavimo sąlygų variantas, todėl būtų galima teigti, kad modelyje numatytas efektas yra išsaugomas.

UML specifikacija nenurodo jokių reikalavimų UML veiksmų atlikimo trukmei ir nerekomenduoja modeliuojant atsižvelgti į numanomas UML veiksmų atlikimo trukmes [4]. Tai reiškia, kad teisingai sudarytame UML veiklos modelyje lenktyniavimo sąlygų, kylančių dėl UML veiksmų vykdymo trukmės, turėtų nebūti arba jų buvimas turėtų neturėti reikšmės modeliuojamo algoritmo veikimui.

Šiame darbe aprašomame programos kodo generavimo iš UML veiksmų semantikos metode yra siekiama UML veiklas išreikšti nuosekliu programiniu kodu. Todėl, atsižvelgiant į

anksčiau paminėtus dalykus, tenka apriboti tose veiklose pasirodančių esybių tipus.

3.3 Apribojimai nuosekliu programinių kodu išreiškiamoms UML veikloms

Siekiant išreikšti UML veiklas nuosekliu programiniu kodu, neiškraipant modeliuotojo intencijų, tenka įvesti vienintelį apribojimą: tose veiklose bus naudojami tik programų inžinerijos modeliavimui skirti dariniai (jie buvo išskirti anksčiau). Tuos darinius galima sujungti lygiagrečiai ir daugumą darinių, skirtų veiklos procesų inžinerijai, galima nesunkiai pakeisti dariniais, skirtais programų inžinerijai.

3.4 Lygiagrečios UML veiklos išreiškimo nuosekliu programiniu kodu metodas

Pagal specifikaciją, UML veiksmas negali pradėti veikti tol, kol visuose jo įėjimuose nebus po žymę [4]. Tai reiškia, kad teisingai sudarytos UML veiklos grafe negalės būti ciklų, nes kitaip kažkuri cikle esanti tos veiklos dalis nesuveiks, laukdama žymių iš kitos to paties ciklo dalies, kuri, savo ruožtu, lauks žymės iš pirmosios.

Lygiagrečiai veikiančią UML veiklą galima išreikšti nuosekliu programiniu kodu, tačiau reikia nustatyti joje esančių UML veiksmų vykdymo eiliškumą. Tam yra sudaromas UML veiklos grafas, kurio viršūnės yra veiksmai, rekursyvių išraiškų pradiniai veiksmai, *InitialNode*, *ActivityFinalNode, ActivityParameterNode* bei *StructuredActivityNode*, o briaunos yra *ControlFlow* bei *ObjectFlow* su to *ObjectFlow* jungiamais *Pin* tipo objektais. Kiekviena UML veiklos grafo viršūnė bei briauna įsimena jas atitinkančius veiklos elementus.

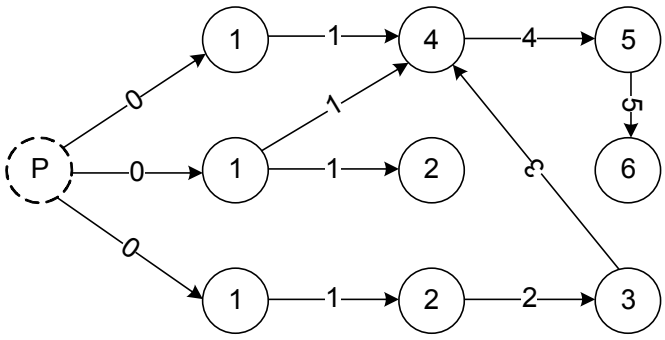
StructuredActivityNode bei jos potipiais esantys *ConditionalNode*, *SequenceNode*, *ExpansionRegion* bei *LoopNode* atitinka programavimo kalbose naudojamus struktūrinius skliaustus. Pagal UML specifikaciją *StructuredActivityNode* turinys nepradedamas vykdyti tol, kol nepradedamas vykdyti atitinkamas *StructuredActivityNode*, o *StructuredActivityNode* sulaiko visas iš jos išeinančias žymes tol, kol jos turinys nebaigia darbo. Tai leidžia interpretuoti *StructuredActivityNode* bei jos potipius kaip savarankiškus vykdymo vienetus ir UML veiklos grafe atvaizduoti viena viršūne. Lygiagretumą *StructuredActivityNode* viduje galima pašalinti, sudarius jai nuosavą vykdymo grafą.

Sudaryto UML veiklos vykdymo grafo pradinės viršūnės (nuo kurių prasideda UML veiklos vykdymas) yra sujungiamos su specialia pseudopradine viršūne. Paskui pasinaudojama modifikuota paieška į plotį [11] ir surandamas ilgiausias kelias nuo pseudopradinės viršūnės iki visų likusių. Tos paieškos algoritmas:

```
N = {Grafo viršūnių aibė, viršūnės nepažymėtos};
```

```
E = {Grafo kraštinių aibė, kraštinės nepažymėtos};
step = 0;
while(true){
   Nr = get_a_set_of_ready_unmarked_nodes(N, E);
   if(set_empty(Nr) && all_marked(N)) break;
   else "klaida: ciklas arba grafas nejungus";
   foreach(Ni in Nr){
      Ni.mark = step;
      Er = get_outgoing_edges(E, Ni);
      foreach(Ei in Er){
         Ei.mark = step;
      }
   }
   step++;
}
```

Paieškos algoritmas veikia žingsniais. Kiekviename žingsnyje iš visų pasiruošusių vykdymui grafo viršūnių išeinančios ir dar nepažymėtos briaunos yra pažymimos to žingsnio numeriu. Žingsnio numeriu pažymimos ir vykdymui pasiruošusios viršūnės. Viršūnė yra laikoma pasiruošusia vykdymui, jeigu visos į ją įeinančios briaunos yra pažymėtos. Algoritmas baigia darbą, kai visos viršūnės ir briaunos tampa pažymėtomis. Pseudopradinė viršūnė vykdymui pasiruošusi visada.



17 pav. Užbaigto paieškos algoritmo iliustracija

UML veiklos grafo viršūnės, su vienodu žymės numeriu, bus pasiruošusios vykdymui vienu metu. Viršūnės su mažesniu žymės numeriu turės suveikti anksčiau, negu viršūnės su didesniu žymės numeriu.

Turint apdorotą UML veiklos vykdymo grafą, jį atitinkančius UML veiklos elementus galima išreikšti nuosekliu programiniu kodu. Tai pradedama daryti nuo viršūnių su mažiausiu žymės numeriu. UML veiklos elementai, kurie atitinka grafo viršūnes su vienodu žymės numeriu, yra sudedami į vieną kodo bloką bet kokia tvarka. Kodo blokai suklijuojami nuosekliai, žymės numerių didėjimo tvarka. Generuojant nuoseklų kodą tokiu būdu, vykdymo grafo briaunos, atitinkančios *ControlFlow,* bus realizuojamos automatiškai, o briaunas, atitinkančias *ObjectFlow,* teks pakeisti tarpinių kintamųjų skaitymu ir rašymu.

Nuoseklaus kodo generavimo iš apdoroto vykdymo grafo algoritmas:

```
N = {Grafo viršūnių aibė, viršūnės pažymėtos};
```

```
E = {Grafo kraštinių aibė, kraštinės pažymėtos};
maxMark = find_max_mark(N);
finalCode = ""; //rezultatas
lastBlock = {};
for( i = 0; i < maxMark; i++ )
{
   curBlock = sef_of_nodes_with_mark(i, N);
   codeBlock = make_code_block(lastBlock, curBlock);
   finalCode = finalCode + codeBlock;
   lastBlock = curBlock;
}
```

```
,kur make_code_block(lastBlock, curBlock) sukuria programos kodo gabalą,
atitinkanti curBlock viršūnes ir susieja jas  tarpusavyje bei su
lastBlock viršūnėmis panaudodamas tarpinius kintamuosius.
```

3.5 Rekursyvios išraiškos turinio išreiškimo nuosekliu kodu metodas

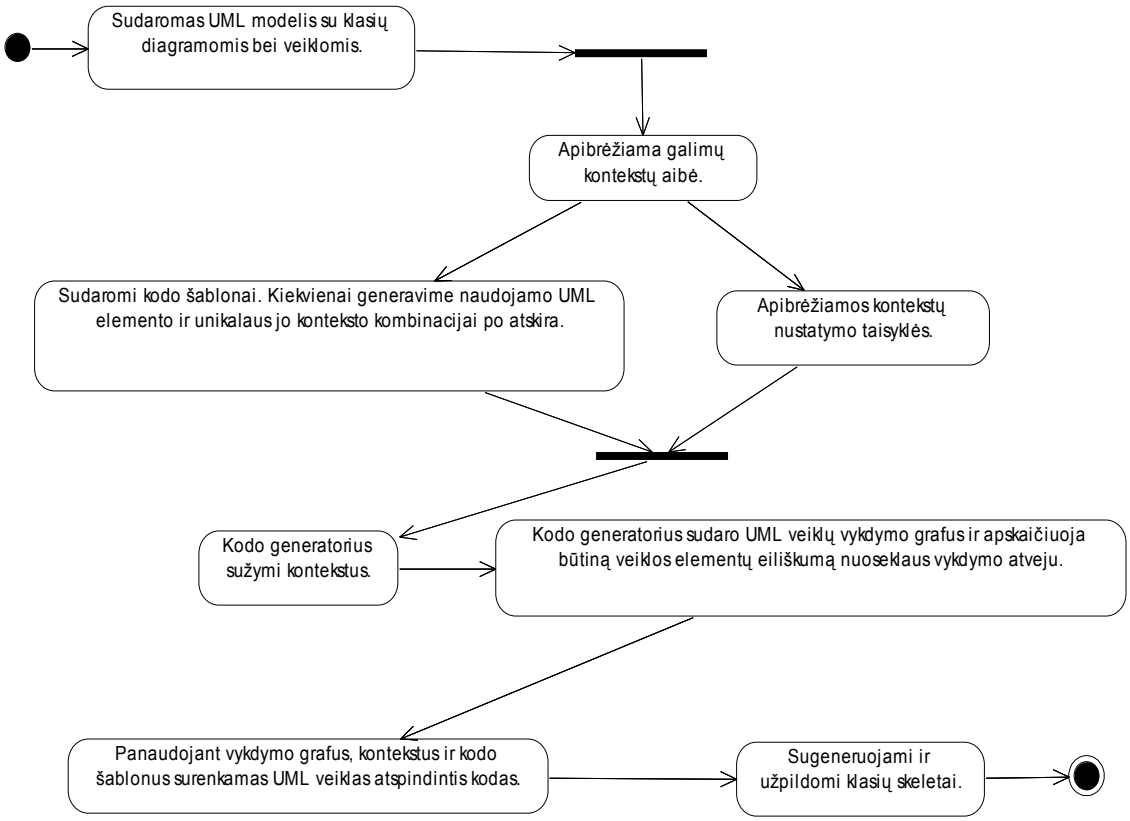
Rekursyvios išraiškos vykdymo grafas yra medis. Todėl jos išreiškimą nuosekliu kodu reikia pradėti nuo giliausiai esančių viršūnių ir nuosekliai eiti prie seklesnių. Vienodame gylyje esančias viršūnes galima sudėti į vieną kodo bloką, išdėstant bet kokia tvarka, o gretimus kodo blokus sujungti tarpiniais kintamaisiais.

3.6 *ConditionalNode* bei *Clause* ypatybės

ConditionalNode (sąlyginis perėjimas) bei *Clause* (sąlyginio perėjimo atšaka) yra ypatingi tuo, kad keletas *Clause* gali naudoti tuos pačius UML veiklos gabalus sąlygų testavime bei kūnuose. Išreiškiant tokius *Clause* programos kodu, reikia sukurti kiekvienam po atskirą naudojamo veiklos gabalo kopiją, tačiau palikti bendrus kintamuosius. Tokiu būdu *Clause* bus galima nesunkiai išreikšti programos kodu, o efektai, numatyti dėl daugkartinio tų pačių kintamųjų panaudojimo, išliks.

3.7 Apibendrintas metodą realizuojantis algoritmas

Apibendrinant tai, kas buvo pasakyta šiame skyriuje, kodo generavimo metodą galima išreikšti toliau pateikiamu algoritmu:



18 Pav. UML veiklos diagrama, iliustruojanti metodą realizuojantį algoritmą

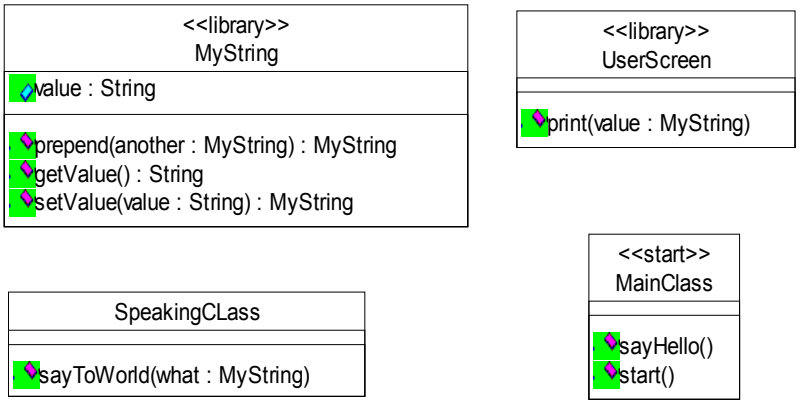
4 APRAŠYTO METODO ILIUSTRACIJA PRAKTINIU PAVYZDŽIU

Šiame skyriuje bus sumodeliuota programa, kuri pasisveikins su pasauliu (angl. *Hello World*). Siekiant pavaizduoti įvairius metodo aspektus, programa bus sumodeliuota šiek tiek sudėtingesnė nei būtina. Bus parodyta, kaip UML veiklos paverčiamos programos kodu, panaudojant anksčiau paminėtus metodus bei principus.

Reikia pabrėžti, kad nėra būdo programos kodui generuoti, kuris tiktų bet kuriai situacijai [10]. Todėl realių programos kodo generatorių veikimo principai gali būti šiek tiek kitokie negu nurodyti pavyzdyje.

4.1 Modeliuojamos sistemos klasių diagrama

Iliustracinės sistemos klasių diagrama pateikiama 19 paveiksle.



19 pav. Modeliuojamos sistemos klasių diagrama.

Klasė *MyString* yra tarnybinė, skirta veiksmams su UML *String* tipo egzemplioriais atlikti. UML specifikacijoje neapibrėžtos operacijos su teksto eilutėmis, o kiekvienoje programavimo kalboje jos atvaizduojamos vis kitaip. Todėl *MyString* yra priskirtas stereotipas „*library*“, kuris nurodo, kad šios klasės realizacija programiniu kodu bus ne generuojama, o apibrėžta iš anksto.

Stereotipo „*library*“ naudojimas yra mechanizmas, skirtas aukštam UML modelio abstrakcijos lygiui palaikyti. Jis leidžia išvengti modelio susiejimo su tiksline platforma ir (tai bus paaiškinta vėliau) leidžia panaudoti tą patį UML modelį generuojant programos kodą keliose skirtingose kalbose.

Klasėje *MyString* yra keletas metodų. Metodas „*prepend*“ papildo atributo „*value*“ turinį kito *MyString* klasės egzemplioriaus atributo „*value*“ turiniu, įdėdamas jį eilutės pradžioje. Metodas „*getValue*“ grąžina atributo „*value*“ turinį. Metodas „*setValue*“ padaro atributo „*value*“ turinį lygų duotajam argumentui.

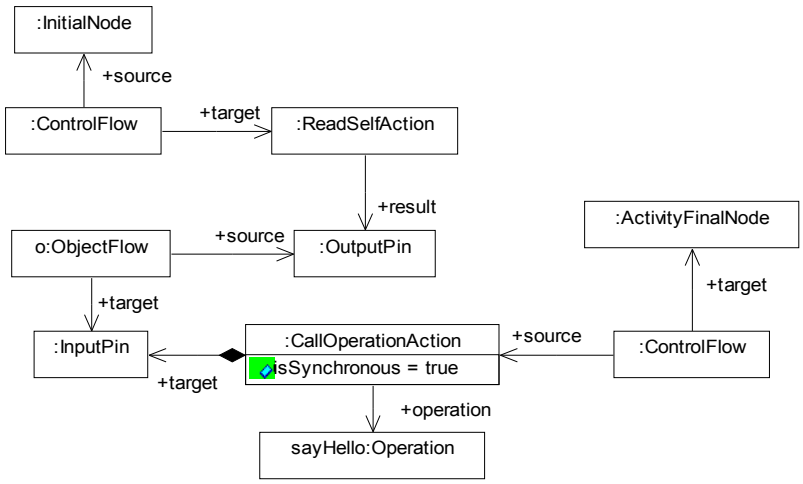
Klasė *UserScreen* turi vienintelį metodą „*print*“, kurio paskirtis yra išvesti duotą eilutę vartotojui į ekraną. Ši klasė taip pat pažymėta stereotipu „*library*“, kadangi teksto išvedimas yra priklausomas nuo platformos, kurioje bus realizuota modeliuojama sistema.

Klasė *MainClass* yra įvadinė programos klasė. Ji pažymėta stereotipu „*start*“, kuris nurodo kodo generatoriui sugeneruoti kodą taip, kad programos darbo pradžioje būtų iškviestas tuo stereotipu pažymėtos klasės metodas „*start*“. Tokiu būdu pasiekiama modelio nepriklausomybė nuo realizacijos platformoje naudojamų priemonių programos įvadiniam taškui aprašyti. Kitas *MainClass* metodas yra „*sayHello*“, kuris sukurs naują *SpeakingClass* egzempliorių bei iškvies tos klasės metodą „*sayToWorld*“ su argumentu „*Hello* “.

Klasė *SpeakingClass* turi vienintelį metodą „*sayToWorld*“, kuris prideda prie savo argumento teksto eilutę „ *World*“ bei, pasinaudodamas klasės *UserScreen* metodu „*print*“, sudėties rezultatą parodo vartotojui.

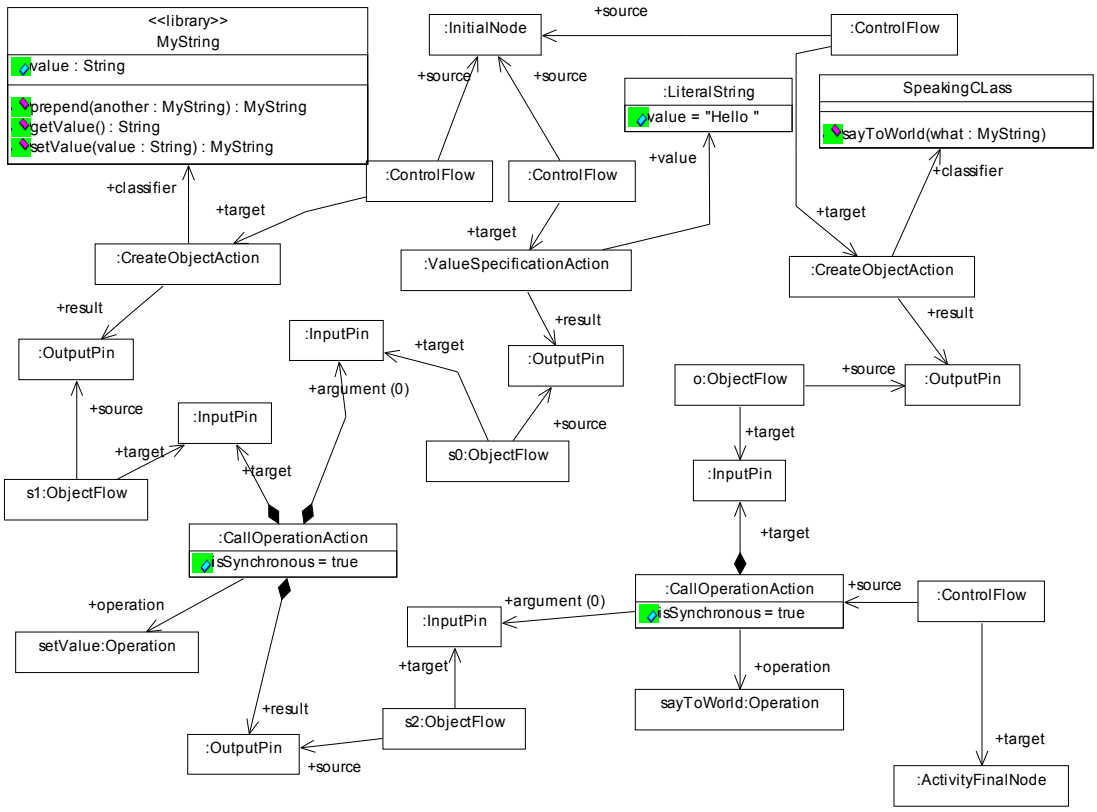
4.2 Veiklos, realizuojančios modeliuojamos sistemos metodus

Visų modeliuojamos sistemos klasių metodų (išskyrus aprašytų klasėse-bibliotekose) elgsena yra apibrėžiama UML veiklų pagalba. UML veiklos yra pavaizduotos UML metamodelio klasių egzemplioriais ir jų ryšiais, kadangi įprasta UML veiklų notacija paslėptų per daug svarbios informacijos. Pateiktus modelius patogiausia yra skaityti pradedant nuo viršuje esančio *InitialNode* egzemplioriaus.



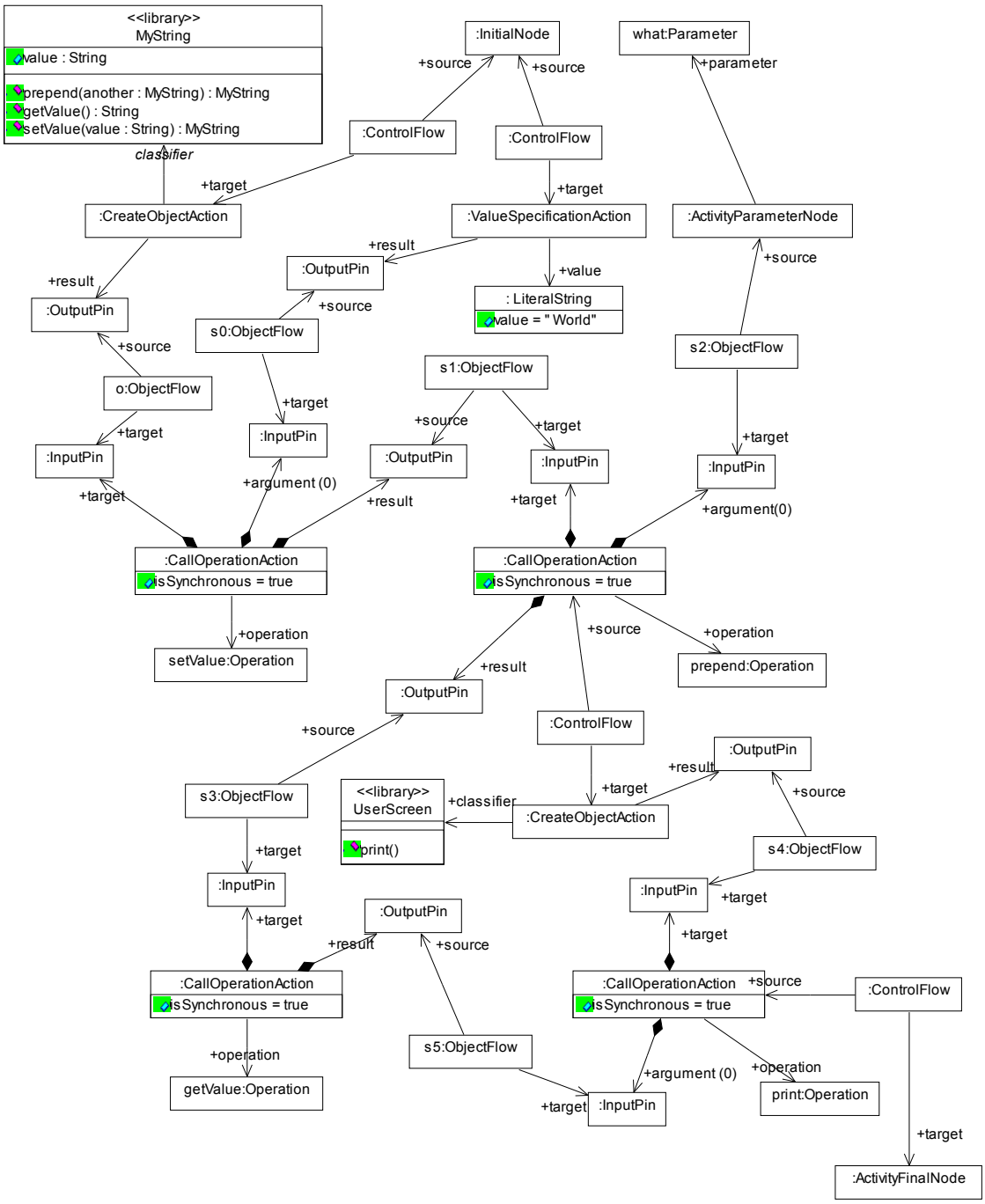
20 pav. Metodą „*MainClass::start*“ specifikuojanti veikla

Metodą „*MainClass::start*“ specifikuojanti veikla per *ReadSelfAction* gauną nuorodą į supančios klasės egzempliorių ir panaudodama *CallOperationAction* iškviečia tos pačios klasės metodą „*sayHello*“.



21 pav. Metodą „*MainClass::sayHello*“ specifikuojanti veikla

Metodą „*MainClass::sayHello*“ specifikuojanti veikla sukuria naują *MyString* klasės egzempliorių ir priskiria jam reikšmę „*Hello* “, kuri gaunama panaudojus *ValueSpecificationAction*. Lygiagrečiai yra sukuriamas klasės *SpeakingClass* egzempliorius, paskui iškviečiamas jo metodas „*sayToWorld*“, kurio argumentas yra neseniai sukurtos klasės *MyClass* egzempliorius su reikšme „*Hello* “.



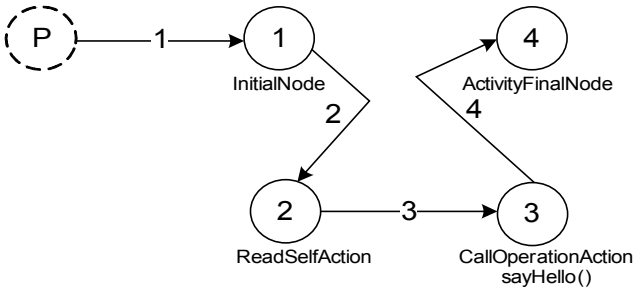
22 pav. Metodą „*SpeakingClass::sayToWorld*“ specifikuojanti veikla

Metodą „*SpeakingClass::sayToWorld*“ specifikuojanti veikla sukuria naują *MyString* klasės egzempliorių, kuriam priskiria reikšmę „ *World*“. Paskui šis egzempliorius yra papildomas metodo argumente gauto *MyString* egzemplioriaus reikšme, taip gaunant teksto eilutę „*Hello World*“. Sekantis veiksmas yra klasės *UserScreen* egzemplioriaus sukūrimas bei metodo „*UserScreen::print*“ iškvietimas su ką tik gauta „*Hello World*“ eilute.

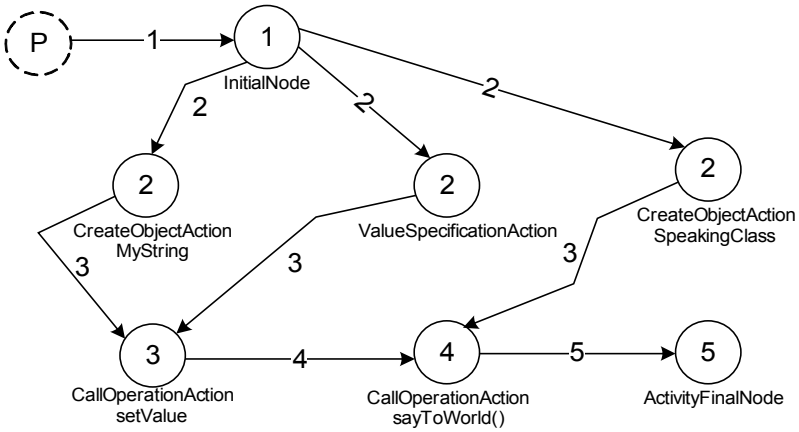
4.3 UML veiklų vykdymo grafų sudarymas ir būtino veiksmų vykdymo eiliškumo nustatymas

Dalis operacijų modelio UML veiklose yra vykdoma lygiagrečiai. Tam, kad galėtume šias veiklas išreikšti nuosekliu programos kodu, teks pasinaudoti anksčiau aprašytu lygiagrečių UML veiklų pavertimo nuosekliu programos kodu metodu.

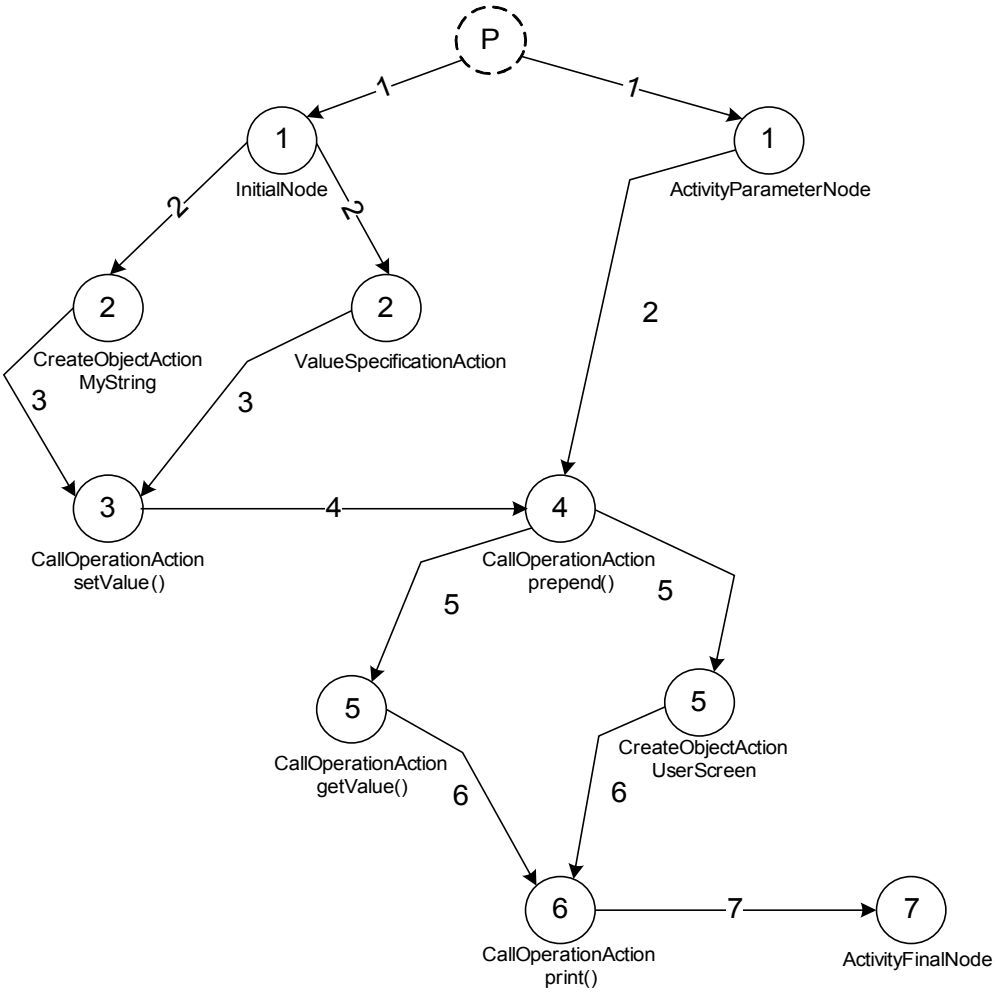
Visų pirma yra sudaromi UML veiklų vykdymo grafai, paskui apskaičiuojamas būtinas veiksmų vykdymo eiliškumas. Sudarytuose grafuose prie viršūnių yra trumpos nuorodos į atvaizduojamos UML veiklos elementus. Ką vaizduoja grafų briaunos, nesunku nustatyti pažvelgus į atitinkamą UML veiklą.



23 pav. Metodą „*MainClass::start*“ specifikuojančios veiklos vykdymo grafas



24 pav. Metodą „*MainClass::sayHello*“ specifikuojančios veiklos vykdymo grafas



25 pav. Metodą „*SpeakingClass:sayToWorld*“ specifikuojančios veiklos vykdymo grafas

4.4 UML elementų kontekstų išskyrimo procesas

UML elemento kontekstas - tai unikalus identifikatorius (arba jų aibė), kuris atspindi to UML elemento aplinką ir savybes. Kontekstų parinkimo kriterijai ir jų galutinė aibė nustatomi pagal konkretaus projekto poreikius.

Šiame projekte naudojami trys konteinerių kontekstai - „*library*“, „*start*“ bei „“ (tuščias kontekstas). Kontekstai „*library*“ bei „*start*“ veikia tik klases ir yra nustatomi pagal atitinkamų pavadinimų jų stereotipus. Tuščias kontekstas yra priskirtas visiems likusiems elementams. Elementai paveldi kontekstus iš savo konteinerių.

UML veiksmų kontekstai priklauso nuo to veiksmo elgseną įtakojančių atributų reikšmių bei veikiamų ar naudojamų objektų tipų. Vienintelis nuo atributo priklausantis UML veiksmas šiame pavyzdyje yra *CallOperationAction*, kuris gali nurodyti sinchroninį (*isSynchronous = true*) arba asinchroninį (*isSynchronous = false*) iškvietimą.

Galutinis kiekvieno elemento kontekstas yra jo konteksto ir visų paveldėtų kontekstų suminė aibė.

4.5 UML elemento šablonas ir jo parinkimas

UML elemento išraiška programos kodu priklausys nuo jo konteksto. Skirtingiems kontekstams reikės skirtingų programos kodo šablonų. Prie kiekvieno UML elemento šablono varianto yra nurodoma kokiam kontekstui jis tinka.

Kaip jau buvo minėta anksčiau, galutinis elemento kontekstas yra sudarytas iš to elemento paveldėto konteksto bei nuo elemento savybių priklausančio konteksto. Todėl nurodant, kokiam kontekstui tinka elemento šablonas, šios dvi kontekstų grupės išskiriamos.

Nuo elemento savybių priklausantis kontekstas apibrėžiamas išvardinant tas savybes bei jų reikšmes. Leistina reikšmių aibė suformuojama apibrėžiant visus jos narius. Jeigu savybė konteksto apibrėžime nepaminėta, laikoma, kad jos reikšmė nesvarbi. Jeigu vieno konteksto apibrėžimas apima ir kito konteksto apibrėžimą, tai pirmiausia bus bandoma panaudoti šabloną su labiau specializuotu konteksto apibrėžimu.

Šiame pavyzdyje elemento šablonui aprašyti bus naudojama lentelė:

2 lentelė. Elemento šablonui aprašyti naudojamos lentelės forma

Elemento tipas, konteksto ID	
A - Paveldėti kontekstai	B - Nuo savybių priklausantys kontekstai
C - Šablono tekstas	

```
Kur:
A = all | ID , ID*;
B = propName (all | ID, ID*) targetContext (all | ID, ID*)
C = tekstas, naudojant "Velocity" šablonų variklio kalbą
propName = savybės vardas
targetContext = tos savybės atspindimo objekto kontekstas

Jeigu kažkuris elementas nenurodytas, tai laikoma, kad jo reikšmė nesvarbi.
```

4.6 UML elementų šablonų panaudojimas programos kodui surinkti

UML elemento šablonas yra tekstinis karkasas, kuris užpildomas tuo šablonu išreiškiamo modelio elemento informacija. Klasės šablono atveju tai būtų klasės pavadinimas, atributai bei metodai.

UML veikloje yra keletas elementų, kurių išraiška programos kode ypatinga. Vienas iš jų yra *ActivityParameterNode*, kuris skirtas perduoti duomenų žymėms iš UML veiklos įėjimo parametrų ir į UML veiklos išėjimo parametrus. *ActivityParameterNode* gali dalyvauti dviejuose scenarijuose: žymės perdavime į UML veiklą bei žymės perdavime iš veiklos. Pirmajame scenarijuje dalyvaujantys *ActivityParameterNode* bus vadinami įeinančiais, o antrajame -

išeinančiais.

Įeinantys *ActivityParameterNode* ypatingi tuo, kad juos atstovaujantis kintamasis jau būna apibrėžtas metodo signatūroje. Visi iš tokių *ActivityParameterNode* išeinantys *ObjectFlow* kodo generavimo metu bus pakeisti atitinkamo metodo signatūroje nurodyto kintamojo vardu.

Išeinantys *ActivityParameterNode* ypatingi tuo, kad į juos patenkančios reikšmės bus grąžinamos kaip metodo rezultatas. UML veikla baigia darbą, kai žymė patenka į veiklos lygyje esantį *ActivityFinalNode* elementą. Šio elemento kodo šablonui bus pateikiamas visų „į išeinančius *ActivityParameterNode* vedančių *ObjectFlow"* atitinkančių kintamųjų vardų sąrašas.

UML veiksmo kodo šablonui yra pateikiamos reikšmės, atitinkančios su juo sujungtų *Pin* reikšmes. Tai gali būti kintamųjų vardai arba UML elementų tipų vardai. Konkreti pateikiamų reikšmių aibė priklauso nuo UML veiksmo tipo ir gali būti nesunkiai nustatyta, pažvelgus į UML kalbos specifikacijoje su tuo UML veiksmu susietus *Pin* bei jų vaidmenis (angl. *Role*).

ObjectFlow tipo elementai tampa tarpiniais kintamaisiais. Tų kintamųjų pavadinimai prilyginami atitinkamų *ObjectFlow* pavadinimams arba (pvz, jeigu *ObjectFlow* be pavadinimo) sukuriami nauji. Prieš generuojant programos kodą, gaunami visų *ObjectFlow* pavadinimai ir deklaruojami atitinkami tarpiniai kintamieji (išskyrus atvejus susijusius su įeinančiu *ActivityParameterNode*).

4.7 Programos kodo generavimas *Java* kalba

4.7.1 Kodo šablonai

Kodo šablonams aprašyti buvo panaudota „*Apache Jakarta*" projekte sukurto šablonų variklio „*Velocity*" naudojama kalba. „*Velocity*" naudojama kalba labai panaši į *PHP* programavimo kalbą, tik tarnybiniai žodžiai turi prefiksą „#".

3 lentelė. Programos kodo šablonas elementui *Class*.

<table>
<tr><td colspan="2">Class, A</td></tr>
<tr><td>library</td><td>name (MyString)</td></tr>
<tr><td colspan="2"><pre>
class MyString{
  private String value = "";
  public MyString prepend(MyString another){
    value = value + another.getValue();
  }
  public setValue(String value){
    this.value = value;
  }
  public String getValue(){
</pre></td></tr>
</table>

```
    return value;
  }
}
```

4 lentelė. Programos kodo šablonas elementui *Class*.

Class, B	
library	name (UserScreen)

```
class UserScreen{
  public print(String what){
    System.out.print(what);
  }
```

5 lentelė. Programos kodo šablonas elementui *Class*.

Class, C	
all	

```
class $className {
  #foreach($method in $methodList)
    public $!method.returnType $method.name(
      #foreach($parameter in $method.parameterList)
        $parameter.type $parameter.name
        #if( $parameter.last == false ) , #end
      #end
      ){
    $method.body
    }
  #end
}
```

6 lentelė. Programos kodo šablonas elementui *InitialNode*.

InitialNode, D	
all	

7 lentelė. Programos kodo šablonas elementui *ActivityFinalNode*.

ActivityFinalNode, E	
all	

```
return $!retNames.firstName; //ActivityFinalNode
```

8 lentelė. Programos kodo šablonas veiksmui *CreateObjectAction*.

CreateObjectAction, F	
all	
`$varName = new $typeName(); //CreateObjectAction`	

9 lentelė. Programos kodo šablonas veiksmui *ValueSpecificationAction*.

ValueSpecificationAction, G	
all	valueInputPinType (LiteralString)
`$name = "$value"; //ValueSpecificationAction`	

10 lentelė. Programos kodo šablonas veiksmui *CallOperationAction*.

CallOperationAction, H0	
all	IsSynchronous (true) resultOutputPinType (all)

`$resultName = $targetName.$operation(`
`  #foreach($argumentName in $argumentList)`
`  $argumentName`
`  #if($argumentName.last == false) , #end`
`  #end); //CallOperationAction H0`	

11 lentelė. Programos kodo šablonas veiksmui *CallOperationAction*.

CallOperationAction, H1	
all	IsSynchronous (true) resultOutputPinType (null)

`$targetName.$operation(`
`  #foreach($argumentName in $argumentList)`
`  $argumentName`
`  #if($argumentName.last == false) , #end`
`  #end); //CallOperationAction H1`	

12 lentelė. Programos kodo šablonas įeinančiam *ActivityParameterNode*.

ActivityParameterNode-Incomming, I	
all	

13 lentelė. Programos kodo šablonas išeinančiam *ActivityParameterNode*.

ActivityParameterNode-Outgoing, J	
all	
`$type $name; //ActivityParameterNode-Outgoing`	

14 lentelė. Programos kodo šablonas elementui *ObjectFlow*.

ObjectFlow, K	
all	
`$type $name;`	

15 lentelė. Programos kodo šablonas veiksmui *ReadSelfAction*.

ReadSelfAction, L	
all	
`$name = this; //ReadSelfAction`	

16 lentelė. Specialus programos kodo šablonas, įvadiniam programos taškui.

EntryPoint (specialus šablonas), M	
all	

```
class Main_{
  public static void main(String[] args){
    $startType startClass = new $startType();
    startClass.start();
  }
}
```

4.7.2 Programos kodo surinkimas

4.7.2.1 Veiklų kodo surinkimas

4.7.2.1.1 MainClass::start

Deklaruojame *ObjectFlow* atitinkančius kintamuosius:

```
MainClass o;
```

Pasinaudodami 23 pav. nupieštu grafu ir kodo šablonais, surenkame veiklą atspindintį kodą:

```
o = this; //ReadSelfaction
o.sayHello(); //CallOperationAction H1
```

4.7.2.1.2 MainClass::sayHello

Deklaruojame *ObjectFlow* atitinkančius kintamuosius:

```
MyString s1;
String s0;
SpeakingClass o;
MyString s2;
```

Pasinaudodami 24 pav. nupieštu grafu ir kodo šablonais, surenkame veiklą atspindintį kodą:

```
s1 = new MyString(); //CreateObjectAction
s0 = "Hello "; //ValueSpecificationAction
o = new SpeakingClass(); //CreateObjectAction
s2 = s1.setValue(o); //CallOperationAction H0
o.sayToWorld(s2); //CallOperationAction H1
```

4.7.2.1.3 SpeakingClass::sayToWorld

Deklaruojame *ObjectFlow* atitinkančius kintamuosius:

```
MyString o;
String s0;
MyString s1;
MyString s3;
String s5;
UserScreen s4;
```

Pasinaudodami 25 pav. nupieštu grafu ir kodo šablonais, surenkame veiklą atspindintį kodą:

```
o = new MyString(); //CreateObjectAction
s0 = " World"; //ValueSpecificationAction
s1 = o.setValue(s0); //CallOperationAction H0
s3 = s1.prepend(what); //CallOperationAction H0
```

```
    s4 = new UserScreen(); //CreateObjectAction
    s5 = s3.getValue(); //CallOperationAction H0
    s4.print(s5); //CallOperationAction H1
```

4.7.2.2 Klasių kodo surinkimas ir užpildymas veiklų kodu

Klasės *MyString* bei *UserScreen* paimamos iš šablonų *A* ir *B* nepakeistos, tad čia jos nebus perrašytos.

Surinkę klasę *MainClass* (pagal šabloną *C*) gausime:

```
class MainClass{
  public start(){
    MainClass o;

    o = this; //ReadSelfaction
    o.sayHello(); //CallOperationAction H1
  }
  public sayHello(){
  {
    MyString s1;
    String s0;
    SpeakingClass o;
    MyString s2;

    s1 = new MyString(); //CreateObjectAction
    s0 = "Hello ";    //ValueSpecificationAction
    o = new SpeakingClass(); //CreateObjectAction
    s2 = s1.setValue(o); //CallOperationAction H0
    o.sayToWorld(s2); //CallOperationAction H1
  }
}
```

Surinkę klasę *SpeakingClass* (pagal šabloną *C*) gausime:

```
class SpeakingClass{
  public sayToWorld(MyString what){
    MyString o;
    String s0;
    MyString s1;
    MyString s3;
    String s5;
    UserScreen s4;

    o = new MyString(); //CreateObjectAction
    s0 = " World"; //ValueSpecificationAction
    s1 = o.setValue(s0); //CallOperationAction, H0
    s3 = s1.prepend(what); //CallOperationAction, H0
    s4 = new UserScreen(); //CreateObjectAction
    s5 = s3.getValue(); //CallOperationAction, H0
    s4.print(s5); //CallOperationAction, H1
  }
}
```

Kodo šablonas *M* apibrėžia specialią įvadinę klasę, kurią paleis *Java* virtuali mašina. Programos kodo generatorius per šį šabloną sukurs klasės su kontekstu „*start*“ (*MainClass*) egzempliorių ir iškvies metodą „*start*“.

```
class Main_{
  public static void main(String[] args){
    MainClass startClass = new MainClass();
    startClass.start();
  }
}
```

Paanalizavę gautą programos kodą pamatysime, kad jį sukompiliavus ir paleidus programą būtų atspausdinta teksto eilutė „*Hello World*“.

4.8 Galimybės generuoti pavyzdžio kodą kitomis kalbomis

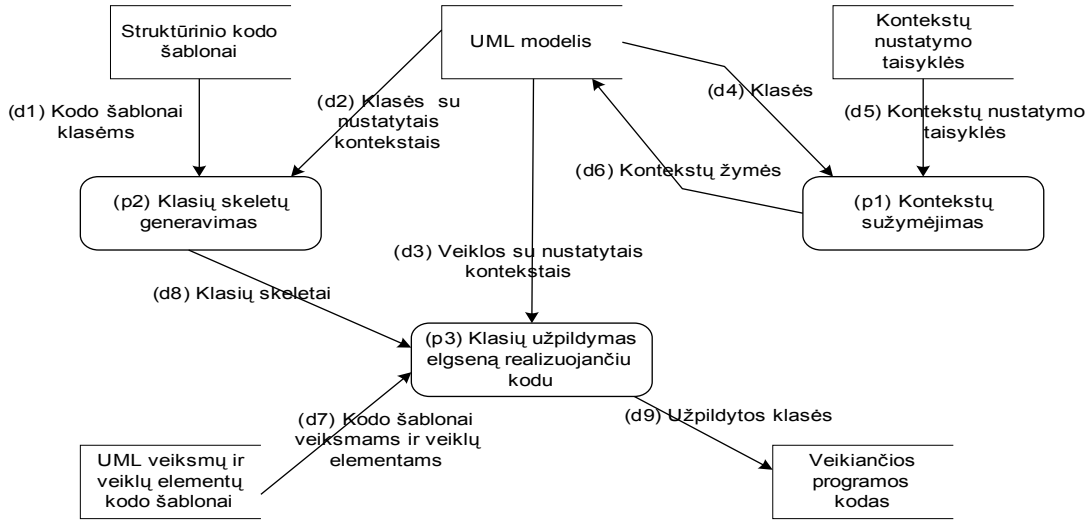
Paanalizavus programos kodo generavimo *Java* kalba pavyzdį, galima pastebėti, kad pavyzdžio UML modelis yra nepriklausomas nuo tikslo programavimo kalbos. Pakeitus kodo šablonus, atitinkamą programą būtų galima gauti ir *PHP*, *C#*, *C++* ar dar kokia nors panašios struktūros kalba.

5 EKSPERIMENTINĖS METODO REALIZACIJOS APRAŠYMAS

5.1 Naudojamos priemonės

Eksperimentinei metodo realizacijai naudojama „*Eclipse*“ platforma bei joje esantys EMF (angl. *Eclipse Modeling Framework*) ir UML2 paketai. Kodo šablonams apdoroti naudojamas „*Velocity*“ šablonų variklis (angl. *Template engine*).

5.2 Duomenys, procesas, rezultatai



26 pav. Duomenų srautų diagrama atspindinti programos kodo generavimo procesą

Programos kodo generavimo procesas prasideda nuo UML modelio elementų kontekstų sužymėjimo (p1). Kontekstai yra priskiriami klasėms arba paketams pagal kodo generatoriui pateiktas taisykles, kurios atspindimos paprasčiausiomis *Java* kalbos klasėmis, iškviečiamomis kontekstų sužymėjimo variklio. Tos klasės gauna nuorodą į UML modelį bei kontekstų žemėlapį (angl. *Map*) ir pagal savo vidinę logiką papildo kontekstų žemėlapį reikiamais kontekstų bei modelio elementų ryšiais. Taisyklės gali atsižvelgti į klasių ar paketų stereotipus, pavadinimus, turinį ir t.t. Jeigu kažkuriam modelio elementui kontekstas nenustatomas, tai tas elementas jį paveldi iš savo konteinerio (UML veiklos atveju tai būtų klasė arba metodas, kuriam priskirta UML veikla).

Turint UML modelį su sužymėtais kontekstais, galima sugeneruoti klasių skeletus. Kontekstai reikalingi klasių skeletų generavime tam, kad būtų galima parinkti tinkamus kodo šablonus, kuriuose gali būti įrašyti pagalbiniai metodai ar kita pagalbinė informacija, reikalinga klasės realizacijai konkrečiame kontekste.

Turint klasių skeletus, galima juos užpildyti klasių elgseną realizuojančiu programiniu

kodu. Tas kodas sugeneruojamas iš atitinkamų UML veiklų, naudojant anksčiau paminėtus UML veiklos pavertimo nuoseklia bei UML veiksmo išreiškimo kodu metodus. Papildomai procesas (p3) įtraukia į klasės failą visas jai reikalingas priklausomybes, nes tos priklausomybės nustatomas pagal tai, kokių tipų objektais manipuliuojama UML veiklose.

UML veiksmų ir veiklų elementų kodų šablonai yra pateikiami kiekvienai modelyje pasitaikančiai veiksmo ar veiklos elemento tipo bei jų unikalaus konteksto kombinacijai.

Užpildytos klasės toliau apdorojamos standartiniais įrankiais (kompiliatoriumi ir t.t.), bei įvykdomos.

6 IŠVADOS

1. Darbe atlikta kodo generavimo iš įvairių UML diagramų galimybių analizė parodė, kad UML veiksmų semantika yra turtingesnis duomenų šaltinis programos kodo generavimui negu UML klasių, būsenų ar sekų diagramos.
2. UML veiksmų semantikos analizė parodė, kad ją naudojant klasių metodų specifikavimui galima sugeneruoti pilnai veikiantį galutinį programos kodą, ne tik skeletą.
3. Atlikta kodą generuojančių CASE įrankių analizė parodė, kad šio darbo rašymo metų egzistavę įrankiai arba nesinaudojo UML veiksmų semantika, arba („*iCCG*“ atveju) neskelbė kodo generavime naudojamų metodų.
4. UML veiklų specifikacija yra pritaikyta dviem sritims – programų inžinerijai bei veiklos procesų inžinerijai. Norint išreikšti UML veiklą nuosekliu programos kodu, tikslinga joje naudoti tik programų inžinerijai skirtus darinius. Jeigu UML veikloje yra ir procesų inžinerijai skirtų darinių, tai ją reikia išreikšti lygiagrečiai veikiančiais procesais.
5. Buvo sukurtas ir iliustruotas pavyzdžiu bendro pobūdžio metodas, leidžiantis panaudoti UML veiksmų semantiką programos kodo generavimui. Šis metodas būtų naudingas kuriant programos kodo iš UML modelių generatorius.
6. Autoriaus žiniomis, panašus kodo generavimo metodas šiuo metu CASE įrankiuose nėra naudojamas. Jis galėtų žymiai padidinti kodo generatorių efektyvumą.
7. Programos kodo generavimas iš UML veiksmų semantikos leidžia panaudoti tą patį UML modelį programos kodui keliose skirtingose kalbose generuoti, kadangi UML veiklas galima pateikti taip, jog jose nebūtų konkrečiai programavimo kalbai būdingų darinių.
8. Darbe sukurtu metodu paremtas kodo generatoriaus prototipas realizuotas panaudojant *Eclipse* platformą bei *EMF*, *UML2*, „*Apache Velocity“* paketus.

7 LITERATŪRA

[1].Niaz, I.A., Tanaka, J. Code generation from UML statecharts. Instititute of Information Sciences and Electronics University of Tsukuba. [interaktyvus]. [2003] [žiūrėta 2006m 03mėn]. Prieiga per internetą: <http://www.iplab.cs.tsukuba.ac.jp/paper/international/niaz_sea2003.pdf>,

[2].Chauvel F., Jezequel JM. Code generation from UML Models with semantic variation points. [interaktyvus]. [2005] [žiūrėta 2006m 03mėn]. Prieiga per internetą: <http://www.irisa.fr/triskell/public/2005/Chauvel05a.pdf >,

[3].Knapp A., Merz S. Model Checking and Code Generation for UML State Machines and Collaborations. [interaktyvus]. [2003] [Žiūrėta 2006m 03mėn] Prieiga per internetą: <http://www.pst.informatik.uni-muenchen.de/veroeffentlichungen/knapp-merz:2003.pdf>

[4].Object Management Group. Unified Modeling Language: Superstructure (version 2.0). [interaktyvus]. [2005] [žiūrėta 2006m 03 mėn]. Prieiga per internetą: <http://www.omg.org/cgi-bin/doc?formal/05-07-04>,

[5].Object Management Group. Unified Modeling Language: Infrastructure (version 2.0). [interaktyvus]. [2005] [žiūrėta 2006m 03 mėn]. Prieiga per internetą: <http://www.omg.org/cgi-bin/doc?formal/05-07-05>,

[6].Miller J., Mukerji J., ... MDA Guide Version 1.0.1. [interaktyvus]. 2003 [žiūrėta 2006 03]. Prieiga per internetą:<http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf>

[7].Fujaba namų puslapis. http://wwwcs.uni-paderborn.de/cs/fujaba/ , peržiūrėta 2006m 03mėn.

[8].ArgoUML namų puslapis. [interaktyvus]. [žiūrėta 2006m 03 mėn]. Prieiga per internetą: <http://argouml.tigris.org/>,

[9].ArcStyler namų puslapis. [interaktyvus]. [žiūrėta 2006m 03 mėn]. Prieiga per internetą: <http://www.arcstyler.com/> ,

[10].Herrington, J. Code generation in action 2003, ISBN 1-930110-97-9 . Manning Publications Co.

[11].Diestel, R. Graph theory (Electronic Edition 2000). Springer-Verlag New York 1997, 2000

[12].Bock, C. UML 2 Activity and Action Models, Part6: Structured Activities. Jornal of Object Technology. [interaktyvus]. [2005] [žiūrėta 2006m 03 mėn]. Prieiga per internetą: <http://www.jot.fm/issues/issue_2005_05/column4>,

[13].Bock, C. UML 2 Activity and Action Models: Owerview. Jornal of Object Technology. [interaktyvus]. [2003] [žiūrėta 2006m 03 mėn]. Prieiga per internetą: <http://www.jot.fm/issues/issue_2003_07/column3>,

[14].iCCG namų puslapis. [interaktyvus]. [žiūrėta 2006m 03 mėn]. Prieiga per internetą:

<http://www.kc.com>.

8 PROGRAM CODE GENERATION USING UML ACTION SEMANTICS

Summary

The recent version of UML 2.0 (in year 2006) specifies activities and actions, which allow describing low level behavior of software system being modeled, in implementation independent fashion. This work analyzes suitability of UML 2.0 activities and actions for generating full program code (or as many as possible). It also proposes a method for generating program code from UML 2.0 activities and actions, which consists of a way to express UML activities with concurrently executing actions in a sequential execution scenario and a way to determine the exact code template (among the few possible) for UML activity elements based on identifying the particular context of element in question.

9 SANTRUMPŲ IR TERMINŲ ŽODYNAS

1. **UML** – (angl. *Unified Modeling Language*) tai standartas ir kalba, daugiausiai skirta programinės įrangos sistemoms modeliuoti. Šiame darbe naudojama UML versija 2.0, kuri dar vadinama UML2.
2. **MDA** – (angl. *Model Driven Architecture*) tai programinės įrangos sistemų kūrimo paradigma, pagal kurią centrinį vaidmenį programinės įrangos kūrimo procese atlieka jos modelis.
3. **UML veikla** – (angl. *Activity*) tai parametrizuotas konteineris UML veiksmams (angl. *Action*) bei kitiems UML veiklų elementams. UML veiklos nereikėtu maišyti su UML veiklos diagrama (angl. *Activity Diagram*), kuri yra grafinė UML veiklos išraiška.
4. **UML veiksmas** – (angl. *Action*) tai smulkiausias veikimą aprašantis UML specifikacijos elementas. UML veiksmų yra įvairių tipų, kiekvienas atlieką vis kitokią operaciją.
5. **CIM** – (angl. *Computation Independent Model*). Tai neformalus modelis, kuris apibūdina modeliuojamos sistemos aplinką bei jai keliamus reikalavimus.
6. **PIM** – (angl. *Platform Independent Model*). Tai formalus modelis, kuriame sistema apibūdinama nepriklausomai nuo realizacijos technologijos.
7. **PSM** – (angl. *Platform Specific Model*). Tai PIM modelis, papildytas arba transformuotas taip, kad jame būtų atspindima tam tikrai realizacijos platformai arba jų aibei būdinga informacija.
8. **Tikslinė programavimo kalba** – (angl. *Target language*). Programos kodo generavime tai yra programavimo kalba, kurioje bus sugeneruotas kodas.
9. **Tikslinė platforma** – (angl. *Target platform*). Programos kodo generavime tai yra platforma, kuriai bus pritaikytas sugeneruotas kodas.
10. **Kodo šablonas** – (angl. *Code template*) tai parametrizuotas programos teksto gabalas.
11. **Šablonų variklis** – (angl. *Template engine*) tai programinis komponentas, kuris leidžia užpildyti kodo šablonus konkrečiais duomenimis.
12. **UML modelio elemento kontekstas** – šiame darbe, tai identifikatorius arba jų aibė, kuris apibūdina unikalią modelio elemento aplinką, to elemento savybes bei jo veikiamų elementų savybes. Kontekstas šiame darbe yra būdas, kaip nurodyti programos kodo generatoriui kokį kodo šabloną reikia naudoti konkrečiam modelio elementui. Kontekstų aibę apibrėžia žmogus, modelio elementams kontekstai priskiriami pagal žmogaus sudarytus kriterijus.

10 I PRIEDAS

11 II PRIEDAS