KAUNO TECHNOLOGIJOS UNIVERSITETAS

KRISTINA BESPALOVA

AUTOMATIZUOTAS METAPROGRAMŲ KŪRIMAS IR SPECIALIZAVIMAS PANAUDOJANT POŽYMIAIS GRINDŽIAMŲ MODELIŲ TRANSFORMACIJAS

Daktaro disertacija
Fiziniai mokslai, Informatika (09P)

2015, Kaunas

UDK 004.4’24 (043.3)

Disertacija rengta 2011–2015 m. Kauno technologijos universiteto Informatikos fakultete, Programų inžinerijos katedroje.

MOKSLINIS VADOVAS:

Prof. habil. dr. Vytautas Štuikys (Kauno technologijos universitetas, fiziniai mokslai, informatika – 09P).

KALBOS REDAKTORIUS:

Jurgita Mikelionienė

ISBN 978-609-02-1167-0

PADĖKA

Visų pirma, nuoširdžiai dėkoju savo moksliniam vadovui prof. Vytautui Štuikiui už pagalbą rengiant šį darbą, suteiktas vertingas mokslines konsultacijas, patarimus ir nuolatinį skatinimą tobulėti.

Taip pat dėkoju Renatai Burbaitei už produktyvias diskusijas ir bendradarbiavimą rengiant straipsnius į tarptautinius žurnalus ir konferencijas.

Dėkoju visiems kitiems tiesiogiai ar netiesiogiai prisidėjusiems prie šio darbo už galimybę dirbti kartu bei perimtas žinias.

Savo šeimai dėkoju už nuolatinį palaikymą, supratimą ir tikėjimą manimi.

TURINYS

PAVEIKSLŲ SĄRAŠAS

LENTELIŲ SĄRAŠAS

TERMINŲ IR SANTRUMPŲ ŽODYNAS

Abstrakcija	Procesas, mažinantis tam tikros informacijos turinį, siekiant išlaikyti tik tą informaciją, kuri atitinka specifinį tikslą.
Abstrakcijos lygmuo	Abstraktumo laipsnio mato vienetas. Kuo aukštesnis abstrakcijos laipsnis, tuo mažesnis detalumo laipsnis.
Adaptavimas	Programinės įrangos pritaikymas darbui tam tikroje aplinkoje (Dagienė, Grigas ir Jevsikova, 2009).
Automatizavimas	Procesų transformacija į automatinį veikimo būdą.
Bendrasis komponentas (angl. *generic component*)	Apibendrintas komponentas, kuriuo aprašoma sintaksiškai arba semantiškai panašių komponentų šeimyna, turintis bendrus metaparametrus, kuriais remdamasis vartotojas gali pasirinkti konkretų komponento egzempliorių.
Bendrybė	Prielaida (atributas), kuri yra teisinga visiems srities objektams.
Daugiapakopis programavimas	Programavimo metodas, kai skirtingos programų pakopos aiškiai atskirtos, vykdomos skirtingu laiku, o ankstesnės pakopos darbo rezultatai yra naudojami kaip programinis kodas sekančioje pakopoje (Inoue, Taha, 2012).
Generavimas	Automatinis žemesnio lygmens programų kūrimas iš aukštesnio lygmens specifikacijos. Iš aukštesnio lygmens metaprogramos kuriama žemesnio lygmens metaprograma, iš metaprogramos – tikslo kalbos programų egzemplioriai.
Heterogeninė transformacija	Transformacija, kai pirminis ir tikslo modeliai yra išreikšti skirtingomis modeliavimo kalbomis.
Heterogeninis metaprogramavimas	Paradigma, kuri remiasi tiesiogine koncepcijų atskirtimi, kai naudojamos mažiausiai dvi nepriklausomos kalbos. Žemesnio lygmens kalba vadinama *tikslo kalba* (angl. *target language*), ji išreiškia bazinį srities

	funkcionalumą. Aukštesnio lygmens kalba vadinama *metakalba* (angl. *meta-language*), ja užrašomas bendrinimo algoritmas, kuriame metaparametrais aprašomas srities variantiškumas (Štuikys, Damaševičius, 2013).
Homogeninė transformacija	Transformacija, kai pirminis ir tikslo modeliai yra išreikšti ta pačia modeliavimo kalba.
Homogeninis metaprogramavimas	Metaprogramavimo atvejis, kai bendrieji komponentai kuriami vienos kalbos aplinkoje, naudojant vienos programavimo kalbos abstrakcijas (Štuikys, Damaševičius, 2013a).
Metakalba	Aukštesnio lygmens kalba, kurios paskirtis yra modifikuoti žemesnio lygmens kalba (dar vadinama tikslo kalba) parašytas programas.
Metametaprograma (angl. *meta-program*)	Aukštesnio lygmens programa, kuri kuria kitas, žemesnio lygmens metaprogramas – tai metaprogramų generatorius.
Metamodeliavimas	Aukštesnio abstrakcijos lygmens modeliavimo procesas (žinių išgavimo iš duotos srities procesas naudojant srities analizės metodus), kurį įvykdžius sukuriami metamodeliai.
Metamodelis	Konkrečios srities modeliams sukurti reikalingas konstrukcijų ir taisyklių modelis; modelių modelis.
Metaparametras (angl. *meta-parameter*)	Metaprogramos parametrai valdantys metakalbos konstrukcijas, kurios aprašo manipuliavimą tikslo kalbos kodu. Nuo pasirinkto metaparametro ar metaparametrų reikšmių priklauso sukuriamas konkretus tikslo kalbos programos egzempliorius.
Metaprograma (angl. *meta-program*)	Aukštesnio lygmens programa, kuri kuria kitas, žemesnio lygmens programas – tai programų generatorius. Metaprograma pateikia gimininių tikslo kalbos programos egzempliorių šeimynos bendrintą aprašą, iš kurio generuojamas konkretus egzempliorius ar egzempliorių grupė (Štuikys, Damaševičius, 2013a).
Metaprogramavimas (angl. *metaprogramming*)	Programavimo metodas, kai manipuliuojama kitomis (žemesnio lygmens) programomis kaip duomenimis. Aukšto lygmens programavimo paradigma, kai

	tikslo kalba užrašomas bazinis (konkretus, ar mažai apibendrintas) srities funkcionalumas, o metakalba išreiškiamas bendrasis funkcionalumas tam, kad programa (komponentas) būtų geriau atkartojama ir geriau pritaikoma.
Metodas	Sąmoningai pasirinktas veikimo būdas, veiklos tvarka, sąmoningai naudojama užsibrėžtam tikslui pasiekti.
Modeliavimas	Abstrakčių ar konceptualių modelių kūrimas ir jų analizės procesas.
Modelis	Realaus objekto, proceso arba reiškinio supaprastintas pavaizdavimas (pateikimas) (Dagienė ir kt., 2009). Modelis – tai sistemos abstrakcija, nusakanti ją tam tikru aspektu.
Modelių transformacija	Vienos modelių aibės atvaizdavimas į kitą ar save pačią, kur projektavimas nustato atitikmenis tarp elementų pradiniame ir paskirties modeliuose (Sendall ir kt., 2004).
Pakartotinis naudojimas (angl. reuse)	Egzistuojančios programinės įrangos ar jos dalių bei žinių apie programinę įrangą naudojimas, siekiant sukurti naują programinę įrangą. Disertacijos kontekste naudojamas transformacinis pakartotinis naudojimas generuojant.
PHP (angl. *hypertext preprocessor)*	Interpretuojamoji kalba scenarijams, vykdomiems serveryje, dinaminiam žiniatinklio turiniui (parametrizuotiems dinaminiams tinklalapiams), o pastaruoju metu ir įvairioms taikomosioms programoms kurti (Enciklopedinis kompiuterijos žodynas, http://aldona.mii.lt/pms/terminai/term/enc.html).
Pirminis modelis	Modelis, kuris yra transformuojamas.
Požymiais grįstas modeliavimas (angl. *feature modeling*)	Sistemų arba srities sąvokų pastovių ir kintamų charakteristikų ir ryšių tarp jų modeliavimas naudojant požymiais grįstus modelius.
Požymis (angl. *feature*)	Vartotojui matoma srities charakteristika (Kang ir kt., 1990), kokybinė ypatybė arba funkcinis reikalavimas.
Požymių diagrama (angl.	Speciali grafinė notacija, skirta požymių

***feature diagram*)**	modeliams aprašyti.
Požymių modelis (angl. *feature model*)	Srities modelis, kuriame sritis, jos sistemos arba sąvokos (koncepcijos) aprašomos ir modeliuojamos naudojant požymio sąvoką.
Probleminė sritis (angl. *problem domain*)	Srities sprendžiamų uždavinių visuma. Disertacijos kontekste tyrinėjama sritis, apimanti mokomųjų robotų valdymo programas.
Programinis kodas	Taisyklių, nurodančių, ką turi daryti kompiuteris, rinkinys, parašytas kokia nors programavimo kalba.
Programos transformacija	Procesas keičiantis programos struktūrą iš esmės nekeičiant programos funkcionalumo.
Programų generatorius	Įrankis, kuris iš aukšto lygmens specifikacijos sukuria žemesnio lygmens specifikaciją (pvz., tikslo kalbos programos kodą).
Programų šeimyna (angl. *Product Line*)	Gimininių programų, apibrėžiamų bendrais požymiais, tenkinančiais specifinius srities reikalavimus, aibė.
Projektavimas dėl pakartotinio naudojimo (angl. *design for reuse*)	Sisteminis procesas, kurio paskirtis sukurti bendrąją sistemos architektūrą ir komponentus / generatorius taikymų šeimynai.
Projektavimas su pakartotiniu naudojimu (angl. *design with reuse*)	Programinės įrangos kūrimas iš pakartotinai naudoti tinkamų komponentų. Pakartotinai naudojami standartiniai šablonai bei algoritmai, kurie yra įdiegti generatoriuje ir kurių parametrai yra nustatomi vartotojo komandomis.
Restruktūrizavimas (angl. *refactoring)*	Transformacija keičianti programos kodo vidinę struktūrą, tačiau nekeičianti programos funkcionalumo (Fowler ir kt., 1999).
Skirtybė (angl. *variability*)	Srities kitokia ypatybė, skirtumas. Prielaida (atributas), kuri yra teisinga tik kai kuriems srities objektams.
Specializavimas (angl. *specialization*)	Disertacijos kontekste – tai yra metaprogramos transformavimas į specializuotą versiją, leidžiančią pritaikyti bendrąją parametrizuotą metaprogramą prie konkrečių poreikių ir taikymų. Specializavimas – tai vienpakopės metaprogramos transformavimas į daugiapakopę.

Sprendimo sritis (angl. *solution domain*)	Technologija, kuri naudojama probleminės srities įgyvendinimui.
Taikymas	Giminingų programų, kuriąs sieja bendros veiklos ir galimybės, klasė.
Tikslo kalba (angl. *target language*)	Kalba, skirta srities funkcionalumui išreikšti.
Tikslo modelis	Modelis, gautas iš pirminio modelio, atlikus transformaciją.
Transformacija	Pirminių artefaktų (modelių, programų) vertimas į paskirties artefaktus (modelius, programas) pagal transformavimo taisykles (Štuikys, Damaševičius, 2008).
Transformacijos taisyklė	Aprašas, nurodantis, kaip viena ar kelios pirminės kalbos konstrukcijos gali būti transformuotos į vieną ar kelias tikslo kalbos konstrukcijas (Štuikys, Damaševičius, 2008).
Srities variantiškumas	Srities objektų, besiskiriančių tam tikromis savybėmis, ypatybėmis, gausa. Variantiškumu apibrėžiama pasirinkimo erdvė.
Variantiškumo modeliavimas	Srities bendrybių, skirtybių, skirtybių sąryšių ir specifiškumo modeliavimas taikant įvairias metodologijas.
Verifikavimas	Formalus modelio teisingumo patikrinimas įrodant, kad modelis yra teisingas.

1. ĮVADAS

1.1. Darbo aktualumas

Transformavimas vienos formos objektų (procesų, energijos, gaminių ir pan.) į kitą yra esminis visų techninių sistemų atributas. Informatikoje šis atributas dar svarbesnis dėl šių priežasčių: (1) transformavimo objektai yra ne fiziniai objektai, o jų abstraktūs atvaizdavimai (duomenys, programos, modeliai); (2) egzistuoja gausybė atvaizdavimo formų; (3) abstraktūs atvaizdavimai įgalina daug lengviau įgyvendinti transformavimus; (4) transformavimas informatikoje lemia praktiškai visų kompiuterinių sistemų funkcionalumą.

Transformavimas informatikoje naudojamas įvairiuose kontekstuose ir apima labai platų spektrą, nuo žemiausio iki aukščiausio lygmens. Žemiausiame lygmenyje yra tradicinės transformacijos: procesoriaus, operacinės sistemos. Aukštesniame lygmenyje yra kompiliavimo transformacijos, dar aukščiau – taikomųjų sistemų projektavimo transformacijos, o aukščiausiame – sistemų-sistemų lygmens transformacijos. Programų inžinerijoje ir informatikoje esminės transformacijos atliekamos su programomis ir modeliais. Programos transformavimas yra taikomas konstruojant, optimizuojant, programų sintezėje, pertvarkyme, programinės įrangos atnaujinime, apgrąžos inžinerijoje ir kt. (Visser, 2001).

Transformavimo tyrimai labai platūs, tačiau galima teigti, kad jų visų tikslas vienas – padidinti kuriamų sistemų našumą ir efektyvumą. Pagrindinis transformavimo siekis – automatizavimas.

Per pastarąjį dešimtmetį mokslo ir technikos srityje stebimas ryškus informacinių technologijų šuolis. Bazinių technologijų raida (turima omenyje lustus) pranoko visus lūkesčius. Šiandien mes jau naudojamės tuo pagrindu sukurtomis naujomis technologijomis, gyvename ir dirbame skaitmeniniame pasaulyje, kuriame pokyčiai yra pastovus reiškinys. Vystantis informacinėms technologijoms (IT) sparčiai auga IT vartotojų kategorijos, atsiranda vis didesnis poreikis kuriamas sistemas pritaikyti prie rinkos reikalavimų. Dar viena esminė ypatybė – nepaliaujamai auga programinio kodo svoris (apimtis) sistemose. Tai geriausiai matoma įterptinėse sistemose (pvz., realaus laiko) ir internetiniuose taikymuose (pvz., daiktų internetas).

Sparti technologijų raida ir rinka taip pat lemia projektuojamų sistemų sudėtingumo, dydžio ir sąveikos laipsnio bei programinio kodo augimą. Tai yra dideli iššūkiai sistemų kūrėjams. Koks galimas atsakas į šiuos iššūkius? Technologijų raidos patikrintas atsakas – abstrakcijos lygmens kėlimas tiek nagrinėjant probleminę sritį (t. y. taikymus), tiek sprendimų sritį (t. y. metodus). Todėl kuriamas sistemas siekiama atvaizduoti aukštesniu abstrakcijos lygmeniu, kuriami nauji projektavimo metodai, sistemos sudalijamos į atskiras dalis (koncepcijų atskirtis), kuriant naudojami automatiniai transformavimo įrankiai. Aukštesnis abstrakcijos lygmuo įvairiuose kontekstuose mokslinėje literatūroje įvardijamas kaip metalygmuo (pvz., plačiai naudojamos sąvokos metamodelis, metaduomenys, metakalba, metaprograma ir kt.).

Kita vertus, šiuolaikinių sistemų kūrimas grindžiamas pakartotinio naudojimo (angl. *reuse*) metodologija. Ši metodologija remiasi programų šeimynos koncepcija (jas galima traktuoti kaip metasistemas), apima srities analizę, modeliavimą (modelių ir metamodelių sukūrimą), bendrųjų (meta) komponentų bei programų generatorių kūrimą.

Pastaraisiais metais dominuoja dvi pakartotiniu naudojimu grindžiamos kūrimo metodologijos: OMG modelių inžinerija (angl. *Model-Driven Engineering*, MDE) (OMG, 2014; OMG-MDA, 2014; Schmidt, 2006), kuri paprastai remiasi objektinėmis abstrakcijomis (UML standartas) ir požymių modelių inžinerija, kuri labiau išryškina ir akcentuoja programų šeimynų koncepciją (angl. *Product Line Engineering*, PLE) (Pohl, Böckle and van der Linden, 2005). Abi metodologijos nagrinėjamos dviejuose lygmenyse (srities inžinerijos ir taikymų inžinerijos) skatina sistemų kūrimo procese naudoti aukšto lygmens modelius, jų transformavimą užtikrinant sisteminį pakartotinį panaudojimą, t. y. siekiant aukštesnio automatizavimo laipsnio, didesnio našumo ir kokybės. Nustatyta, kad modeliais grįstos metodologijos yra vyraujančios naujausiuose tyrimuose. Čia dar daug neišspręstų problemų, siejamų su analize, variantiškumo modeliavimu (Capilla, Bosch ir Kang, 2013), atvaizdavimu, transformavimu ir realizacija (Biehl, 2010; Fioravanti ir kt. 2011; Völter ir kt., 2013; Zhang, 2014).

Disertacijoje keliami ir nagrinėjami uždaviniai yra specifiniai, mažai tyrinėti šių metodologijų atvejai: požymių modelių transformavimas į heterogenines metaprogramas, vidinis metaprogramų transformavimas siekiant jų adaptavimo prie konkretaus taikymo. Metaprogramų kūrimo ir tobulinimo procesai sudėtingi, reikalauja gilių žinių tiek iš taikomosios, tiek iš sprendimo srities. Dėl to metaprogramų kūrimo, transformavimo ir palaikymo procesus tikslinga automatizuoti. Kita vertus, šio tipo metaprogramų automatizuotas kūrimas remiantis požymių modelių transformavimu, mūsų žiniomis, išvis nebuvo nagrinėtas. Kadangi metaprogramos yra tikslo (srities) programų generatoriai, todėl galima drąsiai tvirtinti, kad disertacijoje pasirinkta tema yra aktuali ir savalaikė.

1.2. Tyrimo objektas

Darbe tiriama probleminės srities požymių modeliai, metaprogramos, jų kūrimo ir transformavimo procesai ir metodai.

1.3. Darbo tikslas

Darbo tikslas – sukurti ir ištirti heterogeninių metaprogramų automatizuoto kūrimo ir transformavimo metodiką, įskaitant tuos procesus palaikančius įrankius.

1.4. Darbo uždaviniai

1. Išanalizuoti ir įvertinti modelių ir programų (metaprogramų) transformavimo metodus.
2. Sukurti ir ištirti heterogeninių metaprogramų kūrimo metodą panaudojant požymių modelių transformacijas.
3. Sukurti ir ištirti metodą, kuris transformuotų vienpakopę heterogeninę metaprogramą į daugiapakopę.

4. Sukurti ir ištirti metaprogramų kūrimo ir transformavimo algoritmus ir juos realizuoti atitinkamuose įrankiuose.

1.5. Ginamieji teiginiai

1. Probleminės ir sprendimo sričių požymiais grindžiami modeliai įgalina metodiškai kurti metaprogramas automatizuojant kūrimo procesą.
2. Metaprogramų specializavimo ir adaptavimo formalieji modeliai užtikrina transformavimo įrankio funkcionalumą ir korektiškumą.
3. Sukurti ir išbandyti transformavimo įrankiai užtikrina efektyvų metaprogramų kūrimą, transformavimą ir palaikymą.

1.6. Mokslinis naujumas

1. Pasiūlytas ir ištirtas heterogeninių metaprogramų automatizuotas kūrimo metodas, kuris remiasi požymių modelių transformacijomis.
2. Nustatyta daugiapakopės transformacijos uždavinio sprendinių egzistavimo sąlyga bei maksimalus leistinas pakopų skaičius, įgalinantis apibendrinti dvipakopę metaprogramų transformaciją į daugiapakopę.
3. Pasiūlytas išbaigtas procesas, apimantis (i) modelių ir metaprogramų sukūrimą (panaudojant sukūrimo įrankį), (ii) jų transformavimą (panaudojant restruktūrizavimo įrankį) ir (iii) iš metaprogramų sugeneruotų programų pritaikymą.

1.7. Praktinis naujumas

1. Automatizuotas metaprogramų kūrimas (įrankis) ir automatinis mokomųjų robotų valdymo programų generavimas.
2. Automatizuotas daugiapakopių metaprogramų kūrimas (įrankis) ir automatinis metaprogramų bei mokomųjų robotų valdymo programų adaptavimas prie panaudos konteksto.

1.8. Darbo apimtis ir struktūra

Darbą sudaro: terminų ir santrumpų žodynas, septyni skyriai, išvados ir priedai. Apimtis be priedų – 129 puslapiai. Tekstas iliustruojamas 48 paveikslais ir 20 lentelių, cituojama 217 literatūros šaltinių.

1.9. Darbo aprobavimas

Darbas aprobuotas dviejose recenzuojamuose periodiniuose žurnaluose (į trečią žurnalą pateiktas) ir šešiose recenzuojamose konferencijose, iš viso aštuonios publikacijos.

Straipsniai mokslinės informacijos instituto duomenų bazės „ISI Web of Science“ leidiniuose, turinčiuose citavimo indeksą:

1. Štuikys, V., Bespalova, K. & Burbaitė, R. Refactoring of Heterogeneous Meta-Program into *k*-stage Meta-Program. *Information Technology And Control*. ISSN 1392-124X. 2014, *43*(1), p. 14–27. [ISI Web of Science; INSPEC]

2. Burbaitė, R., Bespalova, K., Damaševičius, R. & Štuikys, V. Context-Aware Generative Learning Objects for Teaching Computer Science. *International Journal of Engineering Education*. 2014, *30*(4), p. 929–936. [ISI Web of Science; Scopus]

Kitų tarptautinių duomenų bazių leidiniuose:

1. Burbaite, R., Damasevicius, R., Stuikys, V., Bespalova, K. & Paskevicius, P. Product variation sequence modelling using feature diagrams and modal logic. CINTI 2011 [elektroninis išteklius]: 12th IEEE International Symposium on Computational Intelligence and Informatics, November 21–22, 2011, Budapest, Hungary: proceedings. Budapest: IEEE, 2011. ISBN 9781457700439. p. 73–77. [IEEE/IEE]

2. Štuikys, V. & Bespalova, K. Methodology and Experiments to Transform Heterogeneous Meta-program into Meta-meta-programs. Information and software technologies: 18th International Conference, ICIST 2012, Kaunas, Lithuania, September 13–14, 2012: proceedings / [edited by] Tomas Skersys, Rimantas Butleris, Rita Butkiene. Berlin, Heidelberg: Springer, 2012. ISBN 9783642333071. p. 210–225. [Conference Proceedings Citation Index]

3. Štuikys, V., Bespalova, K. & Burbaitė, R. Generative Learning Object (GLO) Specialization: Teacher's and Learner's View. Information and software technologies: 20th International Conference, ICIST 2014, Druskininkai, Lithuania, October 9–10, 2014: proceedings / [edited by] Giedrė Drėgvaitė, Robertas Damaševičius, Springer International Publishing, 2014. ISBN 9783319119571. p. 291–301. [Conference Proceedings Citation Index]

4. Burbaitė, R. & Bespalova, K. Model-Driven Processes and Tools to Design GLO for CS Education. SIIE 2014 [elektroninis išteklius]: XVI International Symposium on Computers in Education, November 12–14, 2014, Logrono, La Rioja, Spain: proceedings. Logrono: IEEE, 2014. p. 193–199. [IEEE/IEE]

5. Štuikys, V., Bespalova, K. & Burbaitė, R. Feature Transformation-Based Computational Model and Tools for Heterogeneous Meta-Program Design. CINTI 2014 [elektroninis išteklius]: 15th IEEE International Symposium On Computational Intelligence and Informatics, November 19–21, 2014, Budapest, Hungary: proceedings. Budapest: IEEE, 2014. p. 185–190. [IEEE/IEE]

6. Bespalova, K., Štuikys, V. & Burbaitė, R. CS-Oriented Robot-Based GLOs Adaptation through the Content Specialization and Generation. IFIP TC3 Working Conference A New Culture of Learning: Computing and Next Generations, July 1–3, 2015, Vilnius, Lithuania, p. 29–39.

2. MODELIŲ IR PROGRAMŲ TRANSFORMAVIMO METODŲ ANALIZĖ IR ĮVERTINIMAS

2.1. Įvadas

Jau pirmame skyriuje buvo akcentuota, kad kuriant šiuolaikines sistemas svarbų vaidmenį vaidina pakartotinio naudojimo (angl. *reuse*) metodologija. Pakartotinis naudojimas yra egzistuojančios programinės įrangos ar jos dalių bei žinių apie programinę įrangą naudojimas, siekiant sukurti naują programinę įrangą. Pakartotinis naudojimas apima visus resursus, kurie yra naudojami ir sukuriami programų kūrimo metu (Leach, 2012). *Komponentinio atkartojimo atveju* siekiama pakartotinai naudoti komponentus kaip naujos sistemos elementus (Malik, 2012) jų nekeičiant („*juodosios dėžės*“ modelis) arba juos modifikuojant („*baltosios dėžės*“ modelis). *Generatyvinio atkartojimo atveju* programų variantai yra automatiškai sukuriami panaudojant programų generatorius.

Pakartotinis naudojimas, kaip programų kūrimo strategija, pirmą kartą buvo išskirta Mcilroy‘aus (1968). Jis pasiūlė „programos komponentų masinę gamybą“ kaip programinės įrangos kūrimo pagrindą. Vėliau Parnasas (1972) įvedė programų šeimynų terminą, o Neighborsas (1984) pasiūlė sisteminį požiūrį į programų inžineriją ir taikymų sritį. Jis įvedė tokius terminus kaip „dalykinė sritis“ ir „dalykinės srities analizė“. Vėliau šios idėjos buvo vystomos kitų mokslininkų (Gomaa, Webber, 2004; Kang, Lee ir Donohoe, 2002). Pakartotinio naudojimo metodus, modelius ir sąnaudas detaliai aprašė savo knygoje Leachas (2012). Programų inžinerijos institutas (angl. *Software Engineering Institute*) visas pakartotinio naudojimo idėjas apjungė į vieningą programų šeimynų kūrimo metodą (SEI, 2012). Reikia pažymėti, kad pakartotinio naudojimo metodai (ypač įvairios programų transformacijos) taip pat tiriami ir nagrinėjami informatikos ir kompiuterijos moksliniuose forumuose.

Programų šeimynų inžinerijos (angl. *Product Line Engineering*, PLE) (Pohl ir kt., 2005; van der Linden, Schmid ir Rommes, 2007) metodologija pastaraisiais metais plačiai taikoma kuriant programinę įrangą. Kuriant programų šeimynas siekiama pagerinti kuriamo produkto kokybę, sumažinti kainą ir kūrimo laiką. Pakartotinai naudojami įvairūs anksčiau sukurti aktyvai: reikalavimai, architektūra, metodai, procesai, programos komponentai, testai, įrankiai ir kt. Intensyvūs modeliavimo, modelių ar programų transformavimo ir programų generatorių kūrimo tyrimai.

Šio skyriaus tikslas – pateikti analizę tų mokslinių šaltinių, kurie labiausiai atitinka tiriamąjį objektą ir metodus. Visi jie susiję su pakartotinio naudojimo koncepcijomis. Pasirinktas toks analizės būdas: iš pradžių analizuojami bendri dalykai, po to tematika laipsniškai siaurinama ir nagrinėjamos siauresnės temos bei su jomis susijusios problemos. Pasirinktas analizės būdas realizuotas tokia apžvalgos struktūra:

2.2 skyrelyje pateikiami *srities inžinerijos aspektai*, apimantys srities analizę, variantiškumą, analizės metodus, srities modeliavimą požymiais ir kt. 2.3 skyrelyje apžvelgti šaltiniai, susiję su srities modelių ir programų transformacijomis; 2.4

skyrelyje pateikiama programų kūrimo automatizavimo apžvalga; 2.5 skyrelyje apžvelgti šaltiniai, susiję su metaprogramavimo metodologija; 2.6 skyrelyje pateikiamas disertacijos temos pagrindimas ir metodika; 2.7 skyrelyje suformuluotos skyriaus išvados.

2.2. Srities inžinerija ir taikymų inžinerija

Srities inžinerija (angl. *Domain engineering*) – tai procesas, kuriuo siekiama sukurti bendrinę sistemos architektūrą ir komponentus / generatorius taikymų šeimynai. Tik pažinus sritį galima kurti sistemas ir taikymus, o taikymai kuriami sudarant srities modelius, modeliuojant ir modelius transformuojant į tam tikras programas.

Srities inžinerija, dar vadinama programų šeimynų inžinerija, yra procesas pakartotinai panaudojantis srities žinias kuriant naujas programines sistemas (Bosch, 2000). Apelas ir kt. (2013) aprašė programų šeimynų projektavimą, priežiūrą ir plėtrą, klasifikavo įrankius ir naudojamas technologijas.

Sritį galima apibrėžti kaip rinkinį uždavinių ar funkcijų, kurie gali būti išspręsti per taikymus toje srityje (Harsu, 2002; Tracz, 1994). Sametingeris (1997) sritį apibrėžia kaip veiklos ar žinių erdvę, turinčią taikymų, kuriuos sieja bendros veiklos ir galimybės. Programų inžinerijoje sritis suprantama kaip panašių ar giminingų programų sistemų klasė.

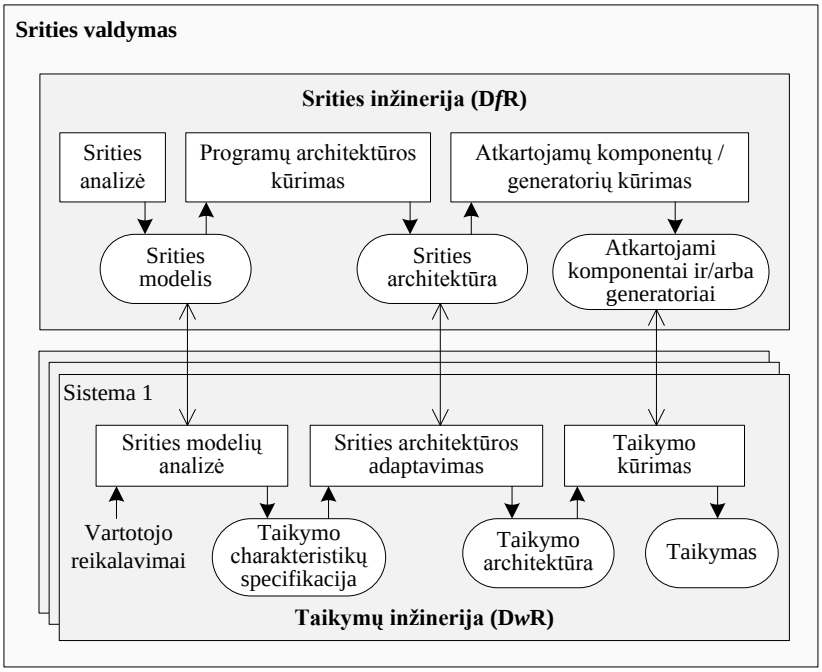
Bet kuri sritis gali būti padalyta į siauresnes sritis, vadinamas posistemėmis. Jei srities funkcionalumas išreiškiamas vienos sistemos vienu posistemiu, sritis vadinama sutrauktąja (ang. *encapsulated*). Jei srities funkcionalumas yra pasiskirstęs vienos ar kelių sistemų keliuose posistemiuose, tuomet sritis yra paskirstytoji (angl. *distributed*).

Srities inžinerija taip pat nagrinėja analizės metodus, žinių ir artefaktų išgavimą, jų atvaizdavimą ir saugojimą. Srities inžinerija susideda iš analizės, projektavimo ir realizacijos etapų, kurių metu sukuriami srities modeliai, srities kalba, bendrieji kodo generatoriai ir pakartotinio panaudojimo komponentai abstrakčiame lygmenyje.

Srities inžinerija apima vienos arba kelių sričių identifikavimą, bendrybių ir skirtybių nustatymą, lengvai pritaikomos architektūros konstravimą ir sprendimus, kaip sukurti sistemą iš generatorių ir pakartotinai naudojamų komponentų. Bendrybės apima esmines srities objektų savybes ir jų elgseną, o skirtybės parodo objektų savybių skirtumus. Srities inžinerija susideda iš analizės, projektavimo ir realizacijos etapų.

Taikymų inžinerija vadinamas procesas arba veikla, kuri sukuria programinius produktus iš modelių, gautų srities inžinerijoje. Taikymų inžinerija susideda iš sistemos analizės, sistemos projektavimo ir sistemos realizacijos etapų, kurių metu sukuriama konkreti realizacija.

Srities inžinerija dar vadinama projektavimu dėl pakartotinio naudojimo (angl. *design-for-reuse*, D*f*R), o taikymų inžinerija – projektavimu su pakartotiniu naudojimu (angl. *design-with-reuse*, D*w*R). Srities inžinerijos ir taikymų inžinerijos procesų sąveiką atspindi dvynių modelis (angl. *twin model* terminas pasiūlytas Sametingerio (1997)) (2.1 pav.).



2.1 pav. Srities inžinerijos ir taikymų inžinerijos sąveika (adaptuota iš Pohl ir kt., 2005)

Srities analizė vaidina labai svarbų vaidmenį pakartotinio naudojimo metodologijoje. Srities analizė apibrėžiama kaip procesas, skirtas analizuoti susijusias sistemas ieškant bendrų ir kintamų dalių. Srities analizės metu nagrinėjamos taikymų šeimynos, nustatomi, išgaunami ir organizuojami duomenys taip, kad jie tiktų pakartotiniam naudojimui kuriant naujas sistemas. Srities analizės procesas sudėtingas, nes sritys yra pasiskirsčiusios ir persidengiančios, sritys tobulėja, o taikymai migruoja.

2.2.1. Srities analizės metodai

Terminą srities analizė (angl. *domain analysis*) pirmasis pasiūlė Neighborsas (1980). Jis nagrinėjo gimininguų taikymų programinės įrangos kūrimą. Sritį sudaro bent du taikymai, o taikymą – bent dvi giminingos programos. Srities analizės rezultatai: srities apibrėžimas, modelis, reikalavimų modelis, architektūros modelis, srities taksonomija, bendrybių ir skirtybių aprašas, srities kalba kaip bendrasis modelis, srities standartai, atsikartojantys komponentai. Srities analizės metu sukuriamas srities modelis. Harsu (2002) pasiūlė srities modelį sudaryti iš šių elementų: srities apimtis, bendrybės, žodynas, notacijos ir reikalavimai. Ferré ir Vegas (1999) suklasifikavo srities analizės metoduose naudojamas duomenų analizės, klasifikavimo, variantiškumo pateikimo technikas (2.1 lent.). Srities analizės metodų yra daug, dalis jų pateikiama 2.2 lent.

2.1 lentelė. Sričių analizės metoduose naudojamos duomenų analizės, klasifikavimo ir variantiškumo pateikimo technikos (Ferré, Vegas, 1999)

Duomenų analizė	Klasifikavimas	Variantiškumas
Objektinė Funkcinė dekompozicija Kiekybinė analizė Kokybinė analizė Atvejais pagrįsta technologija	Aspektai (angl. *Facets*) Požymiai (angl. *Features*) Galimybės (angl. *Capabilities*) Reikalavimai 3C modeliai (angl. *3C's model*) Objektinės analizės metodai Daliniai modeliai (angl. *Partial patterns*)	Deriniai (angl. *Combinations*) Kompromisai (angl. *Compromises*) Specializavimas Patobulinimai (angl. *Refinements*) Sąryšiai (angl. *Relations*) Parametrizacija Kodo generavimas

2.2 lentelė. Srities analizės metodai

Metodas	Naudojama analizės technika	Etapai
FODA (angl. *Feature-Oriented Domain Analysis*) (Bontemps ir kt., 2004; Kang ir kt., 1990)	Požymiai	Konteksto analizė Srities modeliavimas Architektūros modeliavimas
FORM (angl. *Feature-Oriented Reuse Method*) (Kang ir kt., 2002)	Požymiai	Konteksto analizė Požymių modeliavimas Architektūros modeliavimas
FeatureRSEB (angl. *Feature Reuse-Driven Software Engineering Business*) (Griss, Favaro ir d'Alessandro, 1998)	Požymiai, orientuotas į objektą	Srities analizė Modelio projektavimas Architektūros apibrėžimas
CBFM (angl. *Cardinality-Based Feature Modeling*) (Czarnecki, Helsen ir Eisenecker, 2005)	Požymiai	Srities modeliavimas Požymių modelis
ConIPF (angl. *Configuration in Industrial Product Families*) (Wolter ir kt., 2006)	Požymiai	Reikalavimų ir konteksto analizė Srities modeliavimas Požymių ir artefaktų diagrama
DSSA (angl. *Domain-Specific Software Architectures*) (Tracz, 1994)	Požymiai	Reikalavimų analizė Architektūros modeliavimas
DARE (angl. *Domain Analysis and Reuse Environment*) (Frakes, Prieto ir Fox, 1998)	Požymiai	Analizė ir projektavimas Architektūros projektavimas
FAST (angl. *Family Oriented Abstraction, Specification and Translation*) (Weiss, 1999)	Giminingi požymiai	Srities analizė Srities realizacija

Plačiausiai naudojamas 1990 m. SEI (angl. *Software Engineering Institute*) pristatytas FODA metodas (Kang ir kt., 1990). Šis metodas remiasi gimininingų sistemų esminių savybių nustatymu. FODA metodas išskiria srities požymius, juos klasifikuoja, atvaizduoja srities variantiškumą, suskaldo funkcijas, modeliuoja srities savybes ir architektūrą. Tai pirmasis bandymas formaliai aprašyti variantiškumo modelį, įskaitant grafinę notaciją. Naudojant FODA metodą, srities analizės procesas suskirstomas į tris etapus:

1. *Konteksto analizės etapas*. Apibrėžiama srities apimtis, ribos, analizuojami srities sąryšiai su kitomis sritimis ir duomenų srautai tarp jų.

2. *Srities modeliavimo etapas*. Analizuojamos srities taikymų bendrybės ir skirtybės. Atliekama požymių analizė, modeliuojami esybių sąryšiai, atliekama funkcinė analizė. Gaunami modeliai, aprašantys sprendžiamo uždavinio skirtingus aspektus.
3. *Architektūros modeliavimo etapas*. Sukuriamas aukšto lygmens architektūros modelis.

Laikui bėgant FODA notacija įgavo požymių diagramų (angl. *feature diagram*) pavadinimą. Nuo to laiko ši notacija jau pasikeitė ir šiuo metu galima rasti įvairių modifikuotų versijų (Eriksson, Börstler ir Borg, 2005; Hubaux, Tun ir Heymans, 2013; Laguna, Marqués ir Rodriguez-Cano, 2011).

2.2.2. Variantiškumo valdymas

Programų šeimynų inžinerijoje svarbų vaidmenį vaidina variantiškumo valdymas (angl. *variability management*) (Capilla ir kt. 2013; Simmonds ir kt., 2012). Programų šeimynas patogu projektuoti panaudojant požymių modelius, kuriuose išskiriami požymiai bendri visiems atvejams, būdingi tik kai kuriems atvejams arba specifiniai, kurie priklauso tik vienam konkrečiam atvejui. Srities bendrybės ir skirtybės pasireiškia kartu, o specifiškumą sritis gali turėti ir gali neturėti.

Variantiškumo modelis turi turėti aiškią struktūrą ir parametrų atskyrimo mechanizmus, kurie galėtų išskirti požymius, aprašančius konkretų programos šeimynos taikymą. Šio modelio elementai gali būti susieti su tam tikrais pakartotinai naudojamais elementais ir tokiu būdu, tik atlikus atranką iš variantiškumo modelio, galima automatiškai išskirti reikiamą rinkinį programos komponentų, testų ir kt. naujam produktui kurti.

Kruegeris (2002) variantiškumo valdymą skirsto į 9 grupes (2.3 lent.). Eilutėse pateikiami pakartotinai naudojami elementai, pirmuose dviejuose stulpeliuose parodyti šių elementų keitimo būdai, o trečiajame stulpelyje pateikti keitimus palaikantys metodai.

2.3 lentelė. Variantiškumo valdymas (Krueger, 2002)

	Nuoseklus laikas	**Lygiagretus laikas**	**Srities erdvė**
Failai	Versijų valdymas (pasikeitusių failų versijų valdymas)	Išsišakojimų valdymas (išsišakojusių nepriklausomų failų versijų valdymas)	Variantinių taškų valdymas (variantiškumo palaikymas failų lygyje)
Komponentai	Bazinis valdymas (periodinis tarpinių rezultatų fiksavimas)	Išsišakojimų bazinis valdymas (periodinis fiksavimas tarpinių rezultatų šakose)	Pritaikymo valdymas (Komponentų rinkinių, kurie tenkina programų šeimynų reikalavimus, valdymas)
Produktai	Komponavimo valdymas (kuriamo produkto rezultato, kuris sudarytas iš rinkinio komponentų, fiksavimas)	Išsišakojimų komponavimo valdymas (fiksavimas kuriamo produkto rezultato šakose)	Pritaikymo komponavimo valdymas (komponentų, sudarančių galutinį pritaikytą produktą, valdymas)

Pagrindinis variantiškumo modelio tikslas – atspindėti sistemos, įeinančios į programų šeimyną, bendrybes, skirtybes, sąryšius ir specifiškumą. Modeliuojant srities variantiškumą, bendrybėms identifikuoti sudaromi jų scenarijai, o skirtybėms – nustatomos parametrų kitimo reikšmės ir ribos, sudaromi skirtybių scenarijai. Modeliuojami skirtybių persidengimai (priklausomybės ir sąryšiai). Modeliuojant srities variantiškumą stengiamasi nustatyti srities specifiškumą ir jį izoliuoti nuo kitų savybių.

Variantiškumo modeliavimo srityje atliekama daug tyrimų: variantiškumo modelių formalizavimo, skirtingų technologinių aspektų integravimo, procesų standartizavimo ir tobulinimo. Taip pat didelis dėmesys skiriamas įrankiams kurti. Egzistuoja nemažai programinių priemonių, leidžiančių valdyti variantiškumą – „pure::variants“ (Beuche, 2012), „CVL Wiki“ (2012), „Gears“ (2012), „PLUM“ (2011), „TVL“ (Boucher ir kt., 2010), „FeatureIDE“ (Kästner ir kt., 2009), „FAMILIAR“ (2009), „SPLOT“ (Mendonca, Branco ir Cowan, 2009; SPLOT, 2009), „FeatureMapper“ (2007), „FMP“ (2005), „XFeature“ (2004) ir kt. Variantiškumo valdymo įrankių apžvalga pateikiama 2.4 lent.

2.4 lentelė. Variantiškumo valdymo įrankiai

Įrankio pavadinimas	Sukūrimo metai	Aprašymas	Licencija
Gears	1999	Grafinė notacija, BNF (angl. *Backus Naur Form*) gramatika, UML, SysML modeliai, reikalavimai, programos kodo komponentai (java, C, C++, C#, Ada, Perl, XML, HTML), testai (JUnit, NUnit, AUnit ir CppUnit), dokumentacija. Integracija su Microsoft Visual Studio, Eclipse, Serena Dimensions CM, Perforce, IBM Rational ClearCase, IBM Rational Synergy, IBM Rational Team Concert, Subversion, CVS. Produktas sukurtas JAV kompanijos BigLever Software Inc.	Komercinė
pure::variants	2004	Įrankis realizuotas kaip Eclipse įskiepiai. Grafinė notacija, programos kodo komponentai (java, C, C++), XML, UML SysML modeliai, AUTOSAR modeliai. Produktas sukurtas Vokietijos kompanijoje pure-systems GmbH, Magdeburgo Otto-von-Guericke universiteto (angl. *Otto-von-Guericke University Magdeburg*) ir Fraunhoferio kompiuterių dizaino ir programinės įrangos instituto (angl. Fraunhofer Institute for Computer Architecture and Software Technology) mokslininkų.	Komercinė
FeatureIDE	2004	Įrankis, sukurtas Eclipse platformoje, palaiko: Java, JavaCC, C, C++, C#, Haskell ir XML. UML, SysML modeliai, grafinė notacija, BNF gramatika, programinio kodo komponentai. Produktas sukurtas Vokietijos Magdeburgo Otto-von-Guericke universitete.	Nekomercinė, atvirojo kodo

Įrankio pavadinimas	Sukūrimo metai	Aprašymas	Licencija
FMP (angl. *Feature Modeling Plug-in*)	2004	Grafinė notacija, XPath 2.0. Produktas sukurtas Kanados Waterloo universitete (angl. University of Waterloo).	Nekomercinė, atvirojo kodo
XFeature	2005	XML pagrindu sukurtas įrankis, kuria XML dokumentus. Produktas sukurtas Šveicarijos Ciuricho federaliniame technologijos institute (angl. *Swiss Federal Institute of Technology, ETH-Zurich*)	Nekomercinė
PLUM (angl. *Product Line Unified Modeller*)	2007	Įrankis sukurtas Eclipse platformoje. Grafinė notacija, OCL 2.0, DPV (angl. *DirectProduct Variability*) metodas. Produktas sukurtas Ispanijos Europos programinės įrangos instituto (angl. *European Software Institute*).	Nekomercinė
FeatureMapper	2008	Įrankis sukurtas Eclipse platformoje. Grafinė notacija, EMF modeliai. Produktas sukurtas Vokietijos Drezdeno technologijos universitete (angl. *Dresden University of Technology*).	Nekomercinė
FAMILIAR (angl. for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning)	2009	Įrankis sukurtas Java kalba, naudojant Xtext. Grafinė notacija, XML. Produktas sukurtas Prancūzijos Nice-Sophia Antipolis universiteto (angl. *University of Nice Sophia Antipolis*) I3S laboratorijoje (CNRS UMR 6070) ir JAV Kolorado valstybiniame universitete (angl. *Kolorado State University*).	Nekomercinė
SPLOT (angl. *Software Product Lines Online Tool*)	2009	Internetinė sistema sukurta Java Servlet technologijos pagrindu, SXFM kalbos notacija, BNF gramatika, palaiko XML formatą. Produktas sukurtas Kanados Waterloo universitete (angl. University of Waterloo).	Nekomercinė, atvirojo kodo
CVL Wiki	2009	Įrankis sukurtas Eclipse platformoje. Grafinė notacija, XMI modeliai, CVL (angl. *Common Variability Language*) kalba. Produktas sukurtas Norvegijos technologijos instituto (angl. *Norwegian Institute of Technology*) įsteigtos kompanijos SINTEF.	Nekomercinė
TVL (angl. *Text-Based Variability Language*)	2010	Tekstinė kalba, BNF gramatika. Produktas sukurtas Belgijos Namuro universitete (angl. *University of Namur*)	Nekomercinė, atvirojo kodo

Įrankių gausa parodo, kad variantiškumo modeliavimo tyrimai aktyviai vystosi. Dauguma įrankių yra eksperimentiniai, jie sukurti universitetų mokslininkų realizuojant savo mokslines idėjas (išimtis „pure::variants“ ir „Gears“). Todėl įrankiuose siūloma sava kalba, o tai parodo, kad šiuo metu dar nėra nusistovėjusių variantiškumo modeliavimo kalbų. Nėra nusistovėjusių požymių modeliavimo standartų.

2.2.3. Požymių diagramos

Požymių modelis aprašo *privalomuosius* (angl. *mandatory*), *alternatyviuosius* (angl. *alternative*) ir *pasirenkamuosius* (angl. *optional*) srities požymius, tėvo-vaiko sąryšius ir požymių tarpusavio apribojimus (Riebisch, 2003). Požymių modeliai vizualiai atvaizduojami požymių diagramomis.

Požymių diagramos pirmą kartą buvo pasiūlytos K. Kango 1990 metais kaip FODA (angl. *Feature Oriented Domain Analysis*) metodo dalis (Kang ir kt., 1990). Požymių diagrama – tai grafinė kalba naudojama modeliuoti sistemos ar komponento bendrybes ir skirtybes aukštesniame abstrakcijos lygyje.

Kuriant programinę įrangą požymių diagramos dažniausiai naudojamos srities analizei pradiniuose projektavimo etapuose. Kadangi požymių diagramos nagrinėjamą sritį atvaizduoja aukštesniame apibendrintame lygyje, jos padeda lengviau suvokti sudėtingų sistemų galimybes.

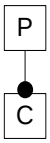
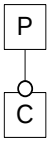
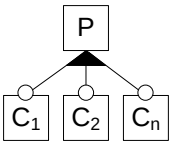
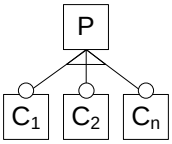
Požymių diagramos naudojamos įvairiuose programinės įrangos kūrimo metodologijose: modeliais grindžiamas kūrimas (angl. *model driven development*) (Trujillo, Batory ir Diaz, 2007), požymiais grįstas programavimas (angl. *feature oriented programming*) (Batory, 2006), generatyvinis programavimas (angl. *generative programming*) (Czarnecki ir kt. 2002; Schlee, Vanderdonckt, 2004) ir kt.

Požymių diagrama yra kryptinis aciklinis grafas, kurį sudaro viršūnių, tiesinių briaunų ir briaunų kombinacijų rinkinys (Štuikys, Damaševičius, 2013a). Šakninis elementas atvaizduoja aukščiausio lygmens požymį (pvz., sritis, sistema, komponentas). Tarpinės viršūnės atvaizduoja sudėtinius požymius, o diagramos paskutinės viršūnės atvaizduoja požymius, kurie nėra skaidomi į smulkesnius požymius. Grafo briaunos nurodo ryšius arba priklausomybes tarp požymių, o žymos ant briaunų nurodo, kokio tipo ryšys yra tarp požymių. Požymis apibrėžiamas reikalavimais ir programinėje įrangoje atsispindi tam tikru funkcionalumu ar kokiu kitu nefunkciniais reikalavimai nustatytu požymiu.

Požymiai tarpusavyje gali būti susieti trijų tipų *tėvo-vaiko* ryšiu: p*rivalomieji* (IR), *neprivalomieji* (ARBA), a*lternatyvieji* (ARBA, variantinis (XOR)). Privalomieji požymiai atspindi srities bendrybes, t. y. būtinas ir nekintančias srities objekto charakteristikas. Neprivalomieji ir alternatyvieji požymiai atspindi srities skirtybes, t. y. kintamas srities objekto charakteristikas. Apribojimai nusako funkcinius ryšius arba apribojimus tarp požymių. Požymių diagramose naudojami požymių tipai pateikti 2.5 lent.

Požymių diagramos grafinis aprašas žmonėms, neturintiems techninių žinių, yra suprantamesnis. Tačiau modeliuojant dideles sistemas požymių diagramos darosi sunkiai skaitomos dėl požymių ir sąryšių gausos, dažnai ryšiai tarp požymių tarpusavyje persipina. Dėl šių priežasčių yra populiarios tekstinės požymių modelių specifikavimo kalbos. Toks aprašymo būdas suteikia galimybę perduoti duomenis iš vienos sistemos į kitą. Tekstinės požymių modelių aprašymo kalbos: SFXM (angl. *Simple XML Feature Model*), FDL (angl. *Feature Description Language*), TVL (angl. *Text-based VariabilityLanguage*), OWL (angl. *Web Ontology Language*), Prolog, FAMILIAR (angl. *FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) ir kt.

2.5 lentelė. Požymių diagramų sintaksė (adaptuota remiantis Thum, Batory ir Kastner, 2009; Benavides, Segura ir Ruiz-Cortés, 2010)

Požymio tipas	**Apibrėžimas**	**Grafinė notacija (sintaksė)**	**Teiginių logikos žymėjimas**	**Ribojimų tenkinimo žymėjimas**
Privalomas: IR tipo ryšys	Sistema turės privalomą požymį, jeigu ji turi to požymio tėvinį požymį (jei P tai ir C, P – tėvas; C – vaikas.		$P \leftrightarrow C$	$P = C$
Neprivalomas: ARBA tipo ryšys	Sistema gali turėti neprivalomą požymį, jeigu ji turi to požymio tėvinį požymį (jei P tai C arba nei vienas požymis).		$C \rightarrow P$	$if\,(P = 0)$ $C = 0$
Alternatyvinis: ARBA pasirinkimas	Sistema turi mažiausiai vieną ARBA tipo alternatyvinį požymį, jeigu ji turi to požymio tėvinį požymį		$P \leftrightarrow (C_1 \vee C_2 \vee$ $... \vee C_n)$	$if\,(P > 0)$ $Sum(C_1, C_2, ..., C_n)$ $in\{1..n\}$ $else\ C_1 = 0,$ $C_2 = 0, ..., C_n = 0$
Alternatyvinis: variantinis pasirinkimas (XOR)	Sistema gali turėti tik vieną variantinį alternatyvinį požymį, jeigu ji turi to požymio tėvinį požymį		$(C_1 \leftrightarrow (\neg C_2 \wedge ...$ $\wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge ...$ $\wedge \neg C_n \wedge P)) \wedge$ $(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2$ $\wedge ... \wedge \neg C_{n-1} \wedge P))$	$if\,(P > 0)$ $Sum(C_1, C_2, ..., C_n)$ $in\{1...1\}$ $else\ C_1 = 0,$ $C_2 = 0, ..., C_n = 0$
Apribojimas *<reikalauja>*	Požymis **A** reikalauja, kad būtinai būtų pasirinktas požymis **B**		$A \rightarrow B$	$if\,(A > 0)$ $B > 0$
Apribojimas *<išskiria>*	Požymiai **A** ir **B** vienu metu negali būti pasirinkti		$\neg(A \wedge B)$	$if\,(A > 0)$ $B = 0$

Šiuo metu požymių diagramos nėra standartizuotos, vyrauja didelė notacijų įvairovė, nėra sukurtų vieningų metamodelių, tyrėjai skirtingai interpretuoja diagramų elementus. Todėl nėra standartinių modeliavimo įrankių. Egzistuojantys įrankiai dažniausiai palaiko duomenų integravimą panaudojant XML.

2.2.4. Konteksto modeliavimas

Informacinės sistemos vis labiau integruojamos į įvairias žmogaus veiklos sritis. Organizacijos nuolatos vystosi, keičiasi jų procesai, o su jais ir procesų automatizavimo reikalavimai. Atsiranda poreikis sistemas adaptuoti pagal vartotojų poreikius. Tai sukuria prielaidas vystyti naujas technologijas, kurios leidžia vykdyti sistemų pritaikymą prie nuolat besikeičiančio konteksto. Prisitaikymas tampa viena iš pagrindinių sistemų savybių, užtikrinančių kūrimo efektyvumą, sistemos funkcionalumą, jų tobulinimą ir sistemų išliekamumą.

Prisitaikančioji (angl. *self-adaptive*) programinės įrangos sistema yra tokia sistema, kuri keičia savo elgesį reaguodama į pokyčius savo veiklos aplinkoje (Oreizy ir kt., 1999). Adaptavimas – tai procesas, kuris keičia arba išplečia posistemės elgesį, įjungdamas ar pagerindamas bendradarbiavimą su aplinkinių sistemų dalimis (kurios yra jos aplinka). Išskiriamos dvi adaptacijos rūšys: funkcionalumo ir korektiškumo (Kell, 2008).

Pagal Oppermanną (1994) sistema laikoma adaptyvia, jei ji atsižvelgdama į vartotojo poreikius pajėgi automatiškai pakeisti savo savybes. Adaptyviosios sistemos valdo sąveiką su vartotoju ir atsižvelgdamos į turimą informaciją keičia vartotojo sąsają arba savo elgesį. Jamesonas (2003) pabrėžia, kad adaptyvioji sistema – tai interaktyvi sistema, kuri savo elgesį pritaiko pagal kiekvieną vartotoją, remiantis sukaupta informacija apie jį. Sistema, gavusi informaciją apie vartotoją ir jo aplinką, pritaiko savo elgesį. Adaptavimas – tai procesas, kurio metu vartotojui, atsižvelgiant į kontekstą, yra pateikiamas dinamiškai sugeneruotas arba parinktas objektas, atitinkantis vartotojo poreikius.

Programinių sistemų prisitaikymo ir tuo pačiu konteksto modeliavimo būtinybę lemia:

1. Technologijų pažanga (mobilūs skaičiavimai, sklindantys (angl. *pervasive*) skaičiavimai, skaičiavimai debesyse, e. mokymasis ir kt.);
2. Didėjantys vartotojų reikalavimai, sistemų sudėtingumo augimas.

Tinkamai parinktas konteksto modelis sumažina kontekstinių (angl. *context-aware*) taikymų sudėtingumą, pagerina jų priežiūrą ir tobulinimo galimybes.

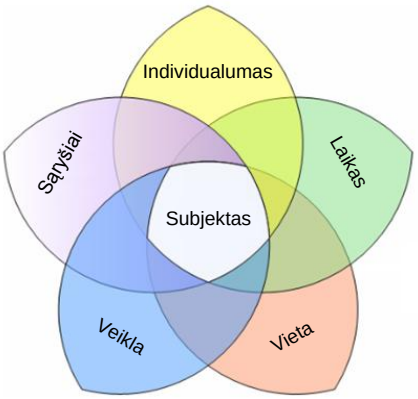
Dey‘as, Abowdas ir Salberas (2001) kontekstą apibrėžia kaip bet kokią informaciją, kuri gali būti naudojama apibūdinti subjekto būsenai. Subjektas yra asmuo, vieta, objektas, kuris susieja vartotoją ir taikomąją programą, taip pat pats vartotojas ir taikomoji programa. Kontekstas gali būti apibrėžiamas kaip aplinkos ir atitinkamų sąlygų rinkinys, sudarantis unikalią ir suprantamą situaciją (Brézillon, 2003). Konteksto apibrėžimą patikslina Zimmermannas, Lorenzas ir Oppermannas (2007), jie aprašant kontekstą informaciją skaido į 5 kategorijas: individualumo, veiklos, vietos, laiko ir santykių (2.2 pav.). Veiklos informacija daugiausia lemia elementų tinkamumą konkrečiose situacijose. Vietos ir laiko informacija nulemia sąryšius ir informacijos mainus tarp subjektų. Visa su subjektu susieta informacija gali būti laikoma nuolat kintančiu to subjekto kontekstu.

Bettinis ir kt. (2010) pateikia konteksto modeliavimo ir valdymo reikalavimų rinkinį:

1. *Nevienalytiškumas ir mobilumas*. Konteksto modelis turi išreikšti įvairių tipų kontekstinę informaciją, o konteksto valdymo sistemos – numatyti

šios informacijos valdymą. Kontekstinė informacija turi būti adaptuojama kintančioje aplinkoje.

2. *Sąryšiai ir priklausomybės*. Konteksto elementus sieja įvairūs ryšiai, kurie užtikrina, kad sistemos veiktų korektiškai. Vienos savybės reikšmės pasikeitimas sukelia kitų savybių reikšmių pokyčius.
3. *Savalaikiškumas*. Kontekstiniai taikymai turi prieiti prie informacijos apie buvusias būsenas (angl. *past states*) ir būsimas būsenas (angl. *futures states*), prognozes. Jei atnaujinimų skaičius yra didelis, savalaikiškumą (konteksto istoriją) valdyti sunku.
4. *Neišbaigtumas*. Dėl heterogeninės prigimties konteksto informacijos kokybė gali kisti (pvz., jutiklių netikslumai). Taip pat konteksto informacija gali būti nepilna ir konfliktuoti su kita informacija.
5. *Samprotavimai*. Kontekstiniai taikymai naudoja konteksto informaciją įvertinimui, ar yra vartotojo ir / ar aplinkos pokytis. Priimant sprendimą naudojamos modelio verifikavimo ir samprotavimo technikos.
6. Formalizmo taikymas palengvina konteksto informacijos panaudojimą kuriant modelius ir kontekstines sistemas.



2.2 pav. Pagrindinės konteksto informacijos kategorijos (Zimmermann ir kt., 2007)

Konteksto modeliavimo rezultatas yra konteksto modelis. Pagal duomenų struktūras, naudojamas kontekstinės informacijos mainuose, konteksto modeliai klasifikuojami į raktinių reikšmių (angl. *Key-Value*), ženklinimu grindžiamus (angl. *Markup-based*), grafinius, objektinius (angl. *Object-oriented*), logika grįstus (angl. *Logic-based*) ir ontologijomis grįstus (angl. *Ontology-based*) modelius (Schmohl, Baumgarten, 2008).

Dauguma kontekstinių sistemų pritaiko savo elgesį prie individualaus vartotojo konteksto, atsižvelgdamos į fizinius ir socialinius aspektus (Dourish, 2004). Kontekstas apima įvairaus tipo informaciją, pavyzdžiui, vartotojo geografinė padėtis, aplinkos sąlygos, buvimo vieta (Cheverst ir kt., 2000), paslaugos ar interesų objektas (Zimmermann, Specht ir Lorenz, 2005). Pavyzdžiui galima pateikti sparčiai tebeaugantį internetą, kuris apima vis daugiau visuomenės sluoksnių ir šalių. Šiuo metu e. verslo strategijose būtinu sėkmės aspektu tampa prisitaikymas prie visuomenės sluoksnių, kultūros dimensijų. Internetinės svetainės turi būti

adaptuojamos įvertinant šalies techninį išsivystymą (Sinkovics, Yamin ir Hossinger, 2007).

Baldaufas, Dustdaras ir Rosenberg (2007) pristato kontekstinių sistemų (angl. *context-aware systems*) tyrimą, bendrus kontekstinių sistemų projektavimo principus. Haake ir kt. (2010) siūlo konteksto lygmenų modelį (angl. *layered context model*), skirtą specifikuoti srities žinias. Šiame modelyje pateikiama konteksto informacija ir taisyklės, kuriomis vadovaujantis atliekamas paslaugos teikimo prisitaikymas.

Kontekstinė informacija plačiai naudojama ir mokymo sistemose, siekiant palengvinti mokymosi ir mokymo uždavinius (Abarca ir kt., 2006; Burbaitė, 2014; Figueiredo, 2010). Siekiama adaptuoti mokymosi kontekstą pagal besimokančiojo poreikius. Mokymasis vyksta įvairialypėje aplinkoje, todėl įtraukimas kontekstinės informacijos apie vartotoją padeda ištirti, kaip kurti intelektualiąsias sistemas (angl. *intelligent systems*), kurios gali geriau planuoti ir numatyti vartotojų poreikius ir efektyviau reaguoti į jų elgesį (Verbertir kt., 2012).

Analizė ir konteksto modeliavimas ir įtraukimas į sistemą yra išmaniųjų ir adaptyviųjų sistemų kūrimo pagrindas. Tokios sistemos gali suvokti aplinkas, mokosi iš atliktų veiksmų istorijos ir priima sprendimus. Kontekstą suvokiančios sistemos jau egzistuoja ir sparčiai tobulėja. Galima teigti, kad šios sistemos yra ateities sistemos (pažangūs robotai, virtualūs patarėjai ir kt.).

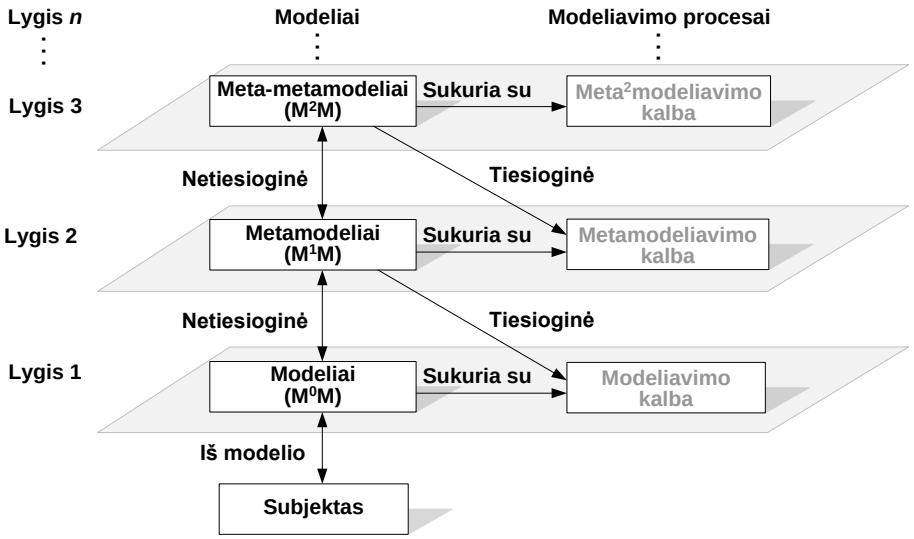
2.2.5. Metamodeliavimas

Naujų sistemų projektavimas yra sudėtingas procesas. Siekiant šį procesą palengvinti keliamas abstrakcijos lygmuo, sudaromi sistemos modeliai, atspindintys tik tam tikras, esmines sistemos charakteristikas. Aukštesnio abstrakcijos lygmens modeliavimo procesas vadinamas metamodeliavimu. Šio proceso metu sukuriami modeliai vadinami metamodeliais. *Metamodelis* – tai modelių modelis. Tai aukštesnio lygmens modelis, aprašantis sąryšius tarp žemesnio lygmens modelių, jų dalių, projektavimo metodų, abstrakcijų ir įrankių (Seidewitz, 2003). Metamodeliai naudojami ne tik modelių apibrėžimui ir analizei, bet ir pačiam modeliavimo procesui aprašyti. Visa informacija metamodeliuose fiksuojama kaip metaduomenys.

Metamodeliavimo sąvoką apibrėžė Štuikys ir Damaševičius (2008): „*Metamodelis yra srities sąvokų (srities artefaktų, kalbų, projektavimo procesų ir įrankių) modelis, kuris aprašo jų semantiką, funkcinius ir struktūrinius sąryšius, identifikuoja pastovius ir kintamus srities aspektus ir specifikuoja realizavimo taisykles*“.

Apibendrintas metamodeliavimo (Clark, 2008) uždavinys yra surasti ir aprašyti gerai išbandytus srities modelius ir šablonus bei pritaikyti juos sistemos kūrimo metu. Metamodeliavimas pagerina sistemų plėtros efektyvumą, teikia tęstinumą pereinant iš vieno projektavimo etapo į kitą, leidžia visuose cikluose į darbus įtraukti dalykinės srities ekspertus. Bendruoju atveju paradigma gali būti suprantama kaip nuoseklus procesas: metamodelis gaunamas iš jo meta-metamodelio, modelio elementas gaunamas iš jo metamodelio (Höfferer, 2007). Siekiant automatizuoti transformavimo procesus modeliai turi turėti apibrėžtą prasmę.

Metamodeliai unifikuoja naudojamas modeliavimo kalbas. Į metamodelį galima žvelgti kaip į kalbą, kurią sudaro abstrakti ir konkreti sintaksė bei semantika. Modeliavimo kalbos savyje turi elementus, leidžiančius generuoti modelius ir aprašyti jų sintaksę, semantiką ir žymėjimus. Modeliavimo procedūra apibrėžia, kaip modeliavimo kalba taikoma generuoti modelių egzempliorius. Tarp metamodelių ir modelių santykius galima pavaizduoti kaip metamodeliavimo hierarchiją (2.3 pav.) (Höfferer, 2007; Karagiannis, Kühn, 2002). Metamodeliai gali būti sukurti naudojant kitą (meta-) modeliavimo kalbą, kuri yra aprašyta pagal meta² modelį.



2.3 pav. Metamodeliavimo hierarchija (Höfferer, 2007)

Aukščiausiame lygmenyje sistemos atvaizduojamos konceptualiaisiais modeliais. Norint sistemos aprašus sukurti žemesniame ar aukštesniame lygmenyje, kad būtų galima juos interpretuoti ir įvykdyti, reikia atlikti transformacijas. Atliekant žeminančią transformaciją žeminamas sistemos abstrakcijos lygmuo: pereinant į žemesnį lygmenį modelis tikslinamas ir detalizuojamas. Vadinasi, transformacijos, pereinant į žemesnį lygmenį, darosi sudėtingesnės ir atliekamos automatiškai, tuo tarpu aukšto lygmens transformacijos gali būti atliekamos rankiniu būdu, pusiau automatiškai arba automatiškai.

Disertacijos kontekste metamodeliavimas suvokiamas siauriau. Pavyzdžiui, požymių diagramų aprašą (2.5 lent.) galima laikyti neformaliu metamodeliu. Savo ruožtu, požymių diagramos gali būti naudojamos metaprogramų modeliams aprašyti (Štuikys, Damaševičius, 2013a).

2.3. Transformacijos kuriant sistemas

Daugelio šiuolaikinių kompiuterijos krypčių kūrimas neįsivaizduojamas be analizės metodų ir transformavimo metodų plėtros. Transformacija apibrėžiama kaip formos, kokybės, pavidalo pakeitimas, pertvarkymas arba procesas, kurio metu tas pakeitimas įvyksta. Pagrindinis transformavimo klausimas – kas į ką yra

transformuojama. Šiame kontekste naudojama artefakto sąvoka. Programų inžinerijoje kaip artefaktai suvokiami reikalavimai, objektų savybės, modeliai, architektūros, algoritmai, programos ir pan.

Transformacija sistemose apibrėžiama kaip pirminių artefaktų (angl. *source artefacts*) atvaizdavimas, pritaikant transformavimo taisykles, į paskirties artefaktus (angl. *target artefacts*) (Štuikys, Damaševičius, 2008). Transformacijoms atlikti kuriamos ir naudojamos automatizuotos ar automatinės transformavimo sistemos ir įrankiai. Sukurti universalius ir efektyvius transformavimo įrankius sudėtinga, nes labai skiriasi kuriamos sistemos, transformavimo erdvės, būdai, abstrakcijos lygmenys ir pats sistemų kūrimas priklauso nuo daugelio veiksnių.

Galima išskirti du požiūrius į transformacijas: deklaratyvų ir operacinį (Mens, Van Gorp, 2006). Deklaratyvus požiūris akcentuoja rezultatą, o ne patį transformavimo algoritmą, sutelkiamas dėmesys į tai, kas turi būti transformuojama ir kas gauta transformacijos dėka. Operacinis požiūris sutelkia dėmesį į pačią transformaciją, kaip ji turi būti atlikta, nurodoma privaloma veiksmų seka, kuri turi būti atlikta siekiant gauti galutinį rezultatą iš pirminio artefakto.

Gali būti transformuojamas bet koks pirminis artefaktas, tačiau esminės transformacijos atliekamos su programomis ir modeliais. Modelių ir programų transformacijos atliekamos siekiant didesnio programų kūrimo, priežiūros ir tobulinimo efektyvumo (Visser, 2001).

2.3.1. Modeliai ir modelių transformacijos

Jau dešimtmetį stebimas akivaizdus perėjimas nuo manipuliacijų programos kodu prie programos modelių transformacijų (Moiz, Rizwanullah, 2012), skaičiavimų-valdymo modeliai (angl. *computational models*) laikomi aukščiausio lygio abstrakcija. To priežastis – nuolat augantys srities reikalavimai ir didėjantis programų ir sistemų sudėtingumas (Mens, 2012). Valdyti projektuojamos sistemos sudėtingumą ir apimtį padeda įvairių abstrakcijos lygmenų modeliai ir modeliavimo metodai. Abstraktūs modeliai suteikia galimybę atvaizduoti sistemą ar jos dalį išskiriant svarbiausias savybes ir ignoruoti informaciją, kuri neaktuali konkrečiame modeliavimo kontekste.

Modelių panaudojimo privalumai:

1. Pagerina programuotojų darbo efektyvumą.
2. Sumažinamas defektų skaičius galutiniame produkte.
3. Susisteminama informacija apie kuriamą sistemą.
4. Sistemos suskaidymo galimybė.
5. Palengvina sistemos palaikymą ir evoliucionavimą.
6. Galimybė iš naujo panaudoti sistemos atskiras dalis.
7. Alternatyvų galimybė.

Modelių kūrimas glaudžiai siejasi su taikomąja sritimi. Visi artefaktai, kurie transformuojami į modelius arba programas, yra tam tikros srities pirminės žinios. Todėl tik pažinus sritį (turint reikiamą informaciją) galima kurti sistemas ir taikymus: sudaryti srities modelius, juos transformuoti į tam tikras programas.

Modelis – tai abstraktus sistemos aprašymas, kuriame atspindėtos esminės sistemos charakteristikos, savybės. Modelyje pateikiama informacija apie sistemą

tam tikroje formoje, kuri suprantama tiek žmonėms, tiek technologinėms darbo priemonėms, naudojamoms įvairiuose sistemos gyvavimo ciklo etapuose.

Modeliai išreiškiami kalba, kuri egzistuoja tam tikrame abstrakcijos lygyje. Modeliai kuriami naudojant įvairias modeliavimo kalbas. Modeliavimo kalbos skirstomos į grafines (pvz., UML, EXPRESS, FAMILIAR), tekstines (pvz., XML, OCL), ir specializuotąsias (pvz., EebML, VRML). Modeliavimo kalbų yra labai daug, todėl modeliuojant sistemą reikia pasirinkti tinkamiausią. Šiuo metu dažniausiai naudojamos vizualios, grafinės modeliavimo priemonės, nes jos lengvai suprantamos vartotojo.

Van Amstelas, Van Den Brandas ir Nguyenas (2010) požiūriu, didžiausią įtaką modelių kokybei turi:

1. Naudojamos modeliavimo kalbos kokybė (jos tinkamumas sričiai ir sudėtingumas).
2. Modeliavimo ir transformavimo įrankių kokybė (suderinamumas su modeliavimo kalba, informacijos derinimas).
3. Projektuotojų žinios ir patirtis (problemos supratimas, modeliavimo kalbos žinios ir įrankių panaudojimas).
4. Naudojamų modeliavimo procesų kokybė.
5. Klaidų paieškos technikos.

Kokybės charakteristikų svarba priklauso ir nuo modeliavimo tikslų, pvz., jei reikia sukurti aukšto lygmens sistemos dokumentaciją, tai suprantamumas yra svarbiau už išsamumą.

Visos modelių transformacijos atliekamos vadovaujantis transformavimo taisyklėmis, kurios dar vadinamos transformacijų aprašu. Transformacijų apraše aprašoma, kaip pirminis modelis realizuotas tam tikra kalba yra transformuojamas į galutinio modelio realizaciją.

Czarneckis ir Helsenas (2003) klasifikavo modelių transformavimo metodus taip:

1. Tiesioginio manipuliavimo (angl. *direct manipulation*) metodas;
2. Struktūra paremtas (angl. *structure-driven*) metodas;
3. Grafais paremtas (angl. *graph-based*) metodas;
4. Sąryšių (angl. *relational*) metodas;
5. Hibridinis (angl. *hybrid*) metodas.

Biehlas (2010) klasifikavo galimas modelių transformacijas pagal transformavimo charakteristikas:

1. *Abstrakcijos keitimas*. Išskiriamos horizontali ir vertikali transformacijos. Horizontalioji transformacija – kai pirminis ir paskirties modeliai yra tame pačiame abstrakcijos lygmenyje. Vertikalioji transformacija – kai pirminis ir paskirties modeliai yra skirtinguose abstrakcijos lygmenyse.
2. *Metamodelių keitimas*. Modelių transformacijos, atsižvelgiant į kalbą, kuria išreikšti pirminis ir paskirties modeliai, skirstomos į egzogenines ir endogenines. Egzogeninės transformacijos atveju pirminis ir paskirties modeliai išreikšti skirtingomis kalbomis. Endogeninės transformacijos atveju pirminis ir paskirties modeliai išreikšti ta pačia kalba. Egzogeninė transformacija dar vadinama transliacija, o endogeninė – perfrazavimu.

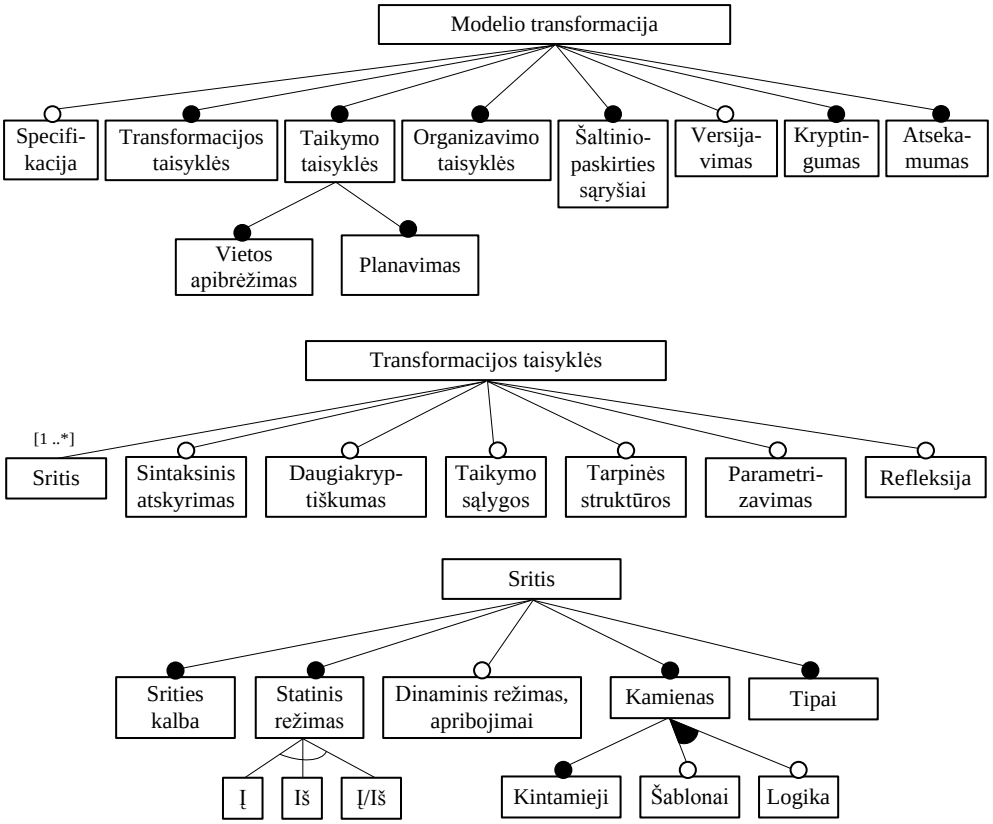
3. *Palaikančios technologijos*. Transformacijas apriboja galimos panaudoti modeliavimo technologijos.
4. *Palaikomas modelių skaičius*. Transformacija gali būti vykdoma su vienu, dviem ar daugiau modelių.
5. *Palaikomas paskirties tipas*. Transformacija gali būti iš modelio į modelį arba iš modelio į programos kodą.
6. *Išsaugojimas savybių*. Transformacijos gali išsaugoti (arba ne) semantiką, elgesį ar sintaksę.

Ši klasifikacija naudojama, kai reikia priimti sprendimą, kuri transformacijos kalba ar skaičiavimų-valdymo variklis tinka sprendžiant transformavimo uždavinį.

Mensas ir Van Gorpas (2006) apibrėžia svarbiausias modelių transformavimo charakteristikas:

1. Automatizavimo laipsnis (rankinis, pusiau automatizuotas, automatizuotas).
2. Sudėtingumas.
3. Pirminio modelio savybių išsaugojimas paskirties modelyje.

Czarneckis ir Helsenas (2006) modelių transformacijas aprašė požymių diagramomis (2.4 pav.), kurių rinkinys sudaro požymių modelį.



2.4 pav. Modelių transformacijos požymių modeliai (Czarnecki, Helsen, 2006)

Pastaruoju metu sparčiai vystosi modeliais grindžiamų sistemų kūrimo metodai. Čia modeliai yra naudojami visuose programinės sistemos kūrimo etapuose. Tačiau maksimalus lankstumas pasiekiamas, kai modeliai naudojami ne tik projektavimo, bet ir eksploatacijos metu: sistema veikia pagal sukurtus modelius, valdančius jos elgseną, keičiant modelius atliekamas sistemos konfigūravimas. Kuriant tokias sistemas srities ekspertai turi dalyvauti ne tik kuriant srities modelius analizės ir projektavimo metu, bet ir eksploatacijos bei adaptavimo metu. Tai įgyvendinama naudojant atitinkamas technologijas, kurios remia srities modeliavimą (angl. *Domain Specific Modeling*) ir specialias srities kalbas (angl. *Domain Specific Languages*) arba srities modeliavimo kalbas (angl. *Domain Specific Modeling Languages*), kurios leidžia projektuotojams dirbti su formaliais modeliais.

Modeliais grįsta inžinerija (angl. *Model-Driven Engineering*, MDE) apibrėžia modelių panaudojimą sričių specifikavimui ir modelių transformavimui (Schmidt, 2006). Ši programinės įrangos kūrimo metodologija orientuojasi į modelių ir abstrakcijų kūrimą. Geriausiai žinoma MDE iniciatyva yra „Object Management Group“ (OMG) konsorciumo pasiūlytas požiūris į programų sistemų kūrimą – modeliais grįsta architektūra (angl. *Model-driven architecture*, MDA). Ši programinės įrangos kūrimo paradigma remiasi modeliais ir jų automatizuotu transformavimu į programinį kodą (OMG-MDA, 2014; Sacevski, Veseli, 2007; Völter ir kt., 2013). Vadovaujantis šia paradigma, projektuojant sistemas kuriami skirtingų abstrakcijos lygių modeliai. Aukštesnės abstrakcijos modeliai iškeliami į aukštesnį abstrakcijos lygį, neprisirišama prie konkrečios platformos, o sukurti modeliai naudojami ne kartą pakartotinai. Tuo tarpu žemesnės abstrakcijos modeliai yra specifiški, pritaikyti konkrečiai platformai. Brown (2004) išskiria pagrindinius modeliu grįstos architektūros principus:

1. Modeliai išreiškiami gerai apibrėžta notacija, tai padeda suprasti sistemą.
2. Sistemų projektavimas gali būti organizuojamas naudojant modelių ir transformacijų tarp modelių rinkinius.
3. Formalus modelių aprašymas palengvina modelių integraciją, transformavimą ir sudaro automatizavimo panaudojant įrankius pagrindą.
4. Modeliais grįstas projektavimo metodas reikalauja standartizavimo.

MDA procesas apima nuo platformos nepriklausomų modelių (angl. *Platform Independent Models*, PIM) apibrėžimą, jų automatizuotą transformavimą į nuo platformos priklausomus modelius (angl. *Platform-Specific Models*, PSM) ir gautų modelių automatizuotą transformavimą į vykdomąją specifikaciją, t. y. veikiantį programos kodą. MDA procesas – abstrakcijos lygį žeminančių transformacijų seka.

Transformacijoms keliami reikalavimai:

1. *Formalumas ir išsamumas*. Transformavimo kalba turi būti aprašytos visos galimos transformacijos. Transformacijų formalus aprašas įgalina automatizuoti transformavimo procesus.
2. *Universalumas*. Transformavimo kalba turi būti pritaikoma daugelyje projektų, ji neturi būti sukurta tik vienam konkrečiam modeliui transformuoti.

3. *Modifikacijos vientisumas*. Po transformavimo pirminiai modeliai kaip ir sukurti modeliai gali būti pakeisti, todėl transformuojant turi būti sukuriami tarp šių modelių sąryšiai.
4. *Transformavimo taisyklių suprantamumas*. Transformavimo kalba turi būti lengvai skaitoma notacija, kuri suprantama ne tik kūrėjams, programuotojams, bet ir tiems, kurie ją naudoja.
5. *Susijusios kelių modelių transformacijos*. Transformacijos kalba turi apimti daugiau nei vieną pirminį ir kuriamus modelius.

MDE technologijos ne visada efektyvios – darbo sąnaudos, susijusios su jų panaudojimu, labai priklauso nuo programinės sistemos dydžio ir struktūrinio sudėtingumo (Kolovos, Paige ir Polack, 2009). Didelius sunkumus sukelia tarpusavyje susijusių modelių keitimai, ypač jei tie modeliai skirti skirtingoms programinės sistemos dalims ar užrašyti skirtingomis programavimo kalbomis (Jouault ir kt., 2010). Sunkiau derinti sugeneruotą nei rašytą ranka kodą.

2.3.2. Programų transformacijos

Programos kodo analizė ir manipuliacijos laikomos vienu iš svarbiausių kompiuterijos aspektu (Harman, 2010). Čia programos kodas suprantamas kaip bet koks programinės sistemos vykdomasis aprašas, kuris apima mašininį kodą, aukšto lygmens kalbas ir sistemos vykdomuosius grafinius aprašus. Programos kodo analizė yra automatinė ar pusiau automatinė procedūra, kuri leidžia suprasti programos kodo prasmę. Manipuliacija – tai automatinė ar pusiau automatinė procedūra, kuri paima ir grąžina pakeistą programos kodą.

Galima rasti įvairių programų transformavimo apibrėžimų. Visseris (2001) programos transformavimą apibrėžia kaip veiksmą keičiantį vieną programą į kitą. Vėliau Visseris (2005) programų transformavimą apibrėžia kaip mechaninę manipuliaciją programa, kuria siekiama ją gerinti, atsižvelgiant į tam tikras sąnaudų funkcijas ir iš esmės tai yra srities skaičiavimai, kuriuose programos yra kaip duomenys. Fioravanti ir kt. (2011) programų transformavimą apibrėžia kaip programos tobulinimo, pažangios kompiliacijos ir programų sintezės techniką. Winteris (2008) programų transformavimą sieja su programų manipuliacijomis. Transformuojama programa vadinama pirmine, o programa į kurią transformuojama – paskirties programa. Transformavimo sistemos pirmines programas naudoja kaip įvesties duomenis ir juos apdorojusios pateikia programas kaip išvesties duomenis.

Programų transformavimo terminas sutinkamas literatūroje įvairiais pavadinimais: metaprogramavimas (Cordy, Sarkar, 2004), programų generavimas, automatinis programavimas (angl. *generative programming*) (Batory, 2004; Czarnecki, Eisenecker, 2000), programų sintezė (Kitzelmann, 2010), programų restruktūrizavimas (angl. *refactoring*) (Apel, Kästner ir Batory, 2008; Kim, Zimmermann ir Nagappan, 2012; Du Bois ir kt., 2004), programų skaičiavimai (angl. *program calculation*) (Tesson ir kt., 2011), pjaustymo technologija (angl. *program slicing*) (Harman, Hierons, 2001). Visomis paminėtomis programų transformacijomis siekiama panašių tikslų – įvairiame kontekste automatizuoti programavimo uždavinius, siekiant padidinti programų kūrimo, priežiūros ir tobulinimo efektyvumą.

Winteris (2008) programų transformacijas skirsto į transformacijas, kurios nekeičia semantinio modelio, ir transformacijas, kurios semantinį modelį keičia. Winter apibrėžė programų transformavimo tikslus:

1. *Aiškumas* – funkcionalumo ir elgsenos sąvokų atskirtis.
2. *Efektyvumas* – vykdomosios programos išteklių naudojimas.
3. *Įvykdomumas* – iš nevykdomos programos gavimas vykdomos (pvz., kompiliatorius tiesiogiai kompiuterio nevykdomą programą transformuoja į vykdomą).
4. *Paprastumas* – pakeisti pirminę programą į tikslo, kur pirminės programos semantinis modelis yra tikslo programos semantinio modelio poaibis.
5. *Funkcionalumas* – keičiamas tikslo kalbos funkcionalumas išlaikant pirminės programos semantiką.
6. *Transliavimas* – keičiama pirminė programa į lygiavertę tikslo programą su skirtinga sintakse ir paprastai skirtingu, bet to paties abstraktumo lygmens semantiniu modeliu.
7. *Skaičiavimai* – atliekami skaičiavimai, gaunama dominanti programos ar išraiškos vertinimo forma.

Programų transformacijų taksonomijas klasifikavo Visseris (2001), jos pateiktos 2.6 lent. Transliavimo metu pirminė programa transformuojama į tikslo programą, abiejų programų kalbos yra skirtingos. Perfrazavimo metu pirminė programa transformuojama į ta pačia programavimo kalba parašytą tikslo programą. Šia transformacija siekiama pagerinti pirminę programą nekeičiant jos semantikos, keičiama tik sintaksė.

2.6 lentelė. Programų transformacijų taksonomijos (Visser, 2001)

Transliacija	**Perfrazavimas**
Migracija (angl. *migration*). Transformacija iš vienos kalbos į kitą išlaikant tą patį abstrakcijos lygmenį. **Sintezė** (angl. *synthesis*). Transformacija iš aukštesnio lygmens į žemesnį lygmenį. – detalizavimas (angl. *refinement*). – kompiliacija (angl. *compilation*). **Apgrąžos inžinerija** (angl. *reverse engineering*). Transformacija iš žemesnio lygmens į aukštesnį lygmenį. – Dekompiliavimas (angl. *decompilation*). – Architektūros gavyba (angl. *architecture extraction*). – Dokumentacijos generavimas (angl. *documentation generation*). – Programinės įrangos vizualizacija (angl. *Software visualization*). **Analizė** (angl. *analysis*). – Valdymo srautų analizė (angl. *control-flow analysis*). – Duomenų srautų analizė (angl. *data-flow analysis*).	**Normalizavimas** (angl. *normalization*). Sintaksinio sudėtingumo mažinimas. – Supaprastinimas (angl. *simplication*). – Konstrukcijų keitimas (angl. *desugaring*). – Apipynimas (angl. *weaving*). **Optimizavimas** (angl. *optimization*). Tam tikrų savybių pagerinimas išlaikant semantiką. – Specializavimas (angl. *specialization*). – Įterpimas (angl. *inlining*). – Suliejimas (angl. *fusion*). **Restruktūrizavimas** (angl. *refactoring*). Vidinės struktūros keitimas, kuris nepaliečia išorinės elgsenos. – Dizaino tobulinimas (angl. *design improvement*). – Supainiojimas (angl. *obfuscation*). **Atnaujinimas** (angl. *renovation*). Egzistuojančių komponentų statinis arba dinaminis pritaikymas vartotojo poreikiams.

Kompiliacija

Šiuo metu geriausiai išnagrinėta metaprogramavimo paradigma yra kompiliatoriai. Kompiliacija – tai transformacija, kurios metu programa iš aukštesnio lygmens kalbos verčiama į žemesnio lygmens kalbą (mašinos kodą) (Štuikys, Damaševičius, 2008). Kompiliatorius – tai programinė įranga, kuri tam tikra programavimo kalba parašytą programą transliuoja į vykdomąją programą. Pirmuoju kompiliatoriumi laikomas 1957 metais pristatytas FORTRAN kalbos kompiliatorius.

Kompiliacija yra sudėtingas procesas, todėl dažnai yra atliekamas keliomis fazėmis: leksikos analizė, sintaksės analizė, semantikos analizė, kodo generavimas ir optimizavimas (Aho ir kt., 2006). Kompiliatoriai ne tik analizuoja programinį kodą, bet ir diagnozuoja programavimo klaidas. Šiuolaikiniai kompiliatoriai komplektuojami su programų fragmentų bibliotekomis, kurių panaudojimas paspartina programos transformavimą.

Restruktūrizavimas

Programos kodo restruktūrizavimas išpopuliarėjo Martinui Fowleriui (1997) išleidus knygą „Refactoring“. Atsiradus šiai knygai, kodo struktūros pertvarkymo funkcionalumas buvo pradėtas diegti daugelyje integruotų programavimo aplinkų.

Restruktūrizavimas (angl. *refactoring*) – tai procesas, keičiantis programos kodo vidinę struktūrą nekeičiant jos funkcionalumo (Fowler ir kt., 1999). Restruktūrizavimas – tai procesas, kuris pradinę programos versiją keičia į patobulintą naują programos versiją (Thomas, 2005). Du Bois ir kt. (2004) restruktūrizavimą apibrėžė kaip programos kodo transformavimą iš vienos formos į kitą, išsaugant programos abstrakcijos lygį ir išorinį elgesį (funkcionalumą ir semantiką).

Liu, Batory ir Lengaueris (2006) pasiūlė požymiais grįstą programos restruktūrizavimą (angl. f*eature oriented refactoring*), dėl kurio programą galima pritaikyti skirtingoms realizacijoms programų šeimynose. Taikant šį metodą, programa sudalijama į požymius, kur kiekvienas požymis atitinką tam tikrą programos funkcionalumą. Šalinant neprivalomus požymius, automatiškai sintetinamas reikiamas programos variantas.

Dažniausiai kodas pertvarkomas norint pagerinti struktūrą, skaitomumą, palaikymą, pašalinti perteklinį, dubliuotą kodą (Mens, Tourw 2004; Thomas, 2005). Restruktūrizavimas naudojamas pertvarkant kintamuosius, išskiriant ar suliejant metodus, klases. Restruktūrizavimo procesas neskirtas programos kodo klaidoms taisyti ar papildomam funkcionalumui įterpti.

Mens ir Tourwé (2014) restruktūrizavimo procesą suskaidė į keletą skirtingų veiklos rūšių:

1. Nustatyti, kuri programinė įranga turi būti restruktūrizuota.
2. Nustatyti, kokie restruktūrizavimo metodai taikomi konkrečiose vietose.
3. Užtikrinti, kad bus išsaugotas programos funkcionalumas.
4. Restruktūrizavimo taikymas.

5. Atlikto restruktūrizavimo įvertinimas. Vertinamos programos charakteristikos: sudėtingumas, suprantamumas, palaikomumas, proceso našumas, sąnaudos ir pastangos.
6. Išlaikyti suderinamumą tarp pertvarkyto programos kodo ir kitų programinės įrangos dalių (pvz., su reikalavimų specifikacija, testais ir kt.).

Restruktūrizavimo procesas daugumoje programavimo aplinkų yra iš dalies automatizuotas. Yra sukurta nemažai restruktūrizavimo procesą palaikančių įrankių: ReSharper (2015), Netbeans (2015), Eclipse (Xing, Stroulia, 2006) ir kt.

Restruktūrizavimas yra neatsiejama ekstremalaus programavimo dalis (Qureshi, Ikram, 2015). Programuotojai kuria naujus funkcionalumus ir, siekdami pagerinti jų logiką ir skaidrumą, juos restruktūrizuoja. Visi pertvarkyti programos elementai testuojami, taip užtikrinamas programos funkcionalumo išsaugojimas. Kita praktikoje naudojama technologija, kuri taip pat transformuoja programos kodą išsaugodama ankstesnį programos funkcionalumą, yra programos specializavimas.

Specializavimas

Sąvokų „programų specializavimas“ (angl. *specialization*) ir „dalinis įvertinimas“ (angl. *partial evaluation*) apibrėžimas yra labai panašus. Jos naudojamos nusakyti programos ar modelio pertvarkas.

Programos specializavimas (dalinis įvertinimas) yra technologija, leidžianti automatiškai pakeisti programą į jos specializuotą versiją atsižvelgiant į naudojamą kontekstą (Le Meur, Lawall ir Consel, 2002). Tai programų transformavimo metodas, kurio metu dalis programos kintamųjų paverčiami konstantomis, taip sumažinama programos argumentų aibė (Futamura, 1999; Jones, Gomard ir Sestoft, 1993). Iš pradžių dalinis įvertinimas buvo naudojamas kompiliatoriuose (Jones, Sestoft ir Søndergaard, 1985). Vėliau imta naudoti kaip kodo-į-kodą transformavimo technika, kurios tikslas pagerinti programos efektyvumą (Jones, 1996). Šiuo metu yra daugybė programų, kurios užsiima programų supainiojimu (Giacobazzi, Jones ir Mastroeni, 2012), modelių transformavimu (Acher ir kt., 2013; Gheyi, Massoni ir Borba, 2011; Hartmann, Trew, 2008; Thum ir kt., 2009), saugumo gerinimu (Murakami, 2012) ir daug daugiau (ACM SIGPLAN 2014; Le Meur ir kt., 2002).

Be programos dalinio įvertinimo dar yra žinomas ir duomenų dalinis įvertinimas. Šio proceso metu atliekami iš anksto žinomi skaičiavimai ir jų rezultatai išsaugomi duomenų struktūrose (pvz., masyvuose). Vėliau, vykdant programą, skaičiavimai neatliekami, o kreipiamasi tiesiogiai į duomenų struktūras ir surandama reikiama reikšmė. Šis metodas efektyvus tik tiems skaičiavimams, kurie reikalauja ilgo vykdymo laiko.

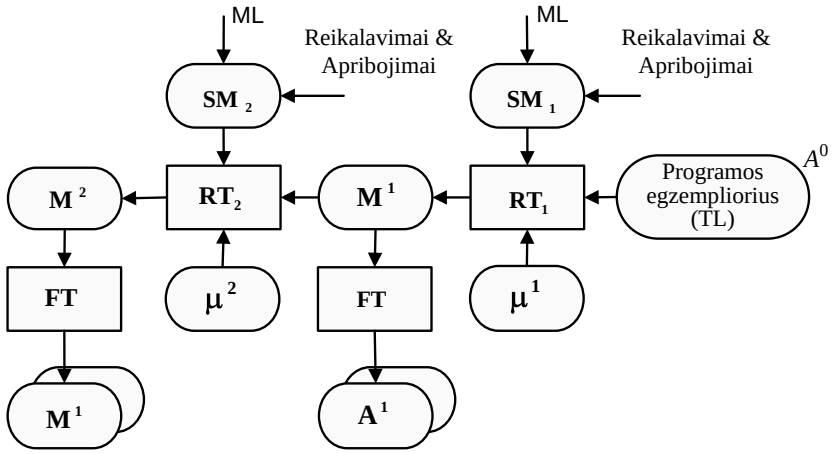
Marletas (2013) specializuojant programas išskiria du atvejus: specializavimas programos kompiliavimo metu ir specializavimas programos vykdymo metu. Vykdymo metu atliekamo specializavimo privalumas yra tas, kad programa modifikuojama atsižvelgiant į informaciją, kuri yra nežinoma (pvz., vartotojo įvesti ar vykdymo metu tinklu gaunami duomenys) tol, kol programa nepaleidžiama.

Programos specializavimas taip pat susijęs su pakopiniu programavimu, metaprogramavimu ir loginiu programavimu (Bachelet, Mahul ir Yon, 2013;

Leuschel, Vidal, 2014; Sheard, 2001; Štuikys, Damaševičius, 2013a; Tourwé, Mens, 2003). Programų skaidymas į pakopas padalija programos vykdymo metu atliekamus skaičiavimus į kelias dalis, kiekvienos pakopos skaičiavimai atliekami skirtingu laiku.

Daugiapakopis programavimas (angl. *multi-stage programming*) – tai programų kūrimo būdas, kai tam tikra programinės įrangos dalis kuria programinį kodą, kuris yra vykdomas vėliau (Inoue, Taha, 2012). Taha pateikė daugiapakopių programų formalų aprašą (Inoue, Taha, 2012; Taha, 2004; Taha, 1999). Tai programos transformacija reorganizuojanti programos vykdymą į etapus. Taha siūlo metaprogramas kurti formalia kalba MetaML, kaip kelių pakopų programas. Ši koncepcija yra susijusi su informacijos slėpimu abstrakcijos lygmenyse (pakopose), o tai leidžia daug lanksčiau projektuoti programas.

Metaprogramos specializavimo uždavinį nagrinėjo Štuikys ir Damaševičius (2013a). Jie pristatė dviejų pakopų metaprogramos kūrimo karkasą (2.5 pav), kuris sudarytas iš pradinių duomenų, transformavimo procesų, procesus remiančių modelių ir transformacijų rezultatų – išeities duomenų.



Legenda: ▭ - procesas; ⬭ - duomenys; ⬭ - duomenų rinkinys;
ML, TL – meta ir tikslo kalba; RT, FT - atvirkštinė ir tiesioginė transformacijos;
M^1, M^2 – metaprograma ir meta-metaprograma; μ^1, μ^2 - M^1 ir M^2 modeliai;
SM – semantinis modelis; A^1 – rinkinys TL programos egzempliorių.

2.5 pav. Dviejų pakopų metaprogramos kūrimo karkasas (Štuikys, Damaševičius, 2013a)

Metaprogramavimas ne tik įgalina kurti programų generatorius, bet ir leidžia apibendrinti metaprogramas į aukštesnio lygmens metaprogramas. Dviejų pakopų metaprograma yra metaprogramų generatorius. Jo dėka galima generuoti specializuotas metaprogramas, o vėliau iš jų, pagal poreikį, automatiškai generuoti tikslo kalbos programų egzempliorius. Metaprogramavimo dėka galima pasiekti didesnį programinės įrangos kūrimo pakartotinį naudojimą ir automatizavimą.

2.4. Programų kūrimo procesų automatizavimas

Automatizuoti programinės įrangos kūrimo procesus siekiama jau seniai. Tai sudėtingas uždavinys, kuriuo siekiama pagerinti kuriamo produkto kokybę ir inžinierių darbo efektyvumą. Šiuolaikinės priemonės padeda sumažinti inžinieriaus veiklas kuriant, analizuojant ar pertvarkant kompiuterinius modelius, atspindinčius kuriamo produkto struktūrą ir pagrindines savybes.

Jau ne vieną dešimtmetį aktyviai vystomi programų modeliavimo, modelių analizės, programos kodo generavimo įrankiai. Tačiau efektas pasirodė ne toks didelis, kaip buvo tikėtasi iš pradžių. Tai susiję su tuo, kad programinės sistemos neturi nei geometrinių, nei fizinių charakteristikų, jos charakterizuojamos dideliu rinkiniu specifinių rodiklių, kurie dažnai yra sunkiai formalizuojami, atsižvelgiant į srities uždavinį, turi skirtingas prasmes. Rodiklių įvairovė skatina įvairių modeliavimo metodų (euristinių, formalių, prototipų) ir programavimo kalbų atsiradimą. Projektuojant realias programines sistemas dažnai modeliavimo metodai susipina ir turi visoms grupėms būdingų bruožų.

Įvairiems projektavimo metodams palaikyti buvo sukurtos formalizuotos, srities kalbos (angl. *domain specific languages*, DSL). DSL apibrėžiama kaip specifikavimo kalba, sukurta tam tikros rūšies problemoms aprašyti analizuojamoje dalykinėje srityje. DSL yra iš prigimties deklaratyvi kalba, t. y. teiginių rinkinys, kuris remiasi matematine logika (Deursen, Klint ir Visser, 2000). DSL yra apibrėžiamos metamodeliais, kurie specifikuoja semantiką ir apribojimus, susijusius su dalykinės srities koncepcijomis.

Automatizuojant programinės įrangos kūrimo procesus iškyla daug sunkumų sprendžiant atsekamumo uždavinius. Atsekamumas (angl. *traceability*) – galimybė atsekti ryšius tarp elementų viename ir skirtinguose abstrakcijos lygiuose. Kuo detaliau atliekamas atsekamumas, tuo lengviau atlikti keitimus. Atsekamumo uždaviniai yra imlūs darbo laikui (Aizenbud-Reshef ir kt. 2006). Šiuo metu jau yra programavimo kalbų ir technologijų, kurios panaikina būtinybę patiems kūrėjams sekti pakeitimus. Aspektinis programavimas (angl. *Aspect-Oriented Programming*) leidžia automatiškai sekti atliekamus keitimus. Aspektinis programavimas sprendžia leidžiantį panaikinti ar bent sumažinti dalykinių turinių susipynimą sistemoje. Tačiau šiai dienai aspektinis kodo generavimas (angl. *aspect-oriented code generation*) dar turi nemažai trūkumų, pvz., nenustatytas atliekamų transformacijų teisingumas, nėra pakankamai palaikančių įrankių (Mehmood, Jawawi, 2013).

2.4.1. Programinio kodo generavimas

Programų generatoriai – tai specialios priemonės, skirtos automatizuotam programų kūrimui. Generatoriai leidžia žymiai sumažinti darbo sąnaudas, kai reikia programuoti dažnai pasikartojančias operacijas, pvz., kūrimas meniu juostų, ataskaitų ir kt.

Pastaruoju metu išplito vizualusis programinių sistemų projektavimas. Todėl tyrimai programinio teksto generavimo iš grafinių modelių yra labai platūs. Grafinius modelius galima padalyti į dvi grupes: statinius ir dinaminius. Objektiniame programavime iš klasių diagramų (statinis modelis) kuriamas programinio kodo karkasas (Usman, Nadeem, 2009), iš būsenų diagramų (dinaminis

modelis) – programos kodas (Park, Youn ir Lee, 2009). Kiekviena programa turi tam tikras būsenas, kurias galima aprašyti būsenų diagrama. Ši diagrama nusako objektų būsenas ir jų pasikeitimus laike. Pačios programos gali būti su aiškiai išreikštomis būsenomis ir be jų. Pirmuoju atveju aprašyti programos elgesį galima pasinaudojant automatų teorija ir būsenų diagramomis (pvz., UML). Antruoju atveju programos turi daug sąryšių ir pakeitimai vienose programos dalyse daro įtaką kitų dalių pakeitimams.

Yra žinoma nemažai įrankių, skirtų programos kodui automatiškai generuoti. Praktiškai naudojami įvairūs sprendimai, kuriuose naudojamos skirtingos programavimo kalbos. Kai kuriose sistemose naudojamos tokios procedūrinės kalbos kaip Asembleris ir C, kitose – objektinės kalbos C++, Java, C# ir kt.

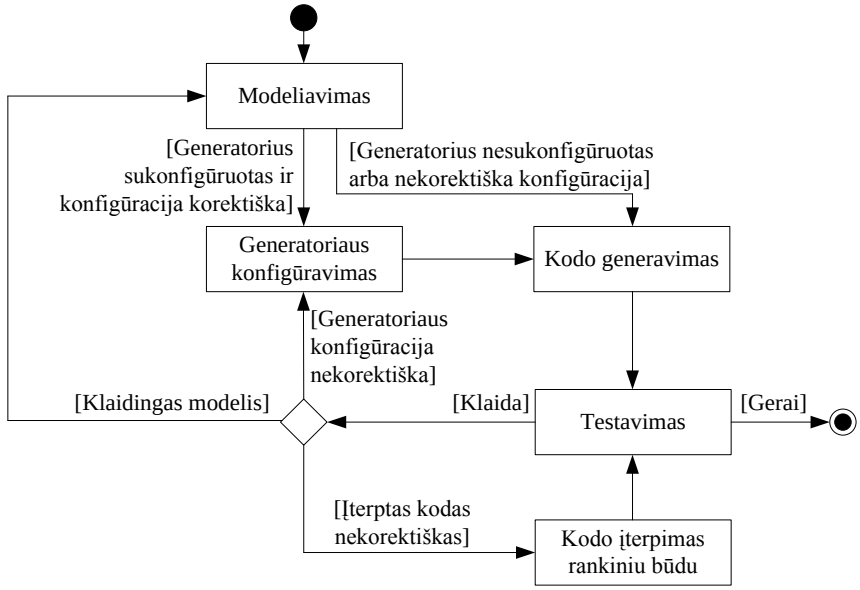
SWITCH technologiją (programinės sistemos projektavimas aiškiai išskiriant būsenas arba automatinį programavimą, objektinio programavimo technologija) palaikantys įrankiai:

1. *Visio2Switch* (Goloveshin, 2002) įrankis, skirtas kodui automatiškai generuoti C programavimo kalba iš būsenų grafų, kurie realizuoti MS Visio redaktoriumi, panaudojant šabloną SWITCH.vss.
2. *MetaAuto* (Kanzhelev, Shalyto, 2005) įrankis, skirtas kodui automatiškai generuoti bet kuria programavimo kalba iš būsenų grafų, kurie realizuoti MS Visio redaktoriumi, panaudojant kiekvienai kalbai skirtą šabloną.
3. *UniMod* (Gurov ir kt., 2005) Eclipse įskiepis, skirtas kodui automatiškai generuoti Java programavimo kalba iš sukurtų UML klasių ir būsenų diagramų.
4. *FSME* (Darovsky, 2003) įrankis skirtas būsenų grafams kurti ir redaguoti bei kodui automatiškai generuoti C++ ir Phython programavimo kalbomis.
5. *Acceleo* (Musset ir kt., 2006) atvirojo kodo įrankis skirtas kodui generuoti Eclipse aplinkoje. Generuojamas kodas bet kuria kalba (Java, PHP, Python ir kt.) iš EMF (angl. *Eclipse Modeling Framework*) modelių, apibrėžtų bet kuriuo metamodeliu (UML, SysML ir kt.).
6. *Altova MapForce* (Altova, 2015) yra grafinis duomenų fiksavimo, konvertavimo ir integracijos įrankis. Kodas generuojamas Java, C# ir C++. Įrankis gali vykdyti pasikartojančias transformacijas.
7. *CodeFluent Entities* (SoftFluent, 2014) grafinis įrankis integruotas į Microsoft Visual Studio, generuojantis .NET kodą C# arba Visual Basic kalba.
8. *RISE* (2015) yra nemokamas modeliavimo paketas, naudojantis ERD (angl. *Entity–relationship diagram*) arba UML. Generuojamas duomenų bazių MySQL, PostgreSQL ir Microsoft SQL Server kodas C# ir PHP kalbomis.
9. *MagicDraw* (2015) – UML modeliavimo įrankis, turintis diagramų semantinio teisingumo tikrinimo mechanizmą. Skirtas Java, C#, C++ ir CORBA IDL programavimo kalboms bei gali vykdyti šių kalbų kodo atvirkštinę inžineriją, duomenų bazių schemų atvirkštinę inžineriją, kodo bei duomenų bazių schemų generavimą. Integruotas su „Borland Jbuilder“, „IBM Eclipse“, „Sun ONE Studio 4“, „NetBeans“ ir „IntelliJ IDEA 4.0“.

Norėdami generuoti programos kodą, turime disponuoti tam tikra informacija (Glass, 1996):

1. Programinės sistemos modeliu.
2. Srities modeliu.
3. Tikslo platformos modeliu (angl. *Target Platform Model*).
4. Transformacijos modeliu.
5. Tikslo platformos modelio esybių (konkrečia programavimo kalba) ir jų jungimo taisyklių rinkiniu.

Kai modeliais grindžiamų sistemų kūrimo procese naudojamas kodo generatorius, visos klaidos sugeneruojamos kode taisomos ne pačiame programos kode, o programos modelio lygyje. Programos kūrimo procesas susideda iš daugelio pakartotinių generavimų ir modelio koregavimų. Programinės įrangos kūrimo procesas, kai naudojamas programinio kodo generatorius, pavaizduotas 2.6 pav.



2.6 pav. Programinės įrangos kūrimo procesas, kai programa konfigūruojama naudojant programos kodo generatorių (Ablonskis, 2007)

Realiai išbaigtą programinį kodą kuriantys generatoriai realizuojami retai ir tik siaurose taikymo srityse. Labai dažnai programos kodą greičiau ir lengviau parašyti rankiniu būdu, nei šiam darbui sukonfigūruoti programos kodo generatorių.

Dalinio programos kodo generavimo atveju, dalį programos kodo tenka rašyti rankiniu būdu, atsiranda būtinybė po kiekvieno programos kodo generavimo proceso atlikti rankines korekcijas. Tokie taisymai yra sudėtingi, nes reikia gerai suprasti, kas keičiasi programiniame kode kiekvienos generacijos metu. Todėl stengiamasi sukurti generatorius, kurių generuojamas ir rankomis rašomas programos kodas kuo labiau atskiriami (Herrington, 2003).

Radosevicas, Orehovackis ir Magdalenicas (2012) pasiūlė idėją programos kodą generuoti ir vykdyti tik pareikalavus (pvz., žiniatinklio programa). Čia

programos kodas generuojamas scenarijų kalbomis (angl. *scripting languages*) ir valdomas kintamaisiais.

Programos kodo generavimas programinės sistemos kūrimo procese yra sudėtingas uždavinys. Realizuojant šį uždavinį susiduriama su eile sunkumų (Herrington, 2003):

1. Programos kodo generatorių sukonfigūravimas reikalauja daug pastangų ir laiko. Be to ne visais atvejais tai įmanoma įgyvendinti.
2. Reikia gerai išmanyti kaip generuojamą kodą parašyti rankiniu būdu, tik tada įmanoma tinkamai sukonfigūruoti generatorių.
3. Sunku atsekti ryšius tarp modelio ir sugeneruoto kodo, nes sugeneruotas kodas sunkiau skaitomas nei parašytas rankiniu būdu.
4. Suradus klaidą sugeneruotame programos tekste reikia surasti už šią klaidą atsakingą modelį ir jį pakeisti, kad pakartotinai generuojant programos kodą klaida būtų ištaisyta.
5. Dažniausiai realizuojamas dalinis programos kodo generavimas, tokiu atveju reikia surasti sprendimus, kaip atskirti sugeneruotą ir ranka parašytą kodą, kad pakeitus modelį ar perkonfigūravus kodo generatorių nekiltų papildomų rūpesčių perkeliant ranka rašytą kodą.

Programų generatoriai realizuoja generatyvinį pakartotinį panaudojimą. Jie pakartotinai naudoja standartinius šablonus ir algoritmus. Išanalizuoti įrankiai paprastai remiasi šablonine technologija, kuri yra paprasta, bet nėra visiškai automatizuota (sukurtas kodas nėra vykdomasis kodas). Todėl darbe buvo pasirinkta metaprogramavimo technologija.

2.5. Metaprogramavimas ir programų generatorių kūrimas

Metaprogramavimas – tai aukšto lygmens programavimo paradigma, kuri nagrinėja, kaip rašyti ar manipuliuoti kitomis programomis kaip duomenimis. Šios manipuliacijos rezultatas yra žemesnio lygio programa. Metaprogramavimas gali būti vertinamas kaip technologija, leidžianti aprašyti srities variantiškumą atkartojimo technologijos kontekste (Štuikys, Damaševičius, 2013a). Tai svarbi programų inžinerijoje technika, kuri yra plačiai naudojama praktikoje.

Metaprogramavimo metodologija nėra nauja. Paprasčiausios formos metaprogramos (makrokomandos (Greenwald, Kane, 1959), kadrų kalba (angl. *frame language*) (Minsky, 1974) skirta žinioms atvaizduoti) jau buvo naudojamos XX a. antroje pusėje. Dabar metaprogramavimas taikomas įvairiose srityse. Formalūs ir pusiau formalūs aprašai – metaprogramavimo, aukštesnio lygio programavimo metodologijų (generatyvinis, aspektinis ir kt.) ir transformacijos, įgyvendinančios aukštesnio lygio programas, intensyviai tyrinėjamos daugelio mokslininkų.

Dėl taikymų gausos yra daug skirtingų požiūrių į metaprogramavimą. Nanevskis (2002) metaprogramavimą apibrėžia kaip programavimo paradigmą, įgalinančią algoritmiškai kurti programas tam tikra (srities) kalba, panaudojant metakalba parašytą programą. Veldhuizenas (2006) metaprogramavimą supranta kaip programos apibendrinimą ir generavimo technologiją. Štuikys ir Damaševičius (2013a) metaprogramavimą apibrėžia kaip programavimo metodologiją, kuri

nagrinėja metodus ir procesus, aprašančius įvairias manipuliacijas su programomis kaip duomenimis.

Metaprogramavimas apima įvairius metodus, programas, įskaitant sintaksės analizatorius, refleksiją (programos savybė stebėti ir modifikuoti savo struktūrą ir elgseną), specialios paskirties metaprogramas ar bendrąsias programas. Metaprogramavimas gali būti atliekamas kompiliuojant programą (Gazagnaire, Madhavapeddy, 2011; Miao, Siek, 2014), generuojant kodą (Trujillo, Azanza ir Diaz, 2007) arba programos vykdymo metu keičiant programos kodą (Porkoláb ir kt., 2009). Taip pat kuriant bendruosius komponentus (Attardi, Cisternino, 2001), programas iš dalies įvertinant (angl. *partial evaluation*) arba specializuojant (angl. s*pecialization*) (Jones, 1996), programas transformuojant (Ludwig, Heuzeroth, 2001), perkuriant (angl. *reengineering*) (Papotti, do Prado ir de Souza, 2012) ir kt.

Löwe ir Noga (2002) metaprogramavimą panaudoja interneto komponentams apjungti į vieną sistemą. Metaprogramavimas gali būti naudojamas įgyvendinant funkcijų bibliotekas. Tokios metaprogramos generuoja nedidelius standartinio kodo kiekius, tai suteikia bibliotekoms daugiau lankstumo (Tobin-Hochstadt ir kt., 2011). Metaprogramavimas naudojamas ir loginiame programavime, kur naudojamos aukštesnės eilės funkcijos (Visser, 2002), šablonų programavime (Abrahams, Gurtovoy, 2004), komponentų generavime (Štuikys, Montvilas ir Damaševičius, 2009).

Metaprogramavimo metodologiją remia nemažai sukurtų ir naudojamų programavimo kalbų: PROMOL (Štuikys, Damaševicius, 2000), MetaML (Taha, Sheard, 2000), Haskell (Sheard, Jones, 2002), MetaD (Pasalic, 2004), Ωmega (Sheard, Pasalic, 2004), Rascal (Bos ir kt. 2011; Storm, 2011), MetaModelica (Fritzson, Pop ir Sjölund, 2011), MetaHaskell (Mainland, 2012) ir kt. Tačiau metaprogramavimas neapsiriboja vien funkcinėmis kalbomis, ir kitos programavimo kalbos gali būti naudojamos kaip metakalbos. Pavyzdžiui, šabloninis metaprogramavimas (Abrahams, Gurtovoy, 2004; Pintoir kt., 2013) C++ naudojamas atlikti programos generavimą kompiliavimo metu. Šis mechanizmas buvo sėkmingai įgyvendintas C++ bibliotekose (Czarnecki ir kt., 2000). MetaAspectJ (Huang, Zook, Smaragdakis, 2008) metaprogramavimo kalba, Java plėtinys, palaikantis AspectJ programų generavimą, sujungia aspektinį ir generatyvinį programavimą. Metaprogramavime naudojamos ir kitos programavimo kalbos: PHP (Gabrysiak, Marr ir Menge, 2005), .NET (Hazzard, Bock, 2013), Java (Miao, Siek, 2014) ir kt. Metaprogramos analizuoja, transformuoja ir generuoja programas, naudodamos jas kaip struktūrizuotus duomenis.

2.5.1. Metaprogramavimo koncepcijų taksonomijos

Metaprogramavimo taksonomijos akcentuoja metaprogramavimo sistemas, įrankius ir kalbas. Metaprogramavimo taksonomijas apibendrino ir pristatė savo darbe Sheardas (2001) ir Pasalicas (2004). Abu autoriai išskyrė dvi metaprogramų rūšis, t. y. programų generatorius ir programų analizatorius. Programų generatoriai generuoja kitas programas pagal tam tikrus įėjimus, o programų analizatoriai analizuoja programą ir atlikę analizę pateikia tam tikrus rezultatus. Kai kurios metaprogramavimo kalbos turi abi metaprogramavimo abstrakcijas, jos yra ir

generatoriai ir analizatoriai (Nanevski, 2002). Apibendrinta metaprogramavimo sąvokų taksonomija pateikta 2.7 lent.

2.7 lentelė. Metaprogramavimo sąvokų taksonomija (Damaševičius, Štuikys, 2008)

Koncepcijos klasė	Koncepcija	Lygiaverčiai terminai naudojami literatūroje	Apibrėžimas
Struktūra	Metaprograma	Metakomponentas; metaspecifikacija; šablonas; generuojamasis, bendrinis, parametrizuotas komponentas	Bendrasis komponentas realizuotas metakalba, kuri aprašo gimininingų komponentų egzempliorių, besiskiriančių tam tikrais funkcionalumo aspektais, šeimyną
	Abstrakcijų lygmenys	Abstrakcijų sluoksniai	Semantinės sistemos, kurios yra sugrupuotos atvaizduojant skirtingus metaprogramos kūrimo aspektus
	Koncepcijų atskirtis	Aspektų atskirtis	Projektavimo uždavinio sudalijimas į atskirus dalinius uždavinius, kurie yra ortogonalūs ir realizuojami skyrium
	Metaduomenys	Anotacijos	Tam tikro metaprogramavimo sistemos sluoksnio aprašas, pateikiantis jo savybes ir koncepcijas
Procesas	Transformavimas	Manipuliavimas, integravimas, kompozicija, modifikavimas, adaptavimas	Procesas keičiantis vieną programos formą į kitą dažniausiai išsaugant programos semantiką
	Generavimas	Programos generavimas, kodo generavimas	Procesas kuriantis išvesties / paskirties programą (sistemą) iš aukštesnio lygio specifikacijos
	Refleksija	Savistaba (angl. *introspection*), Tarpininkavimas (angl. *intercession*)	Programos savybė stebėti ir galimai modifikuoti savo pačios struktūrą ir elgseną
	Bendrinimas	Parametrizavimas	Komponento savybių (aspektų) išplėtimas suteikiant bendrinį aprašą (dažniausiai per parametrus), kuris turi platesnį pritaikomumą.

Metaprogramavimas yra daugiakalbės programavimo paradigmos atvejis. Metaprogramavimas apima dviejų tipų programavimo kalbas: metakalbą (angl. *meta-language*) ir tikslo (srities) kalbą (angl. *target language*). Metakalba – tai aukštesnio lygio programavimo kalba, kuria kuriama pati metaprograma, ji padeda manipuliuoti kitomis programomis. Tikslo kalba – tai žemesnio lygio programavimo kalba, kuria rašomos programos, kuriomis manipuliuoja metaprograma.

Metaprogramavimo sistemas galima suskirstyti į dvi grupes: homogeninio metaprogramavimo ir heterogeninio metaprogramavimo. Homogeninėse sistemose tikslo kalba ir metakalba yra ta pati. Heterogeninėse sistemose yra naudojamos bent dvi nepriklausomos kalbos. Homogeninis ir heterogeninis metaprogramavimas skiriasi tarpusavyje pagal koncepcijų atskirtį aukštesniame ir žemesniame

programavimo lygmenyje, t. y., kokia programavimo aplinka panaudota kuriant metaprogramą – vienalytė ar ne.

Tyrinėti homogeninį ir heterogeninį metaprogramavimą buvo skirta nemažai pastangų (Berger, Tratt, 2010; Pasalic, 2004; Sheard, 2001; Štuikys, Damaševičius, 2013a). Homogeninio metaprogramavimo sąvoką pirmasis įvedė Sheardas (2001). Homogeninis metaprogramavimas kilęs iš formalių metaprogramavimo sistemų ir funkcinių programavimo kalbų (MetaML, Haskell). Šis metaprogramavimo būdas remiasi netiesiogine koncepcijų atskirtimi, kadangi vienalytė programavimo aplinka netiesiogiai sudalijama į du poaibius: aukštesnį ir žemesnį.

Daugelyje taikymų homogeninės metaprogramos analizuoja sintaksinę programos struktūrą, patikrina duomenų tipų teisingumą (Chlipala, 2010; Chen, Xi, 2005; Magalhães ir kt., 2010) ar duomenų tipus ir kontekstą (Chapman, 2009; Devriese ir Piessens, 2013). Refleksija – tai programavimo kalbos savybė, kai programavimo kalba gali būti sau metakalba, tada ji vadinama atspindžiu (Kollár, Forgáč ir Porubän, 2007). Tai programos manipuliavimas savimi kaip duomenimis vykdymo metu. Išskiriami du tokios manipuliacijos aspektai: savistaba (angl. *introspection*) ir interaktyvumas (angl. *intercession*). Savistaba – tai programos savybė stebėti savo būseną, o interaktyvumas – programos savybė keisti savo vykdymo būseną ar pakeisti savo interpretaciją.

Heterogeninis metaprogramavimas remiasi dviejų skirtingų kalbų – srities (tikslo) ir metakalbos – naudojimu toje pačioje specifikacijoje. Šiai koncepcijai realizuoti reikia atskiros programavimo kalbos – metakalbos. Kaip metakalba gali būti panaudota specializuota (pvz., MetaD, Recall ir kt.) arba bet kuri bendrosios paskirties programavimo kalba (pvz., Java, C, PHP ir kt.).

Heterogeninėje metaprogramavimo aplinkoje programų egzemplioriai (angl. *instances*) yra generuojami, vartotojas jais manipuliuoja tiesiogiai, o homogeninėje aplinkoje – programų egzemplioriai konkretinami vykdant metaprogramą ir vartotojui tiesiogiai yra neprieinami (Štuikys, Damaševičius, 2013a).

Metaprogramavimo raida glaudžiai susijusi su programavimo technologijų raida ir pačiomis programavimo kalbomis. Juk metaprograma taip pat yra programa, kuri yra parašyta viena ar keliomis programavimo kalbomis.

2.5.2. Programų ir metaprogramų analogija

Programavimo kalba yra dirbtinė kalba, skirta programoms užrašyti. Pats programavimas gali būti apibrėžiamas kaip veiksmų, kuriuos norima atlikti, apibrėžimas. Programuojant siekiama sukurti programą, kuri spręstų tam tikrą uždavinį ir leistų pasiekti tikslą. Pirmoji aukšto lygio programavimo kalba „Fortran“ buvo sukurta Johno Backuso vadovaujamos grupės 1954 m., šiuo metu pasaulyje yra sukurta tūkstančiai skirtingų programavimo kalbų.

Programavimo kalbos skirstomos pagal jų vykdymo tipą, taikymo sritis, naudojamus metodus ir kt. Kiekvienu klasifikavimo būdu išryškinamos tam tikros kalbos savybės. Galima išskirti dvi dideles programavimo kalbų grupes – bendros paskirties ir specializuotas.

Bendrosios paskirties programavimo kalbos (pvz., C, C++, Java ir kt.) yra skirtos bendriesiems skaičiavimo ir valdymo uždaviniams specifikuoti. Srities

kalbos pritaikytos tam tikros taikomosios srities uždaviniams aprašyti. Srities kalbos paprastai naudojamos tuomet, kai srityje yra daug specifinių abstrakcijų, kurias sudėtinga aprašyti bendrosios paskirties programavimo kalbomis, kai reikia naudoti kelias programavimo paradigmas arba srityje yra daug besikartojančių operacijų ir duomenų sekų. Paprastai srities kalbos yra paprastesnės, jos turi mažiau išraiškos priemonių negu programavimo kalbos, tačiau yra ir išimčių, pvz., VHDL, System C.

Programos aprašo manipuliacijas su duomenimis. Programa duomenų struktūras integruoja pagal pasirinktą algoritmą, taip išreiškiamas reikiamas funkcionalumas (Wirth, 1971). Programa užrašoma (2.1) išraiška:

$$„programa = duomenų\ struktūros + algoritmas“, \qquad (2.1)$$

čia „+“ vaizduoja integravimą.

Metaprogramavimas – tai aukštesnio lygio programavimo paradigma, kai manipuliuojama programomis kaip duomenimis. Metaprogramoje manipuliuojama aukštesnio lygio duomenimis, t. y. programomis pagal pasirinktą bendrinimo algoritmą (Štuikys, Damaševičius, 2013a). Metaprograma užrašoma (2.2) išraiška:

$$„metaprograma = programa + bendrinimo\ algoritmas“ \ (2.2)$$

Metaprogramavimo atveju bendrinimo algoritmas manipuliuoja aukštesnio lygio duomenimis – programomis, t. y. metaduomenys yra programų egzemplioriai. Žemiau pateikiamas programos ir metaprogramos skirtumus demonstruojantis paprastas pavyzdys, nagrinėjamas metodas realizuojantis funkciją $y=x+a$

Programa (Pascal):

```
function Funkcija(x, a: integer);
   var y: integer;
begin
   y:=0;
   y:= x + a;
   writeln (y);
end.
```

Galimas rezultatas, kai $x = 2$, $a = 3$:

```
y = 5.
```

Metaprograma (metakalba – PHP, tikslo kalba Pascal):

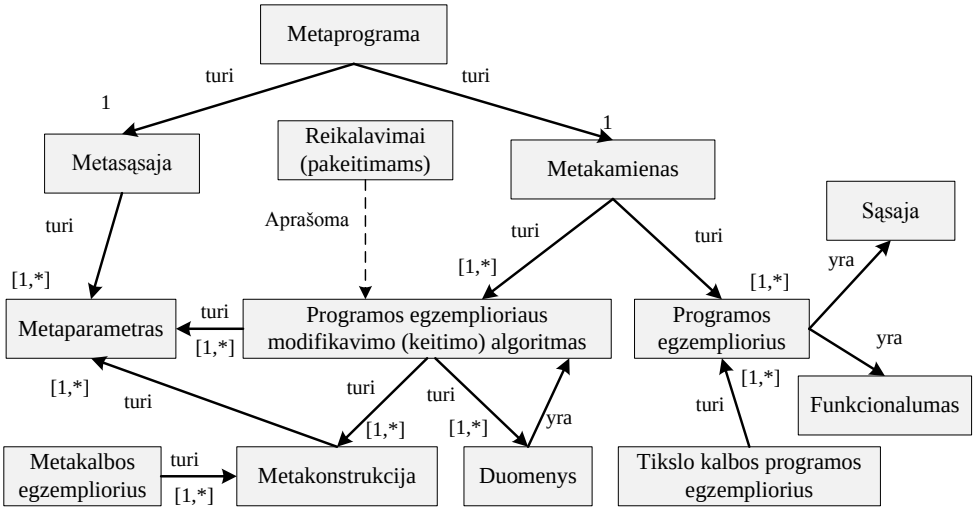
```
<?php
 $mx =$_POST[mx];
 $ma =$_POST[ma];
 $myFile = "result.c";
 $fr = fopen($myFile, 'w');
 fwrite($fr," function Funkcija (x: integer);\n");
 fwrite($fr," var y: integer; \n");
 fwrite($fr," begin \n");
 fwrite($fr," y:=0; \n");
 fwrite($fr," y:= $mx + $ma; \n");
 fwrite($fr," writeln (y); \n");
 fwrite($fr," end. \n");
 fclose($fr);
?>
```

Galimas rezultatas, kai *mx* = x*, *ma* = 2:

```
function Funkcija (x: integer);
   var y: integer;
begin
   y:=0;
   y:= x* + 2;
   writeln (y);
end.
```

Iš pateikto pavyzdžio matome, kad programos darbo rezultatas yra funkcijos reikšmė, paskaičiuota su tam tikromis kintamųjų vertėmis. Metaprogramos darbo rezultatas – sugeneruota funkcija su tam tikromis metaparametrų vertėmis.

Heterogeninė metaprograma – (angl. *meta-program*) tai aukštesnio lygio programa, kuri sukuria kitą, žemesnio lygio programą. Metaprograma – tai programų generatorius. (Štuikys, Damaševičius, 2013a). Metaprograma aprašo pagal semantiką ar sintaksę tarpusavyje giminingų programų šeimyną. Konkretus programos egzempliorius automatiškai generuojamas iš vykdomosios specifikacijos (metaprogramos) panaudojant metakalbos procesorių. Svarbų vaidmenį metaprogramoje vaidina metaparametrai. Metaprogramos vykdymo metu, priklausomai nuo parinktų metaparametrų reikšmių, sukuriamas konkretus tikslo kalbos programos egzempliorius. Metaprogramos modelis (Štuikys, Damaševičius, 2013a) parodytas 2.7 pav.



2.7 pav. Metaprogramos modelis (Štuikys, Damaševičius, 2013a)

Metaprograma taip pat yra programa. Norint sukurti tinkamą metaprogramą, reikia gerai pažinti sritį, išgauti reikiamus srities artefaktus (įskaitant jos modelius), žinoti reikalavimus bei turėti būtinas programos kūrimo priemones (metakalbą, tikslo kalbą). Tik turint reikiamą informaciją galima kurti srities modelius ir juos transformuoti į tam tikras programas.

Metaprogramavimo technologija leidžia pasiekti programų kūrimo automatizavimo tikslus (Štuikys, Damaševičius, 2013a). Programų generatorių

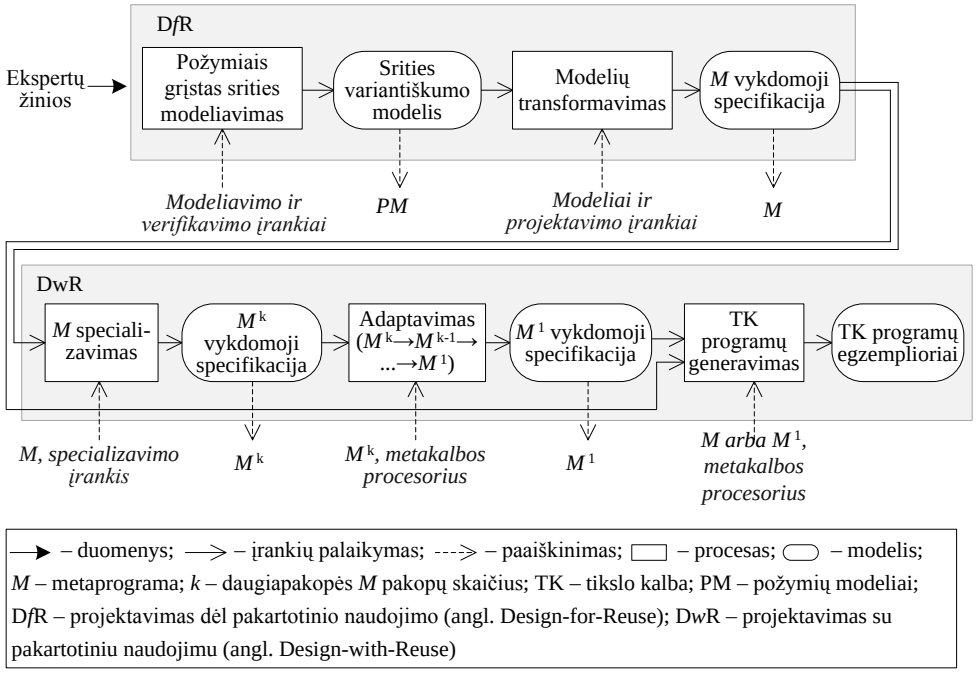
(heterogeninių metaprogramų) kūrimas yra tiesiogiai susijęs su srities bendrybių-skirtybių modeliavimu ir valdymu (Capilla ir kt., 2013).

2.6. Apibendrintas tyrimo karkasas

Atlikta literatūros apžvalga parodė, kad transformavimo tyrimai labai platūs. Dėl sistemų sudėtingumo augimo tyrimai persikelia į aukštesnį abstrakcijos lygmenį, sparčiai vystosi sistemų kūrimo metodai, kuriuose centrinį vaidmenį vaidina kuriamos sistemos modeliai. Sistemos nagrinėjamos pakartotinio naudojimo kontekste, o pakartotinis naudojimas glaudžiai siejasi su komponentika, transformacijomis ir automatizavimu.

Metaprogramavimo technologija leidžia pasiekti programų kūrimo automatizavimo tikslus. Tačiau metaprogramos sukūrimas reikalauja kruopščios srities analizės ir modeliavimo, metaprogramavimo principų ir metodų supratimo, mokėjimo programuoti vienu metu panaudojant bent dvi programavimo kalbas, o programavimo darbų efektyvumui didinti – investicijų į metaprogramavimo įrankių kūrimą.

Metaprogramų kūrimo, transformavimo ir panaudojimo procesuose gali būti pritaikytos programų sistemų inžinerijoje gerai žinomos pakartotinio panaudojimo paradigmos: projektavimas dėl pakartotinio naudojimo (angl. *design-for-reuse*, D*f*R) ir projektavimas su pakartotiniu naudojimu (angl. *design-with-reuse*, D*w*R) (Sametinger, 1997). 2.8 pav. pateiktas apibendrintas disertacijos tyrimo karkasas, apimantis metaprogramų kūrimo, specifikavimo, adaptavimo ir tikslo kalbos programų egzempliorių generavimo procesus.



2.8 pav. Disertacijos apibendrintas tyrimo karkasas

Disertacijos tyrimo karkase projektavimas dėl pakartotinio naudojimo (D*f*R) apima srities modeliavimo ir modelių transformavimo (metaprogramos modeliavimo ir kūrimo) procesus. Sėkmingam metaprogramos sukūrimui reikalingi neprieštaringi srities modeliai, jiems sukurti naudojamos ekspertų žinios ir patirtis. Tik tada, kai problema yra aiškiai apibrėžta, sukurtas srities modelis, galima kurti sprendimą (metaprogramą).
D*w*R veiklos atliekamos vėliau, kai metaprograma jau sukurta. Disertacijos tyrimo karkase projektavimas su pakartotiniu naudojimu (D*w*R) apima metaprogramos specializavimo, adaptavimo ir tikslo kalbos programų egzempliorių generavimo procesus. Vartotojas, žinodamas taikymo kontekstą ir reikalavimus, gali specializuoti metaprogramą ir vėliau ją adaptuoti prie konkrečių poreikių.

2.7. Išvados

1. Srities inžinerijos metodai sudaro metodologinį pagrindą nagrinėti modelių ir programų transformacijas. Tik gerai pažinus sritį ir išgavus reikiamą informaciją galima kurti giminingas programinės įrangos sistemas, programų šeimynas.
2. Modeliais grįstas projektavimas užtikrina efektyvų sistemų kūrimą pakartotinio panaudojimo kontekste. Pastarasis glaudžiai siejasi su komponentika, transformacijomis ir automatizavimu.
3. Nustatyta, kad esminis reikalavimas kuriamiems modeliams bei jų transformacijoms yra *bendrybių-skirtybių ir jų sąveikos identifikavimas*. Modeliams atvaizduoti ir transformacijoms aprašyti aukštame abstrakcijos lygmenyje pasiūlyta požymiais grindžiama notacija, kadangi ji yra grafinė, intuityviai suvokiama ir gerai palaikoma.
4. Heterogeninis metaprogramavimas įgalina pasiekti programų kūrimo automatizavimo tikslus, o programų generatoriai realizuoja generatyvinį pakartotinį panaudojimą.
5. Esminis šio skyriaus rezultatas – apibendrintas tyrimo karkasas disertacijos uždaviniams nagrinėti, kuris buvo pasiūlytas kaip atliktos analizės išdava.

3. METAPROGRAMŲ KŪRIMAS PANAUDOJANT POŽYMIŲ MODELIŲ TRANSFORMACIJAS

3.1. Įvadas

Skyriuje nagrinėjamas heterogeninės metaprogramos (toliau metaprogramos) kūrimo uždavinys. Šio skyriaus tikslas – suformuluoti uždavinio reikalavimus, pagrindines prielaidas ir pateikti sprendimo principą bei sprendimo metodiką taip, kad būtų galima uždavinį spręsti automatizuotai. Taigi, pagrindinis reikalavimas – sukurti prielaidas metaprogramų kūrimo procesui automatizuoti.

Kaip parodė literatūros apžvalga, automatizuotas kūrimas pirmiausia reikalauja probleminės srities abstraktaus atvaizdavimo, t. y. modelio. Sudarytas tyrimo karkasas (žr. 2 skyrių) remiasi požymiais grįstu srities modeliavimu, kadangi jis tiesiogiai atvaizduoja srities variantiškumą. O variantiškumas (kaip nustatyta 2.5 skyrelyje) yra pagrindas metaprogramoms kurti. Todėl šio skyriaus uždavinys yra formuluojamas ir nagrinėjamas kaip požymių modelių transformavimo uždavinys. Abstrakčiai uždavinys formuluojamas kaip probleminės srities požymių modelio *atvaizdavimas* (abstraktus transformavimas) į sprendimo srities modelį. Sprendimo sritimi laikoma metaprogramavimas. Abi sritys (probleminė ir sprendimo) turi būti atvaizduotos tos pačios notacijos modeliais. Atvaizdavimo rezultatas yra metaprogramos modelis.

Šiame skyriuje nagrinėjami tokie uždavinio aspektai: požymių modelių notacija, tų modelių atvaizdavimas abiem sritims, modeliais grindžiami procesai, modelių savybės, transformavimo taisyklės. Taisyklės apima taip pat transformavimą „metaprogramos modelis-metaprogramos vykdomoji specifikacija“. Formalus modelių aprašymas, metaprogramų notacija ir modelių transformavimo taisyklės sukuria prielaidas automatiniams įrankiams kurti (5 skyrius).

Šio skyriaus struktūra tokia: iš pradžių 3.2 skyrelyje suformuluotas metaprogramos kūrimo uždavinys, aprašyti uždavinio reikalavimai; 3.3 skyrelyje aprašyti probleminės srities modelių išgavimo procesai, pateiktas probleminės srities aiškinamasis pavyzdys ir apibrėžtas probleminės srities požymių modelių formalizmas; 3.4 skyrelyje pateiktas sprendimo srities formalizmas; 3.5 skyrelyje apibrėžtos požymiais grindžiamų modelių transformavimo taisyklės; 3.6 skyrelyje pateiktas probleminės srities atvaizdavimo sprendimo srityje požymių modelis ir sukurtos metaprogramos pavyzdys; 3.7 skyrelyje pateikiamas skyriaus apibendrinimas ir 3.8 skyrelyje suformuluotos skyriaus išvados.

3.2. Metaprogramos kūrimo uždavinio reikalavimai ir formulavimas

Metaprogramos kūrimas – tai procesas, kurio metu sukuriama (suprogramuojama) metaprogramos vykdomoji specifikacija. Metaprogramos gali būti kuriamos rankiniu būdu arba automatizuotai, panaudojant tam tikras kūrimo priemones. Kuriant rankiniu būdu, programuotojas rašo metaprogramą pasinaudodamas tam tikrais redaktoriais ir metaprogramos procesoriumi. Naudojamos priemonės padeda programuotojui aptikti programavimo klaidas, tačiau negeneruoja programinio kodo. Šis kūrimo būdas nereikalauja modelių kaip

privalomų atributų, tačiau modelių turėjimas visada palengvina kūrimo procesą. Kuriant rankiniu būdu, programuotojas turi visus reikalavimus apjungti į vieną bendrą metaprogramos algoritmą, o tai yra sudėtingas uždavinys. Metaprogramos kūrimo procesą galima palengvinti panaudojant specializuotus kūrimo įrankius, o įrankių panaudojimas visada reikalauja tikslių ir detalių modelių, t. y. probleminės ir sprendimo srities modelių. Probleminė sritis – tai srities sprendžiamų uždavinių visuma. Sprendimo sritis – tai heterogeninio metaprogramavimo principai ir metodai. Toliau pateikiami uždavinio sprendimui keliami reikalavimai.

Probleminei sričiai keliami reikalavimai:

1. Turi būti apibrėžta probleminė sritis.
2. Probleminė sritis turi būti susiaurinta, išgaunant posričius, kad galima būtų matyti abstrakčius probleminės srities komponentus.
3. Turi būti modeliuojama didelė probleminės srities skirtybių aibė.
4. Turi būti pasirinktas probleminės srities analizės metodas (požymių modelių notacija).
5. Modelis turi būti formalizuotas ir nustatytos jo savybės.
6. Modelis turi būti detalizuotas iki elementų lygmens, t. y. iki variantinių taškų ir variantų, kad būtų galima atvaizduoti srities variantiškumą.

Sprendimų sričiai keliami reikalavimai:

7. Turi būti sudarytas sprendimo srities modelis. Jį aprašyti aibių notacija, kuri apibrėžia modelio savybes.
8. Aibių notacija turi būti išreikšta probleminės srities požymių notacija.
9. Turi būti išskirti sprendimo srities aukščiausio lygmens elementai (sąsaja ir metakamienas).

Bendrieji reikalavimai:

10. Probleminės ir sprendimo srities modeliai turi būti aprašyti ta pačia notacija.
11. Sukurti srities modeliai turi būti neprieštaringi.
12. Paieškos aibė, konkrečiam taikymui (gimininių programų klasei) realizuoti, turi būti kuo didesnė, taip siekiama realizuoti kuo bendresnę metaprogramą.
13. Modeliuose turi būti atspindėtas taikymo kontekstas.
14. Turi būti suformuluotos transformavimo taisyklės, kurios įgalintų konkretų probleminės srities modelį atvaizduoti į abstraktų sprendimo srities modelį, tokiu būdu sukuriant konkretų sprendimo srities modelį.

Norint realizuoti reikalavimus ir suvaldyti sprendžiamo uždavinio sudėtingumą, darbe naudojamas gerai žinomas programų inžinerijoje koncepcijų atskirties principas. Koncepcijų atskirtis – tai procesas, kurio metu bendras projektavimo uždavinys sudalijamas į dalinius nepriklausomus uždavinius. Šie uždaviniai atskirai nagrinėjami, o vėliau integruojami į galutinį produktą. Darbe atskiriami projektavimo aspektai: lygmuo (aukštesnis lygmuo atskiriamas nuo žemesnio lygmens), sritis (probleminė sritis nuo sprendimo srities).

Metaprogramos kūrimo uždavinys formuluojamas kaip probleminės srities atvaizdavimas sprendimo srityje naudojant formalizuotas modelių transformacijas. Pačiame bendriausiame lygmenyje uždavinys užrašomas (3.1) išraiška.

$$SR = PS \rightarrow SS, \tag{3.1}$$

čia *SR* – sprendimo rezultatas, *PS* – probleminė sritis, *SS* – sprendimo sritis.

Probleminė ir sprendimo sritys turi būti atvaizduotos tiesiogiai. Kadangi darbe parinkta požymių diagramų notacija, sritį galime atvaizduoti požymių modeliais ir (3.1) išraišką perrašyti (3.2) išraiška.

$$PM(SR) = PM(PS) \rightarrow PM(SS), \tag{3.2}$$

čia *PM* – požymių modelis.

Iš pradžių požymių modeliai yra abstraktūs. Jų struktūrizavimo lygis nėra pilnas. Norint atlikti transformaciją, modeliai turi būti išreikšti kuo detaliau, t. y. žemiausio lygmens elementais, kurie bus apibrėžti vėliau. Tik turėdami žemiausio lygmens elementus galime formalizuoti transformavimo taisykles.

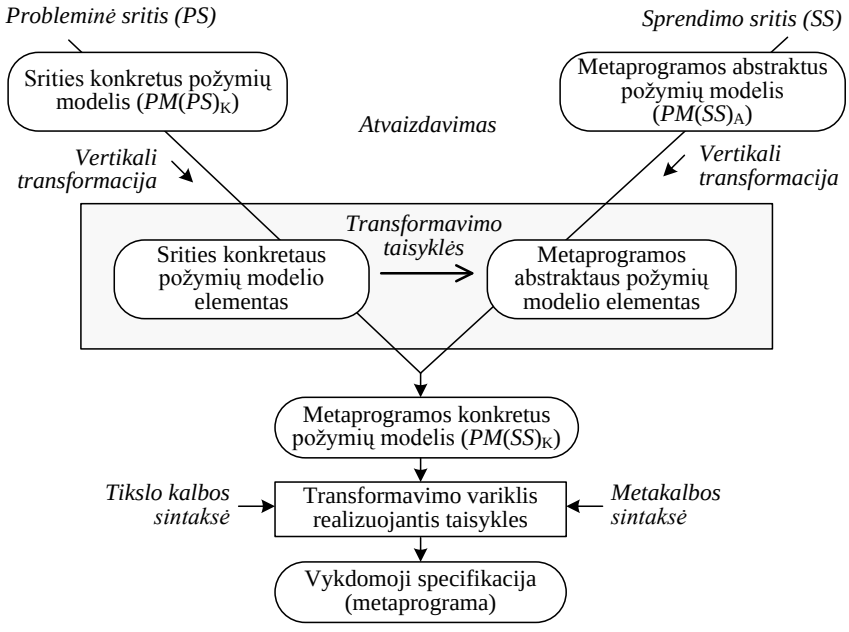
Panagrinėkime (3.2) išraiškos dešiniąją pusę. Čia probleminės srities požymių modelis yra abstraktus. Jį reikia konkretizuoti, kitaip tariant, turime atlikti transformavimą pagal (3.3) išraišką.

$$PM(PS)_A \rightarrow PM(PS)_K, \tag{3.3}$$

čia $PM(PS)_A$ – probleminės srities abstraktus požymių modelis, $PM(PS)_K$ – probleminės srities konkretus požymių modelis (apibrėžimai ir pavyzdžiai bus pateikiami vėliau).

Uždavinio patikslinta formuluotė užrašoma (3.4) išraiška. Bendras sprendimo principas pavaizduojamas Y-ko diagrama (3.1 pav.), kuri realizuoja koncepcijų atskirties principą.

$$PM(SS)_K = PM(PS)_K \rightarrow PM(SS)_A. \tag{3.4}$$



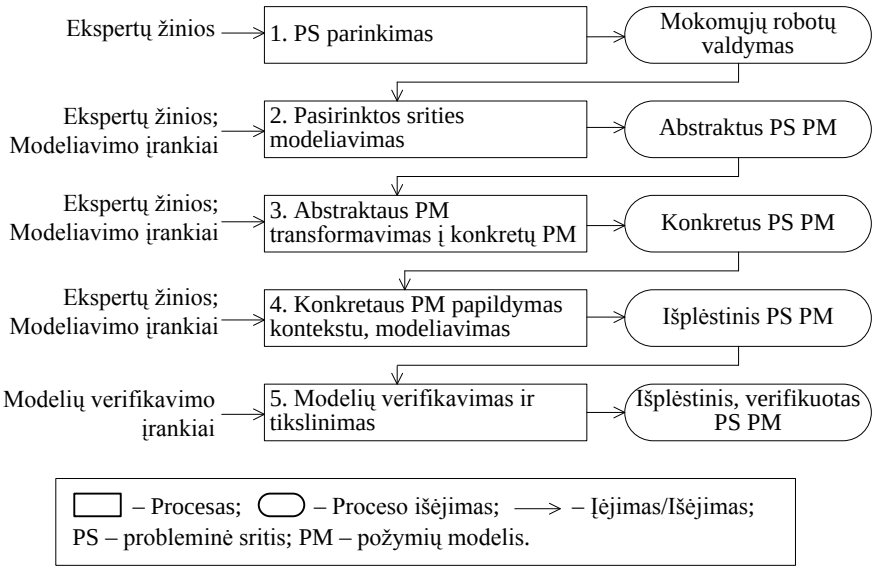
3.1 pav. Metaprogramos kūrimo uždavinio sprendimo principas

Sprendžiant metaprogramos kūrimo uždavinį, pirmiausiai atskirai nagrinėjamas probleminės ir sprendimo sričių modelių daugiapakopis transformavimas. Atliekama vertikali transformacija, kuri išskiria esminius modelių požymius, t. y. ši transformacija žemina modelio abstrakcijos lygmenį. Kai turime sukurtus detalius abiejų sričių modelius, atliekamas probleminės srities atvaizdavimas sprendimų srityje. Ši transformacija atliekama vadovaujantis transformavimo taisyklėmis. Transformavimo taisyklės aprašo, kaip probleminės srities elementas yra transformuojamas į sprendimo srities elementus.

Toliau bus nagrinėjama Y-ko diagramos (3.1 pav.) kairioji šaka.

3.3. Probleminės srities modelių išgavimo procesai

Probleminės srities modelio tikslinimo procesą galima suskirstyti į atskirus etapus (3.2 pav.). Kiekvienas nagrinėjamas etapas turi tikslą, t. y. siekia tam tikro rezultato ir turi įėjimas-procesas-išėjimas struktūrą.



3.2 pav. Probleminės srities modelio tikslinimo procesas (žr. taip pat 3.3.1 skyrelį)

1. *Probleminės srities pasirinkimo etapas*. Parinkta probleminė sritis – mokomųjų robotų valdymas. Ši sritis buvo pasirinkta siekiant darbe pritaikyti Burbaitės (2014) sukurtus probleminės srities modelius ir juos eksperimentiškai išbandyti realiame taikyme sprendžiant šios disertacijos uždavinius (automatizuotą metaprogramų kūrimą ir jų transformavimą).
2. *Probleminės srities modeliavimo etapas*. Apibrėžiama srities apimtis, ribos, analizuojami srities sąryšiai su kitomis sritimis. Analizuojamos srities taikymų bendrybės ir skirtybės. Naudojamas FODA metodas (angl. *Feature-Oriented Domain Analysis*). Kaip rezultatas gaunamas probleminės srities abstraktus modelis.
3. *Abstraktaus probleminės srities požymių modelio transformavimo į konkretų probleminės srities požymių modelį etapas*. Probleminės srities modelis

siaurinamas ir detalizuojamas iki žemiausio lygmens bazinių elementų. Atliekama požymių analizė, modeliuojami esybių sąryšiai, atliekama funkcinė analizė. Rezultate gaunamas konkretus probleminės srities požymių modelis. Darbe naudojami Burbaitės (2014) sukurti konkretūs probleminės srities požymių modeliai.

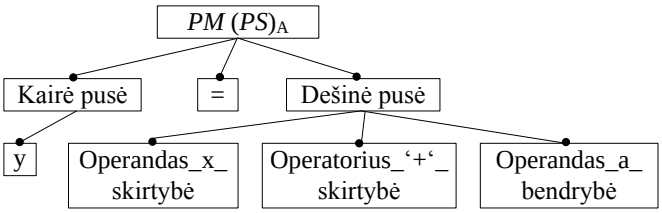
4. *Konkretaus probleminės srities požymių modelio papildymo kontekstu ir modeliavimo etapas*. Atliekama pasirinktos srities konteksto analizė ir modeliavimas. Sukurtas konteksto modelis priklauso nuo eksperto žinių apie sritį ir uždaviniui keliamų reikalavimų. Konkretus probleminės srities požymių modelis papildomas kontekstu, gaunamas išplėstinis probleminės srities požymių modelis.
5. *Modelių verifikavimo ir tikslinimo etapas*. Analizuojami sukurti modeliai, tikrinamas jų teisingumas. Procesas vyksta naudojant požymių modelių analizės įrankį „SPLOT“. Atsižvelgiant į analizės metu gautą modelių charakteristikų ir savybių statistiką, modeliai koreguojami ir pakartotinai verifikuojami. Taip gaunamas galutinis, tikslus probleminės srities modelis.

3.3.1. Probleminės srities aiškinamasis pavyzdys (požymių modelis)

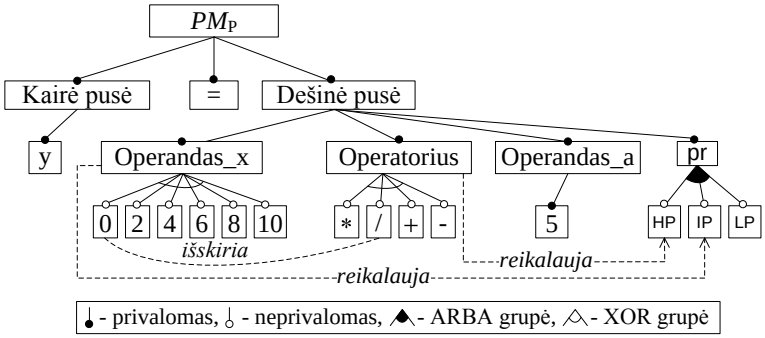
Šio skyrelio tikslas – paaiškinti jau įvestas sąvokas, neformaliai pasirenkant konkretų uždavinį, kuris atspindėtų esminį probleminės srities aspektą. Tokiu aspektu galėtų būti reikalingi aritmetiniai skaičiavimai, kurie yra sudėtinė roboto valdymo dalis. Skaičiavimams aprašyti parenkama tiesinė lygtis: $y = x + a$. Norėdami aprašyti apibendrintą lygtį turime išskirti nagrinėjamos srities bendrybes ir skirtybes. Operandą „*a*“ laikykime srities bendrybe, o operatorių „+“ ir operandą „*x*“ – srities skirtybėmis. Pažymint bendrybes ir skirtybes lygties išraišką galima užrašyti taip:

y = <operandas x skirtybė> <operatorius ‘+‘ skirtybė> <operandas a bendrybė>.

Skirtybių nagrinėjamoje srityje galime atrasti daugiau, tačiau apsiribojame pasirinktomis, kad kuo paprasčiau pailiustruotume probleminės srities požymių modelius. 3.3 pav. parodytas nagrinėjamos tiesinės lygties šeimynos, kaip srities, abstraktus požymių modelis, o 3.4 pav. – išplėstinis požymių modelis. Pastarasis skiriasi nuo ankstesniojo tuo, kad variantiniai taškai turi variantų reikšmes. Šis modelis turi pilną informaciją apie nagrinėjamą sritį, įskaitant konteksto informaciją (požymis *pr*, kuris parodo kitų požymių prioritetinę priklausomybę (žr. 3.14 apibrėžimą). Čia modelis detalizuotas iki elementų lygmens, t. y. iki variantinių taškų ir variantų.



3.3 pav. Aiškinamojo pavyzdžio abstraktus požymių modelis



3.4 pav. Aiškinamojo pavyzdžio išplėstinis požymių modelis

Išplėstiniame modelyje taip pat atvaizduojami apribojimai „*išskiria*“ (angl. *excludes*) ir „*reikalauja*“ (angl. *requires*). Dabar visiškai išaiškėja 3.1 pav. (kairiosios šakos) prasmė.

3.3.2. Požymių modelių formalizavimas

Šio skyrelio tikslas – išskirti probleminės srities požymių modelių elementus ir juos formaliai apibrėžti. Tikslas išplaukia iš uždavinio formulavimo (žr. 3.1 pav. kairiąją šaką).

Požymių modelis – tai srities modelis, kuriame srities artefaktai modeliuojami požymiais. Požymių modelis aprašo *privalomuosius*, *alternatyviuosius* ir *pasirenkamuosius* srities požymius, tėvo-vaiko sąryšius ir požymių tarpusavio apribojimus (žr. 2.2.3 skyrelį). Požymių modeliai vizualiai atvaizduojami požymių diagramomis.

3.1 apibrėžimas. Požymių diagrama (angl. *feature diagram*) – speciali grafinė notacija, aprašanti požymių modelius.

Požymių diagrama – tai kryptinis aciklinis grafas, kurį sudaro rinkinys viršūnių, tiesinių briaunų ir briaunų žymų (žr. 2 skyriaus 2.5 lent.). Šakninis elementas atvaizduoja aukščiausio lygmens požymį (pvz., sritis, sistema, komponentas). Tarpinės viršūnės atvaizduoja sudėtinius požymius, o diagramos paskutinės viršūnės atvaizduoja požymius, kurie nėra skaidomi į smulkesnius požymius. Grafo briaunos nurodo ryšius arba priklausomybes tarp požymių, o žymos ant briaunų nurodo, kokio tipo ryšys yra tarp požymių. Požymiai gali būtie susieti *tėvo-vaiko* tipo ryšiu arba apribojimo ryšiu.

Požymių diagramos nestandartizuotos, todėl darbe parinkta kitų tyrėjų darbuose naudojama notacija (Thum ir kt., 2009; Acher ir kt., 2013).

3.2 apibrėžimas. Požymis (angl. *feature*) išorinė, t. y. vartotojui matoma srities charakteristika (Kang ir kt., 1990), kokybinė ypatybė arba funkcinis reikalavimas.

3.3 apibrėžimas. Variantinis taškas (angl. *variant point*) – tai privalomųjų ir alternatyvinių požymių grupės tėvinis požymis.

3.4 apibrėžimas. Variantas (angl. *variant*) – tai skirtybės variantinio taško reikšmė. Ši reikšmė duotame kontekste neskaidoma į smulkesnes.

3.5 apibrėžimas. Požymio modelio elementas yra bet kuri iš išvardytų esybių: variantinis taškas, variantas, požymis, sąryšiai, apribojimai.

Požymiai tarpusavyje gali būti susieti trijų tipų *tėvo-vaiko* ryšiu: p*rivalomieji* (IR), n*eprivalomieji* (ARBA), a*lternatyviniai* (ARBA, variantinis (XOR)).

3.6 apibrėžimas. Privalomasis (angl. *mandatory, AND*) požymis atspindi būtinas ir nekintančias srities objekto charakteristikas (srities bendrybes). Jis būtinai turi būti pasirinktas, jeigu jo tėvinis požymis yra pasirinktas. Privalomas *tėvo-vaiko* ryšys užrašomas (3.5) išraiška (Thum ir kt., 2009):

$$(P \Rightarrow \wedge_{i \in M} C_i) \wedge (\vee_{1 \le i \le n} C_i \Rightarrow P); M \subseteq \{1,..., n\}, \tag{3.5}$$

čia P žymi tėvo požymį, $C_1, ..., C_n$ yra jo vaikai.

3.7 apibrėžimas. Neprivalomasis (angl. *optional, OR*) požymis atspindi kintamas srities objekto charakteristikas (srities skirtybes). Jis gali būti pasirinktas, jeigu jo tėvinis požymis yra pasirinktas. Neprivalomas *tėvo-vaiko* ryšys užrašomas (3.6) išraiška (Thum ir kt., 2009):

$$P \Leftrightarrow \vee_{1 \le i \le n} C_i. \tag{3.6}$$

3.8 apibrėžimas. Alternatyvinis (angl. *alternative, XOR*) požymis atspindi kintamas srities objekto charakteristikas (srities skirtybes). Jis gali būti pasirinktas tik vienas, jeigu jo tėvinis požymis yra pasirinktas. Alternatyvinis *tėvo-vaiko* ryšys užrašomas (3.7) išraiška (Thum ir kt., 2009):

$$(P \Leftrightarrow \vee_{1 \le i \le n} C_i) \wedge_{i<j} (\neg C_i \vee \neg C_j). \tag{3.7}$$

3.9 apibrėžimas. Apribojimas „*reikalauja*“ modeliuoja sąveiką tarp požymių. Ši sąveika nurodo, kad parinkus požymį F_1 būtinai turi būti parinktas ir požymis F_2. Formaliai, panaudojant teiginių logiką, šis apribojimas užrašomas (3.8) išraiška (Thum ir kt., 2009):

$$\neg F_1 \vee F_2 \tag{3.8}$$

3.10 apibrėžimas. Apribojimas „*išskiria*“ modeliuoja požymių sąveiką, kai vienas požymis išskiria kitą. Ši sąveika nurodo, kad požymiai F_1 ir F_2 vienu metu negali būti parinkti. Formaliai šis apribojimas užrašomas (3.9) išraiška (Thum ir kt., 2009):

$$\neg F_1 \vee \neg F_2 \tag{3.9}$$

3.11 apibrėžimas. Bazinis srities požymių modelis yra junginys (Acher ir kt., 2013) užrašomas (3.10) išraiška:

$$PM_B = \langle G, E_{mand}, G_{xor}, G_{or}, REQ, EX \rangle, \tag{3.10}$$

čia $G = (F, E, r)$ yra medis, F yra baigtinė aibė požymių, $E \subseteq F \times F$ yra baigtinė briaunų aibė, $r \in F$ yra pagrindinis srities požymis; $E_{mand} \subseteq E$ yra briaunų apibrėžiančių privalomus požymius su jų tėviniais požymiais aibė; $G_{xor} \subseteq P(F) \times F$ ir $G_{or} \subseteq P(F) \times F$ apibrėžia alternatyvių ir pasirenkamų požymių grupes ir yra vaikų

požymių bei jų bendrų tėvų požymių porų aibės; *REQ* ir *EX* yra baigtiniai rinkiniai apribojimų „*reikalauja*“ ir „*išskiria*“.

3.12 apibrėžimas. Abstraktus požymių modelis – tai mažo detalumo srities požymių modelis, turintis požymių, kurie kitame kontekste gali būti išskaidyti į smulkesnius požymius. Šiame modelyje atspindimos tik tam tikros, esminės srities charakteristikos (žr. 3.3 pav.).

3.13 apibrėžimas. Konkretus srities požymių modelis – tai aukšto detalumo srities požymių modelis, išgaunamas iš abstraktaus požymių modelio detalizuojant požymius iki elementų lygmens. Šis modelis, duotajame kontekste, neturi požymių, kurie gali būti skaidomi į smulkesnius.

3.14 apibrėžimas. Srities konteksto modelis – tai hierarchinis neraiškios (angl. *fuzzy*) logikos lingvistinių kintamųjų modelis, kuris traktuojamas kaip požymis paimtas iš prioritetų reikšmių aibės {HP, IP, LP}, čia HP – aukštas prioritetas, IP – vidutinis prioritetas ir LP – žemas prioritetas.

3.15 apibrėžimas. Agreguotas požymių modelis – tai požymių modelis, kuris gaunamas sujungus du ir daugiau požymių modelių į vieną.

3.16 apibrėžimas. Probleminės srities išplėstinis (pilnas) modelis yra požymių modelis, kuris gaunamas atlikus konkretaus srities požymių modelio ir konteksto modelio agregavimą, kadangi, bendruoju atveju, sritis turi bazinę informaciją (artefaktus) ir jos (jų) kontekstą (žr. 3.3 pav.). Išplėstinis modelis užrašomas (3.11) išraiška:

$$PM_P = PM_K \oplus PM_C. \qquad (3.11)$$

čia PM_P – probleminės srities išplėstinis požymių modelis, PM_K – srities konkretus požymių modelis, PM_C – srities konteksto modelis; $\oplus$ – agregavimas.

Konkretus srities požymių modelis turi aiškią struktūrą ir aiškiai išskirtus požymius, kuriais aprašomas konkretus programos šeimynos produktas. Konteksto modelis yra aukštesnio prioriteto, jame išskiriama esminė informacija apie srities kontekstą.

3.17 apibrėžimas. Požymių modelio konfigūracija – tai požymio modelio submodelis, kuris apima modelio visus variantinius taškus ir kiekvieno variantinio taško vieną variantą, atsižvelgiant į modelio sąryšius ir apribojimus.

3.18 apibrėžimas. Konfigūracijų skaičius – tai skaičius visų galimų konfigūracijų, kurios išgaunamos iš modelio.

3.3.3. Požymių modelių savybės

Šiame skyrelyje formuluojamos pagrindinės probleminės srities požymių modelių savybės.

3.1 Savybė. Sritis atvaizduojama bendrybėmis, skirtybėmis ir jų sąryšiais.

3.2 Savybė. Srities bendrybės atvaizduojamos privalomaisiais požymiais.

3.3 Savybė. Srities skirtybės atvaizduojamos variantiniais taškais ir variantais.

3.4 Savybė. Požymių sąveika atvaizduojama apribojimais „*reikalauja*“ *ir* „*išskiria*“.

3.5 Savybė. Sritis turi bazinę dalį ir kontekstą. Abi dalys atvaizduojamos agreguotame modelyje.

3.6 Savybė. Konteksto modelis atvaizduojamas konteksto variantiniu tašku ir jo reikšmių variantais.

3.7 Savybė. Agreguotas modelis yra išplėstinis modelis.

3.8 Savybė. Konfigūracijų skaičius apibrėžia visuminę srities bendrybių-skirtybių erdvę.

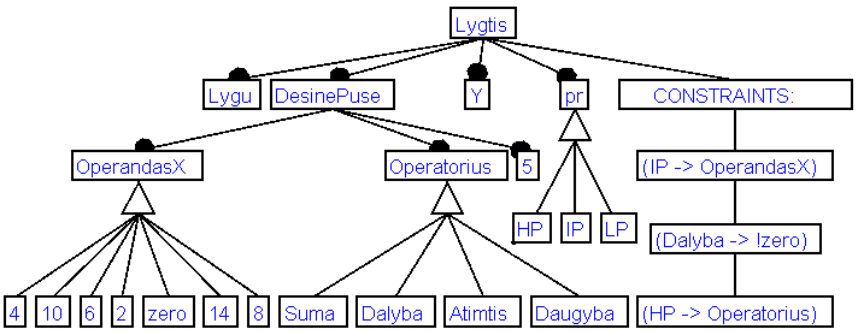
3.9 Savybė. Verifikuoti požymių modeliai laikomi neprieštaringais (jie semantiškai ir sintaksiškai teisingi).

3.10 savybė. Požymių modelio PM_P variantinis taškas atspindintis kontekstą (3.4 pav. „pr“) vaidina ypatingą vaidmenį apibrėžiant metaparametrų seką metaprogramos sąsajoje. Apribojimai *reikalauja* nurodo variantinio taško prioritetinę reikšmę.

3.3.4. Probleminės srities modeliavimo ir verifikavimo procesai

Grįžkime prie probleminės srities procesų diagramos (3.2 pav.). Panagrinėkime 4 ir 5 procesus. Jie apima modeliavimą ir verifikavimą. Modeliavimo tikslas – sukurti išplėstinius požymių modelius, o verifikavimo tikslas – patikrinti jų teisingumą. Procesams realizuoti reikalingi įrankiai. Buvo pasirinkti „FAMILIAR“ (angl. *FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) (Acher ir kt., 2013) ir „SPLOT“ (angl. *Software Product Lines Online Tools*) (Mendonca ir kt., 2009) įrankiai modeliavimui ir verifikavimui (žr. 2.2.2 skyrelį). Pasirinkimą nulėmė įrankių sąlyginis paprastumas ir laisva prieiga bei pakankamas funkcionalumas eksperimentiniam taikymui.

Požymių diagramoms kurti naudojama srities kalba FAMILIAR (įrankis vadinamas tuo pačiu vardu). Ši kalba skirta didelės apimties modeliams valdyti (Acher ir kt., 2013; Collet ir Lahire, 2013) ir papildyti kitus egzistuojančius požymių modeliavimo įrankius. Ji leidžia konstruoti požymių diagramas ir atlikti manipuliacijos su modeliais (skaidymą, suliejimą, agregaciją ir kt.). Žiūrint iš transformacijų pusės, „FAMILIAR“ įrankis gali atlikti dviejų tipų transformacijas: mažinančias modelio abstrakcijos lygį ir išlaikančias tą patį abstrakcijos lygį. Šio įrankio svarbi savybė, kad jis pateikia lengvai suprantamą grafinį modelio atvaizdavimą. 3.5 pav. parodyta aiškinamojo pavyzdžio (žr. 3.3.1 skyrelį) požymių diagrama sukurta naudojant „FAMILIAR“ įrankį.



3.5 pav. Požymių diagrama (panaudojant „FAMILIAR“)

Sukurtiems srities modeliams analizuoti ir verifikuoti naudojamas įrankis „SPLOT“ (Mendonca ir kt., 2009; Simmonds ir kt. 2011). Tai internetinė sistema, kuri modelių analizei naudoja tokius logika pagrįstus metodus, kaip SAT Solver klasės algoritmus ir binarinėmis sprendimų diagramomis pagrįstus požymių modelių struktūrinių metrikų skaičiavimo metodus. Įrankiu „SPLOT“ atliekamas formalus sukurtų modelių verifikavimas. Įrankis vartotojui pateikia esmines modelių charakteristikas. Atsižvelgiant į jas, vartotojas gali priimti sprendimą, ar modeliai yra teisingi, ar gali būti panaudoti kuriant metaprogramas.

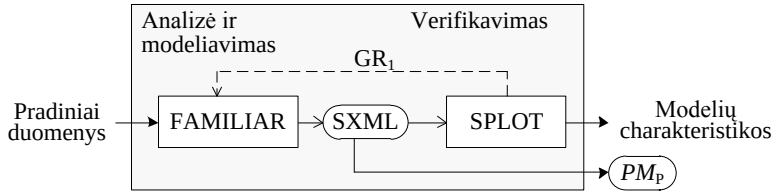
3.1. lent. parodyti požymių modelio (3.5 pav.) kokybės parametrai ir metrikos apskaičiuotos naudojant įrankį „SPLOT“.

3.1 lentelė. Požymio modelio verifikavimo charakteristikos (parametrai apskaičiuoti naudojant įrankį „SPLOT“)

El. Nr.	Parametras	Vertė
1.	Požymių skaičius	22
2.	Neprivalomų požymių skaičius	0
3.	Privalomų požymių skaičius	7
4.	Esminių požymių skaičius	8
5.	Sugrupuotų požymių skaičius	14
6.	OR grupių skaičius	0
7.	XOR grupių skaičius	3
8.	Apribojimų skaičius	3
9.	Apribojimų reprezentatyvumas, %	0.27
10.	Skirtingų kintamųjų skaičius apribojimuose	6
11.	Apribojimų išlygų tankis	0.50
12.	Medžio gylis	4
13.	Konfigūracijų skaičius	81
14.	Variantiškumo laipsnis, %	1.9312E-3
15.	BDD mazgų skaičius	41
16.	Modelio neprieštaringumas	Neprieštaringas
17.	Pertekliniai požymiai	Nėra

„FAMILIAR“ ir „SPLOT“ įrankiai tarpusavyje yra suderinti. „FAMILIAR“ įrankiu sukurtos požymių diagramos gali būti išsaugotos SXML (angl. *Simple Extensible Markup Language*) formatu, kuris vėliau įkeliamas į „SLPOT“ įrankį analizei ir verifikavimui. Požymių diagramų atvaizdavimas XML formatu leidžia realizuoti duomenų integravimą tarp įrankių.

„FAMILIAR“ ir „SPLOT“ įrankių panaudojimas srities analizės ir modeliavimo procese padeda užtikrinti sukurtų modelių semantinį teisingumą. 3.6 pav. pavaizduotas įrankių panaudojimas probleminės srities modeliavimo procese.



3.6 pav. Įrankių panaudojimas srities modeliavimo procese

Taigi, įvykdžius modeliavimo ir verifikavimo procesus, sukuriamas neprieštaringas (angl. *consistent*) išplėstinis modelis, kuris tenkina išraiškos (3.3) sąlygą ir probleminės srities reikalavimus.

Toliau bus nagrinėjama Y-ko diagramos (3.1 pav.) dešinioji šaka.

3.4. Sprendimo srities formalizavimas

Tikslas – išskirti sprendimo srities požymių modelių elementus ir juos formaliai apibrėžti. Tikslas išplaukia iš uždavinio formulavimo (žr. 3.1 pav. dešiniąją šaką).

Heterogeninė metaprograma yra aukšto lygmens bendrinė vykdomoji specifikacija, kuri yra sukurta panaudojant dvi kalbas: tikslo ir metakalbą. Metaprograma generuoja tikslo kalbos programų egzempliorių, kurie tarpusavyje panašūs pagal sintaksę arba semantiką, rinkinį.

Metakalba vadinama aukštesnio lygio kalba. Ji skirta modifikuoti tikslo kalba parašytas programas. Tikslo kalba (angl. *target language*) – tai kalba, skirta srities funkcionalumui išreikšti. Metaprogramavime tikslo kalba užrašomas bazinis srities funkcionalumas, o metakalba išreiškiamas bendrasis funkcionalumas, taip išplečiamas programos atkartojamumo laipsnis ir padidinamas jos pritaikomumas.

3.19 apibrėžimas. Metaspecifikacija yra specifikacija, sudaryta iš metasąsajos (angl. *meta-interface*) ir metakamieno (angl. *meta-body*) (3.7 pav.).

Metasąsaja išreiškia metaparametrų reikšmių aibę. Metasąsaja perduoda metaparametrų reikšmes procesoriui, taip sukuriamas tikslo kalbos programos egzempliorius su pasirinktomis metaparametrų reikšmėmis. Metakamienas išreiškia metakalbos funkcijų aibę, kurios valdo metaprogramos funkcionalumą.

Metaprogramos metasąsaja: *išreiškia metaparametrų reikšmių aibę, leidžiančią sukurti tikslo kalbos programą su pasirinktomis metaparametrų reikšmėmis*
Metaprogramos metakamienas: *apibūdina metaprogramos funkcionalumą; struktūriškai specifikuoja tikslo kalbos programų egzempliorių rinkinį*

3.7 pav. Metaprogramos struktūrinis modelis

3.20 apibrėžimas. Metaprogramos struktūrinis modelis $\mu(M)$ (angl. *structural model*) yra metasąsajos modelio $\mu(M_I)$ ir metakamieno modelio $\mu(M_B)$

kompozicija. Formaliai metaprogramos struktūrinis modelis užrašomas (3.12) išraiška:

$$\mu(M) = \mu(M_I) \cup \mu(M_B). \quad (3.12)$$

3.21 apibrėžimas. Metasąsajos modelį $\mu(M_I)$ sudaro n dimensijų netuščia metaparametrų erdvė (Štuikys, Bespalova ir Burbaitė, 2014a). Formaliai metasąsajos struktūrinis modelis užrašomas (3.13) išraiška:

$$\mu(M_I) = R; \quad (3.13)$$

čia $R = \{P; V\}$, P – pilna metaparametrų vardų aibė ($P \neq \varnothing$), t.y $n = |P|$, n – metaparametrų skaičius, V – visų metaparametrų reikšmių aibė $V = \{V_1,..,V_i,..,V_j,..,V_n\}$.

Kiekvienas metaparametras P_i turi savo reikšmių aibę $\{v_{i_1}, v_{i_2},...,v_{i_q}\} \in V$, $P_i := V_i = \{v_{i_1}, v_{i_2},...,v_{i_q}\} \in V$, kur $P_i \in P$, $i \in [1;n]$;, i_q – metaparametro P_i reikšmių skaičius.

3.22 apibrėžimas. Metaparametrai P_i ir P_j ($P_i, P_j \subseteq P$, $i \neq j$, $i, j \in [1,n]$) laikomi nepriklausomais (angl. *independent*), jeigu su bet kuria pasirinkta jų reikšmių pora $\{v_{i_k}, v_{j_t}\}$ ($v_{i_k} \in P_i, v_{j_t} \in P_j$, kur $k \in [1, i_q]$ ir $t \in [1, j_m]$) metaprograma veikia korektiškai (3.14). Tarp tokių metaparametrų reikšmių nėra apribojimų „*išskiria*“ ir „*reikalauja*“.

$$(v_{i_k}\ reikalauja\ v_{j_t}) \vee (v_{i_k}\ išskiria\ v_{j_t}) = \textbf{false}. \quad (3.14)$$

3.23 apibrėžimas. Metaparametrai P_i ir P_j ($P_i, P_j \subseteq P$, $i \neq j$, $i, j \in [1,n]$) laikomi priklausomais (angl. *dependent*), jeigu egzistuoja jų reikšmių pora $\{v_{i_k}, v_{j_t}\}$ ($v_{i_k} \in P_i, v_{j_t} \in P_j$, kur $k \in [1, i_q]$ ir $t \in [1, j_m]$) tarp kurių yra „*išskiria*“ arba „*reikalauja*“ apribojimas (3.15).

$$(v_{i_k}\ reikalauja\ v_{j_t}) \vee (v_{i_k}\ išskiria\ v_{j_t}) = \textbf{true}. \quad (3.15)$$

Kartais literatūroje priklausomi metaparametrai yra traktuojami kaip sąveikaujantys (kalbant apie požymius arba aspektus (Douence, Fradet ir Südholt, 2002)).

3.24 apibrėžimas. Metaparametrų sąveikos grafas $G(P^w, U)$ yra metaprogramos metasąsajos kontekstinis modelis, čia w – metaparametro prioritetinis svoris, skirtas modeliuoti metaparametrų panaudojimo kontekstą ($w \in \{HP, IP, LP\}$,čia HP – aukštas prioritetas, IP – vidutinis prioritetas ir LP – žemas prioritetas).

3.11 savybė. Metaparametrai metasąsajoje rūšiuojami jų prioritetinio svorio mažėjimo tvarka (nuo aukščiausio iki žemiausio).

Metaparametrų sąveikos grafas $G(P^w,U)$ kuriamas siekiant apibrėžti visų metaparametrų sąveikos erdvę. Grafo viršūnių aibė P atitinka metaparametrus. Briaunų aibė U apibrėžiama: $u_{ij}=1$ (briauna egzistuoja), jei metaparametrai P_i ir P_j yra priklausomi (žr. 3.23 apibrėžimą), priešingu atveju $u_{ij}=0$ (briauna neegzistuoja) $(P_i,P_j\in P,\ u_{ij}\in U)$ $(i\neq j)$.

3.25 apibrėžimas. Metaparametrų reikšmių sąveikos erdvė apibrėžiama metaparametrų reikšmių sąveikos *dvidaliu grafu* $H((V_i,V_j),E)$. Grafo viršūnės V_i ir V_j atitinka metaparametrų P_i ir P_j reikšmes $(V_i,V_j\subset V)$. Briaunos $e_{kt}=(v_{i_k},v_{j_t})$ $(v_{i_k}\in V_i; v_{j_t}\in V_j)$ nurodo metaparametrų reikšmių sąveikos tipą. Kai metaparametrus sieja *reikalauja* apribojimas (v_{i_k} *reikalauja* v_{j_t}), tai $e_{kt}=1$, o kai sieja *išskiria* apribojimas (v_{i_k} *išskiria* v_{j_t}), tai $e_{kt}=0$.

Metaparametrų reikšmių sąveikos grafas yra dvidalis grafas. Atsižvelgiant į sąveikos tipus, metaparametrai modeliuojami kaip variantiniai taškai, o jų reikšmės kaip variantai (Jaring, Bosch, 2004).

3.12 savybė. Metaparametrų sąveikos grafas $G(P^w,U)$ yra *tuščiasis* svorinis grafikas (grafas be briaunų), jeigu kiekvienai metaparametrų P_i ir P_j porai $(P_i,P_j\in P^w)$ metaparametrų reikšmių *dvidalis grafas* $H_b((V_i,V_j),E)$ yra *pilnasis* svorinis grafikas (grafas, kurio viršūnių poaibiai tarpusavyje sujungti). Jam galioja (3.16) savybė:

$$\forall_b(H_b((V_i,V_j),E)\ \text{pilnasis}) = \textbf{true}; \tag{3.16}$$

čia $(b\in[1,|B|];|B|=C_n^2)$; B – metaparametrų skirtingų porų skaičius.

3.13 savybė. Metaparametrų sąveikos grafas $G(P^w,U)$ yra *jungusis* grafas (angl. *disconnected graph*) (t. y. savyje turi rinkinį sujungtų *pografių* (angl. *sub-graphs*)), jam galioja savybė (3.17):

$$\forall_b(H_b((V_i,V_j),E)\ \text{nepilnasis}) = \textbf{true}. \tag{3.17}$$

Metaparametrų sąveikos grafas formaliai aprašomas (3.18) išraiška:

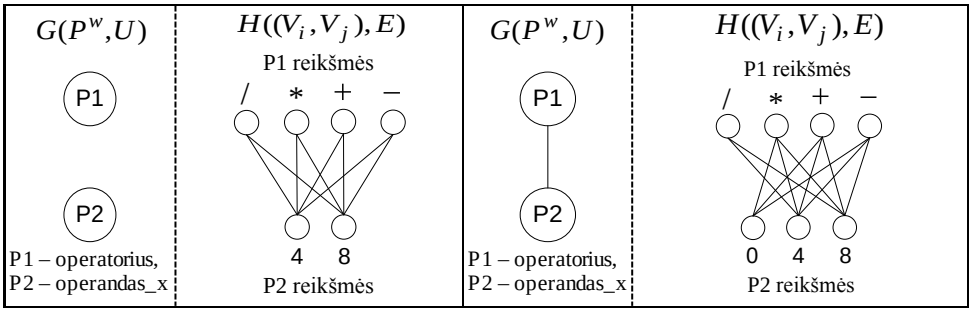
$$G(P^w,U)=\bigcup_{i=1}^{g}G_i^w, G_i^w\cap G_j^w=\varnothing; G_i^w,G_j^w\subseteq G(P^w,U); (i\neq j); \tag{3.18}$$

čia g – sujungtų *pografių* skaičius įtraukiant ir tuščiuosius pografius ($g=|G_i^w|, (g>1)$.

Metaparametrų ir jų reikšmių sąveikos grafus iliustruojantis aiškinamasis pavyzdys pateiktas 3.8 pav. (nagrinėjamas 3.3.1 skyrelyje aprašytas uždavinys). Paveiksle parodyti du atvejai – kai metaparametrai nepriklausomi ir kai priklausomi.

Pirmuoju atveju pateiktas pavyzdys, kuriame metaparametrai yra nepriklausomi, metaparametrų sąveikos grafas $G(P^w,U)$ yra *tuščiasis* grafikas, nes kiekvienai metaparametrų P1 ir P2 porai metaparametrų reikšmių *dvidalis grafas* $H_b((V_i,V_j),E)$ yra *pilnasis* grafikas. Antruoju atveju, metaparametro P2 reikšmių aibė papildyta nulio reikšme. Ši reikšmė įneša ribojimą „*dalyba iš nulio negalima*“. Metaparametrų reikšmių grafas yra nepilnas, vadinasi metaparametrai tarpusavyje priklausomi.

3.14 savybė. Metaparametrų priklausomybės metasąsajoje sukuria programos kodo išsišakojimus (sąlyginius sakinius).



a) b)

3.8 pav. Metaparametrų sąveikos ir metaparametrų reikšmių sąveikos dvidaliais grafais: a) nepriklausomi metaparametrai, b) priklausomi metaparametrai

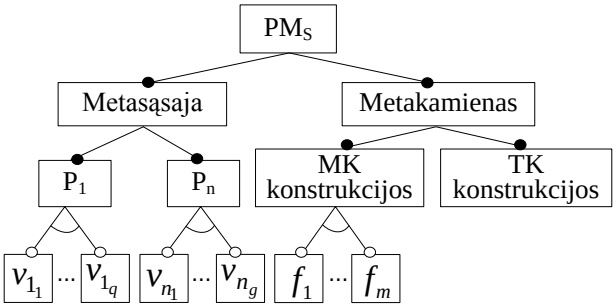
3.25 apibrėžimas. Metakamieno modelį $\mu(M_B)$ sudaro apibrėžta metakalbos funkcijų aibė $f_k(a_j)$, formaliai užrašoma (3.19) išraiška:

$$\mu(M_B)=\{f_k(a_j)\}; \tag{3.19}$$

čia $f_k(a_j)\in L_M$; $a_j\in P$. L_M – metakalba.

Metaprogramos metakamieną sudaro metakalbos funkcijų aibė, kuri įterpiama į tikslo kalba užrašytą programos kodą. Metakamieno specifikacija kuriama rankiniu būdu, čia jau įvedama konkrečios metakalbos ir tikslo kalbos sintaksė. Metakalbos ir tikslo kalbos įvedimo procesas sunkiai automatizuojamas, nes abi kalbos charakterizuojamos dideliu kiekiu specifinių rodiklių rinkinių ir priklauso nuo uždavinio (žr. 2.4 skyrelį). Sukurtą metakamieno specifikaciją darbe vadiname tikslo kalbos bendruoju programos egzemplioriaus modeliu $TKBE_P$ (detaliau šis modelis bus apžvelgtas 5 skyriuje).

3.9 pav. pavaizduotas abstraktus sprendimo srities požymių modelis (metaprogramos požymių diagrama).



*PM*S – sprendimo srities požymių modelis; MK – metakalba; TK – tikslo kalba; P – metaparametrai; ν – metaparametrų reikšmės.

3.9 pav. Sprendimo srities abstraktus požymių modelis

Kaip buvo apibrėžta anksčiau, metaprogramos kūrimas formuluojamas kaip probleminės srities požymių modelio atvaizdavimas į sprendimo srities (metaprogramos) požymių modelį. Kad būtų galima atlikti tokį atvaizdavimą reikia suformuluoti transformavimo taisykles, kurios nustatytų skirtingų modelių elementų tarpusavio atitikimus.

3.5. Požymiais grindžiamų modelių transformavimo taisyklės

Šiame skyrelyje formuluojamos pagrindinės probleminės srities atvaizdavimo į sprendimo sritį taisyklės, t. y. požymių modelių transformavimo taisyklės.

3.1 taisyklė. Probleminės srities požymių modelio PM_P (3.4 pav.) skirtybę vaizduojantis variantinis taškas atitinka metaprogramos metaparametrą.

3.2 taisyklė. Požymių modelio PM_P skirtybes vaizduojančių variantinių taškų variantai atitinka metaprogramos metaparametrų reikšmes.

3.3 taisyklė. Variantinio taško prioritetinė reikšmė atitinka metaparametro prioritetinę reikšmę (žr. 3.10 savybę). Prioriteto požymis vaizduoja metaparametrų galimus svorius, bet ne patį metaparametrą.

3.4 taisyklė. Metasąsajoje metaparametrai rūšiuojami prioritetinių reikšmių mažėjimo tvarka

3.5 taisyklė. Metaprogramos metasąsajoje paprastasis priskyrimas užrašomas:

<metaparametras> = <metaparametro reikšmių aibė>.

3. 6 taisyklė. Metaprogramos metasąsajoje sąlyginis priskyrimas užrašomas:

<metaparametras1> <sąlyga> <metaparametras2> <metaparametras1> = <metaparametro reikšmių aibė>.

3.7 taisyklė. Sąlyginis priskyrimas metasąsajoje kuriamas, kai požymių modelio PM_P variantinis taškas turi apribojimus „*reikalauja*“ arba „*išskiria*“ (žr. 3.9 ir 3.10 apibrėžimus).

3.8 taisyklė. Metakamienas formuojamas naudojant metakalbos funkcijų aibę: *{priskyrimas* („=“), *OPEN-WRITE-CLOSE, šakojimas, kartojimas}*.

Modelius transformuojant į metaprogramą darbe pritaikomas skaičiavimų-valdymo modelis (angl. *computational models*). Kaip skaičiavimų-valdymo medelis

naudojamas baigtinių būsenų automatas (angl. *Finite State Machine*, FSM) (Börger, 1999; Börger, 2010). Žemiau pateikiamas adaptuotas FSM modelis (3.20):

$$\forall_{i,j(i\neq j)} \textbf{(FSM } (i, \textbf{if } cond \textbf{ then } rule, j) = \textbf{if } ctl\_state = i \text{ and } cond \textbf{ then } rule\ ctl\_state := j\textbf{);} \quad (3.20)$$

čia i – esama būsena, j – sekanti būsena, $i, j \in \sum$ (visų galimų deterministinių būsenų rinkinys), *ctl_state* – valdymo būsena, *cond* – sąlygų įtakojančių taisyklių parinkimą rinkinys, *rule* – transformavimo taisyklė.

3.9 taisyklė. FSM variklis sukuria metasąsają pagal 3.1 – 3.7 taisykles.

3.10 taisyklė. Tikslo kalbos bendrasis programos egzempliorius $TKBE_P$ visada kuriamas rankiniu būdu, modelyje aiškiai nurodant metaparametrų vietas (kintančias tikslo kalbos programos vietas).

3.11 taisyklė. Jei tikslo kalbos bendrasis programos egzempliorius $TKBE_P$ egzistuoja, FSM variklis atlieka jo sintaksinę analizę ir sukuria metaprogramos metakamieną.

3.12 taisyklė. Jei tikslo kalbos bendrasis programos egzempliorius $TKBE_P$ neegzistuoja, FSM variklis sukuria metakamieno šabloną.

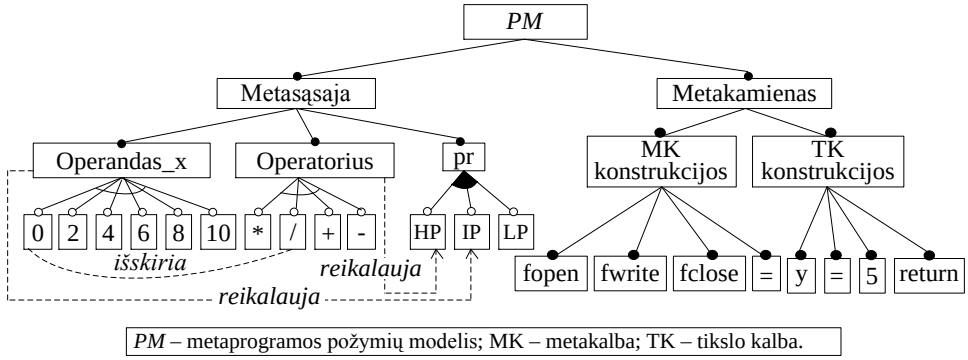
Suformuluotos transformavimo taisyklės įgalina realizuoti probleminės srities požymių modelio atvaizdavimą į sprendimų sritį.

3.6. Probleminės srities požymių modelių atvaizdavimas sprendimo srityje

Tikslas – pademonstruoti probleminės srities išplėstinio požymio modelio atvaizdavimą sprendimų srities požymių modelyje ir pademonstruoti aiškinamajam pavyzdžiui (žr. 3.3.1 skyrelį) sugeneruotą metaprogramą.

Probleminė sritis atvaizduojama sprendimo srityje remiantis anksčiau aprašytais sričių formalizmais (3.3 ir 3.4 skyreliuose) ir apibrėžtomis transformavimo taisyklėmis (3.5 skyrelyje). Sukuriamas konkretus sprendimo srities požymių modelis, kuriame įvedama metakalbos ir tikslo kalbos sintaksė.

Kaip metakalba darbe naudojama PHP programavimo kalba (kaip metakalba gali būti panaudotos ir kitos programavimo kalbos: Java, C++ ir kt. (Štuikys, Damaševičius, 2013a)). Ši kalba yra universali programavimo kalba (angl. *general purpose language*) plačiai taikoma interneto svetainėms kurti. PHP pasirinkimą lėmė tai, kad ji yra nemokama, atvirojo kodo, stabili, greita ir nesudėtinga, taip pat ji leidžia atlikti programinio teksto analizę. Tikslo kalbų pasirinkimas priklausė nuo sprendžiamų uždavinių. Darbe naudojamos RobotC (Burbaitė, Damaševičius ir Štuikys, 2013) ir Arduino (Mellodge, Russell, 2013; Rubio, Hierro ir Pablo, 2013) programavimo kalbos. 3.10 pav. pavaizduotas konkretus sprendimo srities požymių modelis (metaprogramos požymių diagrama). Šiame pavyzdyje kaip tikslo kalba naudojama RobotC programavimo kalba.



3.10 pav. Sprendimo srities konkretus požymių modelis

3.11 pav. pateikiama aiškinamąjį pavyzdį (3.10 pav.) realizuojanti metaprograma.

```
<? //---------------------Metasąsaja------------     --------------
 $Operatorius =$_POST[Operatorius];
 $Operandas_x =$_POST[Operandas_x];
if (!isset($Operatorius) && !isset($Operandas_x)) {
?>
<FORM METHOD = POST ACTION = "">
Select parameter $Operatorius value:
<select name="Operatorius">
<option value="*"> * </option>
<option value="/"> / </option>
<option value="+"> + </option>
<option value="-"> - </option>
</select> <br>
<INPUT TYPE=submit VALUE="Submit" NAME="submit" style="height: 28px">
</FORM>
<? // ----------tolesnis metasąsajos tekstas paslėptas---------------
//                              .....
} if (isset($Operatorius) && isset($Operandas_x)){
//----------  -----------Metakamienas--------------      ------------
$myFile = "result.c";
$fr = fopen($myFile, 'w');
fwrite($fr," int Met(){ \n");
fwrite($fr," int y = 0; \n");
fwrite($fr," y = $Operandas_x $Operatorius 5; \n");
fwrite($fr," return y; \n");
fwrite($fr," } \n");
fclose($fr);
}?>
```

3.11 pav. Metaprogramos programinis tekstas (pilną kodą žr. 1 priede)

3.7. Santrauka ir apibendrinimas

Šiame skyriuje buvo išnagrinėtas heterogeninių metaprogramų automatizuoto kūrimo uždavinys. Automatizuotas intelektualios nuosavybės kūrimas visada yra didžiulis iššūkis, o metaprogramų automatizuotas kūrimas yra dar didesnis, nes metaprogramos yra sudėtingi objektai – programų generatoriai. Todėl uždavinys buvo suformuluotas kaip *dviejų tipų* modelių daugiapakopis transformavimas ir atvaizduotas Y-ko diagrama. *Pirmuoju modeliu* atvaizduojama *probleminė sritis*

kairine Y-ko diagramos šaka, o *antruoju* – sprendimo sritis dešinine šaka. Pasirinkta probleminė sritis buvo mokomųjų robotų valdymas, o sprendimo sritis – metaprogramavimas, nes iškeltas tikslas buvo kurti metaprogramas. Metaprogramos kūrimas formuluojamas kaip probleminės srities požymių modelių *atvaizdavima*s į metaprogramos modelį. Kad būtų galima atlikti tokį atvaizdavimą (jis traktuojamas kaip horizontalus transformavimas; jis apibrėžiamas transformavimo taisyklėmis, kurios nustato skirtingų modelių elementų tarpusavio atitikimus), reikia išpildyti tam tikras sąlygas.

Pirma, probleminis srities modelis turi būti konkretus, jo elementai detalizuoti iki žemiausio lygmens bazinių elementų (požymių modelių variantinių taškų, variantų, apribojimų). Tokiam lygmeniui pasiekti reikia iš pradžių atlikti vertikalų transformavimą žeminant abstrakcijos lygmenį, t. y. einant kairiąja Y-ko šaka žemyn (pereinant nuo abstraktaus prie konkretaus modelio).

Antra, sprendimo sritis turi būti taip pat atvaizduota tos pačios notacijos modeliu, t. y. požymių diagrama. Tačiau prieš horizontalią transformaciją sprendimo srities modelis dar yra abstraktus. Tik atlikus horizontalią transformaciją jis tampa konkrečiu metaprogramos modeliu.

Dar vienas transformavimo lygmuo reikalingas, kad iš metaprogramos modelio būtų gauta jos vykdomoji specifikacija. Šiame lygmenyje įvedama konkrečios metakalbos ir tikslo kalbos sintaksė.

Kad būtų galima realizuoti modeliais grindžiamus transformavimus, reikėjo formalizuoti tiek probleminę, tiek sprendimo sritį. Formalizavimas apėmė ne tik atitinkamų sąvokų apibrėžimus, bet ir modelių savybių (atitinkamų sąryšių) nustatymą. Esminėmis savybėmis reikia laikyti tokias: (1) konteksto modelio sąryšiai su metaprogramos metaparametrų prioritetu; (2) požymių apribojimo sąryšiai su parametrų tarpusavio sąveika; (3) variantinių taškų ir jų variantų atitikimo sąryšis su metaprogramos parametrais ir jų reikšmėmis ir kt. Sąryšių nustatymas įgalino suformuluoti transformavimo taisykles ir sukūrė realias prielaidas algoritmizuoti uždavinį ir galiausiai sukurti įrankį automatizuotai projektuoti probleminės srities metaprogramas.

Daugiapakopis transformavimas (iš pradžių *modelis-modelis*, o toliau *modelis-metaprograma)* ir sudaro *suformuluoto uždavinio sprendimo esmę*. Uždavinio sprendimo metodika pailiustruota aiškinamaisiais pavyzdžiais.

Spendžiant ši uždavinį, nustatyti sunkumai ir apribojimai. Vienas iš paminėtinų sunkumų yra tikslo kalbos sintaksės įvedimas. Jis atliekamas pasirenkant tikslo kalbos scenarijus. Šis įvedimo procesas sunkiai automatizuojamas, nes kalba abstrakti. Nors įrankis „FAMILIAR“ pateikia dvejopą modelio atvaizdavimą (XML tekstu ir grafiškai), tačiau nustatyti tokie apribojimai: (1) XML notacija supaprastinta, pavadinta SXML (traktuotina kaip small XML); (2) apribota kai kurių simbolių vartosena įvardijant variantinius taškus (pvz., =, –, 0 ir kt.); (3) vienodų variantinių taškų ribojimas ir kt. Šie ribojimai mažina srities modeliavimo galimybes, todėl reikalingas nuodugnesnis kitų modeliavimo įrankių pažinimas ar pasirinkimas.

3.8. Išvados

1. Suformuluotas heterogeninių metaprogramų kūrimo uždavinys kaip dviejų tipų modelių daugiapakopis transformavimas, kai pirmasis modelis atstovauja probleminę sritį, o antrasis – sprendimo sritį (metaprogramavimą).

2. Kiekvienai sričiai sukurti jų bendrybes ir skirtybes aprašantys požymių modeliai, kurie detalizuoti iki bazinių elementų lygmens, kad juos būtų galima formalizuoti, nustatyti jų sąryšius, savybes ir sukurti transformavimo taisykles.

3. Srities modelių, metaprogramų notacijų ir požymiais grindžiamų modelių transformavimo taisyklių apibrėžimas sudarė sąlygas automatizuotam įrankiui sukurti, kuris bus nagrinėjamas baigiamuose disertacijos skyriuose.

4. Pasiūlytas metodas, kai jis realizuojamas įrankiu, neįneša papildomų apribojimų galimai parametrų ir jų reikšmių erdvei, todėl pačios metaprogramos gali būti transformuojamos, jas specializuojant ir adaptuojant konkrečiam kontekstui.

4. METAPROGRAMŲ SPECIALIZAVIMAS IR KONTEKSTINIS ADAPTAVIMAS

4.1. Įvadas

Kaip buvo konstatuota 3-ojo skyriaus baigiamojoje išvadoje, automatizuotas metaprogramų kūrimas įgalina sukurti metaprogramas, kurios gali turėti pakankamai didelį metaparametrų ir jų reikšmių skaičių, t. y. galime kurti bendrąsias metaprogramas plačiam pritaikymui toje pačioje srityje. Sukurta metaprograma yra vienpakopė, t. y. metakalbos procesoriaus interpretuojama ir įvykdoma nepertraukiamame cikle (kai visų parametrų reikšmės yra nustatytos). Kadangi sukurta metaprograma bendrinė, t. y. realizuoja projektavimo principą D*f*R (žr. 2.1 pav. skaityti „*desing-for-reuse*“), atsiranda poreikis tokias metaprogramas pritaikyti konkrečiam kontekstui (t. y. realizuoti principą D*w*R (skaityti „*design-with-reuse*“)

Todėl šio skyriaus tikslas – ištirti vienpakopės metaprogramos transformavimą į kitą, tinkamesnį (t. y. daugiapakopį) formatą, kuris leistų pritaikyti bendrines metaprogramas prie konkretaus konteksto. Tikslui pasiekti nagrinėjamas vienpakopės metaprogramos transformavimo į daugiapakopę uždavinys. Čia transformavimas traktuojamas kaip metaprogramos *specializavimas* ir *adaptavimas* pritaikant programų specializavimo idėją (žr. 2.3.2 skyrelį). Dėl transformavimo sudėtingumo uždavinys nagrinėjamas dviem etapais (specializavimo ir adaptavimo).

Literatūroje naudojamos trys sąvokos susijusios su nagrinėjamu transformavimu. Informatikoje naudojama – dalinis įvertinimas (angl. *partial evaluation*) arba specializavimas, o programų inžinerijoje – restruktūrizavimas (angl. *refactoring*). Darbe naudojami du terminai: specializavimas ir restruktūrizavimas. Pirmasis vartojamas, kai turima omeny loginė transformacija, o antrasis – kai kalbama apie fizinį kodo transformavimą įrankyje.

4.2. skyrelyje aprašytas specializavimo uždavinys; 4.3. skyrelyje pateikta daugiapakopės metaprogramos notacija; 4.4. skyrelyje apibrėžtos metaprogramų transformavimo į daugiapakopes taisyklės; 4.5. skyrelyje aprašytas metaprogramos programinio kodo restruktūrizavimo procesas, 4.6. skyrelyje aprašytas adaptavimo uždavinys; 4.7. skyrelyje pateikiamas skyriaus apibendrinimas ir 4.8. skyrelyje suformuluotos skyriaus išvados.

4.2. Specializavimo uždavinio formulavimas

Metaprogramos specializavimui aprašyti pritaikytas Futamura (1999) pasiūlytas programų specializavimo metodas. Todėl jį panagrinėsime plačiau. Jo pasiūlyta formali programos specializavimo išraiška (4.1) užrašoma:

$$\pi(c_1', c_2', ..., c_m', r_1', r_2', ..., r_n') = \alpha(\pi, c_1', c_2', ..., c_m')(r_1', r_2', ..., r_n') \qquad (4.1)$$

Kairėje lygties pusėje pateikiama programos būsena prieš specializavimą. Čia kintamųjų $(c_1, c_2, ..., c_m, r_1, r_2, ..., r_n)$ reikšmės $(c_1', c_2', ..., c_m', r_1', r_2', ..., r_n')$ padalytos į dvi grupes: konstantas, įvertinamas kompiliavimo metu ir kintamuosius, kurie įvertinami programos vykdymo metu. Dešinėje lygties pusėje pateikiama programos

būsena po specializavimo, naudojant „specializavimo algoritmą“ α. Programos kompiliavimo metu įvertinamos $c_1', c_2', ..., c_m'$ reikšmės, o programos vykdymo metu įvertinamos $r_1', r_2', ..., r_n'$ reikšmės. Aukštesnio lygmens programa, kuri įgyvendina „specializavimo algoritmą“ vadinama *specializatoriumi*.

Pavyzdys (Futamura, 1999).

Jei turime funkciją: $f(x, y) = x \times (x \times x + x + y + 1) + y \times y$, kur $x = 1, y = 1, 2, \ldots, n$. Įvertinant $f(1,y)$ visoms y reikšmėms, galima užrašyti:

x:=1; **for** y := 1 *žingsnis* 1 **until** n **do** $f[x, y] := x \times (x \times x + x + y + 1) + y \times y$.

Vykdymo metu, apskaičiuojant kiekvieną y reikšmę, atliekami 3 daugybos ir 4 sudėties veiksmai. Specializuotą programos versiją $\alpha(f, 1)(y) = 1 \times (3 + y) + y \times y$, galima užrašyti taip:

for y := 1 *žingsnis* 1 **until** n **do** $f[1, y] := 1 \times (3 + y) + y \times y$.

Specializuotos programos vykdymo metu, apskaičiuojant kiekvieną y reikšmę, atliekami tik 2 daugybos ir 2 sudėties veiksmai. Specializavimas įgalina vykdymo metu atlikti mažiau skaičiavimų, o tai pagreitina pačios programos darbą.

Iš šio pavyzdžio, žinant metaprogramos struktūrą, galima nustatyti tokius atitikimus (panašumus arba analogiją) tarp programos specializavimo ir dviejų pakopų metaprogramos (Štuikys, Damaševičius, 2013a): 1) programos specializavimas turi du etapus (kompiliacijos ir vykdymo), o metaprogramoje yra dvi pakopos; 2) programos objektas – kintamieji, o metaprogramoje – metaparametrai (atitiktis akivaizdi); 3) skirtingos kintamųjų grupės įvertinamos skirtinguose etapuose (darome prielaidą, kad tai turi galioti ir metaparametrams). Todėl iš pradžių formuluojame metaprogramos dviejų pakopų specializavimo uždavinį: metaparametrų aibė (žr. 3.21 apibrėžimą) $P = \{(p_1, ..., p_n)\}$ padalinama į dvi grupes $P = \{(p_1, ..., p_m), (p_{m+1}, ..., p_n)\}$. Skirstant metaparametrus į grupes, priklausomi metaparametrai (žr. 3.11 apibrėžimą) turi būti priskirti tai pačiai metaparametrų grupei. Metaprogramos specializavimą formaliai užrašome (4.2) išraiška, kuri adaptuota pagal (4.1):

$$\pi(p_1, ..., p_m, p_{m+1}, ..., p_n) = \alpha(\pi, p_1, ..., p_m)(p_{m+1}, ..., p_n). \qquad (4.2)$$

Dviejų pakopų metaprogramos vykdymo metu, metaparametrai $p_1, ..., p_m$ yra aktyvūs. Jie šioje pakopoje įvertinami ir sugeneruotoje vienos pakopos metaprogramoje traktuojami kaip konstantos. Likusieji metaparametrai $p_{m+1}, ..., p_n$ šioje pakopoje yra pasyvūs, jie bus įvertinti vienos pakopos metaprogramos vykdymo metu. *Specializatorius* (α) turi iš anksto užprogramuoti programos pokyčius taip, kad dviejų pakopų metaprogramos vykdymo metu aktyvūs metaparametrai $p_1, ..., p_m$ būtų pakeičiami konstantomis (pasirinktomis metaparametrų reikšmėmis), o pasyvūs metaparametrai $p_{m+1}, ..., p_n$ – aktyvuojami. Vėliau, vienos pakopos metaprogramos vykdymo metu, sugeneruojamas tikslo

kalbos programos egzempliorius, kurio programiniame tekste visi metaparametrai yra traktuojami kaip konstantos.

Toliau išraišką (4.2) apibendriname bet kuriam pakopų skaičiui. Kuriant daugiapakopę metaprogramą, metaparametrai padalijami į tiek grupių, kiek pakopų turės kuriama metaprograma. Daugiapakopis specializavimas užrašomas (4.3) išraiška:

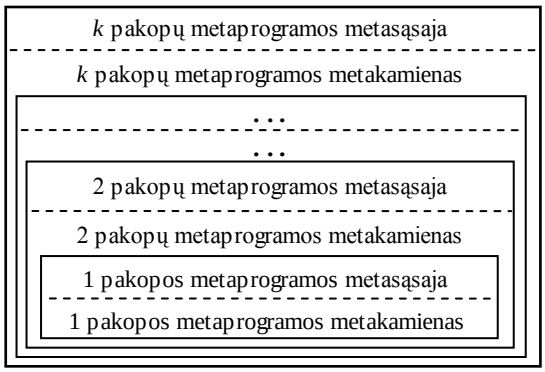
$$\pi(p_1,..., p_m, p_{m+1},..., p_n) = \alpha(\pi, p_1,..., p_m)(p_{m+1},..., p_n)$$
$$\alpha(\pi, p_{m+1},.., p_i)(p_{i+1},.., p_n)...$$
$$\alpha(\pi, p_{i+1},...,p_j)(p_{j+1}...,p_n)... \qquad (4.3)$$

Daugiapakopės metaprogramos vykdymo metu generuojama žemesnės pakopos metaprograma, kuri bus vykdoma vėliau. Aukštesnės pakopos metaprogramos vykdymo metu aktyvūs metaparametrai keičiami konstantomis, o pasyvių metaparametrų mažinamas deaktyvacijos indeksas.

Specializuotos metaprogramos kūrimas formuluojamas kaip vienpakopės metaprogramos transformavimas į daugiapakopę. Kad būtų galima atlikti tokią transformaciją, reikia formaliai apibrėžti daugiapakopę metaprogramą, nustatyti metaparametrų paskirstymo pakopose taisykles bei apibrėžti metakonstrukcijų deaktyvacijos ir aktyvacijos mechanizmą.

4.3. Daugiapakopės metaprogramos formalizavimas

Vienos pakopos metaprograma yra tikslo kalbos programų generatorius, o dviejų pakopų metaprograma yra metaprogramų generatorius. Daugiapakopė metaprograma yra meta-metaprogramų generatorius. Struktūrinis daugiapakopės metaprogramos modelis (4.1 pav.) gaunamas iš vienpakopės metaprogramos modelio (3.7 pav.) pertavarkant šį modelį (Štuikys, Bespalova, 2012). Vienpakopė metaprograma suskaidoma į k lygmenis (k – daugiapakopės metaprogramos pakopų skaičius).



4.1 pav. Daugiapakopės metaprogramos struktūrinis modelis

4.1 apibrėžimas. Daugiapakopės metaprogramos struktūrinis modelis μ^k yra kompozicija metasąsajos modelio $\mu(M_I^k)$ ir metakamieno modelio $\mu(M_B^k)$, kuris savyje turi žemesnio lygio metasąsajos ir metakamieno modelius (žemesnio lygmens

metasąsaja yra sudedamoji daugiapakopės metaprogramos metakamieno dalis). Daugiapakopės metaprogramos struktūrinis modelis užrašomas (4.4) išraiška, o daugiapakopės metaprogramos metakamieno modelis – (4.5) išraiška:

$$\mu^k = \mu(M_I^k) \cup \mu(M_B^k). \quad (4.4)$$

$$\mu(M_B^k) = \mu(M_I^{k-1}) \cup (... \cup (\mu(M_I^1 \cup \mu(M_B^1)))). \quad (4.5)$$

Atlikus vienpakopės metaprogramos transformavimą į daugiapakopę metakonstrukcijos būna dviejų tipų: aktyvios ir pasyvios.

4.2 apibrėžimas. Metakonstrukcija – tai metaparametras arba metakalbos funkcija.

Programavimo kalbos (pvz., C++, PHP) turi nesudėtingą mechanizmą leidžiantį pakeisti metakonstrukcijos būseną iš aktyvios į pasyvią.

4.3 apibrėžimas. Simbolis „\“ arba jų seka \\\... vadinama deaktyvacijos simboliu. Deaktyvuojant metakonstrukciją deaktyvacijos simbolis rašomas prieš ją.

4.4 apibrėžimas. Metakonstrukcija vadinama *aktyvia* (angl. *active*) duotoje pakopoje, jei ji neturi deaktyvacijos simbolio, t. y. atlieka savo apibrėžtą funkciją.

4.5 apibrėžimas. Metakonstrukcija vadinama *pasyvia* (angl. *passive*) duotoje pakopoje, jei ji turi deaktyvacijos simbolį, t. y. neatlieka savo apibrėžtos funkcijos ir traktuojama kaip tikslo kalbos tekstas.

Metakonstrukcijų deaktyvacija leidžia mataprogramą transformuoti į daugiapakopę metaprogramą.

4.6 apibrėžimas. Simbolių „\“ skaičius deaktyvacijos simbolių sekoje vadinamas deaktyvacijos indeksu.

Deaktyvacijos simbolių kiekis priklauso nuo metakalbos ir nuo pakopos, kurioje metakonstrukcija turi būti panaudota, t. y. kurioje daugiapakopės metaprogramos pakopoje metakonstrukcija turi būti aktyvi. Deaktyvacijos indeksas skaičiuojamas pagal (4.6) formulę:

Pakopoje k $DI = 0$, pakopoje (k-1) $DI = 1$, žemesnėse pakopose $DI = \sum_{a=0}^{k-2} 2^a$ (4.6)

čia DI – deaktyvacijos indeksas.

Deaktyvacijos simbolių naudojimo aiškinamieji pavyzdžiai pateikiami 4.1 lent. Lentelėje pateikiami programinio teksto fragmentai, kuriuose kaip metakalba naudojama PHP programavimo kalba (tas pats deaktyvacijos mechanizmas tinka Java ir C++ programavimo kalboms).

4.1 lentelė. Deaktyvacijos simbolių naudojimo pavyzdžiai

Kurioje pakopoje metaparametras naudojamas	Pavyzdžiai PHP programavimo kalba	Deaktyvacijos indeksas
4 pakopoje	`$P1 = "x";`	0
3 pakopoje	`echo "\$P1 = x;";`	1
2 pakopoje	`echo "echo \"\\\$P1 = x;\";";`	3
1 pakopoje	`echo "echo\"echo\\\"\\\\\\\$P1 = x;\\\";\";";`	7

P1 – metaparametras.

4.7 apibrėžimas. Deaktyvacija vadinamas procesas, kurio metu deaktyvuojamos metakonstrukcijos. Metakonstrukcijos deaktyvacija priklauso nuo jos panaudojimo daugiapakopėje metaprogramoje, t. y. kurioje pakopoje konstrukcija bus aktyvi.

4.8 apibrėžimas. Aktyvacija vadinamas procesas, kurio metu sumažinamas deaktyvacijos indeksas per 1 pakopą arba keičiama metakonstrukcijos būsena iš pasyvios į aktyvią.

Metakonstrukcijų aktyvacija yra pakopinis procesas. Metaprogramos vykdymo metu, aktyvacijos procesą atlieka procesorius. Metakonstrukcijų deaktyvacijos ir aktyvacijos procesai neturi įtakos metaprogramos semantikai, šie procesai tik keičia metakonstrukcijų būsenas.

4.9 apibrėžimas. Vienpakopės metaprogramos specializavimas yra atvirkštinė transformacija keičianti metaprogramos struktūrą į daugiapakopę ($M \xrightarrow{T} M^k$) deaktyvuojant metakonstrukcijas pakopose (nuo k-1 iki 1).

4.10 apibrėžimas. Maksimalaus pakopų skaičiaus nustatymo sąlyga (naujumas). Didžiausias galimas pakopų skaičius $k_{\max}$, metaprogramą transformuojant į daugiapakopę randamas pagal (4.7) išraišką:

$$k_{\max} \le g. \tag{4.7}$$

čia g – jungiųjų pografių skaičius įskaitant ir tuščiuosius pografius ($g = \left|G_i^w\right|$, ($g > 1$) (žr. 3.8 pav.).

Specializuojant metaprogramas, svarbų vaidmenį vaidina metaparametrų prioritetiniai svoriai (žr. 3.3 taisyklę). Metaprogramos modelyje (žr. 3.10 pav.) prioritetiniai svoriai leidžia valdyti metaparametrų paskirstymą pakopose metaprogramos specializavimo metu. Kaip buvo minėta anksčiau (žr. 3.3 skyrelį), metaparametrui suteikiamas prioritetinis svoris priklauso nuo srities eksperto žinių ir uždaviniui keliamų reikalavimų. Aukščiausias svoris suteikiamas metaparametrams, kurie daugiapakopės metaprogramos vykdymo metu turi būti įvertinti pirmieji. Pavyzdžiui, nagrinėjant metaprogramas skirtas informatikos (programavimo) mokymuisi, pedagoginiai metaparametrai turi turėti aukščiausią prioritetą, socialiniai – vidutinį prioritetą, turinio ir technologiniai – žemiausią prioritetus (Burbaitė, 2014; Štuikys, 2015).

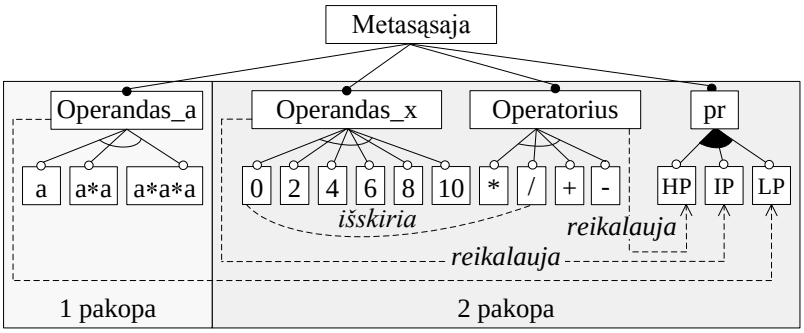
4.11 apibrėžimas. Kontekstinis specializavimas – tai kontekstinės informacijos valdomas procesas, apibrėžiantis metaprogramos galimą pakopų skaičių ir metaparametrų paskirstymą jose.

4.1 savybė. Paskirstant metaparametrus į pakopas, *priklausomų metaparametrų grupė* (metaparametrų sąveikos grafe jungusis pografis) turi būti priskirta tai pačiai pakopai.

4.2 savybė. Kiekvienoje daugiapakopės metaprogramos pakopoje turi būti bent vienas metaparametras arba priklausomų metaparametrų grupė.

4.3 savybė. Skirstant metaparametrus į pakopas aukštesnį prioritetą turintys metaparametrai priskiriami aukštesnei pakopai, o žemesnį – žemesnei.

Toliau bus nagrinėjamas 3.3.1 skyrelyje aprašytas aiškinamasis pavyzdys. Jei norime sukurti dviejų pakopų metaprogramą, metaparametrų sąveikos grafe turime turėti bent du jungiuosius pografius (4.10 apibrėžimas). Nagrinėjamos vienpakopės metaprogramos negalime transformuoti į dviejų pakopų, nes uždavinio metaparametrų sąveikos grafe yra tik vienas jungusis grafas (du priklausomi metaparametrai). Todėl uždavinį modifikuojame papildydami nauju metaparametru. O*perandas_a* laikomas skirtybe, t. y. metaparametru, turinčiu reikšmes $\{a, a*, a**a\}$. 4.2 pav. parodytas nagrinėjamo uždavinio metaparametrų paskirstymas pakopose.



4.2 pav. Metaparametrų paskirstymas į pakopas

Aukščiausią prioritetinį svorį turintys metaparametrai priskiriami antrai pakopai, o žemiausią – pirmai. *Operandas_x* ir *Operatorius* yra priklausomi, jie priskiriami tai pačiai pakopai. Todėl projektuojant metaprogramos požymių modelį, tikslinga visiems priklausomų metaparametrų grupės nariams suteikti vienodą (grupėje didžiausią) prioritetinį svorį.

Skirtingų transformacijų iš vienpakopės į dviejų pakopų metaprogramą skaičius randamas pagal (4.8) formulę, o į trijų pakopų – pagal (4.9) formulę. Šis skaičius parodo, kiek skirtingų tam tikros pakopos daugiapakopių metaprogramų gali būti sukurta iš vienos vienpakopės.

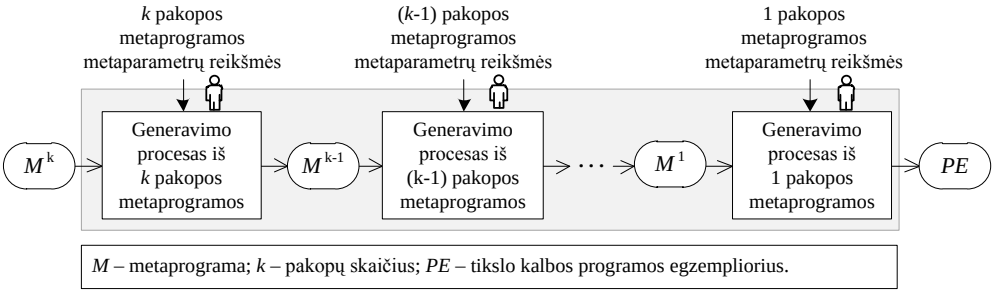
$$|T| = 2^g - 2, \text{ kai } k = 2. \tag{4.8}$$

$$|T| = 3^g - \sum_{i=1}^{g} 3 * 2^{(i-1)}, \text{ kai } k = 3. \tag{4.9}$$

4.12 apibrėžimas. **Specializavimo uždavinio išsprendžiamumo sąlyga (naujumas)**. Vienpakopės metaprogramos transformavimas į daugiapakopę $(M \xrightarrow{T} M^k)$ galimas tada ir tik tada, kai vienpakopės metaprogramos metaparametrų sąveikos grafas $G(P^w, U)$ yra nejungusis grafas, t. y. egzistuoja atskiros jo komponentės (turi būti tenkinama 4.2 savybė).

4.13 apibrėžimas. Daugiapakopis generavimas yra automatizuotas žemesnės pakopos metaprogramų (nuo k pakopos metaprogramos iki 1 pakopos metaprogramos) generavimo procesas, kiekvienoje pakopoje parenkant aktyvių metaparametrų reikšmes (4.3 pav).

4.4 savybė. Daugiapakopis generavimas įgalina pagrįstai adaptuoti metaprogramas prie skirtingų vartotojų poreikių.



4.3 pav. Daugiapakopis generavimo procesas

Vienpakopės metaprogramos transformavimas į daugiapakopę atliekamas vadovaujantis transformavimo taisyklėmis. Toliau formuluojamos specializavimo uždavinio transformavimo taisyklės.

4.4. Metaprogramų transformavimo į daugiapakopes taisyklės

4.1 taisyklė. Informacija apie metaparametrus ir kontekstinė informacija gaunama iš sprendimo srities konkretaus požymių modelio (žr. 2.6 skyrelį).

4.2 taisyklė. Metaparametrų ir kontekstinės informacijos duomenų failas turi būti sukurtas atitinkamos struktūros (bus aprašyta vėliau).

4.3 taisyklė. Tikrinama 4.7 išraiškoje apibrėžta sąlyga. Jei $k \le g$ transformacija galima, priešingu atveju transformacija neįmanoma.

4.4 taisyklė. Priklausomi metaparametrai visada turi būti priskirti tai pačiai pakopai (4.1 savybė).

4.5 taisyklė. Skirstant metaparametrus į pakopas, kiekvienoje pakopoje turi būti priskirtas bent vienas metaparametras arba priklausomų metaparametrų grupė (4.2 savybė).

4.6 taisyklė. Metaparametrai turintys aukštą prioritetinį svorį (HP) turi būti priskirti aukščiausioje pakopoje (4.3 savybė).

4.7 taisyklė. Metaparametrai turintys vidutinį (IP) arba žemą (LP) prioritetinį svorį turi būti priskirti žemesnėse pakopose.

4.8 taisyklė. Pakopų skaičius ir metaparametrų (metaparametrų grupių) paskirstymas pakopose atliekamas automatiškai, atsižvelgiant į kontekstinę informaciją (t. y. pagal metaparametrų prioritetinius svorius). Jei kontekstinės informacijos nepakanka arba ji klaidinga, turi būti pataisytas sprendimo srities konkretus požymių modelis.

4.9 taisyklė. Pakopų skaičiaus parinkimas ir metaparametrų (metaparametrų grupių) paskirstymas jose gali būti atliekamas paties vartotojo.

4.10 taisyklė. 4.8 ir 4.9 taisyklės yra tarpusavio išimtys.

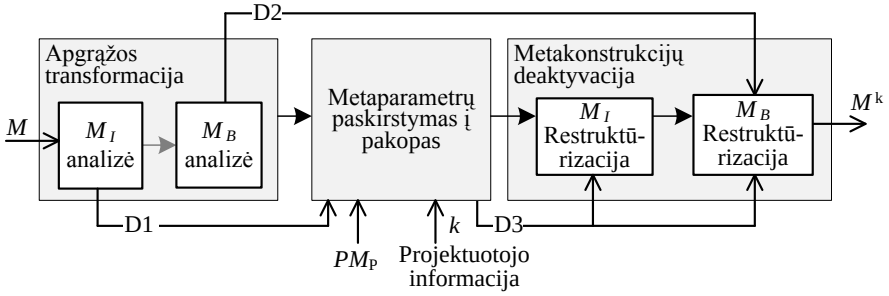
4.11 taisyklė. Metaparametrų deaktyvacija (4.7 apibrėžimas) atliekama vadovaujantis metaparametrų (metaparametrų grupių) paskirstymu pakopose.

4.12 taisyklė. Kiekvienas deaktyvuotas metaparametras reikalauja adekvačios metakalbos funkcijos, kurioje metaparametras buvo panaudotas, deaktyvacijos.

4.5. Metaprogramos programinio kodo restruktūrizavimas

Vienpakopę metaprogramą transformuojant į daugiapakopę metaprogramos programinis kodas restruktūrizuojamas. Restruktūrizavimas – tai transformacija, kuri keičia metaprogramos struktūrą išsaugant pradinį metaprogramos funkcionalumą (žr. 2.3.2 skyrelį). Terminas restruktūrizavimas dažniausiai naudojamas programų inžinerijoje, kai kalbama apie programinio kodo keitimą. Todėl darbe kalbant apie metaprogramos kodo pertvarkymo procesą šis terminas taip pat bus naudojamas.

Metaprograma – taip pat programa, todėl į metaprogramos restruktūrizavimą galima žiūrėti kaip į paprastos programos pertvarkymo procesą. Metaprogramos restruktūrizavimas susideda iš trijų procesų: apgrąžos transformacijos, metaparametrų paskirstymo į pakopas ir metakonstrukcijų deaktyvavimo (4.4 pav.).



M – metaprograma, M_I – metasąsaja, M_B – metakamienas, M^k – daugiapakopė (*k* pakopų) metaprograma, ⟶ – duomenys, ⟶ – valdymo seka, PM_P – probleminės srities požymių modelis, D1– metaparametrų informacija, D2 – metaparametrų ir metafunkcijų priklausomybių informacija, D3 – metaparametrų paskirstymo *k* pakopose modelis.

4.4 pav. Metaprogramos programinio kodo restruktūrizavimo procesas

Apgrąžos transformacijos metu išgaunama informacija apie metaprogramos metaparametrus ir metakalbos funkcijas. Metaparametrų sąveikos ir kontekstinė informacija išgaunama iš probleminės srities požymių modelio PM_P (žr. 3.3 skyrelį). Išgauta informacija toliau naudojama paskirstant metaparametrus į pakopas. Metaparametrai paskirstomi laikantis metaparametrų paskirstymo tarp pakopų reikalavimų (4.1, 4.2 ir 4.3 savybės), tikrinamas paskirstymo korektiškumas. Atsižvelgiant į metaparametrų paskirstymo pakopose informaciją atliekama metakonstrukcijų deaktyvacija. Pirmiausiai deaktyvuojamos metasąsajos, po to – metakamieno metakonstrukcijos. Galiausiai atliekamas restruktūrizavimo įvertinimas. Vertinamos daugiapakopės metaprogramos charakteristikos: sudėtingumas, suprantamumas, proceso nauda.

4.5 pav. parodytas specializuotos metaprogramos pavyzdys, kai sukuriama dviejų pakopų metaprograma. Antroje pakopoje aktyvūs metaparametrai *Operandas_x* ir *Operatorius*, tuo tarpu metaparametras *Operandas_a* – pasyvus.

```
//--------------kodas paslėptas----------------------
?><FORM METHOD = POST ACTION = "">
Select parameter $Operatorius value:
<select name="Operatorius">
<option value="*"> * </option>
<option value="/"> / </option>
<option value="+"> + </option>
<option value="-"> - </option>
</select> <br>
//--------------kodas paslėptas----------------------
fwrite($f1g,"<FORM METHOD = POST ACTION = \"\">\n");
fwrite($f1g,"Select parameter $Operandas_a value:\n");
fwrite($f1g,"<select name=\"Operandas_a\">\n");
fwrite($f1g,"<option value=\"10\"> 10 </option>\n");
fwrite($f1g,"<option value=\"20\"> 20 </option>\n");
fwrite($f1g,"<option value=\"30\"> 30 </option>\n");
fwrite($f1g,"<option value=\"40\"> 40 </option>\n");
fwrite($f1g,"</select>\n");
//--------------kodas paslėptas----------------------
fwrite($f1g," fwrite(\$fr,\" int Met() \\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" { \\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" int y = 0; \\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" y = $Operandas_x $Operatorius \$Operandas_a;\\n\");\n");
fwrite($f1g," fwrite(\$fr,\" return y; \\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" } \\n\"); \n");
//--------------kodas paslėptas----------------------
```

4.5 pav. Specializuotos metaprogramos (dviejų pakopų) programinis tekstas (pilną kodą žr. 2 priede)

Vienpakopių metaprogramų transformavimo į daugiapakopes tikslas – jų adaptavimas konkrečiam kontekstui.

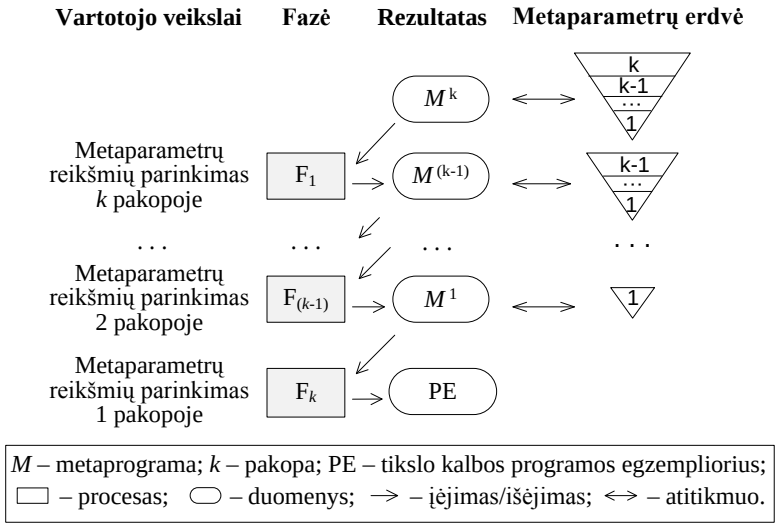
4.6. Adaptavimo uždavinys

Pradiniai duomenys šiam uždaviniui yra specializuota daugiapakopė metaprograma. Daugiapakopės metaprogramos pakopų skaičių nulemia uždaviniui keliami reikalavimai ir tikslai. Daroma prielaida, kad aukščiausios pakopos metaparametrai ir jų reikšmės aprašo skirtingų vartotojų grupių reikalavimus. Todėl vartotojai gali susikurti metaprogramą, adaptuotą jų kontekstui.

Adaptavimas – tai žemesnio lygmens metaprogramos generavimo iš aukštesnio lygmens daugiapakopės metaprogramos procesas, kai aukštesnės pakopos metaparametrus nustato pats vartotojas. Vartotojui parinkus aukštesnės pakopos metaparametrų reikšmes, metakalbos procesorius žemesnės pakopos metaprogramą generuoja automatiškai. Adaptuota žemesnės pakopos metaprograma nuo pirminės skiriasi tuo, kad joje susiaurinta metaparametrų erdvė, t. y. įvertinti aukštesnio lygmens metaparametrai ir juos įvertinus pašalintos nenaudojamos programinio kodo atšakos.

Į adaptavimo procesą galime žvelgti kaip į daugiapakopį generavimo procesą (žr. 4.13 apibrėžimą). Čia kiekvienoje pakopoje sukuriama adaptuota metaprograma, kuri gali būti naudojama kaip savarankiška programa, t. y. ji yra vykdomoji specifikacija, įgalinanti automatizuotai kurti žemesnės pakopos metaprogramas (4.6 pav.). Todėl su daugiapakope metaprograma kiekviename generavimo lygmenyje

gali dirbti skirtingos vartotojų grupės. Taip metaprograma pritaikoma kiekvienos vartotojų grupės poreikiams.

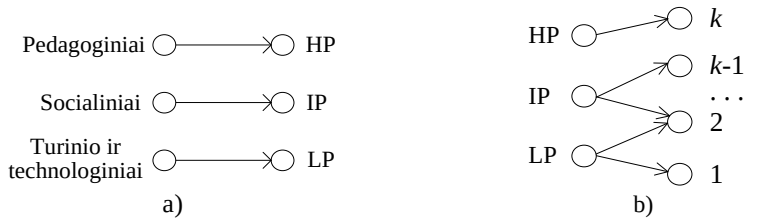


4.6 pav. Pakopinis adaptavimo procesas (Štuikys, 2015)

Tokiu būdu adaptuota metaprograma pasižymi mažesniu sudėtingumu, ji tenkina konkrečių vartotojų poreikius, neturi nenaudojamų metaparametrų ir perteklinio programinio kodo.

Sprendžiant adaptavimo uždavinį buvo nagrinėjamos metaprogramos, kurios įgyvendina CS (angl. *Computing Science*) mokymosi kintamumą automatiškai generuojant ir pritaikant mokymosi turinį pagal iš anksto užprogramuotą kontekstą ir vartotojo poreikius. Čia metaprogramą galima laikyti apibendrintu, parametrizuotu mokymosi objektu, pasižyminčiu pertekliniu funkcionalumu. Metaprograma, pritaikyta mokymo turiniui kurti, išplečia mokymosi variantiškumą, praplečia pakartotinio panaudojimo dimensiją e. mokymesi (Burbaitė ir kt., 2014; Štuikys, 2015). Kuriant mokymui orientuotas metaprogramas įvertinami ir integruojami pedagoginiai, socialiniai ir technologiniai mokymosi aspektai, t. y. galima įvertinti tokias besimokančiojo charakteristikas kaip mokymosi lygis, stilius, amžius ir kt. Pedagoginiai metaparametrai turi turėti aukščiausią prioritetą (HP), socialiniai metaparametrai – vidutinį prioritetą (IP), turinio ir technologiniai metaparametrai – žemiausią prioritetą (LP). Vidutinį prioritetą turinti metaparametrų grupė gali būti išplėsta įvedant Bloomo taksonomijos lygmenis (Airasian ir kt., 2001). Plačiau apie tai galima skaityti Štuikio (2015) monografijoje.

Mokymo srityje aukščiausios pakopos metaprogramos yra skirtos adaptacijai prie mokytojo konteksto. Mokytojas, parinkdamas pedagoginių metaparametrų reikšmes, pritaiko metaprogramą prie savo konteksto, o vėliau gali leisti besimokantiesiems prisitaikyti mokymosi kontekstą pagal jų poreikius (Štuikys, Bespalova, Burbaitė, 2014b). Todėl skirstant metaparametrus į pakopas aukštesnį prioritetą turintys metaparametrai priskiriami aukštesnei pakopai, o žemesnį – žemesnei (4.7 pav.).



HP – žukščiausias prioritetas, IP – vidutinis prioritetas, LP – žemiausias prioritetas, k – aukščiausia pakopa, 1 – žemiausia pakopa, $\rightarrow$ – parametro priskyrimas pakopai.

4.7 pav. Metaparametrų konteksto modelis: a) prioriteto reikšmės priskyrimas ir b) metaparametrų paskirstymas pakopose metaprogramos specializavimo metu (adaptuota remiantis Štuikys, 2015)

Specializuojant metaprogramas sukuriamas įrankis, padedantis mokytojui individualizuoti mokinio darbą parenkant tinkamą ugdymo turinį ir metodus, o mokiniui – prisitaikyti užduotis pagal savo mokymosi galimybes ir poreikius. Tokiu būdu užtikrinamas adaptyvus mokymosi procesas (angl. *Adaptive Learning Process*).

4.7. Santrauka ir apibendrinimas

Šiame skyriuje buvo išnagrinėtas vienpakopės metaprogramos transformavimas į daugiapakopę. Toks transformavimas reikalingas tam, kad būtų galima metaprogramą pritaikyti įvairiems kontekstams tam pačiam taikymui. Transformavimo uždavinys buvo išskaidytas į du etapus: specializavimo ir adaptavimo. Metaprogramos specializavimas buvo suformuluotas remiantis programų specializavimo uždaviniu (Futamuros interpretacija). Buvo nustatyta analogija tarp programos specializavimo ir dviejų pakopų metaprogramos. Remiantis šia analogija ir pritaikyta rekursija, buvo suformuluotas daugiapakopis specializavimo uždavinys. Iki šiol buvo žinoma dviejų pakopų metaprogramos (Štuikys, Damaševičius, 2013a). Šiame skyriuje jos apibendrintos daugiapakopiam atvejui. Apibendrinimas pasiektas, kadangi buvo teoriškai nustatyta tokios transformacijos egzistavimo sąlyga bei galimas maksimalus pakopų skaičius. Be to, surasta metakonstrukcijų deaktyvavimo indekso reikšmės priklausomybė nuo pakopos laipsnio (eilės numerio).

Adaptavimo uždavinys suformuluotas kaip žemesnio lygmens daugiapakopės metaprogramos išvedimas (generavimas) iš aukštesnio lygmens daugiapakopės metaprogramos, kai tos pakopos metaparametrus nustato vartotojas. Adaptavimo procesas pilnai automatinis (po to, kai aukštesnio lygmens parametrai vartotojo jau yra nustatyti adaptavimui). Procesą atlieka metakalbos procesorius. Teorinę dalį sudaro formalūs daugiapakopių metaprogramų bei jų sudėtinių dalių (elementų) apibrėžimai, nustatytos jų savybės ir transformavimo taisyklės ir procesai.

Metaprogramos specializavimo uždavinio sprendime esminį vaidmenį vaidina: (1) konteksto modelio sąryšiai su metaparametrų prioritetu; (2) mataparametrų reikšmių tarpusavio sąveika; (3) taikymo reikalavimai. Srities konteksto modelis yra kuriamas srities ekspertų ir priklauso nuo uždavinio ir tikslo konkrečioje situacijoje.

Sąryšių nustatymas įgalino suformuluoti transformavimo taisykles ir sukūrė realias prielaidas algoritmizuoti uždavinį ir sukurti įrankį automatizuotai specializuoti metaprogramas.

4.8. Išvados

1. Pritaikytas (Futamura, 1999) programų specializavimo uždavinio *interpretavimas* įgalino iš pradžių suformuluoti dviejų pakopų metaprogramų specializavimo uždavinį, o po to, pastarajam pritaikius rekursijos principą, buvo suformuluotas daugiapakopės transformacijos (t. y. specializavimo) uždavinys.

2. Kadangi dviejų pakopų metaprogramos modelis (kitoje notacijoje) jau buvo žinomas, tai atliktas apibendrinimas yra mokslinis naujumas.

3. Esminį teorinį šio skyriaus rezultatą galima apibūdinti taip:

(a) nustatyta apibendrinto specializacijos uždavinio išsprendžiamumo sąlyga, t. y. „*uždavinys išsprendžiamas tada ir tik tada, jei vienpakopės metaprogramos metasąsajos svorinis grafas $G(P^w,U)$ nėra jungusis grafas*“ (čia P – metaparametrų aibė, U – briaunų aibė, vaizduojanti metaparametrų sąveiką, w – neraiškiosios logikos kintamasis, aprašantis metaparametro kontekstą).

(b) nustatyta, kad *maksimalus pakopų skaičius* lygus metasąsajos grafo *visuminiam komponenčių skaičiui* (t. y. jungias ir nejungias komponentes kartu paėmus).

(c) uždaviniui išspręsti pritaikytas metakonstrukcijų deaktyvacijos-aktyvacijos principas ir nustatyta pakopos deaktyvacijos indekso (DI) reikšmė duotai metakalbai, t. y. pakopoje k $DI = 0$; pakopoje (k-1) $DI = 1$, o žemesnėse pakopose – $DI = \sum_{a=0}^{k-2} 2^a$.

4. Gautas teorinis rezultatas pagrindžia specializavimo uždavinio sprendimo metodą ir tuo pagrindu sukurtą algoritmą (taisykles, įrankį).

5. Daugiapakopės metaprogramos adaptavimas yra jos pakopų laipsnio pažeminimas (nuo k iki k-1, arba nuo k-1 iki k-2 ir kt., kol pasiekiama 1 pakopa) vartotojui atitinkamai parenkant tinkamas parametrų reikšmes, o po to automatiškai sukuriant žemesnio lygmens metaprogramą (metaprogramas). Adaptavimas – tai taip pat žemesnių pakopų *metageneratorių* kūrimas.

5. TRANSFORMAVIMO ĮRANKIAI: „MePAG“ ir „MP-ReTool“

5.1. Įvadas

Trečiame ir ketvirtame skyriuose apibrėžti formalizmai ir požymiais grindžiamų modelių transformavimo bei metaprogramų specializavimo taisyklės sudarė sąlygas automatizuotiems įrankiams kurti. Skyriuje nagrinėjamas transformavimo įrankių sukūrimas ir jų panaudojimas.

Šio skyriaus tikslas – aprašyti naujai sukurtus eksperimentinius transformavimo įrankius „MePAG“ (angl. *Meta-Program Automatic Generator*) (Bespalova, Burbaitė ir Štuikys, 2015a) ir „MP-ReTool“ (angl. *Meta-program refactoring tool*) (Bespalova, Burbaitė ir Štuikys, 2015b). Įrankis „MePAG“ skirtas automatizuotai kurti heterogenines metaprogramas. Šiame įrankyje įgyvendintos 3.5 skyrelyje aprašytos požymiais grindžiamų modelių transformavimo taisyklės. Įrankis „MP-ReTool“ skirtas automatizuotai kurti specializuotas (daugiapakopes) metaprogramas. Šiame įrankyje įgyvendintos 4.4 skyrelyje aprašytos metaprogramų transformavimo į daugiapakopes taisyklės.

5.2 skyrelyje bendrai apžvelgiamas įrankių panaudojimas metaprogramų kūrimo ir specializavimo procese; 5.3 skyrelyje aprašytas metaprogramų kūrimo įrankis „MePAG“; 5.4 skyrelyje aprašytas metaprogramų transformavimo į daugiapakopes įrankis „MP-ReTool“; 5.5 skyrelyje pateikiamos apibendrintos įrankių charakteristikos ir 5.6 skyrelyje suformuluotos skyriaus išvados.

5.2. Įrankių panaudojimas kuriant ir specializuojant metaprogramas

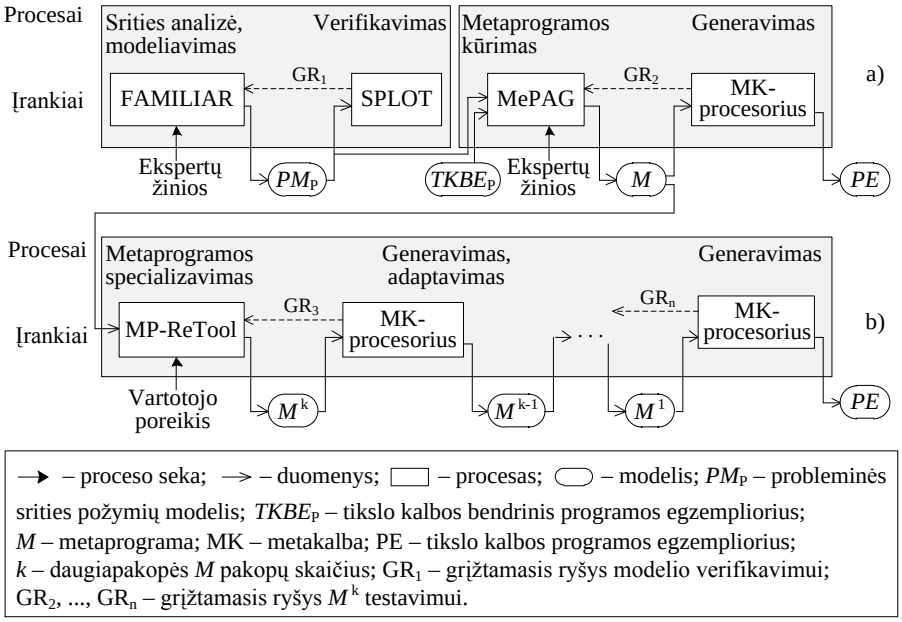
Įrankių panaudojimas, kuriant ar transformuojant metaprogramas, leidžia dirbti efektyviau, palengvina projektavimo bei kūrimo procesą. 5.1 pav. pateikiama bendra įrankių panaudojimo schema, apimanti modelių ir metaprogramų sukūrimą ir jų transformavimą.

Kaip buvo aprašyta anksčiau (žr. 3.3.4 skyrelį), srities analizei ir modeliavimui naudojamas įrankis „FAMILIAR“. Šiuo įrankiu sukuriami srities modeliai. Sukurti požymių modeliai automatizuotai verifikuojami įrankiu „SPLOT“. Teisingi modeliai toliau naudojami kuriant metaprogramas.

Modelių transformavimo į metaprogramą procesas palaikomas naujai sukurto įrankio „MePAG“. Įrankis atlieka modelių abstrakciją mažinančią transformaciją ir generuoja metaprogramą arba metaprogramos šabloną. „MePAG“ įrankis generuoja PHP programavimo kalba parašytas metaprogramas. Įrankio „MePAG“ korektišku veikimu padeda įsitikinti jo sukurtų metaprogramų sintaksinis teisingumas ir sugeneruotų tikslo kalbos programos egzempliorių teisingumas. Tikslo kalbos programos egzempliorių generavimą atlieka standartinis metakalbos procesorius.

„MP-ReTool“ įrankis skirtas transformuoti vienpakopę metaprogramą, parašytą PHP programavimo kalba, į daugiapakopę. Vartotojas, žinodamas tikslių kontekstą ir reikalavimus, gali transformuoti metaprogramą į jos specializuotą versiją ir vėliau adaptuoti pagal savo poreikius. Kaip generavimo įrankis naudojamas standartinis metakalbos procesorius. „MP-ReTool“ įrankio korektišku

veikimu padeda įsitikinti sukurtos daugiapakopės metaprogramos ir generuojamų žemesnės pakopos metaprogramų teisingumas.



5.1 pav. Įrankių naudojimo metaprogramos kūrimo (a) ir specializavimo procese (b) schema

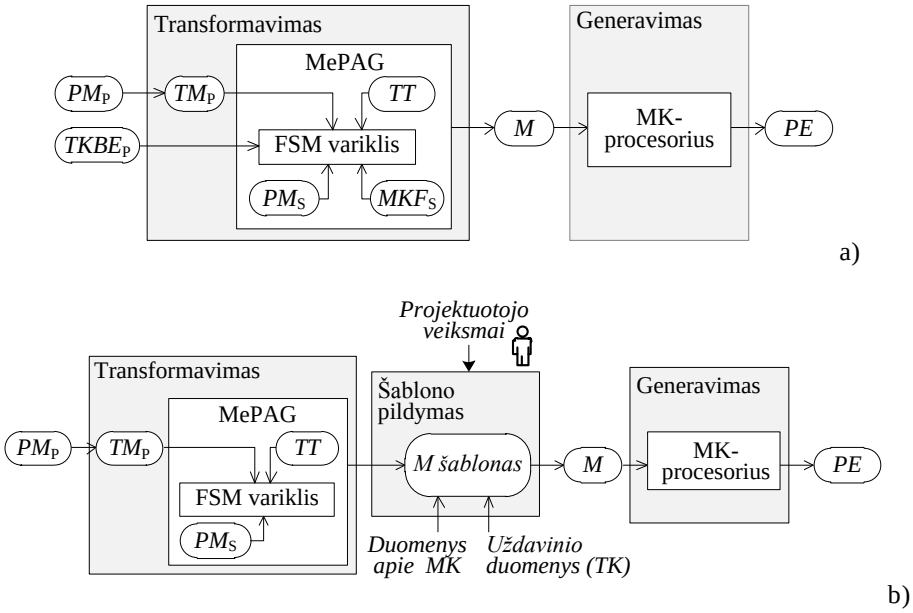
Toliau detaliau apžvelgsime naujai sukurtus įrankius „MePAG“ ir „MP-ReTool“.

5.3. Metaprogramų kūrimo įrankis „MePAG“

Įrankis „MePAG“ įgyvendina 3 skyriuje aprašytą uždavinio sprendimą. Jis transformuoja modelius į vykdomąją specifikaciją (metaprogramą). Transformacijos procese naudojami pusiau formalūs modeliai, to priežastis – heterogeninė metaprogramavimo paradigma, kurioje metakalba ir tikslo kalba yra abstrakčios. Be to, ne visada įmanoma metaprogramą sukurti visiškai automatiniu būdu, dažnai sudėtingiems uždaviniams automatinis metaprogramos kūrimas yra neefektyvus.

Įrankis „MePAG“ gali dirbti dviem režimais: automatiniu (5.2 a) pav.), kai sugeneruojama metaprograma ir pusiau automatiniu (5.2 b) pav.), kai sugeneruojamas metaprogramos šablonas. Pirmuoju atveju galima sukurti nesudėtingas metaprogramas automatiškai. Antrasis atvejis yra bendresnis, jis atspindi realius uždavinius, kai tikslo kalbos programos egzemplioriai yra sudėtingi ir ne visada efektyvu kurti $TKBE_P$, nes sugaištama panašiai tiek pat laiko kaip ir rankiniu būdu sukuriant metakamieną.

Įrankiui dirbant pusiau automatiniu režimu sugeneruojamas metaprogramos šablonas, kuriame yra visiškai realizuota metaprogramos struktūra ir sukurta metasąsaja. Metaprogramos šablone lieka nesukurtas tik metakamieno programinis tekstas. Projektuotojas sugeneruotą metaprogramos šabloną gali parsisiųsti į savo kompiuterį ir toliau jį pildyti rankiniu būdu.



5.2 pav. „MePAG“ įrankio darbo režimai: a) automatinis, b) pusiau automatinis

Sprendžiant metaprogramos kūrimo uždavinį probleminę sritį atspindi srities išplėstinis požymių modelis PM_P ir tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$, o sprendimo sritį – heterogeninio metaprogramavimo principai ir metodai (Štuikys, Bespalova ir Burbaitė, 2014c). Probleminės srities atvaizdavimas sprendimų srityje atliekamas vadovaujantis transformavimo taisyklėmis (žr. 3.5 skyrelyje), kurios apibrėžia, kaip probleminės srities elementas yra transformuojamas į sprendimo realizaciją.

Metaprogramos struktūrinis modelis (žr. 2.9 pav.) yra bendras visiems uždaviniams, todėl jis užprogramuotas „MePAG“ įrankyje kaip pastovi generuojamos metaprogramos struktūra.

Probleminės srities požymių modelis kiekvienam taikymui kuriamas naujas. Šis modelis yra sukuriamas naudojant „FAMILIAR“ įrankį. „FAMILIAR“ nestandartizuotas, ateityje galimi įrankio naudojamų modelių keitimai, todėl šiuo metu nėra realizuota „MePAG“ ir „FAMILIAR“ įrankių integracija ir tenka atlikti papildomą požymių modelio PM_P transformaciją į tarpinį modelį TM_P. TM_P modelyje saugoma informacija apie metaparametrus, jų reikšmes, tarpusavio sąryšius ir kontekstas, t. y. metaparametrų prioritetiniai svoriai. Dirbant įrankiu „MePAG“, TM_P yra naudojamas kaip išorinis probleminės srities įvesties modelis.

Tarpinį modelį TM_P vartotojas gali susikurti rankiniu būdu, naudodamasis bet kuriuo teksto redaktoriumi, arba įrankiu „MePAG“. Naudodamas įrankį „MePAG“, vartotojas net nežinodamas tarpinio modelio TM_P struktūros gali jį lengvai sukurti.

Vartotojo pažingsniui prašoma įvesti su uždaviniu susijusią informaciją, gagal kurią įrankis sugeneruoja probleminės srities tarpinį modelį. Šį modelį vartotojas gali išsisaugoti asmeniniame kompiuteryje ir vėliau naudoti kuriant metaprogramas (TM_P kūrimo algoritmas, realizuotas įrankyje „MePAG“, pateikiamas 3 priede).

Probleminės srities požymių modelio tarpinis modelis TM_P sukuriamas laikantis apibrėžtos duomenų struktūros (5.1 lent.).

5.1 lentelė. Probleminės srities požymių modelio tarpinio modelio struktūra

Stulpelio numeris	Paskirtis	Paaiškinimas
1	Metaparametro prioritetas	Nurodomas aprašomo metaparametro prioritetinis lygmuo. Metaparametras gali turėti vieną iš galimų reikšmių: HP – aukštą prioritetą, IP – vidutinį prioritetą ir LP – žemą prioritetą.
2	Metaparametro vardas	Nurodomas aprašomo metaparametro vardas.
3	Metaparametro priklausomybė	Nurodoma aprašomo metaparametro priklausomybė, t. y. nurodomas metaparametro vardas, nuo kurio šis metaparametras priklauso, arba „0“, jei metaparametras yra nepriklausomas.
4	Metaparametro reikšmių priklausomybė	Nurodoma aprašomo metaparametro reikšmių priklausomybė, t. y. nurodomas metaparametro, nuo kurio priklauso aprašomas metaparametras, reikšmės, kurios priklausomybė aprašoma, numeris. Jei metaparametras nepriklausomas, nurodomas „0“.
5	Metaparametro 1 reikšmė	Nurodoma aprašomo metaparametro pirmoji reikšmė.
…	…	…
n	Metaparametro n reikšmė	Nurodoma aprašomo metaparametro paskutinioji reikšmė.

5.3 pav. parodytas probleminės srities požymių modelio (žr. 3.4 pav.) tarpinio modelio TM_P pavyzdys.

```
HP $Operatorius 0 0 * / + -
HP $Operandas_x $Operatorius 0 0 2 4 6 8 10
HP $Operandas_x $Operatorius 1 0 2 4 6 8 10
HP $Operandas_x $Operatorius 2 2 4 6 8 10
HP $Operandas_x $Operatorius 3 0 2 4 6 8 10
HP $Operandas_x $Operatorius 4 0 2 4 6 8 10
LP $Operandas_a 0 0 a a*a a*a*a
Additional
```

5.3 pav. Tarpinio modelio TM_P pavyzdys

Tikslo kalbos bendrasis programos egzempliorius $TKBE_P$ kuriamas rankiniu būdu. Šiame modelyje įvedama konkrečios metakalbos ir tikslo kalbos sintaksė. Kaip buvo minėta 3.4 skyrelyje, šio modelio sukūrimas sunkiai automatizuojamas, nes abi programavimo kalbos (metakalba ir tikslo kalba) turi didelį rinkinį specifinių rodiklių. Kiekvienas uždavinio sprendimas priklauso nuo uždavinio reikalavimų ir programuotojo patirties.

$TKBE_P$ savyje apjungia visus metaprogramos generuojamus tikslo kalbos programų egzempliorius. Modelyje kintančios tikslo kalbos programos vietos (metaparametrų naudojimo vietos) nurodomos specialiu simboliu „var_“. Metakalbos konstrukcijos užrašomos panaudojant Paskalio programavimo kalbos elementus (Shen, 2011). Jei metaprogramos programiniame tekste yra išsišakojimų ar ciklų, šios konstrukcijos atsiranda ir $TKBE_P$. Sakiniai skirti metakalbos konstrukcijoms kurti išskiriami „@”simboliu sakinio pradžioje.

Priskyrimas užrašomas:

```
@ <kintamojo vardas> := <pradinė reikšmė>;
```

Alternatyvų parinkimui skirtas sąlygos operatorius užrašomas:

```
@ if <Sąlyga> then begin
      <Sakiniai>
@ end
```

Fiksuotam kartojimų skaičiui skirtas ciklo užduoties operatorius užrašomas:

```
@ for <kintamojo vardas> := <pradinė reikšmė> to <galinė
      reikšmė> div <žingsnis> do begin
      <Sakiniai>
@ end
```

5.4 pav. parodytas 3.6 skyrelyje sukurtos metaprogramos $TKBE_P$.

```
int Met()
{
   int y = 0;
   y = var_Operandas_x var_Operatorius var_Operandas_a;
   return y;
}
```

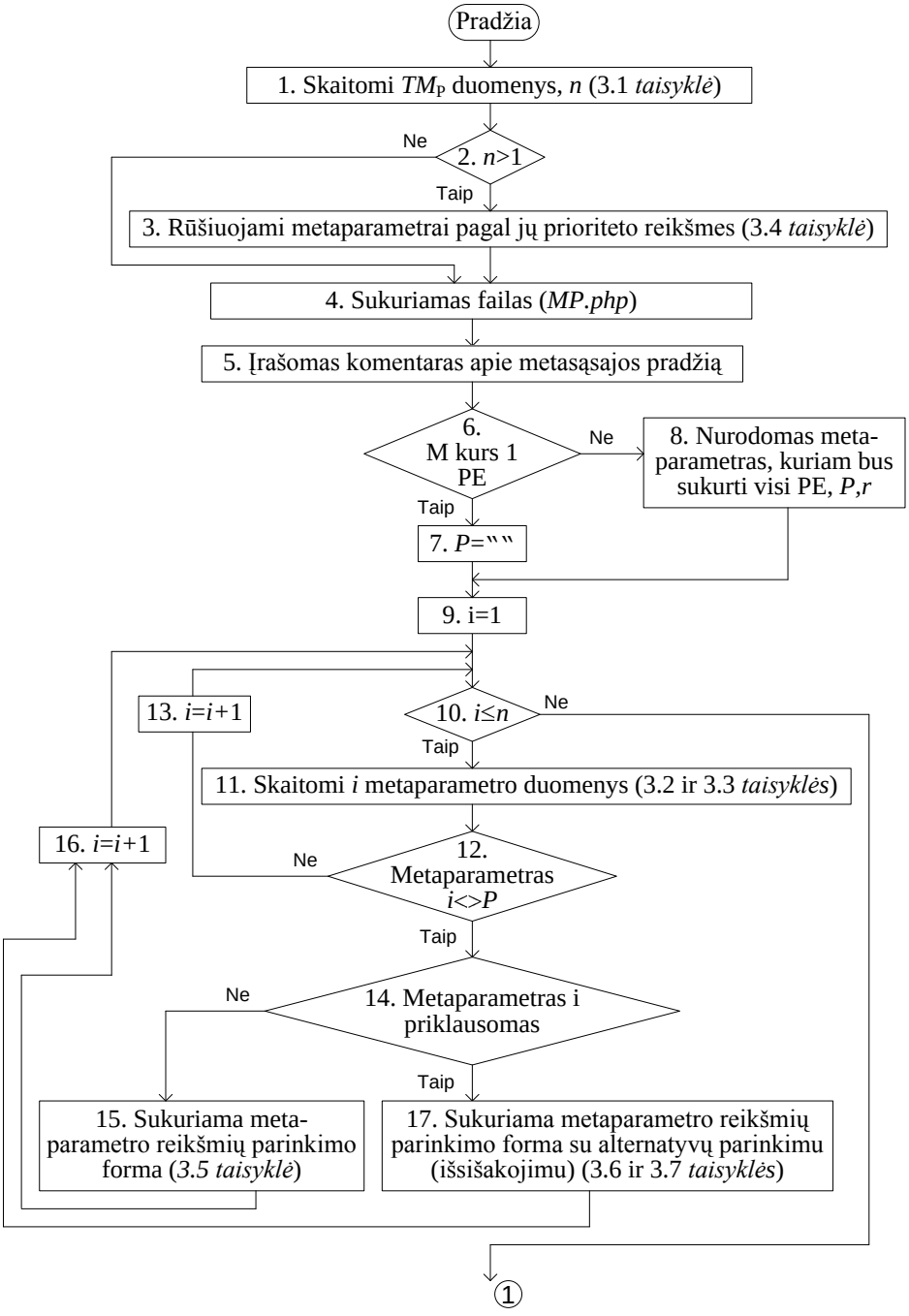
5.4 pav. Tikslo kalbos bendrojo programos egzemplioriaus pavyzdys

Pateiktas pavyzdys yra nesudėtingas, tiesinis. Jame visi veiksmai atliekami nuosekliai, vienas po kito, be alternatyvų ar veiksmų grupių kartojimo. Realaus taikymo, sudėtingesnius $TKBE_P$ pavyzdžius galite žr. 4 priede.

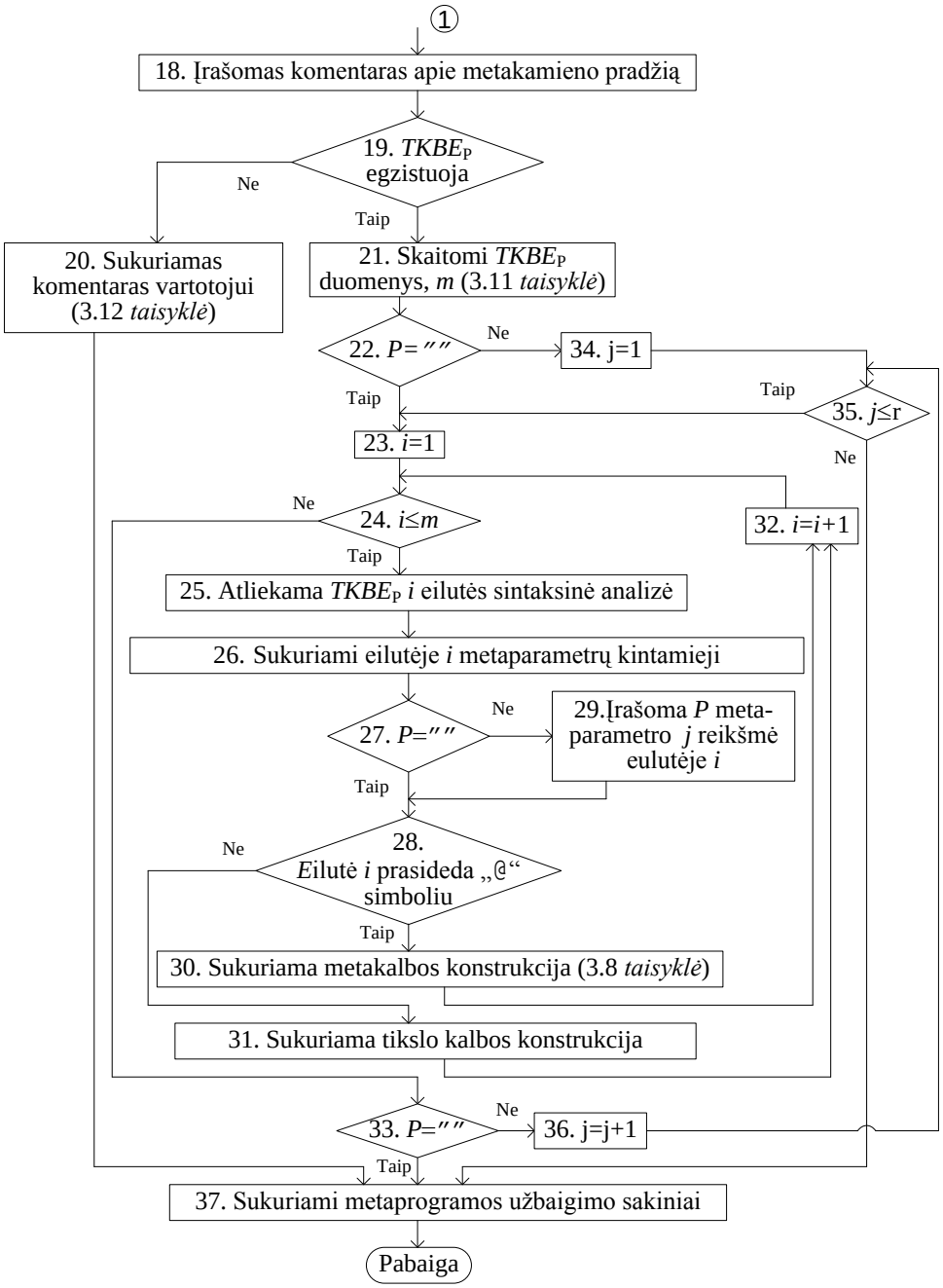
5.3.1. Įrankio MePAG darbo algoritmas

Algoritmu vadinama baigtinė nuosekli veiksmų seka, kurią procesorius turi atlikti su pradiniais duomenimis tam, kad būtų gautas uždavinio sprendinys. 5.5 pav. pateikiamas įrankio MePAG darbo algoritmas. Šiame algoritme naudojamos 3.5 skyrelyje apibrėžtos požymiais grindžiamų modelių transformavimo taisyklės.

Priklausomai nuo vartotojo turimų sukurtų modelių (TM_P; $TKBE_P$) įrankis „MePAG“ gali dirbti dviem darbo režimais: pirmasis, kai generuojama pilnai veikianti metaprograma, antrasis, kai generuojamas metaprogramos šablonas. Vartotojas taip pat gali pasirinkti, kokią metaprogramą jis nori kurti: ar metaprogramą, generuojančią vieną konkretų tikslo kalbos programos egzempliorių, ar metaprogramą, kuri generuoja tikslo kalbos programos egzempliorių šeimyną.

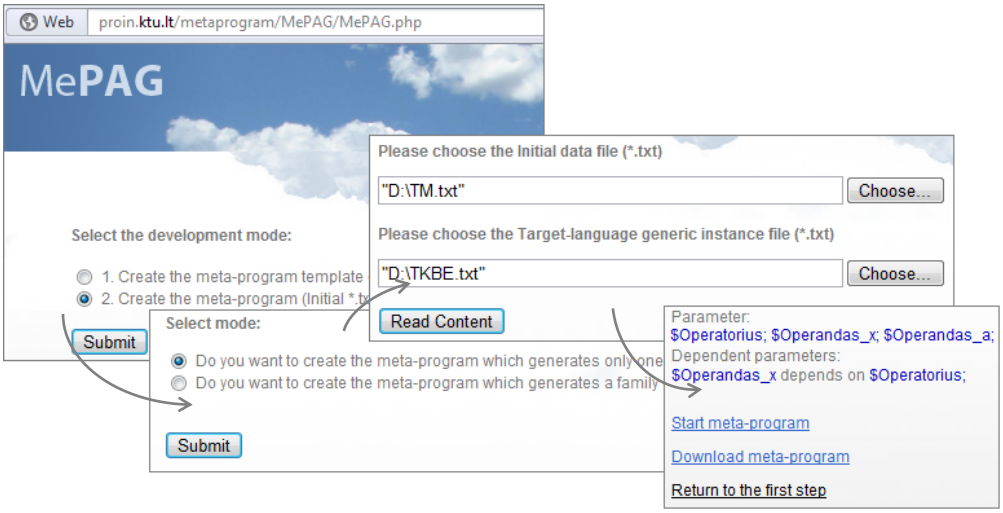


5.5 pav. Įrankio „MePAG“ funkcionavimo algoritmas (tęsinys kitame puslapyje)



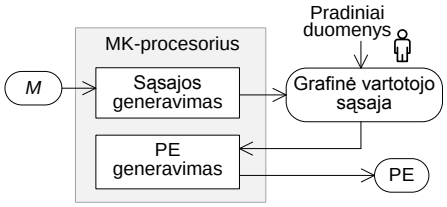
5.5 pav. Įrankio „MePAG“ funkcionavimo algoritmas (pradžia ankstesniame puslapyje)

„MePAG“ įrankio darbo langų vaizdai pateikiami 5.6 pav., o sugeneruota metaprograma pateikiama 1 priede. Sugeneruotos metaprogramos metakalba yra PHP programavimo kalba, o tikslo kalba – RobotC.



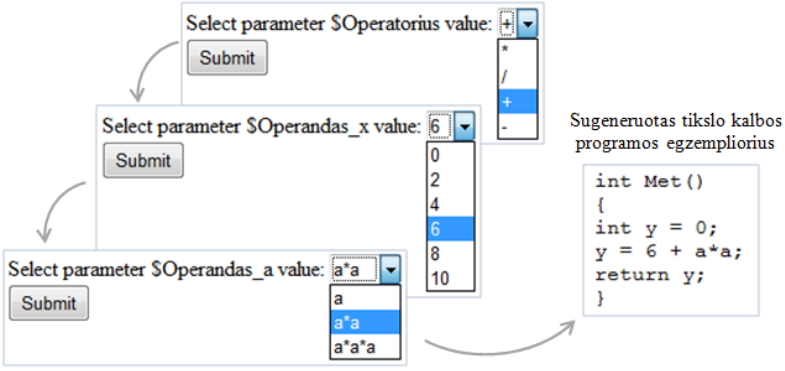
5.6 pav. MePAG įrankio darbo langų vaizdai

Sukurta metaprograma yra aukšto lygmens vykdomoji specifikacija, įgalinanti automatizuotai generuoti tikslo kalbos programos egzempliorius (5.7 pav.). Vartotojas, parinkdamas atitinkamas metaparametrų vertes, generuoja norimą tikslo kalbos programos egzempliorių. 5.8 pav. parodyti tikslo kalbos programos egzemplioriaus generavimo langai ir sugeneruotas programos egzempliorius.



M – metaprograma; MK – metakalba; PE – tikslo kalbos programos egzempliorius.

5.7 pav. Automatizuotas tikslo kalbos programos egzempliorių kūrimas



5.8 pav. Tikslo kalbos programos egzemplioriaus kūrimo langų vaizdai ir sugeneruotas tikslo kalbos programos egzempliorius

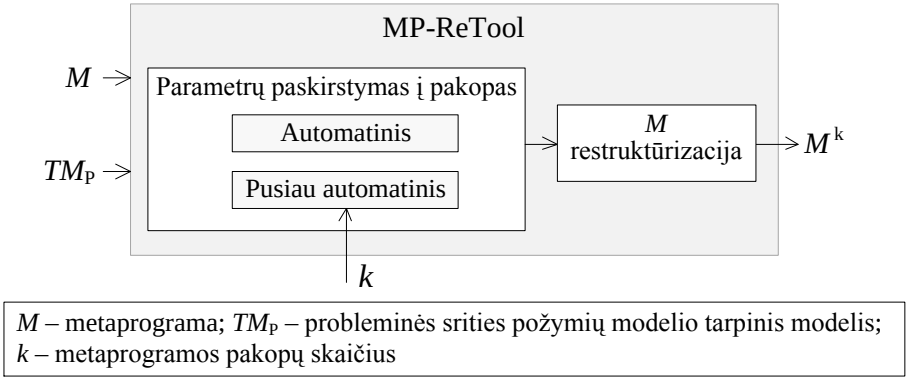
5.4. Metaprogramų specializavimo įrankis „MP-ReTool“

Metaprograma – tai konteksto valdoma specifikacija įgyvendinanti plataus masto srities variantiškumą, o tai suteikia didelę erdvę prisitaikymui prie konkretaus vartotojo poreikių. Metaprogramos adaptavimo uždavinį galima skaldyti į du atskirus etapus. Pirmojo etapo metu atliekamas metaprogramos specializavimas, t. y. vienos pakopos metaprograma transformuojama į daugiapakopę metaprogramą (specializuotą metaprogramą). Specializuota metaprograma leidžia lanksčiau atsižvelgti į vartotojo poreikius ir sukurti metaprogramą, pritaikytą konkrečiam kontekstui. Antrojo etapo metu iš aukštesnės pakopos metaprogramos generuojamos žemesnės pakopos metaprogramos, kiekvienoje pakopoje parenkamos atitinkamos metaparametrų reikšmės, vyksta metaprogramos adaptavimas. Kiekvienoje pakopoje sugeneruojama adaptuota metaprograma, kurioje yra sumažinta metaparametrų aibė, o tuo pačiu ir generuojamų tikslo kalbos programos egzempliorių aibė.

Metaprogramos specializavimas yra sudėtingas uždavinys, jei jį realizuojame rankiniu būdu. Dėl deaktyvacijos simbolių gausos programos kodo rašymas reikalauja didelio atidumo ir skaičiavimų (programuotojas turi suskaičiuoti kiekvienos deaktyvuojamos metakonstrukcijos deaktyvacijos indeksą (žr. 4.6 apibrėžimą), be to didėjant pakopų skaičiui (kai $k > 2$) programos tekstas darosi sunkiai skaitomas.

Įrankis „MP-ReTool“ skirtas automatizuotai transformuoti metaprogramą, parašytą PHP programavimo kalba, į daugiapakopę. Šios transformacijos tikslas – iš anksto užprogramuoti galimą metaprogramos adaptavimą pagal vartotojo poreikius, vykdant specializuotą metaprogramą.

Įrankio darbui vartotojas turi turėti sukurtą teisingą vienpakopę metaprogramą ir probleminės srities požymių modelio tarpinį modelį TM_{P}. Iš šio modelio nuskaitoma pradinė informacija apie visus metaparametrus, jų reikšmes ir sąveikas, taip pat kontekstinė informacija (5.9 pav.). Vienpakopės metaprogramos transformavimas į daugiapakopę atliekamas laikantis apibrėžtų transformavimo taisyklių (žr. 4.4. skyrelį).

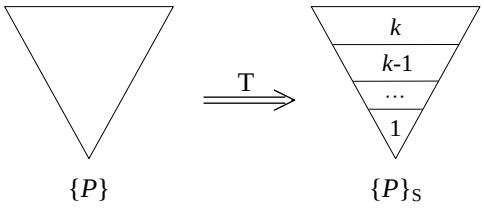


5.9 pav. „MP-ReTool“ įrankio struktūra

Įrankis „MP-ReTool“ gali dirbti dviem darbo režimais: automatiniu ir pusiau automatiniu. Dirbant automatiniu režimu pats įrankis paskaičiuoja pakopų skaičių

(4.7 išraiška) ir paskirsto metaparametrus jose. Pusiau automatinio darbo režimo metu įrankis paprašo įvesti reikiamą pakopų skaičių ir leidžia vartotojui paskirstyti metaparametrus jose.

Specializuojant vienpakopę metaprogramą į daugiapakopę, visa metaparametrų aibė yra suskirstoma į poaibius (žr. 4.2 skyrelį). Metaparametrų poaibiai sudaromi laikantis apribojimų ir iš anksto aprašytų taisyklių (4.1 – 4.3 savybės). Poaibių skaičius atitinka daugiapakopės metaprogramos pakopų skaičių *k* (5.10 pav.).



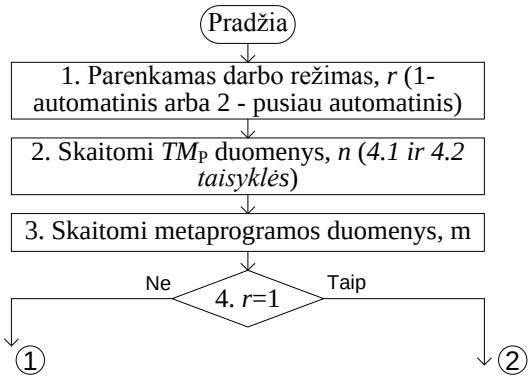
{P} – pilna metaparametrų aibė; T – transformacija; $\{P\}_S$ – suskirstyta metaparametrų aibė; *k* – metaprogramos pakopų skaičius

5.10 pav. Metaparametrų aibės suskirstymas į poaibius metaprogramos specializavimo metu

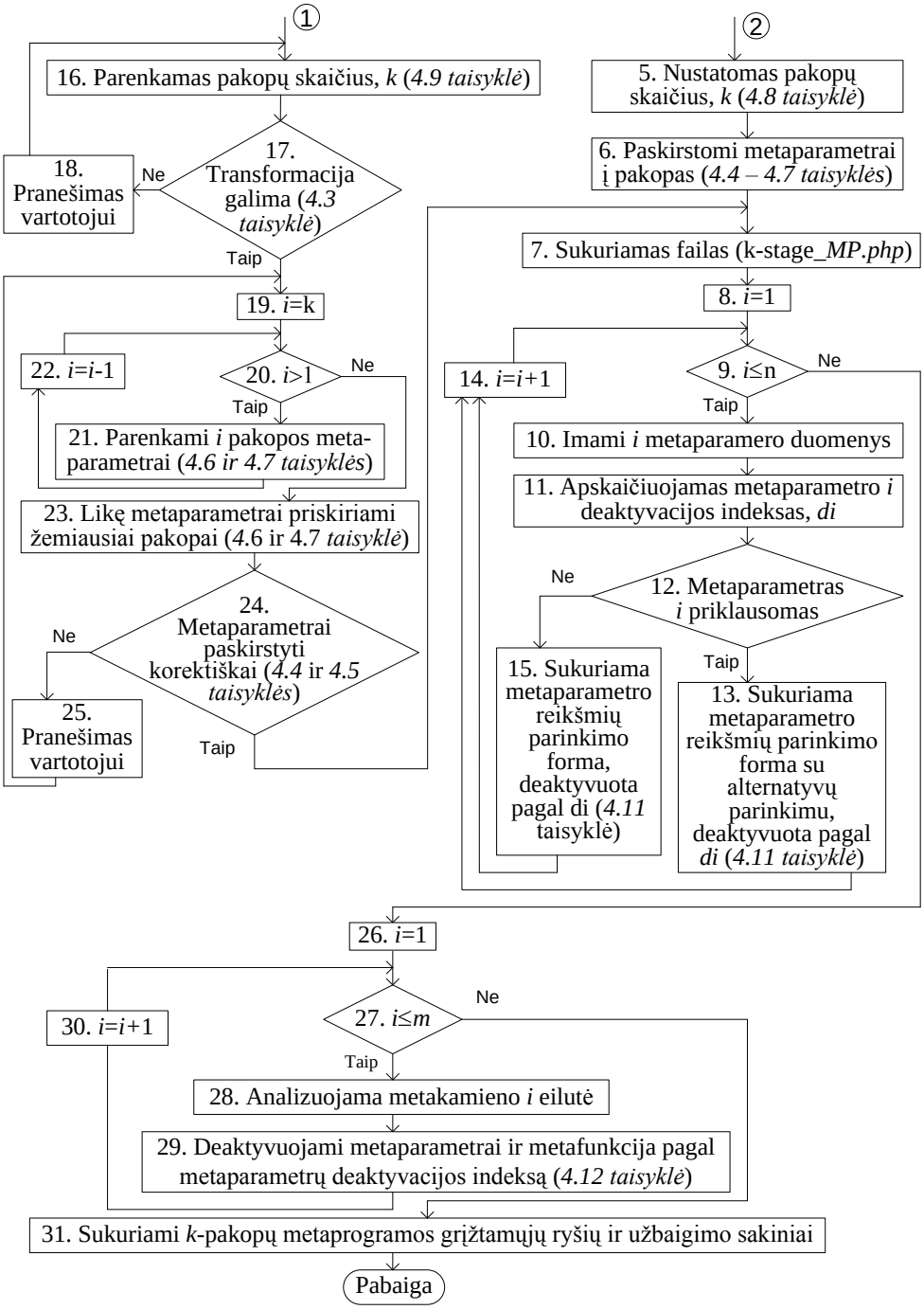
Įrankiui „MP-ReTool“ dirbant bet kuriuo režimu (automatiniu ar pusiau automatiniu), paskirstant metaparametrus pakopose atsižvelgiama į metaparametrų prioritetiniu svorius (žr. 4.3 savybę), metaparametrų tarpusavio priklausomybes (žr. 4.1 savybę) ir kitus reikalavimus (žr. 4.2 savybę).

5.4.1. Įrankio „MP-ReTool“ darbo algoritmas

„MP-ReTool“ įrankio darbo algoritmas pateikiamas 5.11 pav. Šiame algoritme naudojamos 4.4 skyrelyje apibrėžtos metaprogramų transformavimo į daugiapakopes taisyklės.



5.11 pav. „MP-ReTool“ darbo algoritmas (tęsinys kitame puslapyje)



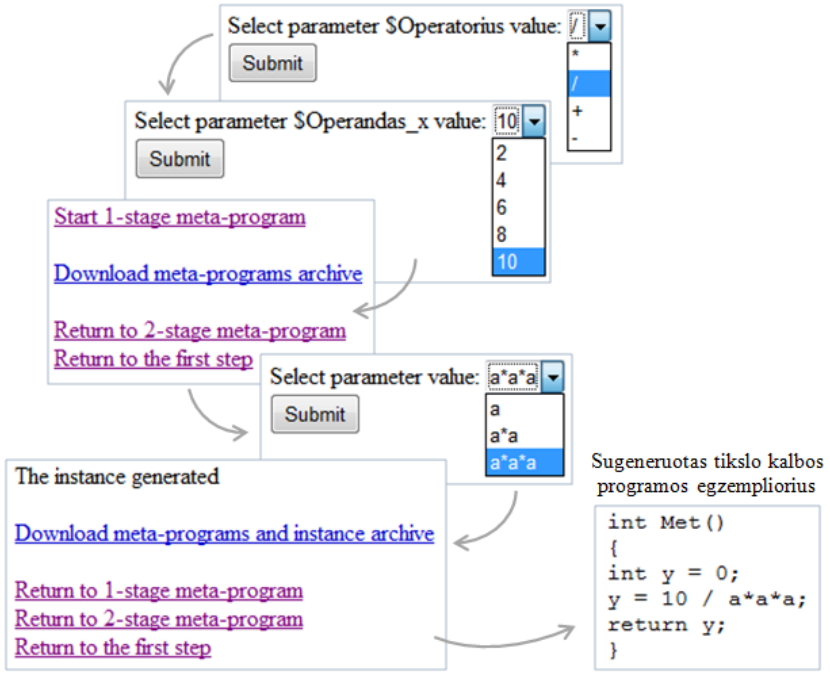
5.11 pav. „MP-ReTool“ darbo algoritmas (pradžia ankstesniame puslapyje)

„MP-ReTool“ įrankio darbo langų vaizdai pateikiami 5.12 pav. (realizuojamas 4.2 pav. nagrinėtas pavyzdys, o sugeneruota dviejų pakopų metaprograma pateikiama 2 priede.

5.12 pav. „MP-ReTool“ įrankio darbo langų vaizdai

Daugiapakopės metaprogramos vykdymo metu sugeneruojama žemesnės pakopos metaprograma, kuri yra vykdoma vėliau. Aukštesnės pakopos metaprogramos vykdymo metu dalis programos kintamųjų yra pakeičiami konstantomis, o gauti darbo rezultatai yra naudojami kaip programinis kodas žemesnėje pakopoje. Galiausiai, įvykdžius vienos pakopos metaprogramą,

sugeneruojamas tikslo kalbos programos egzempliorius, kurio programiniame tekste visi metaparametrai yra traktuojami kaip konstantos. 5.13 pav. parodyti žemesnės pakopos metaprogramų (nuo *k* pakopos metaprogramos iki 1 pakopos metaprogramos) kūrimo langai ir iš 1 pakopos metaprogramos sugeneruotas tikslo kalbos programos egzempliorius.



5.13 pav. Žemesnės pakopos metaprogramų kūrimo langų vaizdai ir sugeneruotas tikslo kalbos programos egzempliorius

Vartotojui parinkus aukštesnės pakopos metaparametrų reikšmes, metakalbos procesorius žemesnės pakopos metaprogramą generuoja automatiškai. Kiekvienoje pakopoje vartotojas gali parsisiųsti sugeneruotą žemesnės pakopos metaprogramą, gali ją startuoti arba grįžti į jau praeitas pakopas ir pakartoti kūrimo procesą.

5.5. Įrankių apibendrintas įvertinimas

Sukurti įrankiai „MePAG“ ir „MP-ReTool“ yra eksperimentiniai. Įrankius testavo 3 tyrėjai. Tai dar iki galo neišbaigti produktai, reikalaujantys tolimesnių išsamių tyrimų. Nustatyti tobulintini įrankių aspektai:

1. Įrankio „MePAG“ integravimas su „FAMILIAR“ ar „SPLOT“ įrankiu, ar kitais įrankiais palaikančiais požymiais grįstą srities modeliavimą.
2. Nuskaityti probleminės srities modelius, kurių medžio gylis didesnis už 3.
3. Realizuoti daugialypius metaparametrų sąryšius.
4. Išplėsti kuriamų metaprogramų metakalbų aibę.
5. Realizuoti nuo metakalbos nepriklausantį metaprogramos transformavimo procesą.

5.2 lent. Pateikiamos apibendrintos įrankių „MePAG“ ir „MP-ReTool“ charakteristikos.

5.2 lentelė. „MePAG“ ir „MP-ReTool“ charakteristikos

Charakteristika	**„MePAG“**	**„MP-ReTool“**
Statusas	Eksperimentinis	Eksperimentinis
Prieinamumas	Laisvai prieinama tinklinė sistema	Laisvai prieinama tinklinė sistema
Veikimo pusė	Serverio pusėje	Serverio pusėje
Įrankio programavimo kalba	PHP, HTML	PHP, HTML
Dydis (LOC/KB)	1783/ 880KB	4589/ 1576 KB
Teorinis pagrindimas	Požymių modeliai, metaprogramų modeliai, metaprogramavimas, FSM skaičiavimų-valdymo modelis.	Požymių modeliai, metaprogramų modeliai, metaprogramavimas, specializavimas, restruktūrizavimas, FSM skaičiavimų-valdymo modelis.
Projektavimo metodas	Modeliu grįstos transformacijos	Kontekstu valdomas restruktūrizavimas, pakopinis metaprogramavimas
Pirminiai modeliai ir duomenys	Probleminės srities variantiškumo modelis, metaprogramos modelis, tikslo kalbos bendrasis programos egzempliorius	Srities variantiškumo modelis, metaprograma, daugiapakopės metaprogramos modelis, pakopų skaičius
Palaikomi bylų formatai	PHP, txt	PHP, txt
Kuriamos metaprogramos programavimo kalbos	PHP, HTML	PHP, HTML
Išvedimo duomenys	Metaprograma arba metaprogramos šablonas	Daugiapakopė metaprograma (specializuota metaprograma)
Sukurtų bylų formatai	PHP, zip	PHP, zip
Apribojimai	Metakalba – PHP (tikslo kalba neprikla usoma), srities variantiškumo modelis reikalauja eksperto žinių, požymio diagramos medžio gylis – 3	Metakalba – PHP (tikslo kalba neprikla usoma), srities variantiškumo modelis reikalauja eksperto žinių, maksimalus pakopų skaičius – 5.
Integracija su kitais įrankiais	„MP-ReTool“	„MePAG“
Kiti naudojami įrankiai	„FAMILIAR“, „SPLOT“, Dramveawer, Interneto naršyklė	Dramveawer, Interneto naršyklė
Vartotojo sąsaja	Patogi vartotojui grafinė sąsaja	Patogi vartotojui grafinė sąsaja
Vartotojo vadovas	Yra	Yra

LOC – programinio kodo eilučių skaičius; KB – programos dydis kilobaitais.

5.6. Išvados

1. Pasiūlyti, išbandyti ir pritaikyti įrankiai (vieni parinkti – „FAMILIAR“ ir „SPLOT“, kiti sukurti – „MePAG“ ir „MP-ReTool“), palaikantys visą metaprogramos gyvavimo ciklą: *modeliavimo*, *modelių transformavimo*, *metaprogramų transformavimo į daugiapakopes*.

2. Įrankis „MePAG“ įgyvendina modeliu grįstą požiūrį kuriant metaprogramas. Įrankis transformuoja modelius į vykdomąją specifikaciją (metaprogramą) arba metaprogramos šabloną.

3. Įrankis „MP-ReTool“ transformuoja vienpakopę metaprogramą į daugiapakopę. Šios transformacijos dėka sukuriama specializuota metaprograma, kuri įgalina metaprogramas adaptuoti skirtingam kontekstui. Automatinis jo darbo režimas yra valdomas konteksto modelio.

4. „MePAG“ ir „MP-ReTool“ yra nepriklausomi nuo tikslo kalbos, tačiau priklausomi nuo metakalbos (PHP programavimo kalba).

5. Naujai sukurti įrankiai „MePAG“ ir „MP-ReTool“ yra eksperimentiniai. Nors jie išbandyti kuriant mokomųjų robotų valdymo realias metaprogramas, tačiau ištirtų metaprogramų variantų skaičius buvo ribotas (9 metaprogramos ir kiekviena turėjo 1–8 variantų; žr. 6.2.1 skyrelį), todėl jie reikalauja tolimesnių išsamių tyrimų.

6. EKSPERIMENTINIS ĮVERTINIMAS

6.1. Įvadas

Šio skyriaus tikslas – įvertinti metaprogramų kūrimo procesą, palyginti metaprogramų kūrimo būdus ir įvertinti metaprogramos struktūrinius pokyčius vienpakopę metaprogramą transformuojant į daugiapakopę.

Eksperimentams atlikti buvo parinkti šie uždaviniai:

1. Testinių atvejų generavimas. Skaitmeninių schemų testavime labai svarbus testinių rinkinių generavimo mechanizmas (Bareisa ir kt., 2005). Metaprograma generuoja testines sekas, skirtas schemų testavimui.
2. L-sistemos. L-sistemos – tai paralelinė perrašymo sistema, skirta modeliuoti augalų augimo ir vystymosi procesus, įvairių organizmų morfologiją, kuriant trimačius vaizdus (Prusinkiewicz, Lindenmayer, 1990).
3. Robotais grindžiamos mokymosi aplinkos. Sukurtos metaprogramos generuoja mokomųjų robotų valdymo programas (Burbaitė ir kt., 2014).

Du pirmieji uždaviniai buvo sprendžiami pirminėje eksperimentavimo stadijoje siekiant išsiaiškinti heterogeninio metaprogramavimo principus ir transformavimo ekvivalentiškumą (Štuikys ir kt., 2014a).

Daugiausiai eksperimentų buvo atlikta su metaprogramomis skirtomis generuoti mokomųjų robotų valdymo programas (Burbaitė, 2014; Burbaitė ir kt., 2013). Šios programos skirtos informatikos (programavimo) mokymuisi. Naudojamos „Lego NXT“ (Burbaitė, 2014) ir „ARDUINO“ (Mellodge, Russell, 2013; Rubio, Hierro ir Pablo, 2013) mokymosi aplinkos.

Eksperimento metu nebuvo vertinami probleminės srities požymių modeliai, nes juos sukūrė, verifikavo ir pateikė tolimesniems eksperimentams Burbaitė (2014). Jos sukurti modeliai buvo papildyti konteksto informacija, pakartotinai verifikuoti ir toliau panaudoti eksperimentiniuose tyrimuose.

6.2 skyrelyje aprašytas metaprogramų kūrimo būdų (rankinis, pusiau automatinis ir automatinis) įvertinimas; 6.3 skyrelyje aprašytas metaprogramų specializavimo ekvivalentiškumo tyrimas bei sukurtų daugiapakopių metaprogramų įvertinimas; 6.4 skyrelyje aprašytas metaprogramų sudėtingumo tyrimas; 6.5 skyrelyje suformuluotos skyriaus išvados.

6.2. Metaprogramų kūrimo įvertinimas

Metaprogramos pirmiausiai buvo kuriamos rankiniu būdu. Kad būtų galima sukurti tokias programas, reikia mokėti programuoti vienu metu panaudojant bent dvi programavimo kalbas: pasirinktą tikslo ir metakalbą. Taip pat reikia suprasti metaprogramos įgyvendinimo principus ir metodus. Tai sudėtingas ir imlus laikui darbas. Tačiau tik gerai pažinus, kaip rankiniu būdu parašyti metaprogramos kodą, buvo įmanoma tinkamai sukonfigūruoti metaprogramos kūrimo įrankį.

6.1 lent. pateikiama apibendrinta metaprogramų kūrimo metodų informacija, apžvelgiami visi išbandyti metaprogramų kūrimo metodai. Lentelėje aprašyti keturi

kūrimo būdai: rankinis, rankinis panaudojant modelius, pusiau automatinis ir automatinis. Du paskutinieji metodai buvo išbandyti sukūrus įrankį MePAG.

6.1 lentelė. Metaprogramos kūrimo metodų palyginimas

M kūrimo būdas / Atributai	Rankinis	Rankinis, panaudojant modelius	Pusiau automatinis	Automatinis
Pradiniai duomenys	MK, TK, reikalavimai, apribojimai	MK, TK, reikalavimai, apribojimai	MK, TK, reikalavimai, apribojimai	MK, TK, reikalavimai, apribojimai
Pradiniu duomenų pateikimas	Pagal pavyzdį	Pagal scenarijų	Pagal PM-ius	Pagal PM-ius
Kūrimo modelis	Intuityvios projektuotojo žinios	Susisteminti modeliai, rankinio projektavimo taisyklės	Susisteminti modeliai, įrankis MePAG	Susisteminti ir išplėsti modeliai, įrankis MePAG
Pradinių duomenų modeliai	Numanomi projektuotojo	Probleminės srities modelis	D*f*R ir D*w*R karkasas, PM_P, PM_S	D*f*R ir D*w*R karkasas, PM_P, $TKBE_P$, PM_S, MKF_S
Transformavimo taisyklės	Intuityvios taisyklės ir kontekstas	Aiškios taisyklės ir kontekstas	Įrankio palaikomos taisyklės ir kontekstas	Įrankio palaikomos išplėstos taisyklės ir kontekstas
Transformavimo variklis	–	Projektuotojo skaičiavimų modelis	Skaičiavimų-valdymo modelis (FSM) *M* šablonui kurti (metasąsają)	Skaičiavimų-valdymo modelis (FSM) *M* kurti (metasąsają ir metakamieną)

M – metaprograma; MK – metakalba; TK – tikslo kalba; PM_P – probleminės srities variantiškumo požymių modelis; PM_S – metaprogramos struktūrinis požymių modelis; $TKBE_P$ – tikslo kalbos bendrasis programos egzemplioriaus modelis; MKF_S – metakalbos funkcijų modelis.

Toliau detaliau apžvelgsime įrankio MePAG panaudojimo efektyvumą.

6.2.1. Įrankio „MePAG“ įvertinimas

Įrankio „MePAG“ (Bespalova ir kt, 2015a) įvertinimui buvo pasirinktos metaprogramos, generuojančios realius mokymosi objektus, skirtus programavimo mokymui vidurinėje mokykloje. Mokymosi objektas – tai nepriklausomas ir savarankiškas mokymosi turinio vienetas, kuris turi tikslą ir kuriamas iš anksto numatant pakartotinio panaudojimo galimybę (Burbaitė, 2014). Sugeneruoti mokymosi objektai buvo integruojami į mokomaisiais robotais grįstas mokymosi aplinkas. Naudojamos „Lego NXT“ (Burbaitė ir kt., 2013) ir „ARDUINO“ (Burbaitė ir Bespalova, 2014; Mellodge, Russell, 2013; Rubio, Hierro ir Pablo, 2013) mokymosi aplinkos. Šių aplinkų panaudojimas nulėmė tikslo kalbų pasirinkimą. Darbe nagrinėjamos metaprogramos, generuojančios roboto valdymo programas, parašytas RobotC (2007) arba Arduino programavimo kalbomis.

PHP programavimo kalba buvo pasirinkta kaip metakalba. Pasirinkimą lėmė šios kalbos paprastumas ir lankstumas, ji veikia daugumoje operacinių sistemų, yra

atvirojo kodo. PHP kalba yra universali programavimo kalba, plačiai taikoma interneto svetainėms kurti, taip pat ji leidžia atlikti programinio teksto analizę.

Vertinant įrankį „MePAG“ buvo tyrinėjami 9 uždaviniai (uždavinių modeliai pateikti 4 priede):

1. Roboto kalibravimas;
2. Linijos sekimas;
3. Ornamentų kūrimas;
4. Reakcija į kliūtį;
5. Spalvos atpažinimas;
6. Pagalbos sistema;
7. Bėganti eilutė;
8. Šviesos sekimas;
9. Šviesoforas.

Pirmieji šeši uždaviniai buvo įgyvendinti naudojant „Lego NXT“ aplinką, o likusieji trys – „ARDUINO“ aplinką. Sukurtų modelių ir metaprogramų charakteristikos pateiktos 6.2 ir 6.3 lent. Metaprogramos buvo kurtos rankiniu būdu ir panaudojant įrankį MePAG. Eksperimentinis įvertinimas atliktas dalyvaujant 3-ų tyrėjų grupei.

6.2 lentelė. Modelių ir metaprogramų kūrimo charakteristikos

Metaprogramos atributai / Uždavinys	TM_P (LOC/ KB)	$TKBK_P$ (LOC/ KB)	Metasąsaja rankiniu būdu (LOC/KB)	Metasąsaja sugeneruota (LOC/ KB)	Metakamienas rankiniu būdu (LOC/KB)	Metakamienas sugeneruotas (LOC/ KB)
	1	2	3	4	5	6
Roboto kalibravimas	16/0,74	46/1,26	207/8,01	213/8,04	52/1,82	54/1,83
Linijos sekimas	19/0,8	137/4,5	279/12,8	287/13	138/4,81	144/4,94
Ornamentų kūrimas	21/0,7	128/3,6	328/14,5	336/14,8	131/5,5	134/5,5
Reakcija į kliūtį	16/0,6	67/1,5	214/8,05	223/8,09	70/2,07	75/2,11
Spalvos atpažinimas	9/0,3	82/1,9	152/6,16	154/6,19	87/2,68	90/2,71
Pagalbos sistema	5/0,2	18/0,4	66/2,33	67/2,33	22/0,90	26/ 0,91
Bėganti eilutė	8/0,3	67/2,3	125/5,19	125/5,32	74/2,86	76/2,87
Šviesos sekimas	10/0,4	100/2,3	176/7,54	176/7,71	109/2,89	112/2,91
Šviesoforas	12/0,4	85/2,1	221/9,98	221/10,2	91/3,33	93/3,34

TM_P – probleminės srities variantiškumo požymių modelio tarpinis modelis; $TKBE_P$ – tikslo kalbos bendrasis programos egzemplioriaus modelis; LOC– programos teksto eilučių skaičius (PHP).

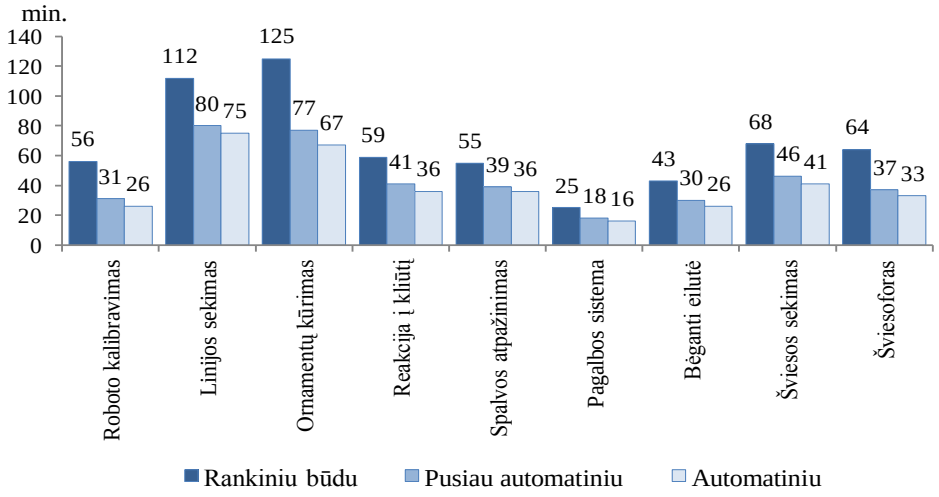
Kiekvieno modelio (TM_P ir $TKBE_P$) ar metaprogramos kūrimo metu buvo fiksuojamas sukūrimui sugaištamas laikas. Vėliau laikai buvo apibendrinti į „vidutinį laiką, reikalingą sukurti modeliams ir programoms (minutėmis)“. Iš 6.3 lent. pateiktų rezultatų galima matyti, kad ne visada efektyvu kurti tikslo kalbos bendrąjį programos egzempliorių, nes jo sukūrimui sugaištama panašiai tiek pat laiko kaip ir rankiniu būdu sukuriant metakamieną. Tikslo kalbos bendrąjį programos egzempliorių tikslinga kurti tik tada, kai tiksliai žinome, kad

metaprograma bus kuriama (ne vieną kartą) pakartotinai. 6.1 pav. pateikiami vidutiniai laikai (minutėmis), sugaištami modeliams ir metaprogramoms kurti.

6.3 lentelė. Metaprogramų kūrimo laiko charakteristikos

Metaprogramos atributai / Uždavinys	Vidutinis laikas, reikalingas sukurti modeliams (minutėmis)		Vidutinis laikas, reikalingas sukurti metaprogramoms (minutėmis)		
	TM_P	$TKBK_P$	Rankiniu būdu (M_I+M_B)	Su „MePAG“ (pusiau automatiniu būdu) (A+ M_B)	Su „MePAG“ (automatiniu būdu) (A)
	1	2	3	4	5
Roboto kalibravimas	6	18	56 (33+23)	apie 25 (2+23)	apie 2
Linijos sekimas	9	64	112 (43+69)	apie 71 (2+69)	apie 2
Ornamentų kūrimas	11	54	125 (61+64)	apie 66 (2+64)	apie 2
Reakcija į kliūtį	7	27	59(31+28)	apie 34 (2+32)	apie 2
Spalvos atpažinimas	4	30	55(22+33)	apie 35 (2+33)	apie 2
Pagalbos sistema	3	11	25(12+13)	apie 15 (2+13)	apie 2
Bėganti eilutė	4	20	43 (19+24)	apie 26 (2+24)	apie 2
Šviesos sekimas	5	34	68 (29+39)	apie 41 (2+39)	apie 2
Šviesoforas	6	25	64 (35+29)	apie 31 (2+29)	apie 2

TM_P – probleminės srities variantiškumo požymių modelio tarpinis modelis; $TKBE_P$ – tikslo kalbos bendrasis programos egzemplioriaus modelis; M_I – metasąsaja; M_B – metakamienas; A – automatinis.



6.1 pav. Metaprogramų kūrimui sugaištamo laiko palyginimas minutėmis

Rankiniu būdu kuriant metaprogramas sugaištama daugiausiai laiko. Pusiau automatinio ir automatinio kūrimo būdų laikai labai panašūs, nes automatinis būdas reikalauja daugiau laiko modeliams sukurti. 6.2 pav. pateikiamas procentinis laikų

palyginimas, kai lyginamas rankinis kūrimo būdas su pusiau automatiniu ir rankinis su automatiniu.

Didžiausią efektą „MePAG“ įrankio naudojimas duoda kuriant metaprogramos šabloną. Probleminės srities tarpiniam modeliui TM_P sukurti sugaištama žymiai mažiau laiko negu metasąsają kuriant rankiniu būdu. Pusiau automatinio kūrimo būdo, lyginant su rankiniu, procentinis pagerėjimas buvo apskaičiuotas pagal (6.1) formulę:

$$\frac{RB - PA}{RB} * 100\%, \tag{6.1}$$

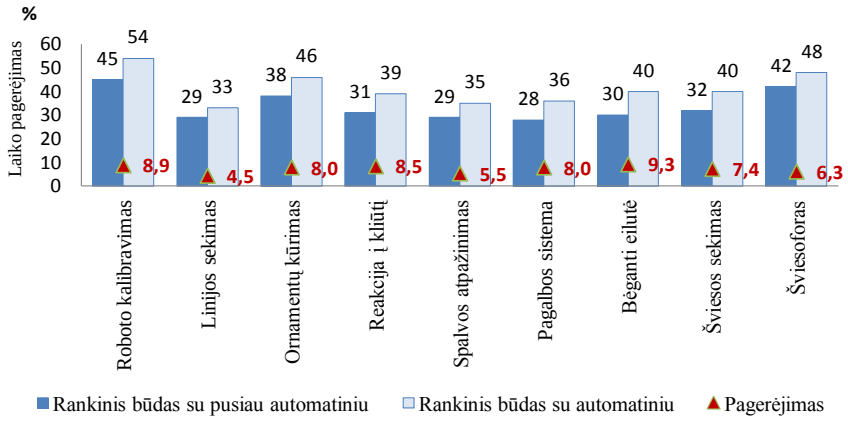
čia, *RB* – rankinis kūrimo būdas, *PA* – pusiau automatinis kūrimo būdas.

Automatinio kūrimo būdo, lyginant su rankiniu, procentinis pagerėjimas buvo apskaičiuotas pagal (6.2) formulę:

$$\frac{RB - A}{RB} * 100\%, \tag{6.2}$$

čia, *RB* – rankinis kūrimo būdas, *A* – automatinis kūrimo būdas.

Metaprogramai kurti pusiau automatiniu būdu sugaištama vidutiniškai 34 % mažiau laiko, negu kuriant metaprogramą visiškai rankiniu būdu. Palyginę metaprogramos automatinį ir pusiau automatinį kūrimo būdus matome, kad kuriant metaprogramas automatiniu būdu sugaištama vidutiniškai 7 procentiniais punktais mažiau laiko. Automatinis metaprogramos kūrimo būdas pasiteisina tik pakartotinio panaudojimo atveju.



6.2 pav. Metaprogramų kūrimui sugaištamo laiko % palyginimas

6.4 lent. pateikiamas metaprogramų kūrimo metodų palyginimas.

6.4 lentelė. Metaprogramų kūrimo būdų palyginimas

Savybė \ Kūrimo būdas	Rankinis	Rankinis, panaudojant modelius	Pusiau automatinis	Automatinis
Modelių neprieštaringumas	-	Modeliai neverifikuojami	Modeliai verifikuojami įrankiu „SPLOT“	Modeliai verifikuojami įrankiu „SPLOT“
Projektavimo valdymo sudėtingumas	Labai žemas	Žemas	Aukštas (jei nevertinamas modelių kūrimas)	Labai aukštas (jei nevertinamas modelių kūrimas)
Pakartotinis panaudojimas	Proginis (angl. *ad hoc*)	Pusiau sistemingas	Sistemingas	Sistemingas generatyvinis
Sisteminimas	Projektuotojų kompetencija	Projektuotojų kompetencija ir paprasti modeliai	Pusiau formalūs probleminės srities modeliai (modeliavimo kompetencija)	Formalūs probleminės ir sprendimo srities modeliai (modeliavimo kompetencija)
Projektavimo pastangos (laikas, klaidos, metaprogramos sudėtingumas)	Reikalauja daug laiko, didelė klaidų tikimybė, metaprogramos nesudėtingos, mažas srities variantiškumas	Reikalauja daug laiko, didelė klaidų tikimybė, metaprogramos nesudėtingos, vidutinis srities variantiškumas	Reikalauja mažiau laiko (apie 50 % (be modelių kūrimo)), vidutinė klaidų tikimybė (metasąsaja be sintaksės klaidų), metaprogramos sudėtingos, aukštas srities variantiškumas	Reikalauja mažiau laiko (apie 80 % (be modelių kūrimo)), žema klaidų tikimybė (metasąsaja ir metakūnas be sintaksės klaidų), metaprogramos sudėtingos, aukštas srities variantiškumas
Apribojimai	Priklauso nuo projektuotojo ir reikalavimų	Priklauso nuo projektuotojo, reikalavimų ir modelių apribojimų	Priklauso nuo reikalavimų, modelių ir naudojamų įrankių	Priklauso nuo reikalavimų, modelių ir naudojamų įrankių
Socialiniai aspektai, projektuotojo perspektyva	Srities ir projektavimo ekspertas	Srities ir projektavimo ekspertas	Srities ir projektavimo ekspertas	Srities ekspertas (jei modeliai sukurti) Srities ir projektavimo ekspertas (jei modeliai nesukurti)
Socialiniai aspektai, vartotojo perspektyva	Naudoja kas yra	Naudoja, kas yra	Galimybė daryti įtaką kūrimo procesui	Galimybė daryti įtaką kūrimo procesui, perkurti tobulinant
Perprojektavimas ir priežiūra	Rankinis	Rankinis	Modeliais grįsta, panaudojant įrankius	Modeliais grįsta, panaudojant įrankius

6.3. Metaprogramos specializavimo įvertinimas

6.3.1. Metaprogramos specializavimo ekvivalentiškumo tyrimas

Heterogeninių metaprogramų ekvivalenčias transformacijas nagrinėjo Štuikys ir Damaševičius (2013b). Jie analizavo apgrąžos transformavimo procesą, kai esant tai pačiai metaparametrų aibei, korektiška vienos pakopos metaprograma keičiama į dviejų pakopų ekvivalenčia metaprogramą.

Metaprogramos pirmiausiai buvo specializuojamos rankiniu būdu. Tik gerai pažinus metaprogramos specializavimo metodiką rankiniu būdu, buvo galima sukonfigūruoti automatizuotą vienpakopės metaprogramos transformavimo į daugiapakopę metaprogramą įrankį.

Darbe nagrinėjamas metaprogramos transformavimo į daugiapakopę procesas. Metaprogramų specializavimas buvo patikrintas eksperimentiškai. Sukurtos metaprogramos rankiniu būdu buvo keičiamos į dviejų ir trijų pakopų metaprogramas, vėliau tikrinamas atliktos transformacijos korektiškumas (Štuikys, Bespalova, 2012). Programos transformacija laikoma teisinga, jei esant tiems patiems pradiniams duomenims, pradinė ir transformuota programos pateikia tuos pačius rezultatus (tą patį tikslo kalbos programos egzempliorių).

Kadangi iš pradžių metaprogramų transformacijos buvo atliekamos rankiniu būdu, ekvivalentiškumo tyrimui atlikti buvo parinkti nesudėtingi uždaviniai:

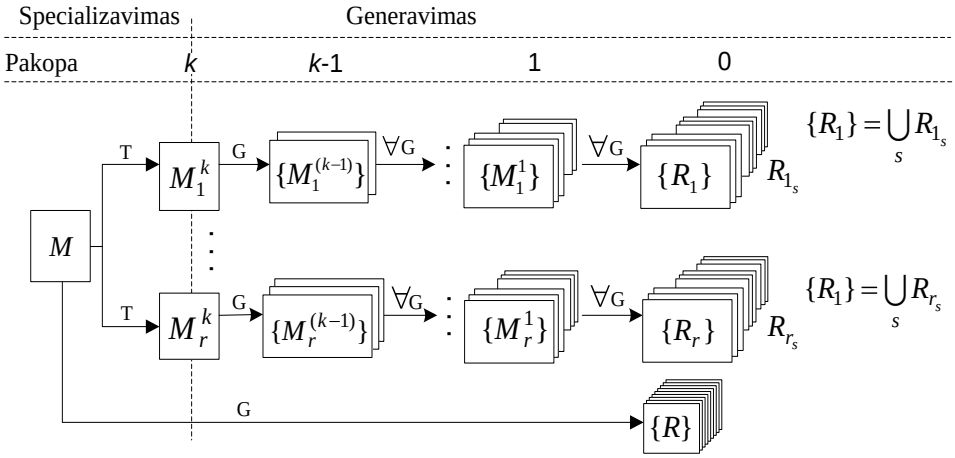
1. Testinių atvejų generavimo;
2. L-sistemų;
3. Mokomųjų robotų valdymo programų kūrimo (suprastinti atsisakant dalies metaparametrų).

Šio eksperimento tikslas – parodyti, kad metaprogramų specializavimas yra galimas ir teisingas. Specializavimo teisingumo išraiška užrašoma (6.3):

$$\forall_{k=2,3}(\forall_{i(i\in[1,r])}(\{R_i\}=\{R\}))\ \textbf{yra tiesa};\qquad (6.3)$$

čia $R_i=\bigcup_s R_{i_s}$, R – pilnas rinkinys tikslo kalbos programų egzempliorių sugeneruotų iš metaprogramos, R_{i_s} – pilnas rinkinys tikslo kalbos programų egzempliorių sugeneruotų iš M_i^k metaprogramos.

6.3 pav. parodyta eksperimentų atlikimo schema. Metaprograma specializuojama į visas galimas daugiapakopes metaprogramas. Iš kiekvienos daugiapakopės metaprogramos automatizuotai generuojamos visos galimos žemesnės pakopos metaprogramos (nuo k pakopos metaprogramos iki 1 pakopos metaprogramos). Galiausiai iš visų sugeneruotų vienos pakopos metaprogramų generuojami visi galimi tikslo kalbos programų egzemplioriai. Automatizuotas generavimo procesas atliekamas kiekvienoje pakopoje parenkant iš anksto apibrėžtų metaparametrų reikšmes.



M – metaprograma; T – transformavimas; G – pavienis generavimas; k – daugiapakopės metaprogramų pakopų skaičius;{X}– X rinkinys; ∀G – daugkartinis generavimas kiekvieno X nario; Pakopoje R – rinkinys tikslo kalbos programų egzempliorių.

6.3 pav. Metaprogramos specializavimo/generavimo uždavinys (Štuikys ir kt., 2014a)

Iš vienos daugiapakopės metaprogramos sugeneruoti tikslo kalbos programų egzemplioriai apjungiami į vieną aibę. Ši aibė lyginama su visomis kitų daugiapakopių metaprogramų rezultatų aibėmis. Taip įsitikinama, kad kiekviena specializuota metaprograma generuoja tą pačią tikslo kalbos egzempliorių aibę.

6.5 lent. pateikiamos metaprogramų, su kuriomis buvo atliekamas ekvivalentiškumo tyrimas, charakteristikos. 6.6 ir 6.7 lent. pateikiamos sukurtų dviejų ir trijų pakopų metaprogramų charakteristikos ir atliktų eksperimentinių tyrimų rezultatai.

6.5 lentelė. Eksperimentinių uždavinių metaprogramų charakteristikos

Uždavinys	Metaparametrų skaičius (prioritetai)	Metaparametrų priklausomybė (pagal numerį)	Metaparametrų reikšmių skaičius	Sugeneruotų tikslo kalbos programų egzempliorių skaičius
L-sistemos	4 (HP-3; LP-1)	Nepriklausomi	5,4,4,4	320 =5∗4∗4∗4
Testiniai atvejai	5 (LP-5)	Nepriklausomi	(2,2,2,2,2)	32=2∗2∗2∗2∗2
Roboto kalibravimas 1	4 (HP-1; LP-3)	Nepriklausomi	(3,4,3,3)	108=3∗4∗3∗3
Roboto kalibravimas 2	5 (HP-1; IP-1; LP-3)	Nepriklausomi	(2,3,4,3,3)	216=2∗3∗4∗3∗3
Linijos sekimas 1	5 (HP-1; IP-2; LP-2)	Priklausomi (2,3)	(2,(4,20),3,4)	960=2∗(2∗10+2∗10)∗3∗4
Linijos sekimas 2	6 (HP-1; IP-3; LP-2)	Priklausomi (2,3,4)	(2,(4,4,6),3,3)	360=2∗ (2∗4+2∗6)∗3∗3
Ornamentų kūrimas	3 (LP-3)	Nepriklausomi	(2,3,2)	12=2∗3∗2

HP – aukštas prioritetas, IP – vidutinis prioritetas ir LP – žemas prioritetas.

6.6 lentelė. Metaprogramos specializavimo į dviejų pakopų metaprogramą charakteristikos

Uždavinys	M^2 gautų iš M skaičius	M^2 metaparametrų skaičius	M^1 metaparametrų skaičius	M^1 sugeneruotų iš M^2 skaičius	TK programos egzempliorių sugeneruotų iš M^1 skaičius	Ekvivalentiškumas
Roboto kalibravimas 1	7	1-HP	3-LP	3	108=3∗ (4∗3∗3)	teisingas
		1-HP, 1- LP	2- LP	12 =3∗4	108=12∗(3∗3)	teisingas
Roboto kalibravimas 2	2	1- HP	1-IP, 3-LP	2	216=2∗(3∗4∗3∗3)	teisingas
Linijos sekimas 1	2	1- HP	2- IP, 2- LP	2	960 =2∗((2∗10+ +2∗10)∗3∗4)	teisingas
Linijos sekimas 2	2	1- HP	3- IP, 2- LP	2	360 =2∗(2∗4+ +2∗6)∗3∗3	teisingas
Ornamentų kūrimas	6	1- LP	2- LP	2	12=2∗(3∗2)	teisingas
		2- LP	1- LP	6=2∗3	12=6∗(2)	teisingas

M – pirminė metaprograma; M^1 – 1-pakopos metaprograma; M^2 – 2-pakopų metaprograma; HP – aukštas prioritetas, IP – vidutinis prioritetas ir LP – žemas prioritetas.

6.7 lentelė. Metaprogramos specializavimo į trijų pakopų metaprogramą charakteristikos

Uždavinys	M^3 gautų iš M skaičius	M^3 metaparametrų skaičius	M^2 metaparametrų skaičius	M^1 metaparametrų skaičius	M^2 sugeneruotų iš M^3 skaičius	M^1 sugeneruotų iš M^2 skaičius	TK programos egzempliorių sugeneruotų iš M^1 skaičius	Ekvivalentiškumas
L-sistemos	12	1- HP	1- HP	1- HP, 1-LP	5	20=4∗4	320=20∗(4∗4)	teisingas
		2- HP	1- HP	1-LP	20=5∗4	80=20∗(4)	320=80∗(4)	teisingas
Testiniai atvejai	150	1- LP	2- LP	2- LP	2	8=2∗(2∗2)	32=8∗(2∗2)	teisingas
		2- LP	1- LP	2- LP	4=2∗2	8=4∗(2)	32=8∗(2∗2)	teisingas
Roboto kalibravimas 1	12	1(HP)	1- LP	2- LP	3	12=3∗(4)	108=12∗(3∗3)	teisingas
		1(HP)	2- LP	1- LP	3	36=3∗(4∗3)	108=36∗(3)	teisingas
Roboto kalibravimas 2	1	1(HP)	1(IP)	3- LP	2	6=2∗(3)	216=6∗(4∗3∗ ∗3)	teisingas
Linijos sekimas 1	1	1(HP)	2(IP)	2- LP	2	80= 2∗(2∗ ∗10+2∗10)	960=80∗(3∗4)	teisingas
Linijos sekimas 2	1	1(HP)	3(IP)	2- LP	2	40= 2∗(2∗ ∗4+2∗6)	360=40∗3∗3	teisingas
Ornamentų kūrimas	6	1- LP	1- LP	1- LP	2	6=2∗(3)	12=6∗(2)	teisingas
		1- LP	1- LP	1- LP	3	6=3∗(2)	12=6∗(2)	teisingas

M – pirminė metaprograma; M^1 – 1-pakopos metaprograma; M^2 – 2-pakopų metaprograma; M^3 – 3-pakopų metaprograma; HP – aukštas prioritetas, IP – vidutinis prioritetas ir LP – žemas prioritetas.

Ekvivalentiškumo tyrimas patvirtino hipotezę, kad metaprogramos specializavimas keičia metaprogramos struktūrą, tačiau išsaugo pradinį metaprogramos funkcionalumą. Eksperimento metu buvo gerai suprastas vienpakopės metaprogramos transformavimo į daugiapakopę metaprogramą procesas, tai suteikė žinių, reikalingų automatizuotam įrankiui kurti.

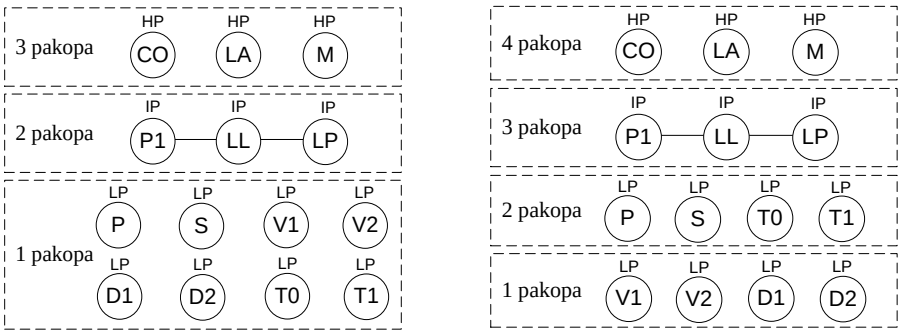
6.3.2. „MP-ReTool“ įrankio įvertinimas

„MP-ReTool“ (Bespalova ir kt., 2015b) įrankis skirtas automatizuotai transformuoti heterogeninę metaprogramą, parašytą PHP programavimo kalba, į atitinkamą daugiapakopę. Šios transformacijos tikslas – iš anksto užprogramuoti galimą metaprogramos adaptavimą pagal vartotojo poreikius.

„MP-ReTool“ įrankio darbui testuoti buvo pasirinktos metaprogramos, generuojančios mokomųjų robotų valdymo programas (naudojami „Lego NXT“ ir „ARDUINO“ robotai). Nagrinėjami 6 uždaviniai (uždavinių modeliai pateikti 4 priede):

4. Roboto kalibravimas;
5. Linijos sekimas;
6. Ornamentų kūrimas;
7. Bėganti eilutė;
8. Šviesos sekimas;
9. Šviesoforas.

Vienpakopės metaprogramos transformavimo į daugiapakopę metu metaparametrų aibė suskirstoma į poaibius. Poaibių skaičius atitinka kuriamos daugiapakopės metaprogramos pakopų skaičių k. Metaparametrų paskirstymas pakopose priklauso nuo uždavinio kontekstinės informacijos ir nuo metaparametrų sąryšių. Kaip buvo minėta 5.4 skyrelyje, „MP-ReTool“ gali dirbti dviem darbo režimais: automatiniu ir pusiau automatiniu. Įrankiui dirbant automatiniu režimu pats įrankis, atsižvelgdamas į probleminės srities požymių modelį, paskaičiuoja pakopų skaičių ir paskirsto metaparametrus jose. Įrankiui dirbant pusiau automatiniu režimu vartotojo paprašoma įvesti norimą pakopų skaičių ir leidžiama paskirstyti metaparametrus jose. 6.4 pav. pateikiamas galimas metaparametrų paskirstymo pakopose pavyzdys, nagrinėjamas ornamentų kūrimo uždavinys.



6.4 pav. Ornamentų kūrimo uždavinys, metaparametrų paskirstymo pakopose pavyzdys: a) automatinis, b) pusiau automatinis

Įrankiui dirbant automatiniu režimu tikrinami metaparametrų sąryšiai ir metaparametrų grupės, turinčios vienodą prioriteto reikšmę. Metaparametrai, turintys vienodą prioriteto reikšmę, sudaro vieną atskirą poaibį. Poaibiui priskirti metaparametrai neturi turėti sąryšių su kito poaibio metaparametrais. Jei toks sąryšis egzistuoja, poaibiai apjungiami į vieną. Kuriamos daugiapakopės metaprogramos pakopų skaičius nustatomas pagal sudarytų metaparametrų poaibių skaičių. 6.4 pav. a) parodytas ornamentų kūrimo uždavinio parametrų paskirstymas pakopose, šiuo atveju bus kuriama trijų pakopų metaprograma.

Kai vartotojas pasirenka pusiau automatinį darbo režimą, jis gali pasirinkti norimą pakopų skaičių ir paskirstyti metaparametrus pakopose (6.4 pav. b)). Tačiau nurodytas pakopų skaičius ir metaparametrų paskirstymas jose turi neprieštarauti transformavimo į daugiapakopę metaprogramą taisyklėms (4.4. skyrelis).

Automatinio režimo metu įrankis „MP-ReTool“ gali sugeneruoti penkių pakopų metaprogamą. Šiuo atveju konteksto modelis papildomas Bloomo taksonomijos lygmenimis, vidutinio prioriteto reikšmė išskaidoma į papildomas reikšmes {IP_L1; IP_L2; IP_L3}, čia L1 – žinios, L2 – supratimas ir L3 – pritaikymas (plačiau žiūrėkite (Štuikys, 2015)). Įrankiui dirbant pusiau automatiniu režimu, vartotojas gali sukurti keturių pakopų metaprogramas. Praktinių uždavinių tyrinėjimas parodė, kad didesnis pakopų skaičius nei keturios yra retai naudojamas.

Metaparametrų paskirstymas pakopose daro įtaką kuriamos daugiapakopės metaprogramos dydžiui. 6.8 lent. pateikiami tyrimų duomenys, parodantys, kaip sukurtų metaprogramų dydis priklauso nuo pakopų skaičiaus ir metaparametrų pasiskirstymo jose (nagrinėjami „LEGO NXT“ uždaviniai). Iš 6.8 lent. pateiktų rezultatų matome, kad daugiapakopės metaprogramos, aukščiausioje pakopoje turinčios daugiau metaparametrų, yra mažesnės. Metaprogramos, žemiausioje pakopoje turinčios daugiausiai metaparametrų, yra didžiausios ir sudėtingos, nes jas kuriant reikia deaktyvuoti daugiau metakonstrukcijų.

6.8 lentelė. Metaprogramos dydžio priklausomybė nuo metaparametrų pasiskirstymo pakopose

Roboto kalibravimas			
Metaparametrai	CO, LA, M, A, LL, LP, S, V, T		
1 pakopos *M* (LOC/KB)	267/10		
Metaparametrų paskirstymo pakopose variantai	4 pakopa: CO 3 pakopa: LA, M 2 pakopa: A, LL, LP 1 pakopa: S, V, T	4 pakopa: CO, LA, M 3 pakopa: A, LL, LP 2 pakopa: S 1 pakopa: V, T	4 pakopa: CO, LA, M 3 pakopa: A, LL, LP 2 pakopa: S, V 1 pakopa: T
4 pakopų *M* (LOC/KB)	286/25,4	286/21,7	286/21
Metaparametrų paskirstymo pakopose variantai	3 pakopa: CO 2 pakopa: LA, M, A, LL, LP 1 pakopa: S, V, T	3 pakopa: CO, LA, M 2 pakopa: A, LL, LP 1 pakopa: S, V, T	3 pakopa: CO, LA, M A, LL, LP 2 pakopa: S, V 1 pakopa: T
3 pakopų *M* (LOC/KB)	278/17,8	274/16,4	277/14,6
Metaparametrų paskirstymo pakopose variantai	2 pakopa: CO 1 pakopa: LA, M, A, LL, LP, S, V, T	2 pakopa: CO, LA, M 1 pakopa: A, LL, LP, S, V, T	2 pakopa: CO, LA, M, A, LL, LP 1 pakopa: S, V, T
2 pakopų *M* (LOC/KB)	278/15,4	268/13,2	268/11,6

Linijos sekimas			
Metaparametrai	CO, LA, M, LP, LL, A, L, S, V		
1 pakopos *M* (LOC/KB)	431/18		
Metaparametrų paskirstymo pakopose variantai	4 pakopa: CO 3 pakopa: LA, M 2 pakopa: LP, LL, A, L 1 pakopa: S, V	4 pakopa: CO, LA 3 pakopa: M 2 pakopa: LP, LL, A; L 1 pakopa: S, V	4 pakopa: CO, LA, M 3 pakopa: LP, LL, A, L 2 pakopa: S 1 pakopa: V
4 pakopų *M* (LOC/KB)	435/39	435/38,8	435/31,8
Metaparametrų paskirstymo pakopose variantai	3 pakopa: CO 2 pakopa: LA, M, LP, LL, A, L 1 pakopa: S, V	3 pakopa: CO, LA, M 2 pakopa: LP, LL, A, L 1 pakopa: S, V	3 pakopa: CO, LA, M LP, LL, A, L 2 pakopa: S 1 pakopa: V
3 pakopų *M* (LOC/KB)	433/27,6	424/25,5	422/22,5
Metaparametrų paskirstymo pakopose variantai	2 pakopa: CO 1 pakopa: LA, M, LP, LL, A, L, S, V	2 pakopa: CO, LA, M 1 pakopa: LP, LL, A, L, S, V	2 pakopa: CO, LA, M, LP, LL, A, L, 1 pakopa: S, V
2 pakopų *M* (LOC/KB)	432/25,5	426/22,6	426/18,4
Ornamentų kūrimas			
Metaparametrai	CO; LA; M; LP; LL; A; L; S; V		
1 pakopos *M* (LOC/KB)	470/19,7		
Metaparametrų paskirstymo pakopose variantai	4 pakopa: CO 3 pakopa: LA, M 2 pakopa: P1, LL, LP 1 pakopa: V1, V2, S, T, D1, D2, P, T1	4 pakopa: CO, LA, M 3 pakopa: P1, LL, LP 2 pakopa: P, S, T, T1 1 pakopa: V1, V2, D1, D2	4 pakopa: CO, LA, M, P1, LL, LP 3 pakopa: P, S, T, T1, 2 pakopa: V1, V2 1 pakopa: D1, D2
4 pakopų *M* (LOC/KB)	472/46,3	450/37,1	443/32,3
Metaparametrų paskirstymo pakopose variantai	3 pakopa: CO 2 pakopa: LA, M 1 pakopa: P1, LL, LP, V1, V2, S, T, D1, D2, P, T1	3 pakopa: CO, LA, M 2 pakopa: P1, LL, LP, 1 pakopa: S, P, T, T1,V1, V2, D1, D2	3 pakopa: CO, LA, M, P1, LL, LP 2 pakopa: P, S T, T1 1 pakopa: V1, V2, D1, D2
3 pakopų *M* (LOC/KB)	462/37,5	448/30,7	443/25,4
Metaparametrų paskirstymo pakopose variantai	2 pakopa: CO LA, M 1 pakopa: P1, LL, LP, V1, V2, S, T, D1, D2, P, T1	2 pakopa: CO, LA, M, P1, LL, LP, 1 pakopa: S, P, T, T1,V1, V2, D1, D2	2 pakopa: CO, LA, M, P1, LL, LP, P, S T, T1 1 pakopa: V1, V2, D1, D2
2 pakopų *M* (LOC/KB)	459/25	444/21,2	442/20,4

M – metaprograma; LOC – programinio kodo eilučių skaičius; KB – programos dydis kilobaitais.

Daugiapakopės metaprogramos vykdymo metu vartotojas kiekvienoje pakopoje parenka reikiamas metaparametrų vertes ir generuoja adaptuotą metaprogramos versiją. 6.9 lent. pateikiamos adaptavimo procese generuojamų žemesnės pakopos metaprogramų charakteristikos.

6.9 lentelė. Iš daugiapakopių metaprogramų generuojamų žemesnės pakopos metaprogramų charakteristikos

M ir jų charakteristikos	Pakopų skaičius $k = 4$				Pakopų skaičius $k = 3$			Pakopų skaičius $k = 2$	
Roboto kalibravimas									
Metaparametrų paskirstymas pakopose	4	3	2	1	3	2	1	2	1
	CO	LA, M	A, LL, LP	S, V, T	CO, LA, M	A, LL, LP	S, V, T	CO, LA, M, A, LL, LP	S, V, T
M_I / M_B LOC (PHP)	11 225	21 180	31 112	37 51	29 178	31 112	37 50	57 123	37 50
M skaičius	1	4	4	45	4	4	45	16	45
PE skaičius	1∗4∗4∗45=720				4∗4∗45= 720			16∗45=720	
Linijos sekimas									
Metaparametrų paskirstymas pakopose	4	3	2	1	3	2	1	2	1
	CO	LA, M	A, LL, LP, L	S, V	CO, LA, M	A, LL, LP, L	S, V	CO,LA, M	A, LL, LP, L, S, V
M_I / M_B LOC (PHP)	11 424	24 386	124 145	42 44	38 386	124 145	42 44	38 388	192 79
M skaičius	1	6	32	12	6	32	12	6	384
PE skaičius	1∗6∗32∗12=2304				6∗32∗12=2304			6∗384=2304	
Ornamentų kūrimas									
Pakopa	4	3	2	1	3	2	1	2	1
Metaparametrų paskirstymas pakopose	CO	LA, M	P1, LL, LP	V1, V2, S, T, D1, D2, P, T1	CO, LA, M	P1, LL, LP	V1, V2, S, T, D1, D2, P, T1	CO, LA, M, P1, LL, LP	V1, V2, S, T, D1, D2, P, T1
M_I / M_B LOC (PHP)	11 461	23 414	73 302	159 89	37 413	73 301	134 114	123 320	133 114
M skaičius	1	4	7	1944	4	7	1944	28	1944
PE skaičius	1∗4∗7∗1944=54432				4∗7∗1944=54432			28∗1944=54432	

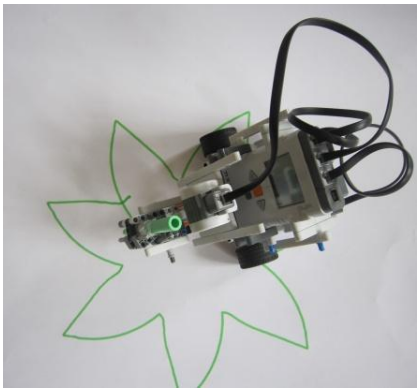
M – metaprograma; M_I – metasąsaja; M_B – metakamienas; LOC – programinio kodo eilučių skaičius; KB – programos dydis kilobaitais; *PE* – tikslo kalbos programos egzempliorius.

Daugiapakopės metaprogramos vykdymo metu generuojama žemesnės pakopos metaprograma, nuo *k* pakopos iki 1 pakopos metaprogramos. Iš 6.9 lent. pateiktų rezultatų matome, kad kiekvienoje pakopoje sukuriama žemesnės pakopos metaprograma yra mažesnė ir paprastesnė, nes dalis metaparametrų yra pakeičiami konstantomis, o įvertinus metaparametrus atmetamos nenaudojamos programinio kodo atšakos.

6.5 pav. pateikiamas vienas iš galimų ornamentų kūrimo uždavinio rezultatų, t. y. sugeneruota tam tikra mokomųjų robotų valdymo programa. 6.5 a) pav. parodytas sugeneruotas tikslo kalbos programos egzempliorius, o 6.5 b) pav. – roboto, vykdančio sugeneruotą programinį kodą, darbo rezultatas.

```
task main(){
//Preparation for drawing
   motor[motorB] = 50;
   wait1Msec(100);
   motor[motorB] = 0;
//Drawing
   for (int j=0; j<7; j++){
      motor[motorC] = 30;
      motor[motorA] = 30;
      wait1Msec(1000);
      motor[motorC] = -30;
      motor[motorA] = 0;
      wait1Msec(1000);
   }
//Drawing of ornament is finished
   motor[motorB] = -50;
   wait1Msec(100);
   motor[motorB] = 0;
}
```

a)



b)

6.5 pav. Ornamentų kūrimo uždavinys: a) sugeneruotas tikslo kalbos programos egzempliorius, b) roboto darbo rezultatas

6.4. Metaprogramų sudėtingumo tyrimas

Sudėtingumo teorija yra informatikos šaka, nagrinėjanti įvairias programų (algoritmų) savybes. Stengiamasi atsakyti į klausimą, kaip palyginti skirtingas programas (algoritmus). Sukurtų metaprogramų įvertinimui darbe buvo naudojamos metaprogramų sudėtingumo metrikos.

Heterogeninių metaprogramų suprantamumas ir skaitomumas yra mažesnis, nes sintaksinės programos konstrukcijos yra sudarytos iš dviejų skirtingų kalbų. Dėl to heterogeninių metaprogramų sudėtingumas yra didesnis nei įprastų programų (Damaševičius, Štuikys, 2010). Metaprogramų sudėtingumas gali būti įvertinamas keletu aspektų:

1. *Informacijos*: metaprograma vertinama kaip simbolių seka (nevertinama sintaksė ir struktūra);
2. *Metakalbos*: sritis aprašoma naudojant tikslo kalbą, o srities variantiškumas aprašomas naudojant metakalbą;
3. *Grafo*: metaprograma yra kaip grafas su vykdymo keliais, kur metaprograma yra šaknis, metakalbos konstrukcijos – mazgai, o lapai yra tikslo kalbos programos egzemplioriai;
4. *Algoritmo*: metaprograma – aukšto lygmens programos specifikacija, kuri yra funkcinių operacijų rinkinys. Operacija gali turėti vieną ar daugiau operandų (metaparametrų);
5. *Pažinimo*: metaprograma kaip skirtingų informacijos vienetų rinkinys. Vienetas gali reikšti metakalbos konstrukciją, jos argumentą ar metaparametrą.

Metaprogramų sudėtingumui vertinti darbe taikomos Štuikio ir Damaševičiaus (2013a) pasiūlytos metaprogramų sudėtingumo metrikos: *Kolmogorovo* (angl.

Relative Kolmogorov Complexity, *RKC*), *Metakalbos turtingumo* (angl. *Metalanguage Richness*, *MR*), *Normalizuoto sudėtingumo* (angl. Normalized Difficulty, *ND*), *Pažinimo sudėtingumo* (angl. *Cognitive Difficulty*, *CD*) ir *Ciklomatinio sudėtingumo* (angl. *Cyclomatic Complexity*, *CC*).

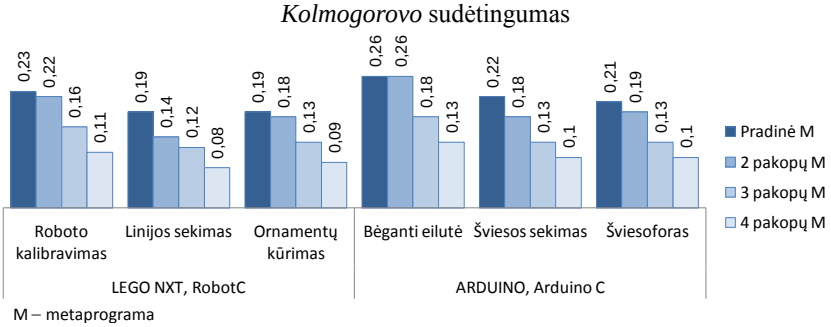
***Kolmogorovo* sudėtingumas**: programos teksto suglaudinimo matas, kuris priklauso nuo teksto atsikartojimo lygio. *Kolmogorovo sudėtingumas* skaičiuojamas kaip suglaudintos programos dydis, padalytas iš nesuglaudintos programos dydžio (6.2):

$$RKC = \frac{\|C(M)\|}{\|M\|}; \tag{6.2}$$

čia $\|M\|$ – metaprogramos dydis, $\|C(M)\|$ – suspaustos metaprogramos dydis.

Metaprogramai suspausti naudojamas glaudinimo algoritmas BWT (angl. *Burrows-Wheeler transforms*).

Didelė RKC vertė rodo, kad yra didelis programinio teksto variantiškumas. Maža RKC vertė rodo, kad metaprogramos kode yra daug pasikartojančių fragmentų, t. y. didesnis sudėtingumas (6.6 pav.).



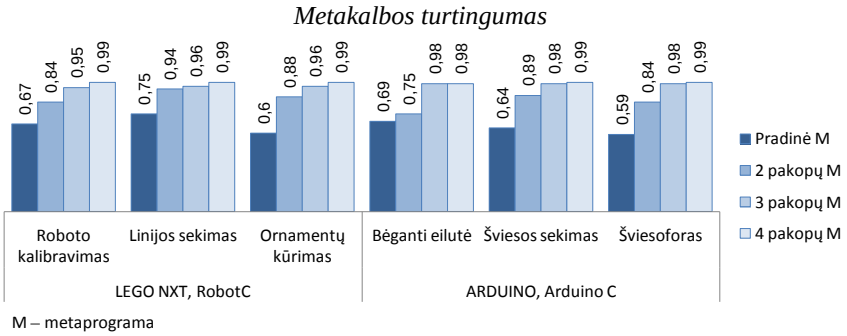
6.6 pav. Metaprogramų *Kolmogorovo* sudėtingumo vertės

Metakalbos turtingumas: metaprogramą galima apibrėžti kaip tikslo kalbos sakinių rinkinį su atitinkamais metaduomenimis. Metaprogramos *metakalbos turtingumas* skaičiuojamas pagal (6.3) formulę:

$$MR = \frac{\sum_{m \in M} \|m\|}{\|M\|}; \tag{6.3}$$

čia $\|M\|$ – metaprogramos dydis, $\|m\|$ – metakalbos konstruktų dydis metaprogramoje.

Didesnė MR vertė parodo, kad metaprogramoje yra daugiau metaduomenų ir jų aprašymas yra sudėtingas (6.7 pav.).

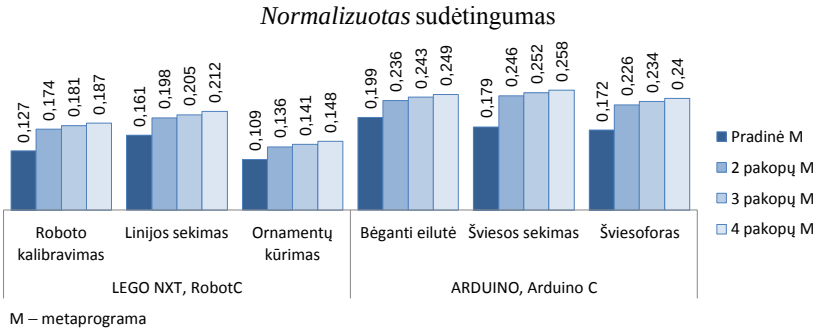


6.7 pav. Metaprogramų *metakalbos turtingumo* vertės

***Normalizuotas* sudėtingumas**: šios metrikos skaičiavimas išvestas iš Halsteado sudėtingumo ir skaičiuojamas pagal (6.4) formulę:

$$ND = \frac{n_1 N_2}{(N_1 + N_2)(n_1 + n_2)}; \qquad (6.4)$$

čia n_1 – metaprogramos skirtingų operatorių skaičius, N_1 – bendras metaprogramos operatorių skaičius, n_2 – metaprogramos skirtingų operandų skaičius, N_2 – bendras metaprogramos operandų skaičius. Didesnė ND vertė rodo, kad metaprograma sudėtinga, t. y. ją sudėtinga suprasti (6.8 pav.).



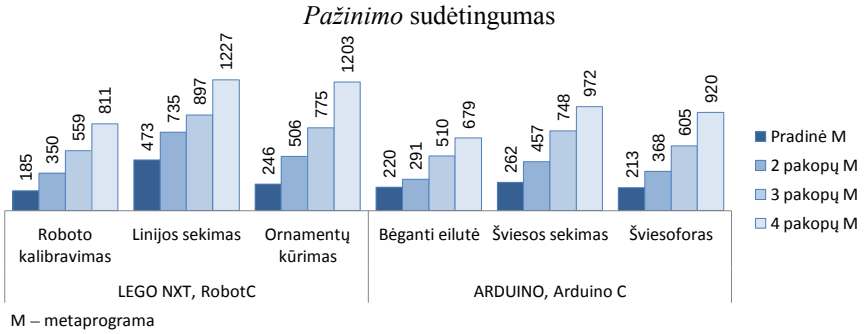
6.8 pav. Metaprogramų *Normalizuotas* sudėtingumas

***Pažinimo* sudėtingumas**: skaičiuojamas kaip metaprogramos maksimalus metalygmenų skaičius (6.5):

$$CD = max\ (P, N_1, N_2); \qquad (6.5)$$

čia P – metaparametrai, N_1 – metakalbos konstrukcijos ir N_2 – jų atitinkami argumentai.

Didėjant metaprogramos sudėtingumui suprantamumas mažėja (6.9 pav.).



6.9 pav. Metaprogramų *Pažinimo* sudėtingumas

Ciklomatinis sudėtingumas: nepriklausomų kelių programos kodo grafe skaičius. Vertinant metaprogramas, jis lygus tikslo kalbos programų egzempliorių skaičiui, kurį gali sugeneruoti metaprograma. Didelė ciklomatinio sudėtingumo vertė rodo, kad metaprograma turi didesnį metaparametrų rinkinį ir sudėtingą metasąsają.

6.5. Išvados

1. Įvertinus metaprogramos kūrimo būdus (rankinį, pusiau automatinį ir automatinį) nustatyta, kad rankiniu būdu kuriant metaprogramas sugaištama daugiausiai laiko. Metaprogramas kuriant pusiau automatiniu būdu sugaištama vidutiniškai 34 % mažiau laiko. Pusiau automatinio ir automatinio kūrimo būdų laikai labai artimi, nes automatinis metaprogramos kūrimo būdas reikalauja didesnių laiko sąnaudų modeliams sukurti.

2. Nustatyta, kad tikslo kalbos bendrąjį programos egzempliorių tikslinga kurti tada, kai žinome, kad metaprograma bus ne kartą kuriama pakartotinai.

3. Atliktas metaprogramos transformavimo į daugiapakopę metaprogramą ekvivalentiškumo tyrimas patvirtino hipotezę, kad metaprogramos specializavimas keičia metaprogramos struktūrą, tačiau išsaugo pradinį metaprogramos funkcionalumą.

4. Nustatyta, kad metaparametrų paskirstymas pakopose turi įtaką kuriamos daugiapakopės metaprogramos dydžiui, tuo pačiu ir sudėtingumui. Daugiapakopės metaprogramos žemiausioje pakopoje turinčios daugiau metaparametrų yra sudėtingesnės, nes jose deaktyvuota daugiau metakonstrukcijų.

5. Atlikus heterogeninių metaprogramų technologinio sudėtingumo tyrimą nustatyta, kad didėjant metaprogramos pakopų skaičiui didėja metaprogramos sudėtingumas pagal visas metrikas. *Pažinimo sudėtingumo* metrikos vertė kiekvienoje aukštesnėje pakopoje auga daugiau nei 50 %. Tai parodo, kad metaprogramą transformuojant į viena pakopa aukštesnę metaprogramą, dvigubai mažėja jos suprantamumas.

7. BAIGIAMASIS ĮVERTINIMAS

Disertacijoje nagrinėjami specifiniai, mažai nagrinėti uždaviniai: požymių modelių transformavimas į heterogenines metaprogramas ir sukurtų metaprogramų transformavimas siekiant jų adaptavimo konkrečiam kontekstui. Heterogeninės metaprogramos kuriamos naudojant bent dvi kalbas: meta ir tikslo. Tokios metaprogramos yra programų generatoriai, kai nustačius metaparametrų reikšmes metakalbos procesorius automatiškai sukuria tikslo kalbos programą.

Daugelis taikomųjų sričių yra heterogeninės, joms būdingas didelis variantiškumas. Realizuojant tokią sritį sukuriama labai daug panašių programų, kurios tarpusavyje skiriasi tik tam tikrais požymiais. Heterogeninis metaprogramavimas gerai tinka tokioms programoms generuoti. Pavyzdžiui, aparatūros, įterptinių sistemų programų, svetainių projektavime, e. mokyme. Plačiau galima skaityti Štuikio ir Damaševičiaus (2013a) monografijoje.

Metaprogramų kūrimas yra sudėtingas intelektualus procesas. Kadangi darbe nagrinėjamas modeliais grįstas metaprogramų kūrimo procesas, svarbų vaidmenį vaidina probleminės ir sprendimo srities analizė ir modeliavimas. Tik turėdami neprieštaringus ir pilnus modelius bei transformavimo taisyklių aprašus galime automatizuoti metaprogramos kūrimo procesą. Tam, kad būtų galima sukurti transformavimo taisykles (o jos yra transformavimo pagrindas), reikėjo formalizuoti tiek probleminės, tiek sprendimo srities (t. y. metaprogramavimo) modelius. Jie darbe detaliai aprašyti, nustatytos jų savybės.

Apibrėžti modelių formalizmai ir požymiais grindžiamų modelių transformavimo taisyklės sudarė sąlygas įrankiui kurti. Taip buvo sukurtas eksperimentinis įrankis „MePAG“, skirtas automatizuotai kurti heterogenines metaprogramas. Sukurtas įrankis buvo ištestuotas. Šiame etape buvo kuriamos metaprogramos, generuojančios mokomųjų robotų valdymo programas.

Pasiūlytas uždavinio sprendimas turi apribojimų. Srities modeliavimo galimybes riboja pasirinktų („FAMILIAR“ ir „SPLOT“) įrankių trūkumai. Taip pat sudėtinga į modelius įvesti programavimo kalbos (meta- ir tikslo) sintaksę, nes kalbos abstrakčios, jos charakterizuojamos dideliu rinkiniu specifinių rodiklių ir jų panaudojimas priklauso nuo uždavinio ir programuotojo patirties. Įrankis „MePAG“ šiuo metu neturi integracijos su modeliavimo įrankiais, todėl modeliams suderinti reikalinga papildoma transformacija. Įrankis yra nepriklausomas nuo tikslo kalbos, tačiau priklausomas nuo metakalbos, todėl įrankį reikia toliau tobulinti išplečiant naudojamų metakalbų aibę.

Modeliuojama sritis gali būti labai didelė. Metaprogramoje gali būti apibendrinti tūkstančiai ar net šimtai tūkstančių generuojamų tikslo kalbos programų egzempliorių. Todėl atsiranda poreikis bendrąsias metaprogramas pritaikyti konkrečiam kontekstui. Todėl darbe buvo sprendžiamas pirminės metaprogramos transformavimo į daugiapakopę uždavinys (*specializavimas* ir *adaptavimas*).

Iki šiol buvo žinomos dviejų pakopų metaprogramos. Šiame darbe, įvedus uždavinio kontekstą, jos buvo apibendrintos daugiapakopiam atvejui. Šiam uždaviniui įgyvendinti buvo teoriškai nustatyta tokios transformacijos egzistavimo

sąlyga, apskaičiuotas maksimalus galimas pakopų skaičius, surasta metakonstrukcijų deaktyvavimo indekso reikšmės priklausomybė nuo pakopos laipsnio.

Formaliai apibrėžtos daugiapakopės metaprogramos ir jų elementai, savybės, vienpakopės metaprogramos transformavimo į daugiapakopę taisyklės ir metaprogramų transformavimo ekvivalentiškumo tyrimo rezultatai sudarė sąlygas įrankiui kurti. Dėl to pasekoje buvo sukurtas eksperimentinis įrankis MP-ReTool, skirtas automatizuotai kurti daugiapakopes metaprogramas. Sukurtas įrankis yra nepriklausomas nuo tikslo kalbos, tačiau priklausomas nuo metakalbos (transformuoja PHP programavimo kalba parašytas metaprogramas), todėl reikia įrankį toliau tobulinti išplečiant naudojamų metakalbų aibę.

Metaprogramos specializavimo dėka sukuriama daugiapakopė metaprograma, kuri leidžia vartotojui susikurti metaprogramą, adaptuotą prie jo konteksto. Adaptuota metaprograma nuo pirminės skiriasi tuo, kad joje susiaurinta metaparametrų erdvė, t. y. įvertinti aukštesnio lygmens metaparametrai ir juos įvertinus pašalintos nenaudojamos programinio kodo atšakos.

Sprendžiant specializavimo-adaptavimo uždavinį buvo nagrinėjamos metaprogramos, pritaikytos mokymo turiniui kurti (mokomųjų robotų valdymo programos). Tokiose metaprogramose aukščiausios pakopos metaparametrai yra skirti adaptacijai prie mokytojo konteksto, o žemesnės pakopos metaparametrai skirti besimokantiesiems prisitaikyti mokymosi kontekstą pagal mokymosi galimybes ir poreikius.

Įrankių panaudojimas kuriant ar transformuojant metaprogramas leidžia dirbti efektyviau, palengvina metaprogramų modeliavimo bei kūrimo procesą. Metodika buvo realiai išbandyta ir įvertinta, tačiau ji reikalauja tolimesnių tyrimų: konteksto modelio pritaikymo platesniu mastu, sukurtų specializuotų įrankių tobulinimo, robotų valdymo programų generavimo ir pritaikymo praktikoje.

IŠVADOS

1. Atlikus literatūros šaltinių analizę nustatyta, kad:
   - esminis reikalavimas kuriamiems probleminės srities modeliams ir jų transformavimui yra *bendrybių-skirtybių ir jų sąveikos identifikavimas, kaip tiriamosios srities variantiškumo išraiška*;
   - heterogeninis metaprogramavimas įgalina pasiekti programų kūrimo automatizavimo tikslus, o programų generatoriai realizuoja generatyvinį pakartotinį panaudojimą.
2. Sukurti bendrybes ir skirtybes aprašantys probleminės ir sprendimų srities formalizuoti požymių modeliai, jų sąryšiai, savybės ir požymiais grindžiamų modelių transformavimo taisyklės įgalino automatizuotai kurti heterogenines metaprogramas.
3. Pritaikytas *Futamuros* programų specializavimo uždavinio *interpretavimas* įgalino suformuluoti dviejų pakopų metaprogramų specializavimo uždavinį, po to, pastarajam pritaikius apgrąžos principą, suformuluotas pradinės (vienpakopės) metaprogramos daugiapakopio transformavimo (t. y. specializavimo) uždavinys. Kadangi dviejų pakopų metaprogramos modelis (kitoje notacijoje) jau buvo žinomas, tai šis apibendrinimas yra moksliškai naujas.
4. Daugiapakopio transformavimo esminis teorinis rezultatas apibrėžiamas taip:
   - nustatyta apibendrinto specializacijos uždavinio išsprendžiamumo sąlyga, t. y. „*uždavinys išsprendžiamas tada ir tik tada, jei vienpakopės metaprogramos metasąsajos svorinis grafas $G(P^w,U)$ nėra jungusis grafas*“ (čia P – metaparametrų aibė, U – briaunų aibė, vaizduojanti metaparametrų sąveiką, w – neraiškiosios logikos kintamasis, aprašantis metaparametro kontekstą);
   - nustatyta, kad *maksimalus pakopų skaičius* lygus metasąsajos grafo *visuminiam komponenčių skaičiui* (t. y. jungias ir nejungias komponentes kartu paėmus);
   - uždaviniui išspręsti pritaikytas metakonstrukcijų deaktyvacijos-aktyvacijos principas ir nustatyta pakopos deaktyvacijos indekso (DI) reikšmė duotai metakalbai, t. y. pakopoje k $DI = 0$, pakopoje (k-1) $DI = 1$, o žemesnėse pakopose $DI = \sum_{a=0}^{k-2} 2^a$.
5. Pasiūlyti, išbandyti ir pritaikyti įrankiai (vieni– „FAMILIAR“ ir „SPLOT“, parinkti, kiti – „MePAG“ ir „MP-ReTool“ sukurti), palaikantys pilną metaprogramos gyvavimo ciklą: *modeliavimo, modelių transformavimo, metaprogramų transformavimo į daugiapakopes*.
6. Nustatyta, kad naujai sukurto įrankio „MePAG“ panaudojimas leido metaprogramas kurti efektyviau. Metaprogramas kuriant pusiau automatiniu būdu sugaištama vidutiniškai 34 % mažiau laiko nei kuriant rankiniu būdu. Pusiau automatinio ir automatinio kūrimo būdų laikai labai artimi, nes automatinis būdas reikalauja didesnių laiko sąnaudų modeliams sukurti.

Nustatyta, kad tikslo kalbos bendrinį programos egzempliorių tikslinga kurti tada, kai žinome, kad metaprograma bus ne kartą kuriama pakartotinai.

7. Atliktas metaprogramos transformavimo į daugiapakopę metaprogramą ekvivalentiškumo tyrimas patvirtino hipotezę, kad metaprogramos specializavimas keičia metaprogramos struktūrą, tačiau išsaugo pradinį metaprogramos funkcionalumą.
8. Nustatyta, kad naujai sukurtas įrankis „MP-ReTool“ leidžia automatizuotai transformuoti vienpakopę metaprogramą į daugiapakopę. Šios transformacijos dėka sukuriama specializuota metaprograma, kuri įgalina metaprogramas adaptuoti prie skirtingo konteksto.
9. Atlikus heterogeninių metaprogramų technologinio sudėtingumo tyrimą nustatyta, kad didėjant metaprogramos pakopų skaičiui didėja metaprogramos sudėtingumas. *Pažinimo sudėtingumo* metrikos vertė kiekvienoje aukštesnėje pakopoje auga daugiau nei 50 %. Tai parodo, kad metaprogramą transformuojant į viena pakopa aukštesnę metaprogramą, dvigubai mažėja jos suprantamumas.

LITERATŪRA

1. ABARCA, M. G., ALARCON, R. A., BARRIA, R. and FULLER, D. Context-based e-learning composition and adaptation. In: *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*. Springer Berlin Heidelberg, 2006. p. 1976-1985.
2. ABLONSKIS, L. A suggestion for improvement of program code generators. *Perspectives in Business Information Research-BIR'2007*, 2007, 1.
3. ABRAHAMS, D. and GURTOVOY, A. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.
4. ACHER, M., COLLET, P., LAHIRE, P. and FRANCE, R. B. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 2013, 78.6: 657-681.
5. ACM SIGPLAN 2014. *Workshop on Partial Evaluation and Program Manipulation* (PEPM'14). 2014. [žiūrėta 2015-03-20]. Prieiga per internetą: http://www.program-transformation.org/PEPM14.
6. AHO, A. V., SETHI, R. and ULLMAN, J. D. *Compilers*: *Principles, Techniques, and Tools* (*2nd Edition*). Addison Wesley, 2006.
7. AIRASIAN, P. W., CRUIKSHANK, K. A., MAYER, R. E., PINTRICH, P. R., RATHS, J. and WITTROCK, M. C. A taxonomy for learning, teaching, and assessing: A revision of Bloom's Taxonomy of Educational Objectives. *Anderson LW and Krathwohl DR. New York: Addison Wesley Longmann*, 2001.
8. AIZENBUD-RESHEF, N., NOLAN, B. T., RUBIN, J. and SHAHAM-GAFNI, Y. Model traceability. *IBM Systems Journal*, 2006, 45.3: 515-526.
9. Altova. *MapForce – Graphical Data Mapping, Conversion, and Integration Tool*. 2015. [žiūrėta 2015-02-27]. Prieiga per internetą: http://www.altova.com/mapforce.html.
10. APEL, S., BATORY, D., KÄSTNER, C. and SAAKE, G. *Feature-Oriented Software Product Lines*. Berlin: Springer, 2013.
11. APEL, S., KÄSTNER, C. and BATORY, D. Program refactoring using functional aspects. In: *Proceedings of the 7th international conference on Generative programming and component engineering*. ACM, 2008. p. 161-170.
12. ATTARDI, G. and CISTERNINO, A. Reflection support by means of template metaprogramming. In: *Generative and Component-Based Software Engineering*. Springer Berlin Heidelberg, 2001. p. 118-127.
13. BACHELET, B., MAHUL, A. and YON, L. Template metaprogramming techniques for concept-based specialization. *Scientific Programming*, 2013, 21.1-2: 43-61.
14. BALDAUF, M., DUSTDAR, S. and ROSENBERG, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2007, 2.4: 263-277.
15. BAREISA, E., JUSAS, V., MOTIEJUNAS, K. and SEINAUSKAS, R. Functional test generation remote tool. In: *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*. IEEE, 2005. p. 192-195.
16. BATORY, D. A tutorial on feature oriented programming and the ahead tool suite. In: *Generative and Transformational Techniques in Software Engineering*. Springer Berlin Heidelberg, 2006. p. 3-35.
17. BATORY, D. The road to utopia: A future for generative programming. In: *Domain-Specific Program Generation*. Springer Berlin Heidelberg, 2004. p. 1-18.
18. BENAVIDES, D., SEGURA, S. and RUIZ-CORTÉS, A. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 2010, 35.6: 615-636.

19. BERGER, M. and TRATT, L. Program logics for homogeneous meta-programming. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer Berlin Heidelberg, 2010. p. 64-81.
20. BESPALOVA, K., BURBAITĖ, R.. and ŠTUIKYS, V. MePAG tools. 2015a. [žiūrėta 2015-03-27]. Prieiga per internetą: http://proin.ktu.lt/metaprogram/ MePAG/.
21. BESPALOVA, K., BURBAITĖ, R. and ŠTUIKYS, V. MP-ReTool tools. 2015b. [žiūrėta 2015-03-27]. Prieiga per internetą: http://proin.ktu.lt/metaprogram/ MP-ReTool/.
22. BETTINI, C., BRDICZKA, O., HENRICKSEN, K., INDULSKA, J., NICKLAS, D., RANGANATHAN, A. and RIBONI, D. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 2010, 6.2: 161-180.
23. BEUCHE, D. Modeling and building software product lines with pure:: variants. In: *Proceedings of the 16th International Software Product Line Conference-Volume 2*. ACM, 2012. p. 255-255.
24. BIEHL, M. Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*, 2010.
25. BONTEMPS, Y., HEYMANS, P., SCHOBBENS, P. Y. and TRIGAUX, J. C. Semantics of FODA feature diagrams. In: *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation–Towards Tool Support*. 2004. p. 48-58.
26. BÖRGER, E. High level systemdesign and analysis using abstract state machines. In: *Applied Formal Methods—FM-Trends 98*. Springer Berlin Heidelberg, 1999. p. 1-43.
27. BÖRGER, E. The abstract state machines method for high-level systemdesign and analysis. In: *Formal Methods: State of the Art and New Directions*. Springer London, 2010. p. 79-116.
28. BOS, J. V. D., HILLS, M., KLINT, P., Van Der STORM, T. and VINJU, J. J. Rascal: From algebraic specification to meta-programming. *arXiv preprint arXiv:1107.0064*, 2011.
29. BOSCH, J. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.
30. BOUCHER, Q., CLASSEN, A., FABER, P. and HEYMANS, P. Introducing TVL, a text-based feature modelling language. In: *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January*. 2010. p. 27-29.
31. BRÉZILLON, P. Context dynamic and explanation in contextual graphs. In: *Modeling and Using Context*. Springer Berlin Heidelberg, 2003. p. 94-106.
32. BROWN, A. An introduction to model driven architecture. 2004.
33. BURBAITĖ, R. *Išplėstiniai generatyviniai mokymosi objektai informatikos mokymuisi: koncepcija, modeliai ir realizacija*. 2014. PhD. Kauno technologijos universitetas. [žiūrėta 2015-04-26]. Prieiga per internetą: http://vddb.library.lt/fedora /get/LT-eLABa-0001:E.02~2014~D_20141105_161316-71164/DS.005.0.01.ETD
34. BURBAITE, R. ir BESPALOVA, K. Model-driven processes and tools to design GLO for CS education. In: *Computers in Education (SIIE), 2014 International Symposium on*. IEEE, 2014. p. 139-144.
35. BURBAITE, R., Bespalova, K., Damaševičius, R. and Štuikys, V. Context Aware Generative Learning Objects for Teaching Computer Science. *International Journal of Engineering Education*, 2014, 30.4: 929-936.
36. BURBAITĖ, R., DAMAŠEVIČIUS, R. and ŠTUIKYS, V. Teaching of Computer Science Topics Using Meta-Programming-Based GLOs and LEGO Robots. *Informatics in Education-An International Journal*, 2013, Vol12_1: 125-142.

37. CAPILLA, R., BOSCH, J. and KANG, K. C. *Systems and Software Variability Management*. Springer, 2013.
38. CHAPMAN, J. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 2009, 228: 21-36.
39. CHEN, C. and XI, H. Meta-programming through typeful code representation. *Journal of functional programming*, 2005, 15.06: 797-835.
40. CHEVERST, K., DAVIES, N., MITCHELL, K. and SMITH, P. Providing tailored (context-aware) information to city visitors. In: *Adaptive Hypermedia and Adaptive Web-Based Systems*. Springer Berlin Heidelberg, 2000. p. 73-85.
41. CHLIPALA, A. Ur: statically-typed metaprogramming with type-level record computation. In: *ACM Sigplan Notices*. ACM, 2010. p. 122-133.
42. CLARK, T., SAMMUT, P. and WILLANS, J. Applied metamodelling: a foundation for language driven development. 2008.
43. COLLET, P. and LAHIRE, P. Feature modeling and separation of concerns with FAMILIAR. In: *Comparing Requirements Modeling Approaches Workshop (CMA@ RE), 2013 International*. IEEE, 2013. p. 13-18.
44. CORDY, J. R. and SARKAR, M. S. Metaprogram implementation by second order source transformation. In: *Workshop at Generative Programming and Component Engineering Conference*. 2004. p. 5-6.
45. CVL Wiki. *Common Variability Language Wiki*. 2012. [žiūrėta 2015-02-08]. Prieiga per internetą: http://www.omgwiki.org/variability/doku.php.
46. CZARNECKI, K., EISENECKER, U., GLÜCK, R., VANDEVOORDE, D. and VELDHUIZEN, T. Generative programming and active libraries. In: *Generic Programming*. Springer Berlin Heidelberg, 2000. p. 25-39.
47. CZARNECKI, K., BEDNASCH, T., UNGER, P. and EISENECKER, U. Generative programming for embedded software: An industrial experience report. In: *Generative Programming and Component Engineering*. Springer Berlin Heidelberg, 2002. p. 156-172.
48. CZARNECKI, K. and EISENECKER, U. W. Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, 2000, 15.
49. CZARNECKI, K. and HELSEN, S. Classification of model transformation approaches. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. 2003. p. 1-17.
50. CZARNECKI, K. and HELSEN, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 2006, 45.3: 621-645.
51. CZARNECKI, K., HELSEN, S. and EISENECKER, U. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 2005, 10.1: 7-29.
52. DAGIENĖ, V., GRIGAS, G. and JEVSIKOVA, T. *Enciklopedinis kompiuterijos žodynas. II papildytas leidimas*. 2009. [žiūrėta 2015-03-31]. Prieiga per internetą: http://www.likit.lt/term/enc.html.
53. DAMAŠEVIČIUS, R. and ŠTUIKYS, V. Metrics for evaluation of metaprogram complexity. *Computer Science and Information Systems*, 2010, 7.4: 769-787.
54. DAMAŠEVIČIUS, R. and ŠTUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 2008, 37.2: 124-132.
55. DAROVSKY, A. *Finite State Machine Editor*. 2003. [žiūrėta 2015-02-03]. Prieiga per internetą: http://fsme.sourceforge.net.
56. DEY, A. K., ABOWD, G. D. and SALBER, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 2001, 16.2: 97-166.

57. DEVRIESE, D. and PIESSENS, F. Typed syntactic meta-programming. In: *ACM SIGPLAN Notices*. ACM, 2013. p. 73-86.
58. DIAS DE FIGUEIREDO, A. Learning Contexts: a Blueprint for Research. *Digital Education Review*, 2010, 11: 127-139.
59. DOUENCE, R., FRADET, P. and SÜDHOLT, M. A framework for the detection and resolution of aspect interactions. In: *Generative Programming and Component Engineering*. Springer Berlin Heidelberg, 2002. p. 173-188.
60. DOURISH, P. What we talk about when we talk about context. *Personal and ubiquitous computing*, 2004, 8.1: 19-30.
61. Du BOIS, B., Van GORP, P., AMSEL, A., Van EETVELDE, N., STENTEN, H., DEMEYER, S. and MENS, T. A discussion of refactoring in research and practice. *Reporte Técnico. Universidad de Antwerpen, Bélgica*, 2004.
62. ERIKSSON, M., BÖRSTLER, J. and BORG, K. The PLUSS approach–domain modeling with features, use cases and use case realizations. In: *Software Product Lines*. Springer Berlin Heidelberg, 2005. p. 33-44.
63. FAMILIAR. *FAMILIAR Project*.2009. [žiūrėta 2015-02-08]. Prieiga per internetą: http://familiar-project.github.io/.
64. FeatureMapper. *Mapping Features to Models*. 2007. [žiūrėta 2015-02-08]. Prieiga per internetą: http://featuremapper.org/.
65. FERRÉ, X. and VEGAS, S. An evaluation of domain analysis methods. In: *4th CAiSE/IFIP8*. 1999.
66. FIORAVANTI, F., PETTOROSSI, A., PROIETTI, M. and SENNI, V. Program transformation for development, verification, and synthesis of programs. *Intelligenza Artificiale*, 2011, 5.1: 119-125.
67. FMP. *FMP: Feature Modeling Plug-in*. 2005. [žiūrėta 2015-02-08]. Prieiga per internetą: http://gp.uwaterloo.ca/fmp.
68. FOWLER, M. Refactoring: Improving the Design of Existing Code. 1997
69. FOWLER, M., BECK, K., BRANT, J., OPDYKE, W. and ROBERTS, D. Refactoring: Improving the Design of Existing Code, Addison Wesley. 1999. [žiūrėta 2015-03-31]. Prieiga per internetą: http://www.refactoring.com/.
70. FRAKES, W., PRIETO, R. and FOX, C. DARE: Domain analysis and reuse environment. *Annals of Software Engineering*, 1998, 5.1: 125-141.
71. FRITZSON, P., POP, A. and SJÖLUND, M. Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0. 2011.
72. FUTAMURA, Y. Partial evaluation of computation process--an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 1999, 12.4: 381-391.
73. GABRYSIAK, G., MARR, S. and MENGE, F. Meta Programming and Reflection in PHP. 2005. [žiūrėta 2015-03-31]. Prieiga per internetą: http://instantsvc. sourceforge.net/docs/metaprogramming-and-reflection-with-php-paper.pdf.
74. GAZAGNAIRE, T. and MADHAVAPEDDY, A. Dynamics for ML using Meta-Programming. *Electronic Notes in Theoretical Computer Science*, 2011, 264.5: 3-21.
75. Gears. *BigLever. Pragmatic Product Line Engineering Solutions for Systems and Software*. 2012. [žiūrėta 2015-02-08]. Prieiga per internetą: http://www.biglever.com/.
76. GHEYI, R., MASSONI, T. and BORBA, P. Automatically Checking Feature Model Refactorings. *J. UCS*, 2011, 17.5: 684-711.
77. GIACOBAZZI, R., JONES, N. D. and MASTROENI, I. Obfuscation by partial evaluation of distorted interpreters. In: *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*. ACM, 2012. p. 63-72.
78. GLASS, R. L. Some thoughts on automatic code generation. *ACM SIGMIS Database*, 1996, 27.2: 16-18.

79. GOLOVESHIN, A. *Converter Visio2Switch*. 2002. [žiūrėta 2015-03-31]. Prieiga per internetą: http://is.ifmo.ru/progeny/visio2switch/.
80. GOMAA, H. and WEBBER, D. L. Modeling adaptive and evolvable software product lines using the variation point model. In: *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*. IEEE, 2004. p. 10.
81. GREENWALD, I. D. and KANE, M. The share 709 system: programming and modification. *Journal of the ACM (JACM)*, 1959, 6.2: 128-133.
82. GRISS, M. L., FAVARO, J. and D'ALESSANDRO, M. Integrating feature modeling with the RSEB. In: *Software Reuse, 1998. Proceedings. Fifth International Conference on*. IEEE, 1998. p. 76-85.
83. GUROV, V. S., MAZIN, M. A., NARVSKY, A. S. and SHALYTO, A. A. Unimod: Method and tool for development of reactive object-oriented programs with explicit states emphasis. *Proceedings of St. Petersburg IEEE Chapters*, 2005, 2: 106-110.
84. HAAKE, J. M., HUSSEIN, T., JOOP, B., LUKOSCH, S., VEIEL, D. and ZIEGLER, J. Modeling and exploiting context for adaptive collaboration. *International Journal of Cooperative Information Systems*, 2010, 19.01n02: 71-120.
85. HARMAN, M. Why Source Code Analysis and Manipulation Will Always be Important. In: *Source Code Analysis and Manipulation, SCAM*. 2010. p. 7-19.
86. HARMAN, M. and HIERONS, R. An overview of program slicing. *Software Focus*, 2001, 2.3: 85-92.
87. HARSU, M. *A survey on domain engineering*. Tampere University of Technology, 2002.
88. HARTMANN, H. and TREW, T. Using feature diagrams with context variability to model multiple product lines for software supply chains. In: *Software Product Line Conference, 2008. SPLC'08. 12th International*. IEEE, 2008. p. 12-21.
89. HAZZARD, K. and BOCK, J. *Metaprogramming in. NET*. Manning Pub, 2013.
90. HERRINGTON, J. *Code generation in action*. Manning Publications Co., 2003.
91. HÖFFERER, P. Achieving Business Process Model Interoperability Using Metamodels and Ontologies. In: *ECIS*. 2007. p. 1620-1631.
92. HUANG, S. S., ZOOK, D. and SMARAGDAKIS, Y. Domain-specific languages and program generation with meta-AspectJ. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2008, 18.2: 6.
93. HUBAUX, A., TUN, T. T. and HEYMANS, P. Separation of concerns in feature diagram languages: A systematic survey. *ACM Computing Surveys (CSUR)*, 2013, 45.4: 51.
94. INOUE, J. and TAHA, W. Reasoning about multi-stage programs. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2012. p. 357-376.
95. JAMESON, A. Systems that adapt to their users: An integrative Overview. In: *Tutorial presented at 9th International Conference on User Modelling. Johnstown, PA, USA*. 2003.
96. JARING, M. and BOSCH, J. A taxonomy and hierarchy of variability dependencies in software product family engineering. In: *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. IEEE, 2004. p. 356-361.
97. JONES, N. D. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 1996, 28.3: 480-503.
98. JONES, N. D., GOMARD, C. K. and SESTOFT, P. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

99. JONES, N. D., SESTOFT, P. and SØNDERGAARD, H. An experiment in partial evaluation: the generation of a compiler generator. In: *Rewriting techniques and applications*. Springer Berlin Heidelberg, 1985. p. 124-140.
100. JOUAULT, F., VANHOOFF, B., BRUNELIERE, H., DOUX, G., BERBERS, Y. and BÉZIVIN, J. Inter-DSL coordination support by combining megamodeling and model weaving. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010. p. 2011-2018.
101. KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E. and PETERSON, A. S. *Feature-oriented domain analysis (FODA) feasibility study*. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1990.
102. KANG, K. C., LEE, J. and DONOHOE, P. Feature-oriented product line engineering. *IEEE software*, 2002, 19.4: 58-65.
103. KARAGIANNIS, D. and KÜHN, H. Metamodelling platforms. In: *EC-Web*. 2002. p. 182.
104. KÄSTNER, C., THÜM, T., SAAKE, G., FEIGENSPAN, J., LEICH, T., WIELGORZ, F. and APEL, S. FeatureIDE: A tool framework for feature-oriented software development. In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009. p. 611-614.
105. KELL, S. A Survey of Practical Software Adaptation Techniques. *J. UCS*, 2008, 14.13: 2110-2157.
106. KIM, M., ZIMMERMANN, T. and NAGAPPAN, N. A field study of refactoring challenges and benefits. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012. p. 50.
107. KITZELMANN, E. Inductive programming: A survey of program synthesis techniques. In: *Approaches and Applications of Inductive Programming*. Springer Berlin Heidelberg, 2010. p. 50-73.
108. KOLLÁR, J., FORGÁČ, M. and PORUBÄN, J. Adaptiveness of software systems using reflection. *Acta Electrotechnica et Informatica No*, 2007, 7.1: 3.
109. KOLOVOS, D. S., PAIGE, R. F. and POLACK, F. AC. The grand challenge of scalability for model driven engineering. In: *Models in Software Engineering*. Springer Berlin Heidelberg, 2009. p. 48-53.
110. KRUEGER, C. W. Variation management for software production lines. In: *Software Product Lines*. Springer Berlin Heidelberg, 2002. p. 37-48.
111. LAGUNA, M. A., MARQUÉS, J. M. and RODRIGUEZ-CANO, G. Feature diagram formalization based on directed hypergraphs. *Computer Science and Information Systems*, 2011, 8.3: 611-633.
112. LE MEUR, A. F., LAWALL, J. L. and CONSEL, C. Towards bridging the gap between programming languages and partial evaluation. In: *ACM SIGPLAN Notices*. ACM, 2002. p. 9-18.
113. LEACH, R. J. *Software Reuse: Methods, Models, Costs*. AfterMath, 2012.
114. LEUSCHEL, M. and VIDAL, G. Fast offline partial evaluation of logic programs. *Information and Computation*, 2014, 235: 70-97.
115. LIU, J., BATORY, D. and LENGAUER, C. Feature oriented refactoring of legacy applications. In: *Proceedings of the 28th international conference on Software engineering*. ACM, 2006. p. 112-121.
116. LÖWE, W. and NOGA, M. L. Metaprogramming applied to web component deployment. *Electronic Notes in Theoretical Computer Science*, 2002, 65.4: 106-116.
117. LUDWIG, A. and HEUZEROTH, D. Metaprogramming in the Large. In: *Generative and Component-Based Software Engineering*. Springer Berlin Heidelberg, 2001. p. 179-188.

118. MAGALHÃES, J. P., DIJKSTRA, A., JEURING, J. and LÖH, A. *A generic deriving mechanism for Haskell*. ACM, 2010.
119. MagicDraw. No Magic. 2015. [žiūrėta 2015-06-16]. Prieiga per internetą: http://www.nomagic.com/.
120. MAINLAND, G. Explicitly heterogeneous metaprogramming with MetaHaskell. In: *ACM SIGPLAN Notices*. ACM, 2012. p. 311-322.
121. MALIK, N. Proposed Classification Approach for Software Component Reuse. *International Journal of Electronics and Computer Science Engineering (IJECSE, ISSN: 2277-1956)*, 2012, 1.04: 1993-1999.
122. MARLET, R. *Program Specialization*. John Wiley & Sons, 2013.
123. MCILROY, M. D. Mass-produced software components. In: *Proceedings of the 1st International Conference on Software Engineering, Garmisch, Germany*. 1968.
124. MEHMOOD, A. and JAWAWI, D. NA. Aspect-Oriented Code Generation for Integration of Aspect Orientation and Model-Driven Engineering. *integration*, 2013, 7.2.
125. MELLODGE, P. and RUSSELL, I. Using the arduino platform to enhance student learning experiences. In: *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. ACM, 2013. p. 338-338.
126. MENDONCA, M., BRANCO, M. and COWAN, D. SPLOT: software product lines online tools. In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 2009. p. 761-762.
127. MENS, T. On the complexity of software systems. *Computer*, 2012, 45.8: 0079-81.
128. MENS, T. and TOURWÉ, T. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 2004, 30.2: 126-139.
129. MENS, T. and VAN GORP, P. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 2006, 152: 125-142.
130. MIAO, W. and SIEK, J. Compile-time reflection and metaprogramming for Java. In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. ACM, 2014. p. 27-37.
131. MINSKY, M. A framework for representing knowledge. 1974.
132. MOIZ, S. A. and RIZWANULLAH, M. Model based Software Develeopment: Issues & Challenges. *arXiv preprint arXiv:1203.1314*, 2012.
133. MURAKAMI, M. An application of partial evaluation of communicating processes to systemsecurity. *International Journal in Foundations of Computer Science & Technology (IJFCST)*, 2012, 2.4.
134. MUSSET, J., JULIOT, É., LACRAMPE, S., PIERS, W., BRUN, C., GOUBET, L. and ALLILAIRE, F. Acceleo user guide. 2006.
135. NANEVSKI, A. *Meta-programming with names and necessity*. ACM, 2002.
136. NEIGHBORS, J. M. The Draco approach to constructing software from reusable components. *Software Engineering, IEEE Transactions on*, 1984, 5: 564-574.
137. NEIGHBORS, J. M. *Software construction using components*. 1980. PhD Thesis. University of California, Irvine. [žiūrėta 2015-04-25]. Prieiga per internetą: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.4397&rep=rep1&type=pdf.
138. NetBeans. NetBeans IDE - The Smarter and Faster Way to Code. 2015. [žiūrėta 2015-02-21]. Prieiga per internetą: https://netbeans.org/features/index.html.
139. OMG. Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation. 2014. [žiūrėta 2015-01-26]. Prieiga per internetą: http://www.omg.org/spec/QVT/index.htm.

140. OMG-MDA. *Model Driven Architecture*. 2014. [žiūrėta 2014-01-23]. Prieiga per internetą: http://www.omg.org/mda/.
141. OPPERMANN, R. Adaptively supported adaptability. *International Journal of Human-Computer Studies*, 1994, 40.3: 455-472.
142. OREIZY, P., GORLICK, M. M., TAYLOR, R. N., HEIMBIGNER, D., JOHNSON, G., MEDVIDOVIC, N., and WOLF, A. L. An architecture-based approach to self-adaptive software. *IEEE Intelligent systems*, 1999, 14.3: 54-62.
143. PAPOTTI, P. E., DO PRADO, A. F. and DE SOUZA, W. L. Reducing time and effort in legacy systems reengineering to MDD using metaprogramming. In: *Proceedings of the 2012 ACM Research in Applied Computation Symposium*. ACM, 2012. p. 348-355.
144. PARK, J., YOUN, H. and LEE, E. An Automatic Code Generation for Self-Healing. *J. Inf. Sci. Eng.*, 2009, 25.6: 1753-1781.
145. PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 1972, 15.12: 1053-1058.
146. PASALIC, E. *The role of type equality in meta-programming*. 2004. PhD. Oregon Health & Science University. [žiūrėta 2015-04-25]. Prieiga per internetą: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.214.826&rep=rep1&type=pdf.
147. PINTO, S., CASTRO, T., MENDES, J., LOPES, S., EKPANYAPONG, M. and TAVARES, A. Exploiting Template Metaprogramming to customize an object-oriented operating system. In: *Industrial Electronics (ISIE), 2013 IEEE International Symposium on*. IEEE, 2013. p. 1-6.
148. PLUM. *Product Line Unified Modeller*. 2011. [žiūrėta 2015-02-07]. Prieiga per internetą: http://www.esi.es/plum/index.php.
149. POHL, K., BÖCKLE, G., VAN DER LINDEN, F. J. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.
150. PORKOLÁB, Z., MIHALICZA, J., PATAKI, N. and Sipos, Á. Analysis of profiling techniques for C++ template metaprograms. *Ann. Univ. Sci. Budapest. Sect. Comput*, 2009, 30: 97-116.
151. PRUSINKIEWICZ, P. and LINDENMAYER, A. The algorithmic beauty of plants. 1990. New York, Springer-Verlag.
152. QURESHI, M. R. J. and IKRAM, J. S. Proposal of Enhanced Extreme Programming Model. 2015.
153. RADOSEVIC, D., OREHOVACKI, T. and MAGDALENIC, I. Towards software autogeneration. In: *MIPRO, 2012 Proceedings of the 35th International Convention*. IEEE, 2012. p. 1076-1081.
154. ReSharper. *How ReSharper Helps Visual Studio Users*. 2015. [žiūrėta 2015-02-21]. Prieiga per internetą: http://www.jetbrains.com/resharper/index.html.
155. RIEBISCH, M. Towards a more precise definition of feature models. *Modelling Variability for Object-Oriented Product Lines*, 2003, 64-76.
156. Rise. RISE Visual Modeling. 2015. [žiūrėta 2015-02-27]. Prieiga per internetą: https://visualstudiogallery.msdn.microsoft.com/bb041300-1468-441d-ac1b-ead01301c41a.
157. RobotC. RobotC–Improved movement. 2007. *Robotics Academy*. [žiūrėta 2015-03-27]. Prieiga per internetą: https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/ROBOTC%20- %20Improved%20Movement.pdf
158. RUBIO, M. A., HIERRO, C. M. and PABLO, Á. P. M. Using arduino to enhance computer programming courses in science and engineering. In: *Proceedings of the EDULEARN13 Conference*. 2013. p. 5127-5133.

159. SACEVSKI, I. and VESELI, J. Introduction to Model Driven Architecture (MDA). In: *Seminar Paper, University of Salzburg*. 2007.
160. SAMETINGER, J. *Software engineering with reusable components*. Springer Science & Business Media, 1997.
161. SCHLEE, M. and VANDERDONCKT, J. Generative programming of graphical user interfaces. In: *Proceedings of the working conference on Advanced visual interfaces*. ACM, 2004. p. 403-406.
162. SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering.*Computer*, 2006, 39.2: 0025-31.
163. SCHMOHL, R. and BAUMGARTEN, U. Context-aware computing: a survey preparing a generalized approach. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists*. 2008. p. 744-750.
164. SEI. A Framework for Software Product Line Practice. 2012. [žiūrėta 2015-02-07]. Prieiga per internetą: http://www.sei.cmu.edu/productlines/frame_report/index.html.
165. SEIDEWITZ, E. What models mean. *IEEE software*, 2003, 20.5: 26-32.
166. SENDALL, S., HAUSER, R., KOEHLER, J., KÜSTER, J. and WAHLER, M. Understanding model transformation by classification and formalization. In: *Proceedings of Workshop on Software Transformation Systems*. 2004.
167. SHEARD, T. Accomplishments and research challenges in meta-programming. In: *Semantics, applications, and implementation of program generation*. Springer Berlin Heidelberg, 2001. p. 2-44.
168. SHEARD, T. and JONES, S. P. Template meta-programming for Haskell. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 2002. p. 1-16.
169. SHEARD, T. and PASALIC, E. Meta-programming with built-in type equality. In: *Fourth International Workshop on Logical Frameworks and Meta-Languages*. 2004.
170. SHEN, A. *Algorithms and programming: problems and solutions*. Springer Science & Business Media, 2011.
171. SIMMONDS, J., BASTARRICA, M., SILVESTRE, L. and QUISPE, A. Analyzing Methodologies and Tools for Specifying Variability in Software Processes. *Universidad de Chile, Santiago, Chile*, 2011.
172. SIMMONDS, J., BASTARRICA, M. C., SILVESTRE, L. and QUISPE, A. Modeling variability in software process models. *Computer Science Department, Universidad de Chile, Santiago, Chile*, 2012.
173. SINKOVICS, R. R., YAMIN, M. and HOSSINGER, M. Cultural adaptation in cross border e-commerce: a study of German companies. *Journal of Electronic Commerce Research*, 2007, 8.4: 221-235.
174. SoftFluent. CodeFluent Entities. 2014. [žiūrėta 2015-02-27]. Prieiga per internetą: http://www.softfluent.com/store/codefluent-entities.
175. SPLOT. Software Product Lines Online Tools. 2009. [žiūrėta 2015-02-08]. Prieiga per internetą: http://www.splot-research.org/.
176. STORM, T. The Rascal Language Workbench. *Software Engineering [SEN]*, 2011, SEN-1111: 1-28.
177. ŠTUIKYS, V. Smart Learning Objects for the Smart Education in Computer Science: Theory, Methodology and Robot-Based Implementation. Springer. 2015.
178. ŠTUIKYS, V. and BESPALOVA, K. Methodology and Experiments to Transform Heterogeneous Meta-program into Meta-meta-programs. In:*Information and Software Technologies*. Springer Berlin Heidelberg, 2012. p. 210-225.

179. ŠTUIKYS, V., BESPALOVA, K. and BURBAITĖ, R. Refactoring of Heterogeneous Meta-Program into k-stage Meta-Program. *Information Technology And Control*, 2014a, 43.1: 14-27.
180. ŠTUIKYS, V., BESPALOVA, K. and BURBAITĖ, R. Generative Learning Object (GLO) Specialization: Teacher's and Learner's View. In:*Information and Software Technologies*. Springer International Publishing, 2014b. p. 291-301
181. ŠTUIKYS, V., BESPALOVA, K. and BURBAITE, R. Feature transformation-based computational model and tools for heterogeneous meta-program design. In: *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on*. IEEE, 2014c. p. 185-190.
182. ŠTUIKYS, V. and DAMAŠEVIČIUS, R. *Meta-Programming and Model-Driven Meta-Program Development: Principles, Processes and Techniques*. Springer Science & Business Media, 2013a.
183. ŠTUIKYS, V. and DAMAŠEVIČIUS, R. Equivalent Transformations of Heterogeneous Meta-Programs. *Informatica*, 2013b, 24.2: 315-337.
184. ŠTUIKYS, V. ir DAMAŠEVIČIUS, R. *Modelių ir programų abstrakčiosios transformacijos*. Kauno technologijos universitetas. 2008.
185. ŠTUIKYS, V. and DAMAŠEVIČIUS, R. Scripting language open PROMOL and its processor. *Informatica*, 2000, 11.1: 71-86.
186. ŠTUIKYS, V., MONTVILAS, M. and DAMAŠEVIČIUS, R. Development of web component generators using one-stage metaprogramming.*Information Technology and Control*, 2009, 38.2: 108-118.
187. TAHA, W. A gentle introduction to multi-stage programming. In: *Domain-Specific Program Generation*. Springer Berlin Heidelberg, 2004. p. 30-50.
188. TAHA, W. *Multi-stage programming: Its theory and applications*. 1999. PhD. Oregon Graduate Institute of Science and Technology. [žiūrėta 2015-04-25]. Prieiga per internetą: http://www.researchgate.net/profile/Walid_Taha2/publication /2623619_Multi-Stage_Programming_Its_Theory_and_Applications/links/ 0deec538a36497119f000000.pdf.
189. TAHA, W. and SHEARD, T. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 2000, 248.1: 211-242.
190. TESSON, J., HASHIMOTO, H., HU, Z., LOULERGUE, F. and TAKEICHI, M. Program Calculation in Coq. In: *Algebraic Methodology and Software Technology*. Springer Berlin Heidelberg, 2011. p. 163-179.
191. THOMAS, D. A. Refactoring as Meta Programming. *Journal of Object Technology*, 2005, 4.1: 7-12.
192. THUM, T., BATORY, D. and KASTNER, C. Reasoning about edits to feature models. In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009. p. 254-264.
193. TOBIN-HOCHSTADT, S., ST-AMOUR, V., CULPEPPER, R., FLATT, M. and FELLEISEN, M. Languages as libraries. In: *ACM SIGPLAN Notices*. ACM, 2011. p. 132-141.
194. TOURWÉ, T. and MENS, T. Identifying refactoring opportunities using logic meta programming. In: *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE, 2003. p. 91-100.
195. TRACZ, W. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *ACM SIGSOFT Software Engineering Notes*, 1994, 19.2: 52-56.
196. TRUJILLO, S., AZANZA, M. and DIAZ, O. Generative metaprogramming. In: *Proceedings of the 6th international conference on Generative programming and component engineering*. ACM, 2007. p. 105-114.

197. TRUJILLO, S., BATORY, D. nad DIAZ, O. Feature oriented model driven development: A case study for portlets. In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007. p. 44-53.
198. USMAN, M. and NADEEM, A. Automatic generation of Java code from UML diagrams using UJECTOR. *International Journal of Software Engineering and Its Applications*, 2009, 3.2: 21-37.
199. VAN AMSTEL, M. F., VAN DEN BRAND, M. GJ. and NGUYEN, P. H. Metrics for model transformations. In: *Proceedings of the Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL 2010), Lille, France (December 2010)*. 2010.
200. VAN DER LINDEN, F. J., SCHMID, K. and ROMMES, E. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.
201. VAN DEURSEN, A., KLINT, P. and VISSER, J. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 2000, 35.6: 26-36.
202. VELDHUIZEN, T. L. Tradeoffs in metaprogramming. In: *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2006. p. 150-159.
203. VERBERT, K., MANOUSELIS, N., OCHOA, X., WOLPERS, M., DRACHSLER, H., BOSNIC, I. and DUVAL, E. Context-aware recommender systems for learning: a survey and future challenges. *Learning Technologies, IEEE Transactions on*, 2012, 5.4: 318-335.
204. VISSER, E. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 2001, 57: 109-143.
205. VISSER, E. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 2005, 40.1: 831-873.
206. VISSER, E. Meta-programming with concrete object syntax. In: *Generative programming and component engineering*. Springer Berlin Heidelberg, 2002. p. 299-315.
207. VÖLTER, M., STAHL, T., BETTIN, J., HAASE, A. and HELSEN, S. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.
208. WEISS, D. M. Software product-line engineering: a family-based software development process. 1999.
209. WINTER, V. L. Program Transformation: What, How, and Why. *Wiley Encyclopedia of Computer Science and Engineering*, 2008.
210. WIRTH, N. Program development by stepwise refinement. *Communications of the ACM*, 1971, 14.4: 221-227.
211. WOLTER, K., KREBS, T., DEELSTRA, S., SINNEMA, M., NIJHUIS, J. and MACGREGOR, J. *Configuration in industrial product families*. Amsterdam: IOS, 2006.
212. XFeature. XFeature -- feature modelling tool. 2004. [žiūrėta 2015-02-08]. Prieiga per internetą: http://www.pnp-software.com/XFeature/.
213. XING, Z. and STROULIA, E. Refactoring practice: How it is and how it should be supported-an eclipse case study. In: *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 2006. p. 458-468.
214. ZHANG, X. Developing Model-Driven Software Product Lines. 2014. PhD. University of Oslo.
215. ZIMMERMANN, A., LORENZ, A. and OPPERMANN, R. An operational definition of context. In: *Modeling and using context*. Springer Berlin Heidelberg, 2007. p. 558-571.

216. ZIMMERMANN, A., SPECHT, M. and LORENZ, A. Personalization and context management. *User Modeling and User-Adapted Interaction*, 2005, 15.3-4: 275-302.
217. KANZHELEV, S. and SHALYTO A. Преобразование графов переходов, представленных в формате MS Visio в исходные коды программ для различных языков программирования (инструментальное средство MetaAuto). Проектная документация. *Режим доступа: http://is. ifmo. ru/projects/metaauto [Дата обращения: 19.07. 2013]*, 2006.

PRIEDAI

1 priedas. Aiškinamojo pavyzdžio metaprogramos programinis tekstas

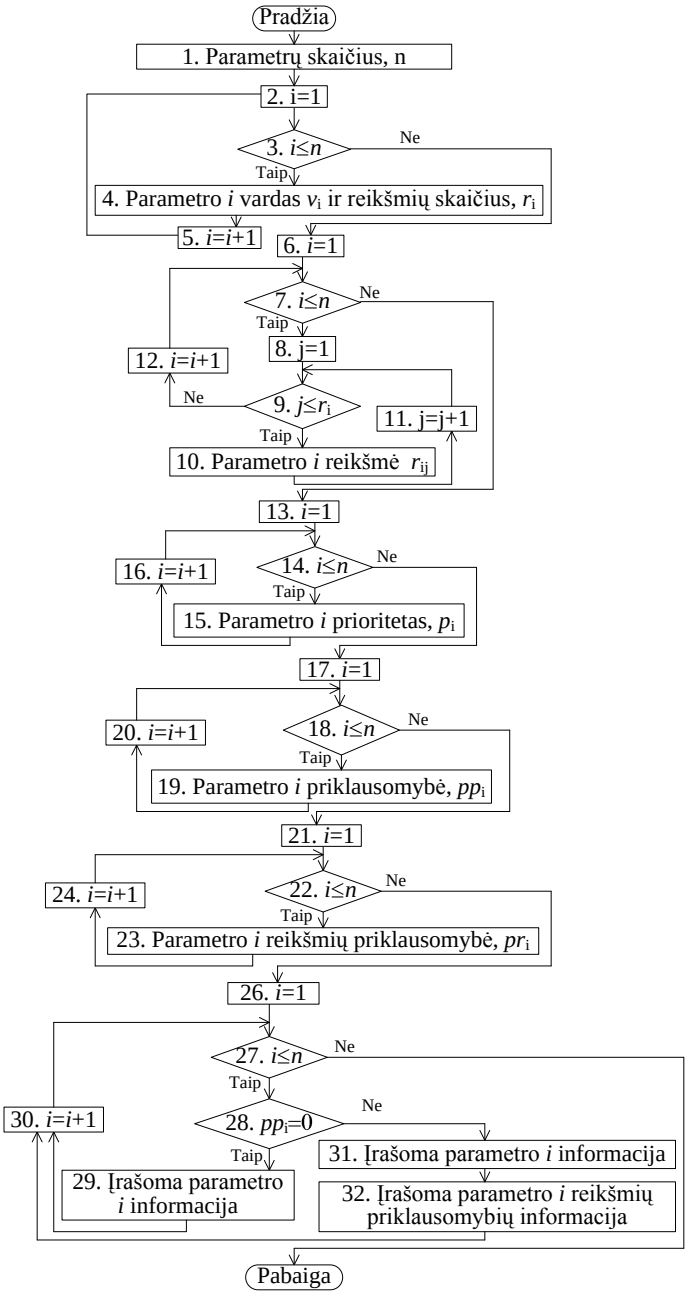
```
<?
//--------------------Meta-interface-------------------------
 $Operatorius =$_POST[Operatorius];
 $Operandas_x =$_POST[Operandas_x];
if (!isset($Operatorius) && !isset($Operandas_x)) {
?>
<FORM METHOD = POST ACTION = "">
Select parameter $Operatorius value:
<select name="Operatorius">
<option value="*"> * </option>
<option value="/"> / </option>
<option value="+"> + </option>
<option value="-"> - </option>
</select> <br>
<INPUT TYPE = submit VALUE ="Submit" NAME = "submit" style="height: 28px">
</FORM>
<?
} if (isset($Operatorius) && !isset($Operandas_x)) {
?>
<FORM METHOD = POST ACTION = "">
Select parameter $Operandas_x value:
<select name="Operandas_x">
<?
if("$Operatorius"=="/"){
?>
<option value="2"> 2 </option>
<option value="4"> 4 </option>
<option value="6"> 6 </option>
<option value="8"> 8 </option>
<option value="10"> 10 </option>
<? } else { ?>
<option value="0"> 0 </option>
<option value="2"> 2 </option>
<option value="4"> 4 </option>
<option value="6"> 6 </option>
<option value="8"> 8 </option>
<option value="10"> 10 </option>
<? } ?>
</select> <br>
 <INPUT TYPE="hidden" name="Operatorius" value="<?php echo
$_POST[Operatorius];?>">
<INPUT TYPE = submit VALUE ="Submit" NAME = "submit" style="height: 28px">
</FORM>
<?
}if (isset($Operatorius) && isset($Operandas_x)){
//--------------------Meta-body-------------------------
$myFile = "result.c";
$fr = fopen($myFile, 'w');
fwrite($fr," int Met(){ \n");
fwrite($fr," int y = 0; \n");
fwrite($fr," y = $Operandas_x $Operatorius 5; \n");
fwrite($fr," return y; \n");
fwrite($fr," } \n");
fclose($fr);
echo" <br><a href=\"result.c\">Program file</a></span></span></strong><br>";
}?>
```

2 priedas. Aiškinamojo pavyzdžio dviejų pakopų metaprogramos programinis tekstas

```
<?
$Operatorius =$_POST[Operatorius];
$Operandas_x =$_POST[Operandas_x];
$f1g = fopen("1_stage_g.php", 'w');
fwrite($f1g,"<?\n");
fwrite($f1g,"\$Operandas_a =\$_POST[Operandas_a];\n");
if (!isset($Operatorius) && !isset($Operandas_x)) {
?>
<FORM METHOD = POST ACTION = "">
Select parameter $Operatorius value:
<select name="Operatorius">
<option value="*"> * </option>
<option value="/"> / </option>
<option value="+"> + </option>
<option value="-"> - </option>
</select>
<br>
<INPUT TYPE = submit VALUE ="Submit" NAME = "submit" style="height: 28px">
</FORM>
<?
}
if (isset($Operatorius) && !isset($Operandas_x)) {
?>
<FORM METHOD = POST ACTION = "">
Select parameter $Operandas_x value:
<select name="Operandas_x">
<?
if("$Operatorius"=="/"){
?>
<option value="2"> 2 </option>
<option value="4"> 4 </option>
<option value="6"> 6 </option>
<option value="8"> 8 </option>
<option value="10"> 10 </option>
<?
}
else{
?>
<option value="0"> 0 </option>
<option value="2"> 2 </option>
<option value="4"> 4 </option>
<option value="6"> 6 </option>
<option value="8"> 8 </option>
<option value="10"> 10 </option>
<?
}
?>
</select>
<br>
 <INPUT TYPE="hidden" name="Operatorius" value="<?php echo
$_POST[Operatorius];?>">
<INPUT TYPE = submit VALUE ="Submit" NAME = "submit" style="height: 28px">
</FORM>
<?
}
fwrite($f1g,"if (!isset(\$Operandas_a)) {\n");
fwrite($f1g,"?>\n");
fwrite($f1g,"<FORM METHOD = POST ACTION = \"\">\n");
fwrite($f1g,"Select parameter $Operandas_a value:\n");
```

```
fwrite($f1g,"<select name=\"Operandas_a\">\n");
fwrite($f1g,"<option value=\"a\"> a </option>\n");
fwrite($f1g,"<option value=\"a*a\"> a*a </option>\n");
fwrite($f1g,"<option value=\"a*a*a\"> a*a*a </option>\n");
fwrite($f1g,"</select>\n");
fwrite($f1g,"<br>\n");
fwrite($f1g,"<INPUT TYPE=submit VALUE =\"Submit\" NAME = \"submit\"
style=\"height: 28px\">\n");
fwrite($f1g,"</FORM>\n");
fwrite($f1g," <? \n");
fwrite($f1g," } \n");
fwrite($f1g," if (isset(\$Operandas_a)) \n");
fwrite($f1g," { echo\"The instance generated\";\n");
fwrite($f1g," echo\"<br><br>\";\n");
fwrite($f1g," \$myFile = \"result.c\"; \n");
fwrite($f1g," \$fr = fopen(\$myFile, 'w'); \n");
fwrite($f1g," fwrite(\$fr,\" int Met(){ \\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" int y = 0; \\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" y = $Operandas_x $Operatorius \$Operandas_a;
\\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" return y; \\n\"); \n");
fwrite($f1g," fwrite(\$fr,\" } \\n\"); \n");
fwrite($f1g," fclose(\$fr); \n");
fwrite($f1g," echo\" <br> <a href=\\\"result.c\\\">Program
file</a></span></span></strong><br>\"; \n");
fwrite($f1g," } \n");
fwrite($f1g," ?> \n");
if (isset($Operatorius) && isset($Operandas_x)){
 echo "<a href=\"1_stage_g.php\">Start 1-stage meta-program <br><br></a>";
}
?>
```

3 priedas. Probleminės srities požymių modelio tarpinio modelio kūrimo algoritmas



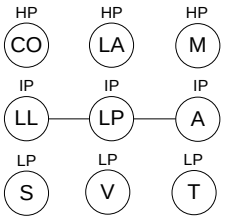
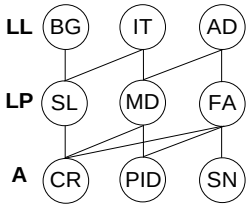
1 pav. Tarpinio modelio TM_P kūrimo algoritmas

4 priedas. Uždavinių aprašas ir sukurti modeliai

1. Roboto kalibravimas

Sukuriama metaprograma, kuri generuoja roboto kalibravimo programas, kurias panaudojant susipažįstama su roboto konstrukcija ir valdymu, nagrinėjami 3 tiesiaeigio judėjimo algoritmai.

1. lentelė Roboto kalibravimo uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**CO** (SA); **LA** (CT; PS); **M** (PR; PB); **LL** (BG; IT; AD); **LP** (SL, MD; FA)
Turinio metaparametrai	**A** (CR; PID; SN); **S** (AB; AC; BC); **V** (10; 30; 50; 70; 90); **T**(1000; 3000; 5000)
Metaparametrų sąveikos modelis $G(P^w, U)$ 	Priklausomų metaparametrų reikšmių sąveikos modelis $H_b((V_i, V_j), E)$ 
Galimas tikslo kalbos programų egzempliorių skaičius	$1*2*2*9*3*5*3 = 1620$

Konteksto metaparametrai: **CO** – curriculum objective (SA - Sequential algorithms), **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*)), **M** – learning method (PR - Project-based; PB - Problem-based), **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced), **LP** – learning pace (SL - Slow, MD - Medium; FA - Fast).
Turnio metaparametrai: **A** – algorithm's type (CR - Without corrections; PID - Using PID principles; SN - Using motors synchronization) (RobotC, 2007), **S** – selected motor (A, B, C – names of motors), **V** - velocity of motors in % calculated of the maximum value, **T** – robot's movement time in ms.

```
HP $objective 0 0 Sequential_algorithms
HP $activity 0 0 Case_study Practise
HP $method 0 0 Project_based Problem_based
IP $pace 0 0 Beginner Intermediate Advanced
IP $level $pace 0 Slow Medium Fast
IP $level $pace 1 Slow
IP $level $pace 2 Slow Medium
IP $level $pace 3 Medium Fast
IP $algorithm $level 0 Without_corrections Using_PID_principles
Using_motors_synchronization
IP $algorithm $level 1 Without_corrections
IP $algorithm $level 2 Without_corrections Using_PID_principles
IP $algorithm $level 3 Without_corrections Using_PID_principles
Using_motors_synchronization
LP $motors 0 0 AB AC BC
LP $v 0 0 10 30 50 70 90
LP $t 0 0 1000 3000 5000
Additional $myFile $fd $motor1 $motor2 $motor3
```

2 pav. Roboto kalibravimo uždavinio probleminės srities tarpinis modelis TM_P

```
//Curriculum objective: var_objective
//Learning activity: var_activity
//Learning method: var_method
//Learner's level: var_level
//Learning pace: var_pace
//Calibration algorithm: var_algorithm
task main()
{
@ if var_motors = AB then begin
@ var_motor1 := motorA;
@ var_motor2 := motorB;
@ var_motor3 := motorC;
@ end
@ if var_motors = AC then begin
@ var_motor1 := motorA;
@ var_motor2 := motorC;
@ var_motor3 := motorB;
@ end
@ if var_motors = BC then begin
@ var_motor1 := motorB;
@ var_motor2 := motorC;
@ var_motor3 := motorA;
@ end
   motor[var_motor3] = 50;
   wait1Msec(100);
   motor[var_motor3] = 0;
@ if var_algorithm = Without_corrections then begin
     motor[var_motor1] = var_v;
     motor[var_motor2] = var_v;
@ end
@ else if var_algorithm = Using_PID_principles then begin
     nMotorPIDSpeedCtrl[var_motor1] = mtrSpeedReg;
     nMotorPIDSpeedCtrl[var_motor2] = mtrSpeedReg;
     motor[var_motor1] = var_v;
     motor[var_motor2] = var_v;
@ end
@ else if var_algorithm = Using_motors_synchronization then begin
     nSyncedMotors = synchvar_motor1var_motor2;
     nSyncedTurnRatio = 100;
     motor[var_motor1] = 50;
@ end
   wait1Msec(var_t);
   motor[var_motor3] = -50;
   wait1Msec(100);
   motor[var_motor3] = 0;
}
```

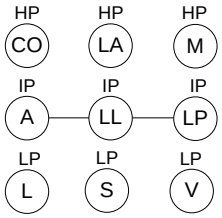
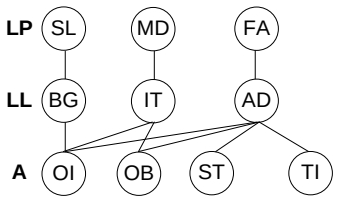
3 pav. Roboto kalibravimo uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

2. Linijos sekimas

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, kurių vykdymo metu robotas seka juodos spalvos liniją popieriaus lape. Aprašomi 4 linijos sekimo algoritmai.

2. lentelė Linijos sekimo uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**CO** (LC); **LA** (CT; PS; AK); **M** (PR, PB); **LL** (BG; IT; AD); **LP** (SL, MD, FA)

Turinio metaparametrai	**A** (OI; OB; ST; TI); **L** (S1; S2; S3; S4; S1&S2; S1& S3; S1&S4; S2&S3; S2&S4; S3&S4); **S** (AB; AC; BC); **V** (10; 20; 30; 40)
Metaparametrų sąveikos modelis $G(P^W, U)$ 	Priklausomų metaparametrų reikšmių sąveikos modelis $H_b((V_i, V_j), E)$ 
Galimas tikslo kalbos programų egzempliorių skaičius	1*3*2*7*10*3*4 = 5040

Konteksto metaparametrai: **CO** – curriculum objective (LC - Loop-based and conditional algorithms), **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*); AK - Assessment of knowledge), **M** – learning method (PR - Project-based; PB - Problem-based); **LP** – learning pace (SL - Slow, MD - Medium, FA - Fast); **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced). *Turinio metaparametrai*: **A** – algorithm (OI - One Inside; OB - One Bounce; ST - Straddle; TI - Two Inside), **L** – Light sensors' inputs (S1; S2; S3, S4), **S** – selected motor (A; B; C), **V**- velocity of motors in %.

Probleminės srities požymių modelio tarpinį modelį TM_P:

```
HP $objective 0 0 Loop-based_and_conditional_algorithms
HP $activity 0 0 Case_study Practise Assesment_of_knowledge
HP $method 0 0 Project-based Problem-based
IP $pace 0 0 Slow Medium Fast
IP $level $pace 0 Beginner Intermediate Advanced
IP $level $pace 1 Beginner
IP $level $pace 2 Intermediate
IP $level $pace 3 Advanced
IP $algorithm $level 0 One_inside One_bounce Straddle Two_inside
IP $algorithm $level 1 One_inside
IP $algorithm $level 2 One_inside One_bounce
IP $algorithm $level 3 One_inside One_bounce Straddle Two_inside
LP $sensor 0 0 left right
LP $leftsensor 0 0 S1 S2 S3 S4
LP $rightsensor 0 0 S1 S2 S3 S4
LP $leftmotor 0 0 A B C
LP $rightmotor 0 0 A B C
LP $v 0 0 10 20 30 40
Additional $tab $movement1 $movement2 $myFile $fd
```

4 pav. Linijos sekimo uždavinio probleminės srities tarpinis modelis TM_P

```
//Curriculum objective: var_objective
//Learning activity: var_activity
//Learning method: var_method
//Learner's level: var_level
//Learning pace: var_pace
@ if var_algorithm = One_inside || var_algorithm = One_bounce then begin
@ if var_sensor = left then begin
     #pragma config(Sensor, var_leftsensor, lightSensorleft,
sensorLightActive)
@ end
@ if var_sensor = right then begin
     #pragma config(Sensor, var_rightsensor, lightSensorright,
sensorLightActive)
@ end
@ end
```

```
@ else if var_algorithm = Straddle || var_algorithm = Two_inside then begin
#pragma config(Sensor, var_leftsensor, lightSensorleft,
sensorLightActive)
#pragma config(Sensor, var_rightsensor, lightSensorright,
sensorLightActive)
@ end
task main()
{
   nMotorEncoder[motor".var_leftmotor."] = 0;
   nMotorEncoder[motor".var_rightmotor."] = 0;
   while(true){
@ if var_algorithm = One_inside || var_algorithm = One_bounce then begin
@ if var_sensor = left then begin
     float i = SensorValue(lightSensorleft);
@ end
@ if var_sensor = right then begin
     float k = SensorValue(lightSensorright);
@ end
@ end
@ else if var_algorithm = Straddle || var_algorithm = Two_inside then begin
     float i = SensorValue(lightSensorleft);
     float k = SensorValue(lightSensorright);
@ end
@ if var_method = Problem-based then begin
   //Write a code of program
@ end
@ else if var_method = Project-based then begin
@ if var_algorithm = Straddle || var_algorithm = Two_inside then begin
     if (i < 45)
@ end
@ else if var_algorithm = One_inside then begin
@ if var_sensor = left then begin
     if (i < 45)
@ end
@ if var_sensor = right then begin
     if (k < 45)
@ end
@ end
@ else if var_algorithm = One_bounce then begin
@ if var_sensor = left then begin
     if (i > 45)
@ end
@ if var_sensor = right then begin
     if (k > 45)
@ end
@ end
     {
@ if var_algorithm = Straddle || var_algorithm = Two_inside then begin
      motor[motor".var_leftmotor."] = 0;
      motor[motor".var_rightmotor."] = var_v;
@ end
@ else if var_algorithm = One_inside || var_algorithm = One_bounce then begin
@ if var_sensor = left then begin
      motor[motor".var_leftmotor."] = 0;
      motor[motor".var_rightmotor."] = var_v;
@ end
@ if var_sensor = right then begin
       motor[motor".var_leftmotor."] = var_v;
      motor[motor".var_rightmotor."] = 0;
@ end
@ end
     }
@ if var_algorithm = One_inside then begin
      else
      {
```

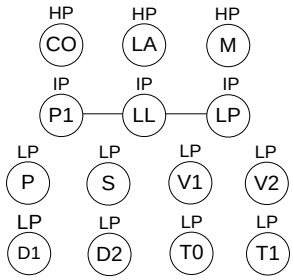
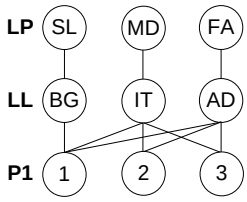
```
@ if var_sensor = left then begin
       motor[motor".var_leftmotor."] = var_v;
       motor[motor".var_rightmotor."] = 0;
@ end
@ else if var_sensor = right then begin
      motor[motor".var_leftmotor."] = 0;
      motor[motor".var_rightmotor."] = var_v;
@ end
@ end
@ else if var_algorithm = One_bounce then begin
@ if var_sensor = left then begin
                  if (i <= 45)
                    {
                       motor[motor".var_leftmotor."] = var_v;
                       motor[motor".var_rightmotor."] = 0;
                    }
                  if (i > 45)
                    {
                       motor[motor".var_leftmotor."] = var_v;
                       motor[motor".var_rightmotor."] = 0;
@ end
@ else if var_sensor = right then begin
                  if (k <= 45)
                    {
                       motor[motor".var_leftmotor."] = 0;
                       motor[motor".var_rightmotor."] = var_v;
                    }
                  if (k > 45)
                    {
                       motor[motor".var_leftmotor."] = 0;
                       motor[motor".var_rightmotor."] = var_v;
@ end
@ end
@ else if var_algorithm = Straddle then begin
                  else if (k > 45)
                    {
                       motor[motor".var_leftmotor."] = var_v;
                       motor[motor".var_rightmotor."] = 0;
@ end
@ else if var_algorithm = Two_inside then begin
                   else if (i >= 45)
                     {
                       motor[motor".var_leftmotor."] = var_v;
                       motor[motor".var_rightmotor."] = 0;
                     }
                   else if (k > 45)
                     {
                       motor[motor".var_leftmotor."] = var_v;
                       motor[motor".var_rightmotor."] = 0;
                     }
                   else if (k <= 45)
                     {
                       motor[motor".var_leftmotor."] = 0;
                       motor[motor".var_rightmotor."] = var_v;
@ end
       }
@ end
       }
         motor[motor".var_leftmotor."] = 0;
         motor[motor".var_rightmotor."] = 0;
       }
```

5 pav. Linijos sekimo uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

3. Ornamentų kūrimas

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, kurių vykdymo metu robotas piešia ornamentus.

3. lentelė Ornamentų kūrimo uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**CO** (LN); **LA** (CT; PS); **M** (PR; PB); **LL** (BG; IT; AD); **LP** (SL, MD, FA);
Turinio metaparametrai	**P1** (1, 2; 3); **P** (4; 5; 6); **S** (AB; AC; BC); **V1** (10; 30; 50); **V2** (10; 30; 50); **D1** (10; 30); **D2** (10; 30); **T0** (1000; 3000; 5000); **T1** (200; 500);
Metaparametrų sąveikos modelis $G(P^W, U)$  	Priklausomų metaparametrų reikšmių sąveikos modelis $H_b((V_i, V_j), E)$ 
Galimas tikslo kalbos programų egzempliorių skaičius	$1*2*2*7*3*3*3*3*2*2*3*2 = 54432$

Konteksto metaparametrai: **CO** – curriculum objective (LN – loops and nested loops), **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*)), **M** – learning method (PR - Project-based; PB - Problem-based), **LP** – learning pace (SL – Slow; MD – Medium; FA - Fast), **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced).
Turinio metaparametrai: **P1** – number of ornaments, **P** – number of ornament's parts, **S** – selected motor, **V1**, **V2**– *drawing velocity* of motors, **T0** – robot's drawing time, **D1, D2** – *moving velocity* of motors, **T1** – robot's moving time.

```
HP $objective 0 0 Loops_and_nested_loops
HP $activity 0 0 Case_study Practise
HP $method 0 0 Project_based Problem_based
IP $pace 0 0 Slow Medium Fast
IP $level $pace 0 Beginner Intermediate Advanced
IP $level $pace 1 Beginner
IP $level $pace 2 Intermediate
IP $level $pace 3 Advanced
IP $p1 $level 0 1 2 3
IP $p1 $level 1 1
IP $p1 $level 2 1 2 3
IP $p1 $level 3 1 2 3
LP $p 0 0 4 5 6
LP $s 0 0 AB AC BC
LP $v1 0 0 10 30 50
LP $v2 0 0 10 30 50
LP $t0 0 0 1000 3000 5000
LP $d1 0 0 10 30
LP $d2 0 0 10 30
LP $t1 0 0 200 500
Additional $i $j $m1 $m2 $mp $tt $e0 $e1 $f0 $f1 $mb1 $mb2 $a1 $a2 $d
$fin $myFile $fd $tab
```

6 pav. Ornamentų kūrimo uždavinio probleminės srities tarpinis modelis TM_P

```
//Curriculum objective: var_objective
//Learning activity: var_activity
//Learning method: var_method
//Learner's level: var_level
//Learning pace: var_pace
@ if var_method = Problem_based then begin
 Parameters of ornaments:
//Motors: var_s
@ if var_level = Intermediate || var_level = Advanced then begin
//Number of ornaments: var_p1
//Distance between ornaments:
//Moving velocity of motor1: var_d1
//Moving velocity of motor2: var_d2
//Moving time: var_t1
@ end
//Drawing velocity of motor1: var_v1
//Drawing velocity of motor2: var_v2
//Drawing time: var_t0
//Number of parts of one ornament: var_p
@ end
task main()
{
@ if var_method = Problem_based then begin
//Write commands to robot to draw ornaments
@ end
@ else if var_method != Problem_based then begin
@ if var_level = Intermediate || var_level = Advanced then begin
   for (int $i = 0; $i < var_p1; $i++) {
@ end
@ if var_s = AB then begin
//-------------------------------------------------------
//Preparation for painting
motor[motorC] = 50;
wait1Msec(100);
$mb1 motor[motorC] = 0;
//-------------------------------------------------------
//Painting
for (int $j = 0; $j < var_p; $j++) {
motor[motorA] = var_v1;
motor[motorB] = var_v2;
wait1Msec(var_t0);
//-------------------------------------------------------
motor[motorA] = -var_v1;
motor[motorB] = 0;
wait1Msec(var_t0);
}
//-------------------------------------------------------
//Painting of ornament is finished
motor[motorC] = -50;
wait1Msec(100);
motor[motorC] = 0;
@ if var_level = Intermediate || var_level = Advanced then begin
//-------------------------------------------------------
  //Distance between two neighboring ornaments
  motor[motorA] = var_d1;
  motor[motorB] = var_d2;
  wait1Msec(var_t1);
@ end
@ end
@ else if var_s = AC then begin
//-------------------------------------------------------
//Preparation for painting
motor[motorB] = 50;
wait1Msec(100);
```

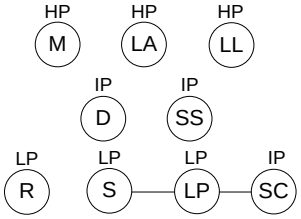
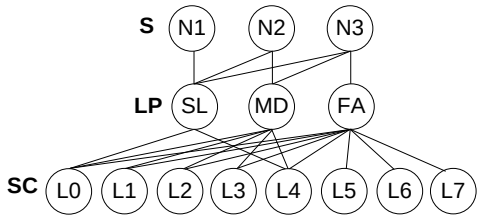
```
$mb1 motor[motorB] = 0;
//-----------------------------------------------------
//Painting
for (int $j = 0; $j < var_p; $j++) {
motor[motorA] = var_v1;
motor[motorC] = var_v2;
wait1Msec(var_t0);
//-----------------------------------------------------
motor[motorA] = -var_v1;
motor[motorC] = 0;
wait1Msec(var_t0);
}
//-----------------------------------------------------
//Painting of ornament is finished
motor[motorB] = -50;
wait1Msec(100);
motor[motorB] = 0;
@ if var_level = Intermediate || var_level = Advanced then begin
//-----------------------------------------------------
  //Distance between two neighboring ornaments
  motor[motorA] = var_d1;
  motor[motorC] = var_d2;
  wait1Msec(var_t1);
@ end
@ end
@ else if var_s = BC then begin
//-----------------------------------------------------
//Preparation for painting
motor[motorA] = 50;
wait1Msec(100);
$mb1 motor[motorA] = 0;
//-----------------------------------------------------
//Painting
for (int $j = 0; $j < var_p; $j++) {
motor[motorB] = var_v1;
motor[motorC] = var_v2;
wait1Msec(var_t0);
//-----------------------------------------------------
motor[motorB] = -var_v1;
motor[motorC] = 0;
wait1Msec(var_t0);
}
//-----------------------------------------------------
//Painting of ornament is finished
motor[motorA] = -50;
wait1Msec(100);
motor[motorA] = 0;
@ if var_level = Intermediate || var_level = Advanced then begin
//-----------------------------------------------------
  //Distance between two neighboring ornaments
  motor[motorB] = var_d1;
  motor[motorC] = var_d2;
  wait1Msec(var_t1);
@ end
@ end
@ end
}
```

7 pav. Ornamentų kūrimo uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

4. Reakcija į kliūtį

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, kurių vykdymo metu robotas reaguoja į kliūtis, esančias skirtingais atstumais.

4. lentelė Reakcijos į kliūtį uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**M** (PR; PB); **LA** (CT; PS); **LL** (BG; IT; AD), **LP** (SL; MD; FA).
Turinio metaparametrai	**D** (20; 30; 50); **SS** (S1; S2; S3; S4); **R** (Screen; Sound), **S** (N1; N2; N3), **SC** (L0; L1; L2; L3; L4; L5; L6; L7).
Metaparametrų sąveikos modelis $G(P^W, U)$ 	Priklausomų metaparametrų reikšmių sąveikos modelis $H_b((V_i, V_j), E)$ 
Galimas tikslo kalbos programų egzempliorių skaičius	2*2*3*3*4*2*24= 6912

Konteksto metaparametrai: **M** – learning method (PR - Project-based; PB - Problem-based), **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*)), **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced), **LP** – learning pace (SL – Slow; MD – Medium; FA - Fast). *Turinio metaparametrai*: **D** – reaction distance in cm, **SS** – selected sensor, **R** – result output (audible signalo or NXT Block display), **S** – selected sound (N1 – sound 1; N2 – sound 2; N3 – sound 3), **SC** – selected screen line (8 line).

```
HP $Method 0 0 Project-based Problem-based
HP $Activity 0 0 Case-Study Practise
HP $Level 0 0 Beginner Intermediate Advanced
IP $Distance 0 0 20 30 50
IP $Sensor 0 0 S1 S2 S3 S4
LP $Result 0 0 Screen Sound
LP $Pace 0 0 Slow Medium Fast
LP $Sound $Pace 0 Sound1 Sound2 Sound3
LP $Sound $Pace 1 Sound1
LP $Sound $Pace 2 Sound1 Sound2
LP $Sound $Pace 3 Sound1 Sound2 Sound3
LP $Screen $Pace 0 L0 L1 L3 L4 L5 L6 L7
LP $Screen $Pace 1 L0 L4
LP $Screen $Pace 2 L0 L1 L2 L3 L4
LP $Screen $Pace 3 L0 L1 L2 L3 L4 L5 L6 L7
Additional $fr $myFile
```

8 pav. Reakcijos į kliūtį uždavinio probleminės srities tarpinis modelis TM_P

```
#pragma config (Sensor,var_Sensor, sonarSensor, sensorSONAR)
// Learning activity: var_Activity
// Learning method: var_Method
// Learner's level: var_Level
task main()
{
while (true) {
```

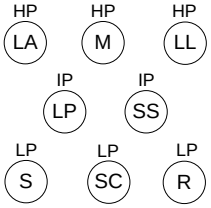
```
@ if var_Method = 'Project-based' && var_Activity = 'Case-Study' then
begin
int distance_in_cm = var_Distance;
while(SensorValue[sonarSensor] < distance_in_cm) {
@ if var_Result = 'Screen' then begin
@ if var_Screen = L0 then begin
@ var_line := 0;
@ }
@ if var_Screen = L1 then begin
@ var_line := 1;
@ }
@ if var_Screen = L2 then begin
@ var_line := 2;
@ }
@ if var_Screen = L3 then begin
@ var_line := 3;
@ }
@ if var_Screen = L4 then begin
@ var_line := 4;
@ }
@ if var_Screen = L5 then begin
@ var_line := 5;
@ }
@ if var_Screen = L6 then begin
@ var_line := 6;
@ }
@ if var_Screen = L7 then begin
@ var_line := 7;
@ }
nxtDisplayCenteredTextLine(".var_line.', "Labas"'.");
nxtDisplayClearTextLine(".var_line.");
@ end
@ if var_Result = 'Sound' then begin
@ if var_Sound = 'Sound1' then begin
@ var_s := 'soundBeepBeep';
@ }
@ if var_Sound = 'Sound2' then begin
@ var_s := 'soundLowBuzz';
@ }
@ if var_Sound = 'Sound3' then begin
@ var_s := 'soundBlip';
@ }
PlaySound (".var_s.");
@ end
@ end
@ if var_Method = 'Problem-based' then begin
//Define instance
while(SensorValue[sonarSensor] < distance_in_cm) {
@ if var_Result = 'Screen' then begin
//Define line number
//Define display of the text on the screen
//Define screen cleaning
@ end
@ if var_Result = 'Sound' then begin
//Define sound
//Define PlaySound
@ end
@ end
}
}
}
```

9 pav. Reakcijos į kliūtį uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

5. Spalvos atpažinimas

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, kurių vykdymo metu robotas atpažįsta spalvas.

5. lentelė Spalvos atpažinimo uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**LA** (CT; PS); **M** (PR; PB); **LL** (BG; IT; AD), LP (SL; MD; FA).
Turinio metaparametrai	**SS** (S1; S2; S3; S4); **S** (N1; N2; N3), **SC** (L0; L1; L2; L3; L4; L5; L6; L7), **R** (Screen; Sound).
Metaparametrų sąveikos modelis $G(P^W, U)$	
Galimas tikslo kalbos programų egzempliorių skaičius	2*2*3*3*4*3*8*2= 6912

Konteksto metaparametrai: **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*)), **M** – learning method (PR - Project-based; PB - Problem-based), **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced), **LP** – learning pace (SL – Slow; MD – Medium; FA - Fast). *Turinio metaparametrai*: **SS** – selected sensor, **S** – selected sound (N1 – sound 1; N2 – sound 2; N3 – sound 3), **SC**– selected screen line (8 line), **R** – result output (audible signalo or NXT Block display).

```
HP $Activity 0 0 Case-Study Practise
HP $Method 0 0 Project-based Problem-based
HP $Level 0 0 Beginner Intermediate Advanced
IP $Pace 0 0 Slow Medium Fast
IP $Sensor 0 0 S1 S2 S3 S4
LP $Sound 0 0 Sound1 Sound2 Sound3
LP $Screen 0 0 L0 L1 L2 L3 L4 L5 L6 L7
LP $Result 0 0 Screen Sound
Additional $fr $myFile
```

10 pav. Spalvos atpažinimo uždavinio probleminės srities tarpinis modelis TM_P

```
#pragma config (Sensor,var_Sensor, colorPort, sensorCOLORFULL)
// Learning activity: var_Activity
// Learning method: var_Method
// Learner's level: var_Level
task main()
{
while (true){
@ if var_Method = 'Project-based' &&  var_Activity = 'Case-Study' then
begin
@ if var_Result = Screen then begin
string sColor;
@ if var_Screen = L0 then begin
@ var_line := 0;
@ }
@ if var_Screen = L1 then begin
@ var_line := 1;
@ }
@ if var_Screen = L2 then begin
@ var_line := 2;
@ }
@ if var_Screen = L3 then begin
```

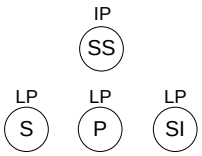
```
@ var_line := 3;
@ }
@ if var_Screen = L4 then begin
@ var_line := 4;
@ }
@ if var_Screen = L5 then begin
@ var_line := 5;
@ }
@ if var_Screen = L6 then begin
@ var_line := 6;
@ }
@ if var_Screen = L7 then begin
@ var_line := 7;
@ }
switch (SensorValue[colorPort])
{
case BLACKCOLOR: sColor = 'Black'; break;
case BLUECOLOR: sColor = 'Blue'; break;
case GREENCOLOR: sColor = 'Green'; break;
case YELLOWCOLOR: sColor = 'Yellow'; break;
case REDCOLOR: sColor = 'Red'; break;
case WHITECOLOR: sColor = 'White'; break;
default: sColor = '???'; break;
}
nxtDisplayCenteredTextLine(var_line, "Labas");
nxtDisplayClearTextLine(var_line);
@ end
@ if var_Result = 'Sound' then begin
@ if var_Sound = 'Sound1' then begin
@ var_s := 'soundBeepBeep';
@ }
@ if var_Sound = 'Sound2' then begin
@ var_s := 'soundLowBuzz';
@ }
@ if var_Sound = 'Sound3' then begin
@ var_s := 'soundBlip';
@ }
@ var_s1 := 'soundException';
TSounds sSound;
switch (SensorValue[colorPort])
{
case BLACKCOLOR: sSound =". var_s."; break;
default: sSound = ".var_s1."; break;
}
PlaySound (sSound);
wait1Msec(100);
}
@ end
@ end
@ if var_Method = 'Problem-based' then begin
@ if var_Result = 'Screen' then begin
//Define line number
//Define display of the text on the screen
//Define screen cleaning
//Define program code
@ end
@ if var_Result = 'Sound' then begin
//Define sound
//Define program code
@ end
@ end
}
```

11 pav. Spalvos atpažinimo uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

6. Pagalbos sistema

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, kurių vykdymo metu vartotojas naudodamasis lietimosi jutikliais gali išsikviesti pagalbą.

6. lentelė Pagalbos sistemos uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Turinio metaparametrai	**SS** (NP; P); **S** (N1; N2), **P** (P; NP), **SI**(S1; S2; S3; S4).
Metaparametrų sąveikos modelis $G(P^W, U)$	
Galimas tikslo kalbos programų egzempliorių skaičius	2*2*2*4= 32

Turinio metaparametrai: **SS** – tactile sensor position (NP – not pressed; P - pressed), **S** – selected sound (N1 – sound 1; N2 – sound 2), **P** – selected sensor position (P – pressed; NP – not pressed), **SI** – selected sensor input.

```
IP $SensorPosition 0 0 NotPressed Pressed
LP $Pressed 0 0 Sound1 Sound2
LP $NotPressed 0 0 Pressed NotPressed
LP $SensorInput 0 0 S1 S2 S3 S4
Additional $fr $myFile
```

12 pav. Pagalbos sistemos uždavinio probleminės srities tarpinis modelis TM_P

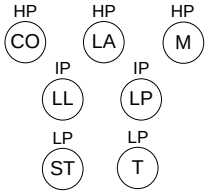
```
#pragma config (Sensor,var_SensorInput, touchSensor, sensorTouch)
task main()
{
while (true){
while(SensorValue(touchSensor) == 0)
nxtDisplayCenteredTextLine(4, "OK");
while(SensorValue(touchSensor) == 1){
@ if var_Pressed = Sound1 then begin
@ var_s :='soundBeepBeep';
@ }
@ if var_Pressed = Sound2 then begin
@ var_s :='soundLowBuzz';
@ }
PlaySound (var_s);
wait1Msec(500);
}
}
}
```

13 pav. Pagalbos sistemos uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

7. Bėganti eilutė

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, kurių vykdymo metu robotas išveda ekrane vartotojo norimą informaciją.

7. lentelė Bėgančios eilutės uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**CO** (SA; BA); **LA** (CT; PS); **M** (PR; PB); **LL** (BG; IT; AD); **LP** (SL, MD, FA)
Turinio metaparametrai	**ST** (text1, text2); **T** (200; 500; 3000; 5000)
Metaparametrų sąveikos modelis $G(P^W, U)$	
Galimas tikslo kalbos programų egzempliorių skaičius	$2*2*2*3*3*2*4 = 576$

Konteksto metaparametrai: **CO** – curriculum objective (SA - Sequential algorithms; BA - Binary addition), **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*)), **M** – learning method (PR - Project-based; PB - Problem-based), **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced), **LP** – learning pace (SL - Slow, MD - Medium, FA - Fast).
Turinio metaparametrai: **ST** – selected string, **T** – robot's work time.

```
HP $objective 0 0 Sequential_algorithms Binary_addition
HP $activity 0 0 Case_study Practise
IP $method 0 0 Project_based Problem_based
IP $level 0 0 Beginner Intermediate Advanced
LP $pace 0 0 Slow Medium Fast
LP $tstring 0 0 as tu
LP $time 0 0 200 500 3000 5000
Additional $tab $myFile $fd $simb $simb1 $simb2 $i $a $b
```

14 pav. Bėgančios eilutės uždavinio probleminės srities tarpinis modelis TM_P

```
//Curriculum objective: var_objective
//Learning activity: var_activity
//Learning method: var_method
//Learner's level: var_level
//Learning pace: var_pace
#include <font_5x4.h>
#include <HT1632.h>
#include <images.h>
int j = 0;
int wd;
@ for var_i := 1 to var_tstring div 1 do begin
@ if var_level = Beginner then begin
@ var_a := mt_rand (1, 5);
@ var_b := mt_rand (6, 10);
@ }
@ if var_level = Intermediate then begin
@ var_a := mt_rand (1, 10);
@ var_b := mt_rand (10, 15);
@ }
@ if var_level = Advanced then begin
@ var_a := mt_rand (5, 10);
@ var_b := mt_rand (11, 20);
@ end
@ end
void setup () {
   Serial.begin(9600);
   HT1632.begin(9, 10, 11);
@ if var_objective = Sequential_algorithms then begin
```

```
wd = HT1632.getTextWidth(".'"'.var_tstring.'"'.", FONT_5X4_WIDTH,
FONT_5X4_HEIGHT);
@ end
@ if var_objective = Binary_addtion then begin
@ if var_activity = Case_study then begin
     wd =
HT1632.getTextWidth(".'"'.decbin(var_a)."+".decbin(var_b)."=".decbin(var_
a + var_b).";".'"'.", FONT_5X4_WIDTH, FONT_5X4_HEIGHT);
@ end
@ if var_activity = Practise then begin
     wd =
HT1632.getTextWidth(".'"'.decbin(var_a)."+".decbin(var_b)."=".'?'.";".'"'
.", FONT_5X4_WIDTH, FONT_5X4_HEIGHT);
@ end
@ end
}
void loop () {
   HT1632.drawTarget(BUFFER_BOARD (1));
   HT1632.clear();
@ if var_objective = Sequential_algorithms then begin
   HT1632.drawText(".'"'.var_tstring.'"'.", 2*OUT_SIZE - j, 2, FONT_5X4,
FONT_5X4_WIDTH, FONT_5X4_HEIGHT, FONT_5X4_STEP_GLYPH);
@ end
@ if var_objective = Binary_addtion then begin
@ if var_activity = Case_study then begin
   HT1632.drawText(".'"'.decbin(var_a)."+".decbin(var_b)."=".decbin(var_a
+ var_b).";".'"'.", 2*OUT_SIZE - j, 2, FONT_5X4, FONT_5X4_WIDTH,
FONT_5X4_HEIGHT, FONT_5X4_STEP_GLYPH);
@ end
@ if var_activity = Practise then begin
HT1632.drawText(".'"'.decbin(var_a)."+".decbin(var_b)."=".'?'.";".'"'.",
2*OUT_SIZE - j, 2, FONT_5X4, FONT_5X4_WIDTH, FONT_5X4_HEIGHT,
FONT_5X4_STEP_GLYPH);
@ end
@ end
@ if var_method = Problem_based then begin
   //Write a code of scrolling text
@ end
   HT1632.render();
   j = (j + 1) % (wd + OUT_SIZE * 2);
@ if var_method = Project_based then begin
    delay(".var_time.");
@ end
@ if var_method = Problem_based then begin
   //Write a code of delay time
@ end
}
```

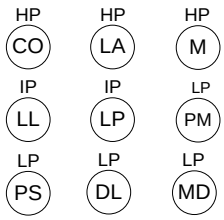
15 pav. Bėgančios eilutės uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

8. Šviesos sekimas

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, kurių vykdymo metu robotas seka šviesą.

8. lentelė Šviesos sekimo uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**CO** (LC); **LA** (CT; PS; AK); **LL** (BG; IT; AD); **M** (PR; PB); **LP** (SL, MD, F)

Turinio metaparametrai	**PM** (2-3, 3-2, 2-4, 4-2, 3-4, 4-3); **PS** (1-2; 2-1; 1-3; 3-1; 2-3; 3-2); **DL** (25; 50; 75); **MD** (R; L; S)
Metaparametrų sąveikos modelis $G(P^W, U)$	
Galimas tikslo kalbos programų egzempliorių skaičius	1*3*3*2*3*6*6*3*3 = 17496

Konteksto metaparametrai: **CO** – curriculum objective (LC - Loop-based and conditional algorithms), **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*); AK - Assessment of knowledge), **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced), **M** – learning method (PR - Project-based; PB - Problem-based), **LP** – learning pace (SL - Slow, MD - Medium, FA - Fast).
Turinio metaparametrai: **PM** – output pins on Arduino to control left and right motors, **PS** – left and right light sensors input pins on Arduino, **DL** – difference of the lightning, **MD** – movement direction (R – right; L – left; S – straight).

```
HP $objective 0 0 Loop_based_and_conditional_algorithms
HP $activity 0 0 Case_study Practise Assesment_of_knowledge
HP $method 0 0 Project_based Problem_based
IP $level 0 0 Beginner Intermediate Advanced
IP $pace 0 0 Slow Medium Fast
IP $pm 0 0 2-3 3-2 2-4 4-2 3-4 4-3
LP $ps 0 0 1-2 2-1 1-3 3-1 2-3 3-2
LP $dl 0 0 25 50 75
LP $md 0 0 R L S
Additional $tab $myFile $fd $dr1 $dl1 $ds1
```

16 pav. Šviesos sekimo uždavinio probleminės srities tarpinis modelis TM_P

```
//Curriculum objective: var_objective
//Learning activity: var_activity
//Learning method: var_method
//Learner's level: var_level
//Learning pace: var_pace
@ if var_pm = '2-3' then begin
      const int rm = 2;
      const int lm = 3;
@ end
@ if var_pm = '3-2' then begin
      const int rm = 3;
      const int lm = 2;
@ end
@ if var_pm = '2-4' then begin
      const int rm = 2;
      const int lm = 4;
@ end
@ if var_pm = '4-2' then begin
      const int rm = 4;
      const int lm = 2;
@ end
@ if var_pm = '3-4' then begin
      const int rm = 3;
      const int lm = 4;
@ end
@ if var_pm = '4-3' then begin
      const int rm = 4;
      const int lm = 3;
@ end
```

```
@ if var_ps = '1-2' then begin
      const int rs = 1;
      const int ls = 2;
@ end
@ if var_ps = '2-1' then begin
      const int rs = 2;
      const int ls = 1;
@ end
@ if var_ps = '1-3' then begin
      const int rs = 1;
      const int ls = 3;
@ end
@ if var_ps = '3-1' then begin
      const int rs = 3;
      const int ls = 1;
@ end
@ if var_ps = '2-3' then begin
      const int rs = 2;
      const int ls = 3;
@ end
@ if var_ps = '3-2' then begin
      const int rs = 3;
      const int ls = 2;
@ end
int sl, sr, sd;
void setup () {
   pinMode (lm, OUTPUT);
   pinMode (rm, OUTPUT);
   pinMode (ls, INPUT);
   pinMode (rs, INPUT);
   Serial.begin(9600);
}
void loop() {
   sl = 1023 - analogRead(ls);
   delay(1);
   sr = 1023 - analogRead(rs);
   delay(1);
   sd = abs(sl - sr);
@ if var_method = Project_based then begin
   if (sl > sr && sd > var_dl) {
      digitalWrite(rm, HIGH);
      digitalWrite(lm, LOW);
   }
   if (sl < sr && sd > var_dl) {
      digitalWrite(rm, LOW);
      digitalWrite(lm, HIGH);
   }
@ if var_md = R then begin
   else if (sd < var_dl) {
      digitalWrite(rm, HIGH);
      digitalWrite(lm, LOW);
   }
@ end
@ if var_md = L then begin
   else if (sd < var_dl) {
      digitalWrite(rm, LOW);
      digitalWrite(lm, HIGH);
   }
@ end
@ if var_md = S then begin
   else if (sd < var_dl) {
      digitalWrite(rm, HIGH);
      digitalWrite(lm, HIGH);
   }
@ end
```

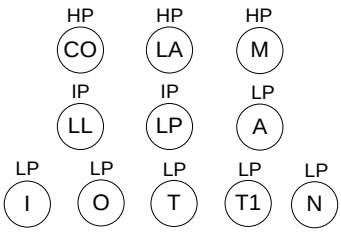
```
@ end
@ if var_method = Problem_based then begin
//Write a code of robot's straight movement, if difference of lightning
is less than var_dl
@ end
}
```

17 pav. Šviesos sekimo uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

9. Šviesoforas

Sukuriama metaprograma, kuri generuoja roboto valdymo programas, imituojančias šviesoforų darbą.

9. lentelė Šviesoforo uždavinio charakteristikos

Metaprogramos charakteristikos	Metaparametrai (metaparametrų reikšmės)
Konteksto metaparametrai	**CO** (LN); **LA** (CT; PS); **M** (PR; PB); **LL** (BG; IT; AD); **LP** (SL, MD, FA)
Turinio metaparametrai	**A** (FT; DB); **I** (1; 2; 3); **O** (1-2-3; 1-3-2; 2-1-3; 2-3-1; 3-1-2; 3-2-1); **T** (500; 1000; 1500); **T1** (100; 200; 300)
Metaparametrų sąveikos modelis $G(P^W, U)$	
Galimas tikslo kalbos programų egzempliorių skaičius	$1*2*2*3*3*2*3*6*3*3 = 11664$

Konteksto metaparametrai: **CO** – curriculum objective (LN – loops and nested loops), **LA** – learning activity (CT - Case study (given by *Teacher*); PS - Practise (done by *Learner*)), **M** – learning method (PR - Project-based; PB - Problem-based), **LL** – learner's previous knowledge level (BG - Beginner; IT - Intermediate; AD - Advanced), **LP** – learning pace (SL - Slow, MD - Medium, FA - Fast).
Turinio metaparametrai: **A** – algorithm's type (FT – fixed time control; DB – demand button control), **I** – input pin on Arduino for button (NO – without button), **O** – output pins on Arduino for red, yellow and green LEDs, **T** – duration of light phase in ms, **T1** – duration of warning in ms, **N** – number of warning signals.

```
HP $objective 0 0 Loops_and_nested_loops
HP $activity 0 0 Case_study Practise
IP $method 0 0 Project_based Problem_based
IP $level 0 0 Beginner Intermediate Advanced
IP $pace 0 0 Slow Medium Fast
LP $a 0 0 Fixed_time_control Demand_button_control
LP $inp 0 0 1 2 3
LP $out 0 0 1-2-3 1-3-2 2-1-3 2-3-1 3-1-2 3-2-1
LP $t 0 0 500 1000 1500
LP $t1 0 0 100 200 300
LP $n 0 0 3 4 5
Additional $tab $myFile $fd
```

18 pav. Šviesoforo uždavinio probleminės srities tarpinis modelis TM_P

```
//Curriculum objective: var_objective
//Learning activity: var_activity
//Learning method: var_method
//Learner's level: var_level
//Learning pace: var_pace
@ if var_out = '1-2-3' then begin
const int Red = 1;
const int Yellow = 2;
const int Green = 3;
@ end
@ else if var_out =='1-3-2' then begin
const int Red = 1;
const int Yellow = 3;
const int Green = 2;
@ end
@ else if var_out = '2-1-3' then begin
const int Red = 2;
const int Yellow = 1;
const int Green = 3;
@ end
@ else if var_out = '2-3-1' then begin
const int Red = 2;
const int Yellow = 3;
const int Green = 1;
@ end
@ else if var_out = '3-1-2' then begin
const int Red = 3;
const int Yellow = 1;
const int Green = 2;
@ end
@ else if var_out = '3-2-1' then begin
const int Red = 3;
const int Yellow = 2;
const int Green = 1;
@ end
@ if var_a = Demand_button_control then begin
     int buttonState = 0;
@ end
void setup () {
   pinMode(Red, OUTPUT);
   pinMode(Yellow, OUTPUT);
   pinMode(Green, OUTPUT);
@ if var_a = Demand_button_control then begin
     pinMode(Button, INPUT);
@ end
}
void loop () {
@ if var_a = Demand_button_control then begin
     buttonState = digitalRead(Button);
     if (buttonState ==HIGH) {
@ end
@ if var_method = Problem_based then begin
//Write a code of program with defined parameters:
//Duration of light phase in ms: var_t
//Duration of warning in ms: var_t1
//Number of warnings: var_n
@ end
@ else if var_method = Project_based then begin
   digitalWrite (Red, HIGH);
   delay (var_t);
   for (int k = 1; k <= var_n; k++) {
    digitalWrite (Red, LOW);
    delay (var_t);
```

```
     digitalWrite (Red, HIGH);
     delay (var_t);
    }
    for (int k = 1; k <= var_n; k++) {
     digitalWrite (Yellow, HIGH);
     delay (var_t);
     digitalWrite (Yellow, LOW);
     delay (var_t);
    }
    digitalWrite (Green, HIGH);
    delay (var_t);
    for (int k = 1; k <= var_n; k++) {
     digitalWrite (Green, LOW);
     delay (var_t);
     digitalWrite (Green, HIGH);
     delay (var_t);
    }
@ end
}
@ if var_a = Demand_button_control then begin
 else { digitalWrite (Red, HIGH); } }
@ end
```

19 pav. Šviesoforo uždavinio tikslo kalbos bendrasis programos egzemplioriaus modelis $TKBE_P$

SL344. 2015-11-04 19,25 leidyb. apsk.1. Tiražas 10 egz. Užsakymas 393.
Išleido Kauno technologijos universitetas, K. Donelaičio g. 73, 44249 Kaunas
Spausdino leidyklos „Technologija“ spaustuvė, Studentų g. 54, 51424 Kaunas