VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Modeliais grįstos programų sistemų gamyklos

Model driven software product lines

Magistro baigiamasis darbas

Atliko: Vaidotas Ratkus (parašas)

Darbo vadovas: Lektorius Donatas Čiukšys (parašas)

Recenzentas: Asistentas Petras Petkus (parašas)

Vilnius – 2011

Santrauka lietuvių kalba

Šio magistrinio darbo tikslas – pasiūlyti būdą, kaip panaudoti modeliais grįstas technikas (angl. Model Driven Development) (toliau MDD) programų sistemų gamyklų (angl. Software prduct lines) (toliau SPL) konfigūratoriams ir generatoriams kurti, integruojant kitų autorių metodų idėjas, jas papildžius ar modifikavus. Darbą sudaro trys dalys: literatūros apžvalga, pasiūlymas, eksperimentas.

Literatūros apžvalgos dalyje aprašyti straipsniai, kurie naudoja MDD technikas SPL etapuose, norint pasiekti tam tikro etapo tikslus. Analizuoti straipsniai neaprašo visų SPL dalių, apibūdina juos pakankamai skirtingai. Daugelis tų metodų pristatomi abstrakčiai.

Pasiūlymo dalyje, remiantis literatūros apžvalgos analizės rezultatais ir literatūroje aprašytomis idėjomis, suformuluotas pasiūlymas, kaip prasmingai taikyti MDD technikas kuriant SPL konfigūratorius ir generatorius. Pasiūlymo skyriuje analizuojami SPL etapų artefaktai, galimybės naudoti MDD technikas bei galimi spendimai, taikytini MDD kiekvienoje SPL dalyse.

Eksperimento dalyje aprašyta, kaip pasiūlymo dalyje pateiktas metodas realizuojamas praktiškai, kokie naudojami įrankiai, kokios atliktos veiklos ir kokie gauti rezultatai.

Santrauka anglų kalba

This MA tesis aim – suggest strategy use of model driven techniques (MDD) in software product line (SPL) for creating generators and configurators by modification and itegration ideas of other authors. In order to achieve this work is divided into three parts: literature review, a proposal and experiment.

In lliterature review part articles about MDD techniques in SPL steps to acheve its goals are described. Reviewed methods do not cover all parts of the SPL. Proposed solutions are quite different. Many of these methods are abstract.

In proposal part MDD techniques in creating SPL configurators and generators is proposed on the basis of reviewed ideas in literature. This proposal has meaningfull MDD technique use in SPL steps. Also proposal analyze SPL steps artifacts and the possibilities of using MDD techniques in them.

In the experiment part proposed method use in practice is described. It covers needed tools, technologies and activities.

Turinys

Įvadas

Pasaulyje yra sukurta labai daug programinės įrangos. Vienos iš jų sprendžia vienokias problemas, atlieka vienokias užduotis, kitos – panašias, dar kitos – sprendžia visiškai kitokias problemas. Programinė įranga skirstoma pagal sprendimus tam tikrai sričiai, kaip antai įmonių veiklai padeda ERP (angl. enterprise resource planing), ryšiams su klientais valdyti padeda CRM (angl. customer relationship management), tai pat skirstoma pagal tipą, pavyzdžiui: tvarkyklės, žaidimai, teksto redaktoriai. Taip suskirstyti programinės įrangos paketai turi tam tikrus vienodus ar bendrus, tačiau ir skirtingus, tam tikrai naudotojų daliai reikalingus bruožus. Darbe bendri bruožai vadinami bendrybėmis, o skirtingi – kintamybėmis. Programiniuose paketuose, esant tam tikriems bendriems bruožams, kyla idėja, kad juos galėtų realizuoti tie patys komponentai, sukurdami savotišką platformą. Konkrečius atskiro kliento pageidavimus galėtų realizuoti kiti – integruojami – komponentai, o užsakovo norimas programinės įrangos paketas galėtų būti sudėtas pagal konkrečius komponentus, atitinkančius užsakovo norus (pvz., vienas nori saugoti dokumentus XML, o kitas – PDF formatu ir abu nori tekstų redaktoriaus). Taip tarsi užsisakius automobilį, jį pagamins ant tos pačios platformos, ją papildydami kliento norima audio aparatūra, salono spalva ir medžiaga ir t.t. Panašiai apibūdintas programų sistemų kūrimo metodas yra analizuojamas šiame darbe. Jis vadinamas programų sistemų gamykla (angl. Software product lines). Metodas kuria ne vieną, o visą produktų grupę (darbe autorius tai vadina programų sistemų šeima), turinčią bendrų savybių (bendrybių) ir skirtumų (kintamybių). Programų sistemų šeimų kūrimą, vystymą ir panašias susijusias veiklas nagrinėja programų sistemų šeimų inžinerija. Kuriant ir vystant programų sistemų šeimas, iškyla nemažai galimybių tokių kaip kodo komponentų pakartotinio panaudojamumo galimybė, nes programų sistemų šeimos nariai (programų sistemos) turi daug bendrybių. Iškyla ir problemų, pavyzdžiui, kintamybės gali būti viena kitai prieštaraujančios ir taip įtakojančios bendrybės. Programų sistemų gamyklų metodas įgalina aprašyti programų sistemų šeimą, sukurti jai architektūrą ir mechanizmus, lengvai ir greitai jungti pakartotinai panaudojamus kodo komponentus iš norimų savybių ir kurti programų sistemų šeimos narius – programų sistemas. Šis metodas yra pakankamai abstraktus, iš specialistų reikalauja nemažai darbo, pritaikant jį praktiškai ir atliekant jo veiklas. Šio magistrinio darbo tikslas yra pasiūlyti MDD technikų pritaikymą SPL konfigūratoriams (priemonės išrinkti reikiamas savybes iš kintamybių-bendrybių modelio) ir generatoriams

(priemonės sugeneruoti programų sistemos kodą įgyvendinančias išrinktas, norimas savybes) kurti.

Modeliais grįstos technikos yra naudingos dėl įvairių dalykų tokių kaip:

Aprašymas. Norint tiksliai aprašyti koki nors objektą, jam reikia modelio. Modeliais grįstos technikos įgalina formaliai susikurti tam tikras aprašymo taisykles, nurodyti galimus aprašomojo objekto elementus. Toks aprašymas darbe vadinamas meta modeliu (modelių šeimos aprašymu). Panašiai XML dokumentus galima aprašyti XML schemomis, tačiau meta modelis skirtas ne vien dokumentų formato aprašymui, juo galima aprašyti bet kokių objektų aprašymo kalbą. Dar vienas didelis meta modelių privalumas yra tas, kad šiuo metu yra programinės įrangos, sugeneruojančios redaktorių kurti ir modifikuoti meta modeliu aprašytus modelius. Nebeprireikia patiems programuoti redaktorių.

Kodo generavimas. Iš modelių įmanoma generuoti kodą. Nemažai UML įrankių generuoja programavimo kalba aprašytą kodą iš klasių diagramų, tačiau modeliais grįstos technikos įgalina daugiau. Jos leidžia ir UML, ir kita modeliavimo kalba aprašytus modelius transformuoti į bet kokį norima kodą (xml, dokumentacija, programavimo kalba aprašytą kodą ir pan.). Tereikia aprašyti būdą, kaip modelį versti į tekstą, ir tai vadinama transformacija.

Modelių kūrimas. Dažnai skirtingos rūšies modeliai turi kažką bendro (pvz., UML klasių diagramose pavaizduoti paketai ir UML dislokavimo diagramose naudojami paketai). Akivaizdu, kad įmanoma generuoti modelio dalį (pvz., paketų elementus klasių diagramose iš dislokavimo diagramų) iš kito modelio, tai pat generuoti kodą tereikia aprašyti transformavimo būdą transformavimo kalba.

Magistriniame darbe nagrinėjamos kitų autorių idėjos, kaip panaudoti modeliais grįstas technikas tam tikrose programų sistemų gamyklų vietose. Deja, atskirai nagrinėjami autorių darbai neaprėpia visų programų sistemų gamyklų veiklų, todėl autorius pasiūlo savo idėją. Ji integruoja kelių autorių idėjas jas praplečiant ar šiek tiek modifikuojant, kad viena su kita būtų integralios. Programų sistemų gamyklų veiklas, kurias kiti autoriai neaprašo, šio darbo autorius aprašo pats. Gautas rezultatas – modeliais grįstų technikų pritaikymo kuriant programų sistemų gamyklų konfigūratorius ir generatorius pasiūlymas. Pasiūlymas pritaikytas praktiškai – atliktas eksperimentas, realizuojamos pasiūlyme aprašytos veiklos.

Tyrimo objektas

Šio magistrinio darbo tyrimo objektas yra modeliais grįsto programų sistemų kūrimo (angl. Model Driven Development) (toliau MDD) technikų taikymas programų sistemų gamyklose (angl. Software Product Lines) (toliau SPL). Kitaip tariant, domimasi modeliais grįstomis programų sistemų gamyklomis (angl. Model Driven Software Product Lines) (toliau MDSPL).

Tyrimo aktualumas ir naujumas

MDD technikų naudojimo aktualumą, kuriant programų sistemų gamyklą, autorius pademonstruoja pavyzdžiu. Programų sistemų šeimos analizės vienas iš rezultatų yra kintamybių modelio sukūrimas. Šį modelį kaip įeinantį darbo produktą naudoja programų sistemos konfigūravimo instrumentas. Taigi reikia apibrėžti formalią kintamybių modelio notaciją, saugojimo formatą, sukurti kintamybių modelių kūrimo instrumentinę priemonę ir tikrai reikės kintamybių modelio notacijos sintaksinio analizatoriaus (angl. parser). Visa tai yra labai laikui imlūs programavimo darbai. Naudojant MDD technikas, tereikia apibrėžti kintamybių modelio meta modelį (t.y., kintamybių DSL), o sintaksinis analizatorius ir kintamybių modelių (kaip ir bet kokio modelio apibrėžto meta modeliu) redagavimo instrumentinė priemonė būtų sukuriami (sugeneruojami) automatiškai, naudojant egzistuojančius MDD įrankius (pvz., naudojant xText[Ope10]).

MDD technikas taip pat galima naudoti integruojant komponentus. Tam tereikia produkto konfigūracijos modelio, jį aprašančio meta modelio ir kodo integracijas aprašančių M2C transformacijų. M2C transformacijos įrankiai gali tinkamai sujungti programų sistemų šeimos komponentų artefaktus (išeities kodus, konfigūracijos failus ir pan.) į programų sistemų artefaktus (išeities tekstus, konfigūracijos failus ir pan.). Tam tereikia tinkamai aprašyti jungimo taisykles transformacijų kalba. Tokiu būdu sukuriamas programų sistemų šeimos komponentų jungimo generatorius.

2009 m. birželio mėnesį Europos modeliais grindžiamų programų sistemų konferencijos ECMDA (angl. European Conference on Model-Driven Architecture) metu vyko pirmasis seminaras apie modeliais grįstas programų sistemų gamyklas. Versta ištrauka iš ataskaitos [MBM09].

„Modeliais grįstų programinės įrangos kūrimo (MDD) ir programų sistemų gamyklų (SPL) konceptai, technikos ir įrankiai, be abejonės, turi potencialą reikšmingai padidinti programų sistemų inžinerijos procesų produktyvumą ir kokybę. Industrijos konkuruoja, siekdamos adaptuoti šias idėjas dideliu mąstu. Tačiau akademiniu ir praktiniu požiūriu vis dar nežinoma, kaip sistemingai integruoti šias dvi sritis. Pavyzdžiui, kaip adaptuoti ir pritaikyti MDD technikas, kuriant programų sistemų gamyklas ir su jomis susijusius inžineriniuos įrankius, ir kaip integruoti SPL konceptus, kuriant modelių šeimas, meta modelius, transformacijas ir modeliavimo kalbas.“

MDD ir SPL idėjos yra vystomos, vystomas ir jų apjungimas, tačiau visa tai dar nėra iki galo įgyvendinta.

Darbo tikslai ir uždaviniai

Darbo tikslas yra pasiūlyti MDD technikų pritaikymą SPL konfigūratoriams ir generatoriams kurti, integruojant kitų autorių metodų idėjas, jas papildant ar modifikuojant.

Tam, kad darbo tikslas būtų pasiektas, reikia atlikti šiuos uždavinius:

- Atlikti MDSPL metodų literatūros apžvalgą.
- Išanalizuoti MDD technikas.
- Sukurti SPL metodo modifikaciją, numatant vietas, kuriose gali būti prasmingai taikomos MDD technikos SPL konfigūratoriams ir generatoriams kurti.

Laukiami rezultatai

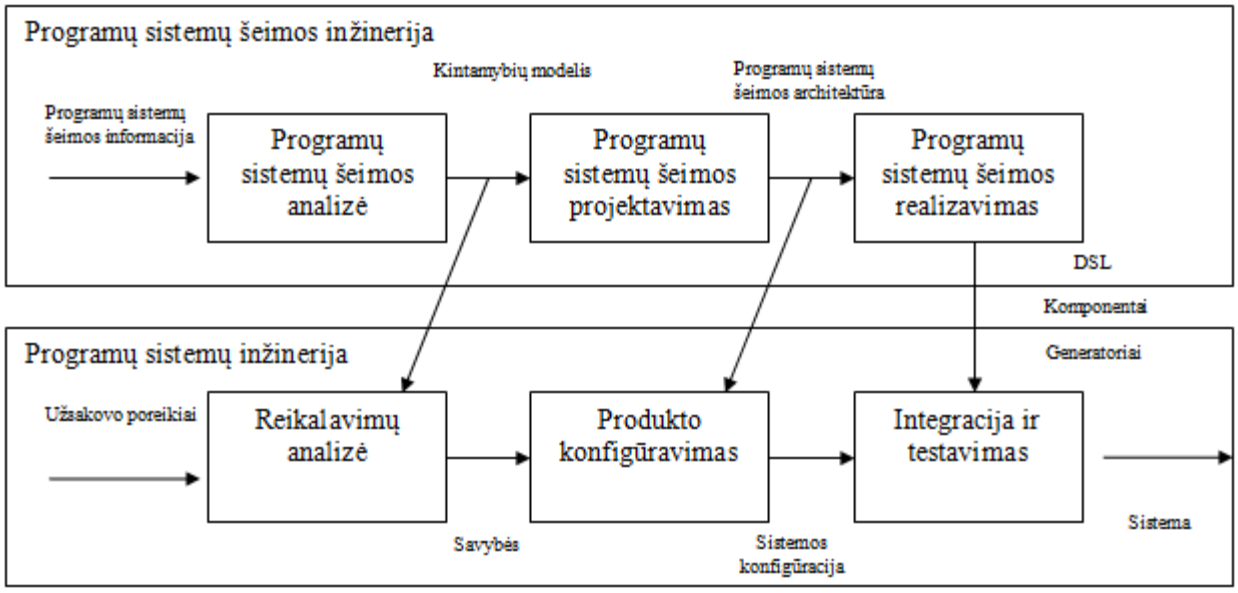
Magistrinio darbo metu planuojama pasiekti šį rezultatą:

- SPL konfigūratorių ir generatorių kūrimo pasiūlymai, paremti MDD technikomis.

1. Programų sistemų gamyklos apžvalga

Pirmas straipsnis programų sistemų gamyklos tematika, kaip tai apibrėžia SEI (angl. Software engineering institute) savo žiniatinklyje, buvo parašytas 1998 m. pavadinimu „Perspective of software reuse“ [Per88]. Šiame straipsnyje be kitų idėjų yra idėja apie sistemos išvedimą iš bendro sistemų modelio, parenkant reikalingas sistemos savybes.

Praėjus tam tikram laikui, SEI apibrėžė programų sistemų kūrimo metodą – programų sistemų gamyklas SPL (angl. Software prduct lines) [CN07]. Programų sistemų gamyklas SEI apibrėžia kaip programų sistemų artefaktų rinkinį ir būdą iš jų kurti programų sistemas. Šios programų sistemos orientuotos į tam tikrą dalykinę sritį ar rinkos segmentą ir patenkina to segmento poreikius. Šios programų sistemos kuriamos apibrėžtu būdu, naudojant bendrus pirminius šaltinius (išteklius).



1 pav. SPL metodas

Programų sistemų gamyklą sudaro šios veiklos: (1 pav.) [Cza98]:

1. Programų sistemų šeimos inžinerija

• Programų sistemų šeimos analizė. Šios veiklos tikslas yra aprašyti programų sistemų šeimą. Tai padaroma analizuojant įvairią informaciją apie potencialius programų sistemų šeimos narius. Pagrindinis šios veiklos darbo produktas yra kintamybių modelis, kuriame nurodomos programų sistemų šeimos kintamybės ir bendrybės.

• Programų sistemų šeimos projektavimas. Šios veiklos tikslas – sukurti programų sistemų šeimos architektūrą, kurioje aprašomi apibendrinti komponentai ir jų tarpusavio sąveika. Architektūra turi būti bendra visiems programų sistemų šeimos egzemplioriams.

• Programų sistemų šeimos realizavimas. Šios veiklos tikslas yra realizuoti programų sistemų šeimos komponentus, generatorius, skirtus automatiniam komponentų jungimui.

2. Programų sistemos inžinerija:

• Reikalavimų analizė. Šios veiklos tikslas yra išsiaiškinti, kurios iš kintamų savybių yra esminės šiai konkrečiai programų sistemai. Tai daroma analizuojant užsakovo poreikius.

• Programų sistemos konfigūravimas. Šios veiklos tikslas yra parinkti programų sistemų šeimos kintamybes ir sukurti sistemos konfigūraciją. Tai daroma paimant kintamybių modelį, išsirenkant kintamybes, kurias analizuoja SPL įrankis – konfiguratorius. Naudodamas kintamybių modelyje apibrėžtas taisykles konfiguratorius tikrina, ar pasirinktos kintamybės yra neprieštaringos. Jei taisyklės nėra pažeidžiamos, tai konfiguratorius pateikia rezultatą – sistemos konfigūraciją.

• Integracija ir testavimas. Šios veiklos tikslas yra sukurti programų sistemų šeimos egzempliorių – programų sistemą. Tai daroma pasinaudojant generatoriais ir DSL pagal konfigūraciją sugeneruojant reikalingą kodą ir sujungiant reikalingus komponentus.

2. Programų sistemų gamyklos konfigūratorius ir generatorius

Vieni iš programų sistemų gamyklų įrankių yra konfiguratorius ir generatorius. Konfigūratorius tai programinė įranga leidžianti parinkti reikalingas savybes (kintamybes) iš kintamybių-bendrybių modelio norint patenkinti tam tikrą reikalavima (poreikį) ir iš tų savybių

suformuluoti sistemos konfigūraciją .pvz. vienas klientas nori dokumentus saugoti doc faile kitas-xml, tai vieno kliento norimo produkto sistemos konfigūracijoje bus doc failo savybe atitinkanti kintamybė kito – xml. Toks įrankis tai pat turi užtikrinti ,kad norima produktą galima realizuoti ir jis veiks pvz. užtikrinti ,kad nebūtu įmanoma išsirinkti automobilio su elektriniu varikliu ir benzino baku.

Generatoriaus tikslas yra turint norima sistemos konfigūraciją (kliento reikalaujamų savybių) sugeneruoti programų sistemas. Tokie generatoriai jungia esamus programų sistemų šeimos modulius, juos reikiamai integruoja ir surenka produktą. Generatoriai užtikrina ,kad kliento reikalaujamas savybes realizuojantys modeliai atsidurtu produkte (programų sistemoje) ir ,kad modeliai būtu tinkamai sukonfigūruoti, kad tinkamai vienas su kitu dirbtu.

3. Modeliais grįstos technikos

Šiame skyriuje autorius supažindina su modeliais grįstomis technikomis ir galima jų nauda. Informacija paimta iš M. Karlsch darbo „A model-driven framework for domain specific languages“ [Kar07].

Kaip ir tradiciniame programų sistemų kūrimo procese, pavyzdžiui, krioklio (ang. Watterfall) galutinis produktas (programų sistema) evoliucionuoja iš tam tikrų modelių (reikalavimus apibrėžiančio – reikalavimų specifikacijos, pagal ją paruošto sprendimo aprašymo – architektūros, pagal ją pagaminto produkto – programų sistemos). Ši evoliucija transformuoja vienus modelius į kitus (pvz., reikalavimų specifikaciją į architektūrą). Transformacija dažnai yra neformali, nes ir modeliai neformalūs (pvz. reikalavimų specifikacija aprašyta naudojimo scenarijais). Modeliais grįstų technikų pagrindas yra formalūs modeliai. Juos aprašo meta modeliai – modeliai aprašantys modelius. Meta modeliai leidžia formaliai aprašyti, kokias klases turės modelis, kokius atributus turės tos klasės ir t.t. Formalus modelis leidžia aprašyti formalias transformacijas į kitą formalų modelį, aprašant taisykles, kaip panaudoti šaltinio modelio elementus, kuriant rezultato modelį, pavyzdžiui, kiekvienam UML dislokavimo diagramos modelyje esančiam paketui sukurti po paketą UML klasių diagramoje. Tai pat galima iš modelio generuoti kodą, tai atlieka daugelis UML įrankių iš klasių diagramos, kiekvieną klasę diagramoje aprašant ją programavimo kalbos sintakse. Transformacijas galima atlikti automatiškai pasinaudojant MDD įrankiais.

Kadangi meta modelis formaliai aprašo modelio struktūrą, egzistuoja įrankiai (pvz. xText[Ope10], GMF [EF10a]), kurie sugeneruoja tokių modelių redaktorius. Modeliai dažnai modeliuojami juos aprašius modeliavimo kalba. Tokia kalba dar vadinama dalykinei sričiai specifinė kalba (ang. Domail Specific Language, toliau DSL). Ji gali būti grafinė (skirta modeliuoti grafiniais primityvais) ir tekstinė (skirta modeliuoti žodžiais). Taip pat modelių redaktoriai, naudojantys DSL, yra grafiniai arba tekstiniai. Tekstiniai redaktoriai analizuoja teksto sintakse ir verčia tekstą į modelį, saugomą tam tikru formatu (pvz., MOF). Grafiniai redaktoriai modelio elementus vaizduoja grafiniais primityvais, pavyzdžiui, UML įrankiai UML klasių diagramose klases vaizduoja stačiakampiais.

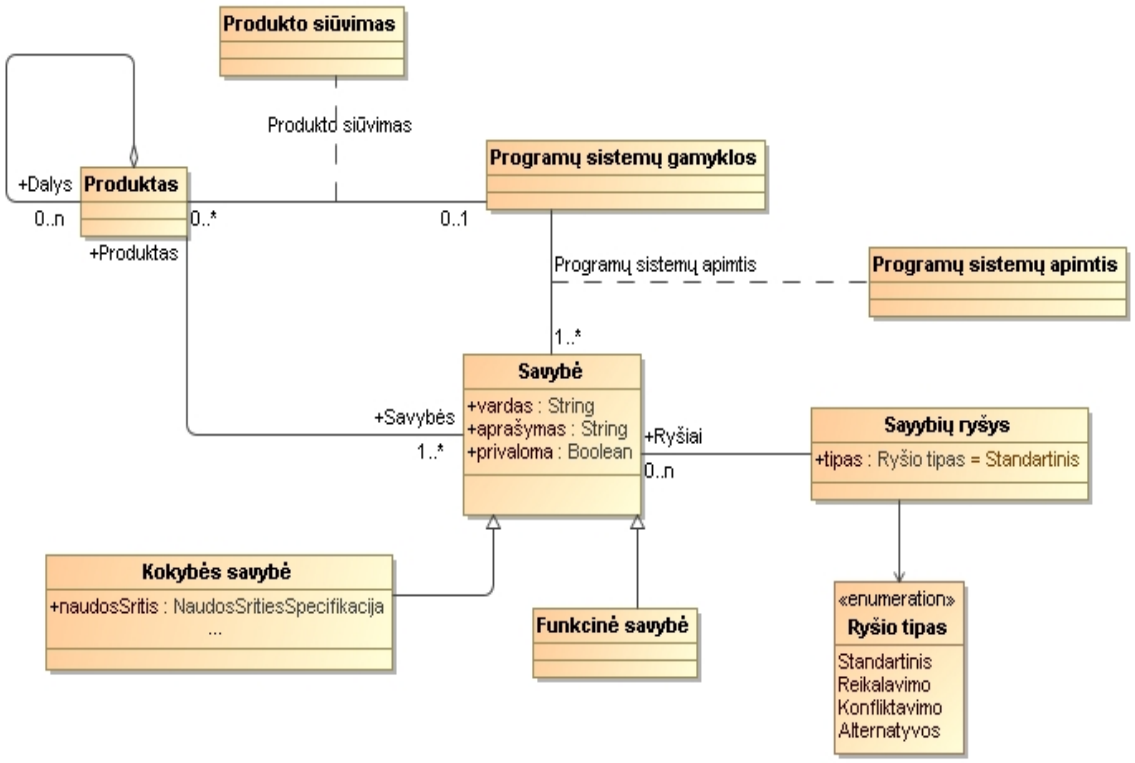
Apibendrinant, MDD technikos leidžia formaliai modeliuoti modelius, juos automatizuotai transformuoti į kitus modelius ar kodą, naudotis sugalvota modeliavimo kalba ir notacija (tekstine ar grafine), modeliuoti pasinaudojus modelių redaktoriais.

4. Modeliais grįstų programų sistemų gamyklų apžvalga

Iki šiol yra sukurta nemažai modeliais gristų programų sistemų gamyklų metodų [TP08]. Šiame skyriuje bus aptarti keli iš jų. Moksliniai straipsniai susieti pagal jų idėjas realizuoti programų sistemų gamyklų etapus.

4.1. Programų sistemų šeimos analizė

Programų sistemų šeimos analizės siekis – aprašyti programų sistemų šeimą [CN07]. Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Arnor Solberg [HPOS05] straipsnyje „An MDA®-based framework for model-driven product derivation“ pasiūlė orientuotis ne į kintamybių modeliavimą, o į rinkai svarbių savybių modeliavimą. Metodo esmė yra modeliuoti į rinką orientuotas savybių terminais, pavyzdžiui, Microsoft office programinė įranga (Word, Excel ir kitos). Ji yra produktų grandinė, tenkinanti tam tikro rinkos segmento poreikius, tačiau jos produktai nebutinai galėtų būti pagaminti iš pakartotinai panaudojamų komponentų (pvz. Word ir Excel nebutinai didžioji dalis sistemos turetų bendrus komponentus), nors patenka į programų sistemų rinkos segmentą, skirta patenkinti sekretorių darbą ir parduodamas kaip vienas paketas. Šis metodas aprašo kelias programų sistemų šeimas, naudojant produktų šeimos modelius.

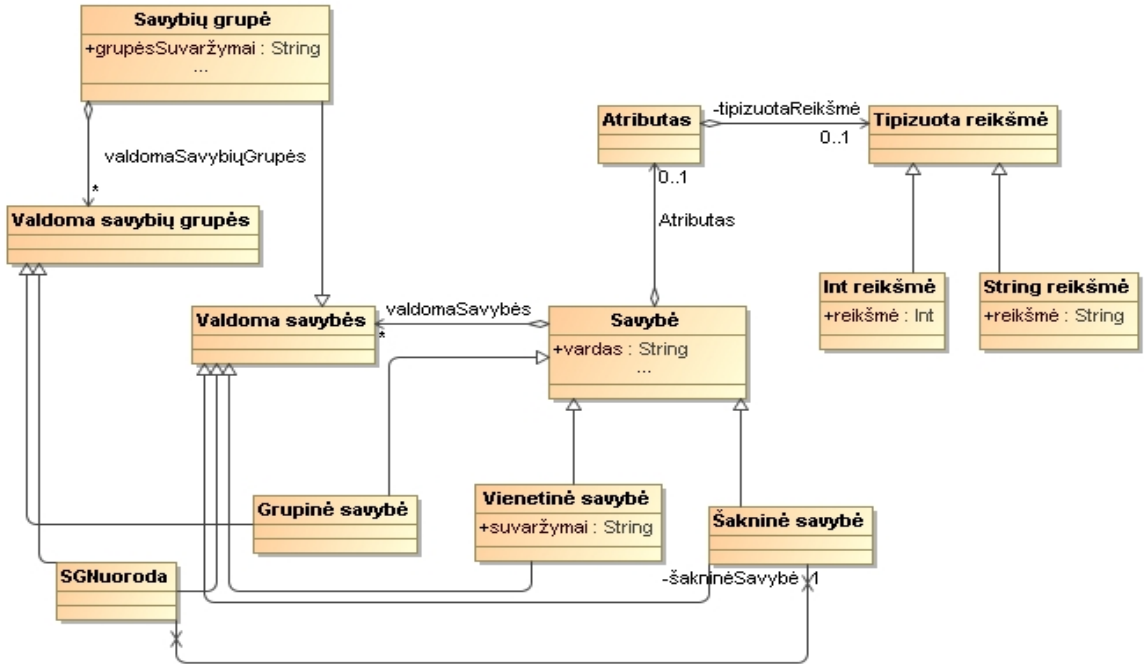


2 pav. Produktų šeimos meta modelis [HPOS05]

Produktų šeimos meta modelis (2 pav.) apibudina programų sistemų gamyklas ir produktus naudojant į rinką orientuotų savybių rinkinius. Programų sistemų šeimos egzempliorius (produktus) gamina programų sistemų gamykla, tad šiame modelyje programų sistemų šeimą apibudina jos egzempliorius gaminanti programų sistemų gamykla. Kaip paveikslėlyje matoma programų sistemų gamykla nurodo, kokias savybes (kokybines, funkcines) ji gali realizuoti ir kokius produktus ji gali gaminti pasinaudodama produkto siuvimo objektu. Šis objektas nurodo, kaip produktas turi būti konstruojamas, gaminamas. Savybės šiame modelyje turi du tipus: funkcinį, aprašantį funkcionalumą, ir kokybinį, aprašantį funkcionalumo kokybę. Šie tipai turi tarpusavio ryšius: standartinį, reikalavimo, konfliktavimo, alternatyvos. Deja, daugiau informacijos apie ryšius straipsnis nepateikia, todėl jis yra pakankamai abstraktus.

Šis metodas naudoja meta modeliavimą ir modeliavimą, reikalingą MDD transformacijoms. Šios transformacijos yra aprašytos kituose skyriuose.

Orlando Avila-García, Antonio Estévez García, E. Victor Sánchez Rebull savo straipsnyje „Using Software Product Lines to Manage Model Families in Model-Driven Engineering“ [GGR07] siūlo programų sistemų šeimas aprašyti CBFM (angl.Cardinality-Based Feature Model) modeliu (3 pav).



3 pav. CBFM meta modelis[GGR07]

Modelis aprašo programų sistemų šeimas, pasitelkiant medžio vaizdinį, kur medis prasideda nuo šakninės savybės. Savybės turi tekstinius ir skaitinius atributus ir jos gali būti jungiamos į savybių grupes. Tai pat savybės turi nuorodas („SGNuoroda“ pavadinta klasė paveikslėlyje) į kitas paveikiamas, valdoma savybes („Valdoma savybės“ pavadinta klasė paveikslėlyje). Modelyje tai pat aprašyti savybių nuorodų ir valdymo ryšiai.

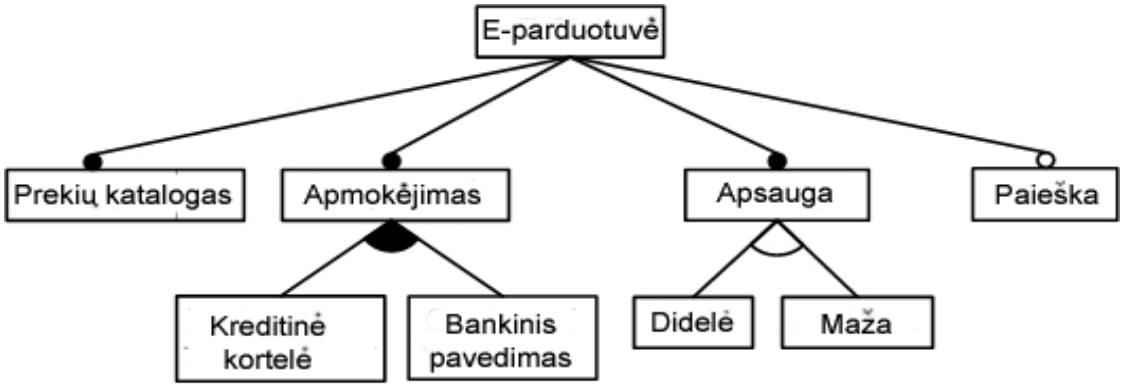
Šis metodas naudoja meta modeliavimo (formaliai apibrėžiant CBFM modelio struktūra meta modeliu) ir modeliavimo (kuriant CBFM meta modeliu aprašytus modelius) MDD technikas. Straipsnyje aprašoma, kaip iš meta modelio sugeneruojamas redaktorius, galintis kurti ir redaguoti meta modelio aprašytus modelius.

J. Oldevik ir O. Haugen straipsnyje „Higher-Order Transformations for Product Lines" [OH07] nesiūlo tikslių programų sistemų šeimų kintamybių modeliavimo metodų, tačiau siūlo kintamybes grupuoti į platformos (pvz., programavimo kalbos) ir dalykinės srities savybes (angl. Domain feature). Šis straipsnis programų sistemų šeimų aprašymą apibūdina labai abstrakčiai.

Alexandre Bragança and Ricardo J. Machado straipsnyje „Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines"[BM07] siūlo programų sistemų šeimas apibrėžti *use case* diagramomis. Metode *use case* modelio elementai turi sąryšius ir ribojimus į savybių modelį, o pastarieji į konfigūracijos modelį. *Use case* modelis susiejamas su savybių modeliu naudojant anotacijas. Savybių modelis yra medis, kuris susideda iš šakninių savybių, pastarieji turi vaikines savybes. Visos savybės turi atributus. Deja, straipsnis neaprašo konfigūracijos modelio struktūros.

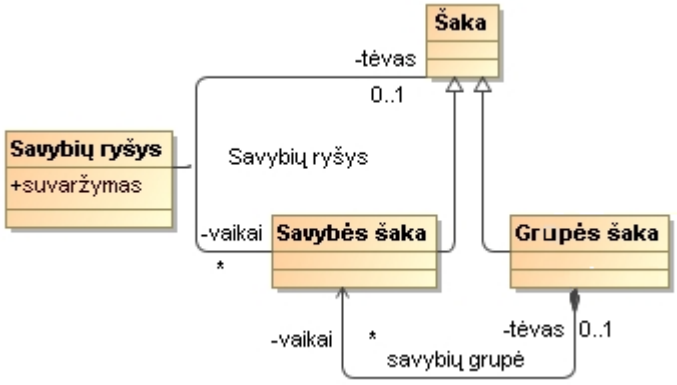
Naudojant šį metodą, programų sistemų šeimos yra aprašomos gerai žinoma ir populiaria UML modeliavimo kalba, tai pat yra pasiekiamas reikalavimų (*use case* elementų) ir juos įgyvendinančių savybių trasuojamumas. Metodas naudojasi tuo, kad UML *use case* turi savo meta modelį ir jame yra galimybė formaliai turėti nuorodas (pvz., per anotacijas) į kitą modeli (šiuo atveju kintamybių modelį).

Panašų sprendimą siūlo Krzysztof Czarnecki, Michał Antkiewicz,Chang Hwan Peter Kim, Sean Lau, Krzysztof Pietroszek straipsnyje „Model-Driven Software Product Lines" [CAK+05]. Autoriai siūlo programų sistemų šeimas aprašyti savybių kintamybių bendrybių taksonomine forma (angl. taxonomic form). Savybės apibrėžia programų sistemų šeimų funkcines ir nefunkcines charakteristikas. Toks savybių modelis yra vaizduojamas savybių hierarchiniu medžiu ir ryšiais. Toks modelis pavaizduotas 4 paveikslėlyje. Bendros ir būtinos savybės pažymėtos užtušuotu rutuliuku. Savybės, jungiamos lanku, reiškia vieną pasirenkamą savybę. Savybė, sujungta linija, reiškia, kad ji yra nebūtina.



4 pav. Savybių kintamybių bendrybių modelis.

Šis modelis turi savo abstraktų meta modelį, aprašytą Krzysztof Czarnecki, Simon Helsen, Ulrich Eisenecker straipsnyje „Formalizing Cardinality-based Feature Models and their Specialization“ [CHU05] (5 pav.). Jame pavaizduota, kaip realizuojama hierarchija. Kiekviena savybė turi tėvą ir vaikus. Savybės gali būti jungiamos į grupes bei gali turėti ryšius, pavyzdžiui, būtinumo, alternatyvos.



5 pav. Savybių modelio meta modelis [CHU05]

Kintamybių naudojimui autoriai siūlo FBMT (angl. EATURE-BASED MODEL TEMPLATES) šablonus. FBTM susideda iš savybių modelių (modelis aprašytas programų sistemų šeimų analizės skyriuje) ir anotuotų modelių, įgyvendinančių savybes. Anotacijos nurodo į savybes savybių modelyje. Pavyzdžiui, anotacijos gali savyje turėti:

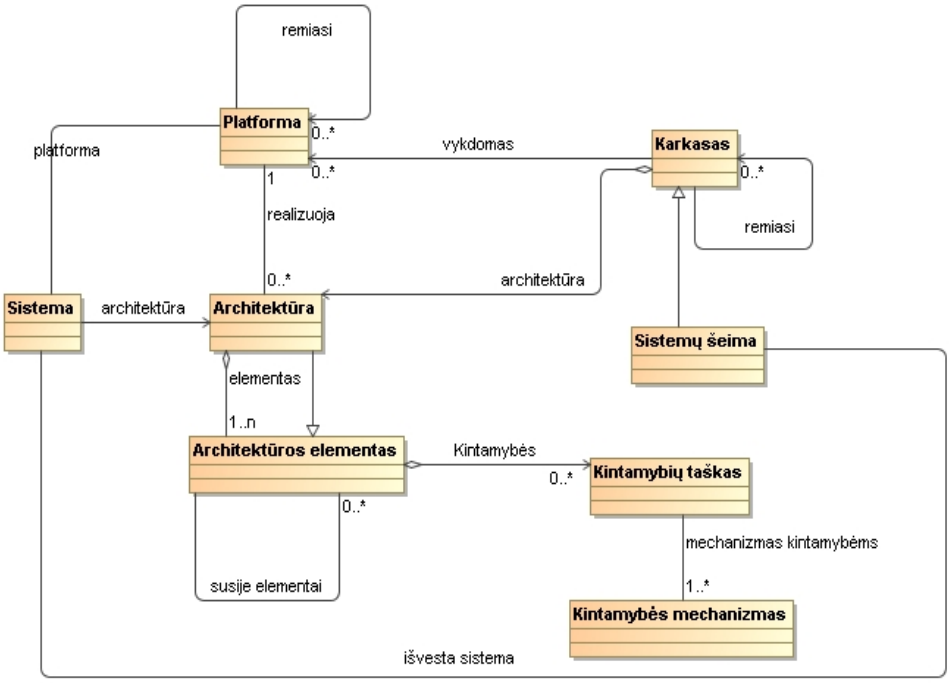
- aprašytas sąlygas; šios sąlygos panašios į #ifdef direktyvą C preprocesoriui, bet taikoma modeliu elementams.
- iteracijų nurodymus; šie nurodymai gali būti naudojami iteracijoms šablonuose.
- ir/arba meta išraiškas; šios išraiškos gali būti naudojamos modelio elementų manipuliavimui.

Autorių aprašytas metodas veikia naudojat MOF modelius tokius kaip UML, bet gali būti adaptuotas ir kitiems meta modeliavimo formalizmams.

FBMT atvaizduoja reikalavimų, projektavimo ir realizavimo lygio modelių variantus super įdėtine (angl. Superimposed) forma. Variantai gali būti naudojami kaip vienas artefaktas. Be to, anotacijos suteikia trasavimą tarp savybių savybių modelyje ir elementų, kurie jas realizuoja. Šiam konceptui įrodyti yra sukurti keli įrankiai: fmp, fmp2rsm, template verifier. Nors ir yra realizuoti įrankiai, kaip jie veikia straipsnio nepakanka. Straipsnyje aprašytas labai abstraktus metodas, nuodugniai neaiškinama, kokie galėtų būti modeliai ir šablonai. Nepaisant to, idėja yra pakankamai aiški – MOF modelį (pvz. UML fmp2rsm įrankyje) praplėsti anotacijomis, norint modeliuoti kintamybes.

4.2. Programų sistemų šeimos projektavimas

Programų sistemų šeimų analizės skryriuje Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Arnor Solberg [HPOS05] straipsnyje „An MDA®-based framework for model-driven product derivation“ aprašytas programų sistemų šeimų modelis autorių aprašytame metode naudojamas programų sistemų šeimų apibrėžimai, tačiau jų realizavimą aprašo sistemų šeimos modelis (angl. system model). Jei programų sistemų šeimų modelis programų sistemų šeimos analizės etape aprašo daug programų sistemų šeimų, tai sistemų šeimos modelis aprašo daug programų sistemų šeimų architektūrų. Modelis aprašo produktų šeimos architektūrą, kokiai platformai ji skirta, kokiame karkase veikia, kur ir kaip realizuojamos kintamybės (6 pav.).



6 pav. Sistemų modelio meta modelis [HPOS05]

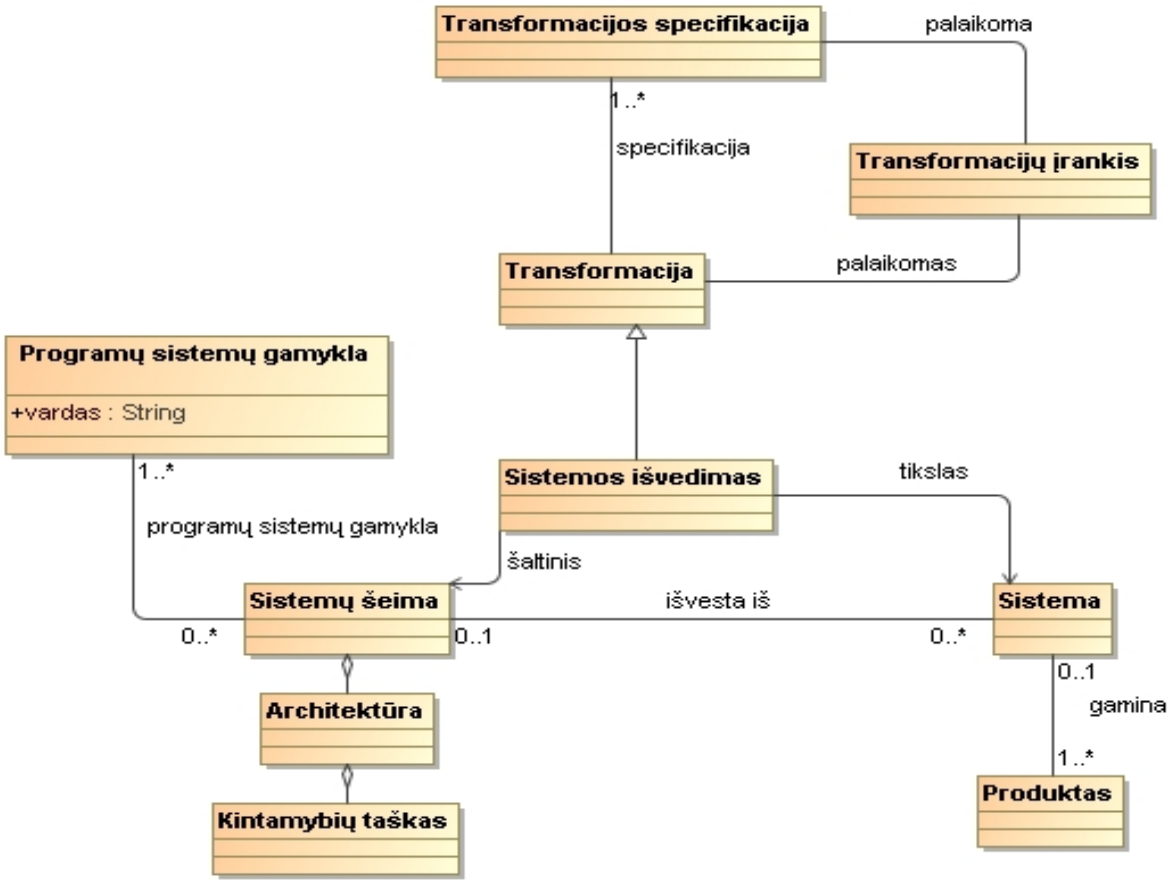
Metodas yra pakankamai abstraktus ,nes straipsnyje projektavimas yra keturiomis pastraipomis aprašoma tik kaip idėja, jis naudoja meta modeliavimo ir modeliavimo MDD technikas.

4.3. Programų sistemų šeimos realizavimas

Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Arnor Solberg [HPOS05] straipsnyje „An MDA®-based framework for model-driven product derivation“ programų sistemų šeimos realizavimui siūlo sistemų išvedimo (angl. system derivation) procesą. Šis procesas siekia specifikuoti produktą per savybes ir programų sistemų gamyklomis iš programų sistemų šeimos modelio. Kitais žodžiais tariant, programų sistemų gamyklos susiejamos su produktų savybėmis vienas su vienu ryšiu. Produktų šeimos modelis yra aprašytas programų sistemų šeimos analizės skyriuje, o sistemų šeimos modelis – programų sistemų šeimos projektavimo skyriuje. Šis procesas susideda iš dviejų smulkesnių procesų:

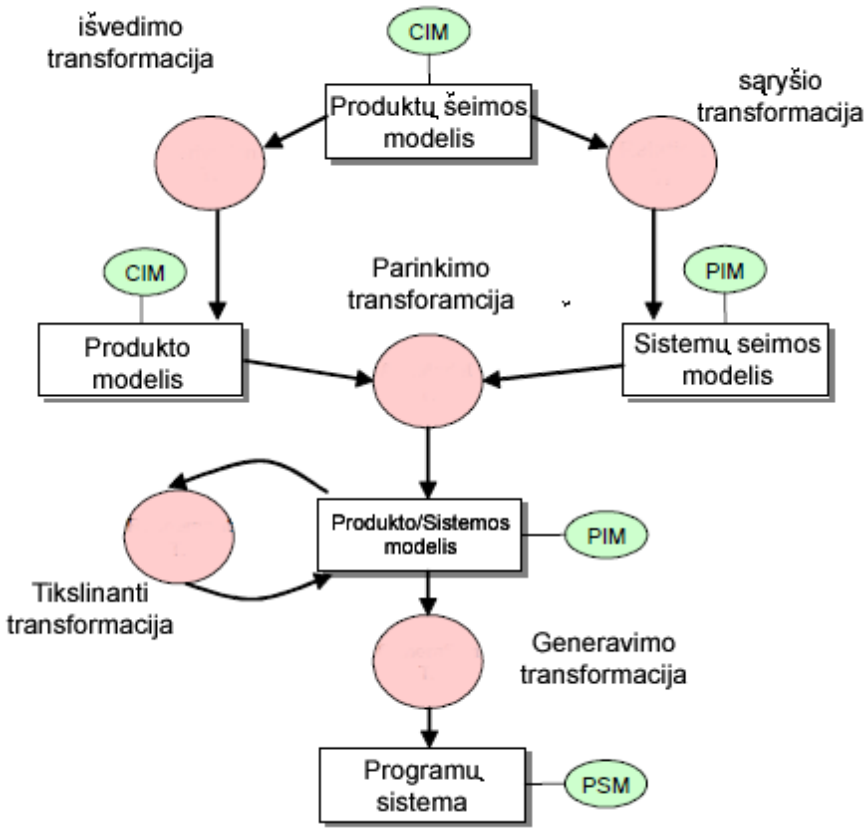
- Iš programų sistemų gamyklų išrenkamos savybės, kurios visiškai specifikuoja produktą.

- Iš sistemų šeimos modelio išvedama sistema, naudojant produktų savybes kaip konfigūraciją.



7 pav. Sistemos išvedimo modelis [HPOS05]

Šis metodas naudoja MDA [JM03] abstrakcijos lygmenis ir MDD technikas – transformacijas (8 pav.).



8 pav. MDA modelių lygiai atvaizduoti į metodo modelius[HPOS05]

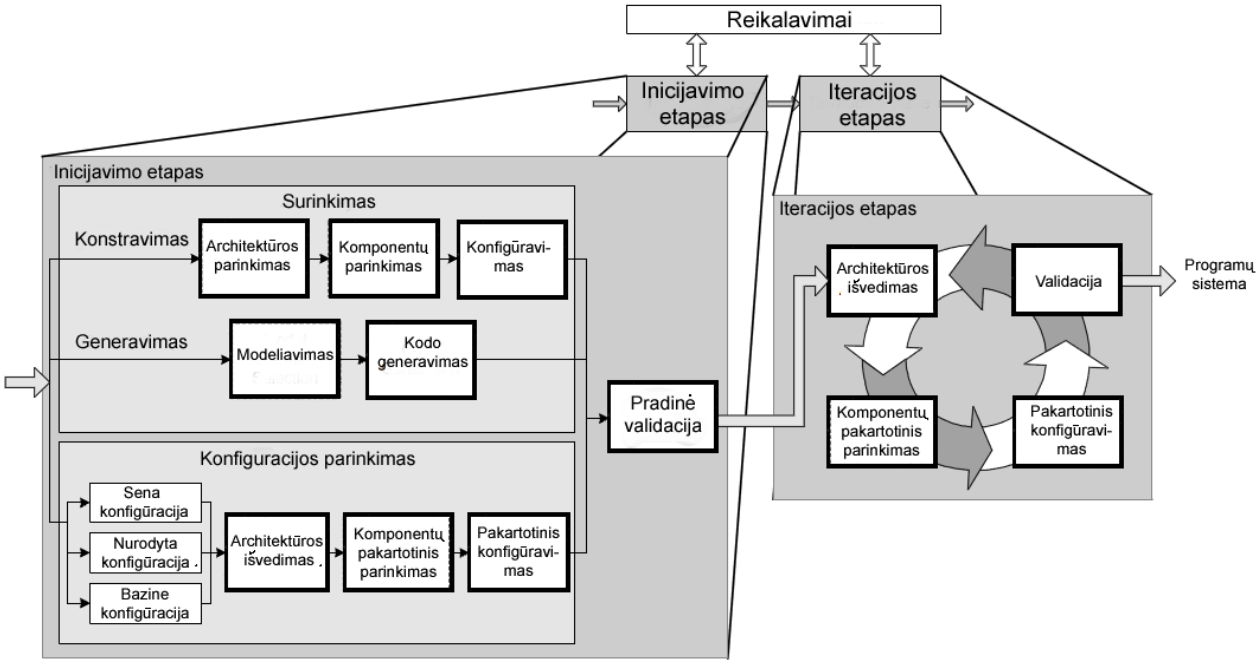
Produktų šeimos ir produkto modelis suprantamas kaip CIM modelis. Produktų šeimos modelis aprašo bendras į rinką orientuotas programų sistemų gamyklos realizuojamas gamyklas, o produkto modelis – konkretaus produkto savybes (išrinktas iš galimų programų sistemų šeimų savybių). Sistemų šeimos ir sistemos modelis suprantamas kaip PIM modelis. Jis aprašo architektūrinius sprendimus. Gautas sugeneruotas kodas – PSM modelis. Programų sistemų išvedimas yra kelių transformacijų rezultatas:

- Sąryšis tarp produktų ir sistemų šeimos modelių aprašomas „Sąryšio“ transformacija.
- Straipsnyje transformacija iš sistemų šeimos modelio į produkto/sistemos modelį nėra apibrėžta, tačiau pasakyta, kad ši transformacija gali būti rankinė ar pusiau automatizuota.

- „Generavimo“ transformacija iš Produkto/Sistemos modelio generuoja konkrečią sistemą.

Deja, kitos (tikslinimo , išvedimo) transformacijos straipsnyje nėra apibrėžtos. Pats metodas yra abstraktus nes dalis transformacijų net nėra apibrėžtos.

Sybren Deelstra, Marco Sinnema, Jan Bosch straipsnyje „Product derivation in software product families: a case study“ [DSB03] programų sistemų gamyklų kūrimas suskirstomas i du etapus: inicijavimo ir iteracijos (9 pav.).



9 pav. Inicijavimo ir iteracijų etapai [DSB03]

Inicijavimo etapo įvestis yra reikalavimai. Inicijavimo etapas susideda iš trijų dalių.

- Surinkimas. Šis etapas turi du kelius: konstravimo ir generavimo. Konstravimo kelyje pradinė konfigūracija (reikalingi komponentai, įrankiai ir pan. įgyvendinantys tam tikrą fonkcionalumą) sudaroma iš architektūros parinkimo, bendrų komponentų parinkimo ir tų komponentų konfigūravimo. Generavimo keliu yra naudojama modeliavimo ir transformacijų MDD technika. Meta modeliu apibrėžiami artefaktai reikalingiems modeliavimo tikslams (pvz., komponentų funkcionalumo parinkimui). Po to iš jų

generuojama konfigūracija vietoj to, kad tai būtų rašoma rankomis. Po to modelio artefaktai surenkami į bendrą modelį ir pradinė konfigūracija yra integruojama ir komponuojama programų sistema.

- Konfigūracijos parinkimas. Šio etapo metu visų pirma parenkama konfigūracija. Ji gali būti:
    - Sena konfigūracija. Ankstesnių projektų produktų konfigūracija.
    - Nurodyta konfigūracija. Sena konfigūracija su nuorodomis į naujo produkto specifinius parametrus.
    - Bazinė konfigūracija. Nauja konfigūracija, kurios senesni projektai neįtakoja.

  Po to yra pertvarkoma architektūra, iš naujo parenkami komponentai, pertvarkomi komponentų parametrai.
- Pradinė validacija. Pradinės validacijos metu patikrinama, ar parinkta konfigūracija atitinka reikalavimus, jei ne, tai inicijavimo etapas pradedamas iš pradžių.

Inicijavimo etapo rezultatais jau galima konstruoti reikalavimus atitinkančius produktus, tačiau dažnai reikia dar kelių iterakcijų, kad produktai atitiktų patikslintus reikalavimus, ar, esant reikalavimų pasikeitimui, sistemos evuliocinavimui.

Iteracijų etapo metu yra atliekamos modifikavimo ir validavimo veiklos. Modifikavimo veiklos atliekamos architektūriniu, komponentiniu ir konfigūracijos abstrakcijos lygmeniu. Validavimo veiklos esmė yra validuoti konfigūraciją, kad ji atitiktų reikalavimus.

Šis metodas naudoja modeliavimo ir transformacijų MDD technikas. Metodas yra gana senas ir generavimo strategija yra parenkama kaip vienas iš būdų. Šio metodo nauda – iteratyvus procesas.

Orlando Avila-García, Antonio Estévez García, E. Victor Sánchez Rebull savo straipsnyje „Using Software Product Lines to Manage Model Families in Model-Driven Engineering“ [GGR07] įvairių artefaktų generavimui siūlo naudoti DSTL (angl. Domain specific transformation language). DSTL yra transformacijų kalba, skirta aprašyti transformacijas iš savybių modelio į kodo šablonus (M2C transformacijas). Darbe aprašyta DSTL vadinama MTTL (angl. model template transformation language), šios kalbos aprašytų transformacijų rezultatas – ATL (angl. Atomic Transformation Language) kalbos tekstas. ATL yra bendros paskirties transformacijų kalba, palaikoma įrankių. Kitais žodžiais tariant, DSTL transformacijų rezultatas – ATL transformacijos.

Savybių modeliui šiame metode yra CBFM modeliai, aprašyti programų sistemų šeimų analizės skyriuje. Pagrindinė straipsnio idėja yra naudoti į programų sistemų gamyklų dalykinę sritį orientuotą transformacijų kalbą, kuri skirta aprašyti su programų sistemų šeimos savybių modeliu susijusias problemas ir transformuoti ta kalba parašytas transformacijas į bendrosios paskirties transformacijų kalbą. Pastaroji kalba yra skirta spręsti bendros paskirties problemas. Metodas naudoja meta modeliavimo (apibrėžiant ir naudojant CBFM meta modelius) ir transformacijų (transformuojant CBFM modelius į ATL) bei DSL (aprašant DSTL kalbą) MDD technikas.

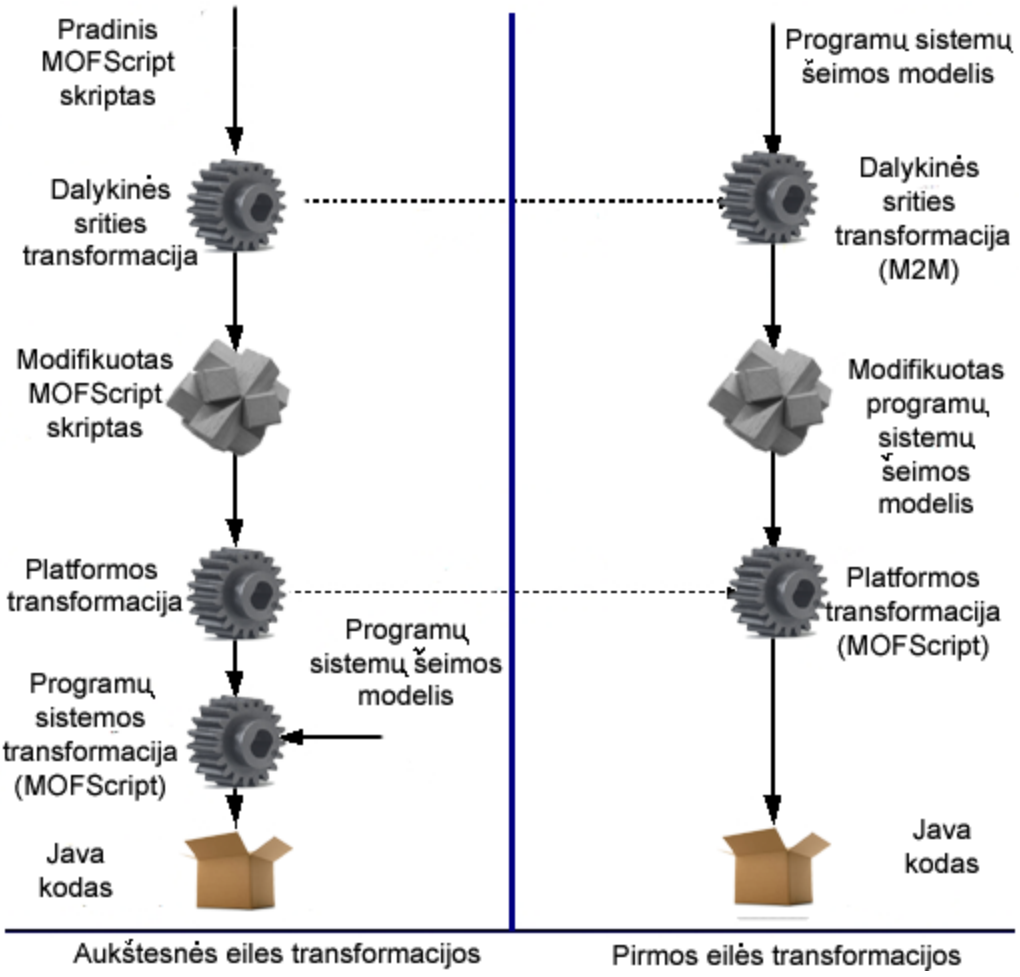
Panašus sprendimas yra aprašytas J. Oldevik ir O. Haugen straipsnyje „Higher-Order Transformations for Product Lines" [OH07]. Autoriai transformacijoms siūlo naudoti aukštesnės eilės (angl. High order) transformacijas vietoj standartinių M2M ir M2C transformacijų. Autoriai taip pat sukuria įrankį, pavadintą Hi-Transform. Šis įrankis praplečia MOFScript (bendrosios paskirties M2C transformacijų kalba) galimybes. Aukštesnės eilės transformacijos pasižymi tuo, kad jos įeitis ir išeitis yra MOFscript tekstas (įeitis tai pat turi ir sistemos konfiguracinį modelį). Naudojant šį metodą, MOFscript tekstas papildomas aspektais bei aprašytomis kintamybėmis. Po to jis paduodamas Hi-transform įrankiui kaip įvestis. Hi-transform įrankio MOFscript tekstas yra modifikuojamas aspektais aprašytomis taisyklėmis.

Yra aprašytos šios dvi taisyklės:

- Hi-Query. Aprašoma transformuojama vieta, pavyzdžiui, MOFscript kodo antra eilutė.
- Hi-Rule. Aprašoma transformacija, kaip ji turi būti modifikuota Hi-Query nurodyta vieta. Pavyzdžiui, su sąlyga sistemos konfigurafijos modelyje yra aprašyta, kad produktas bus „java" kalba ir rezultatas bus MOFscriptas, gebantis generuoti java kodą iš to paties sistemos konfigūracijos modelio, tačiau iš kitų savybių (pvz., produktas turės paiešką, vartotojų autorizavimą ar pan.)

Autoriai išskiria du transformacijos abstrakcijos lygmenis:

- Platformos. Transformacija, kuri nurodytą funkcionalumą įgyvendina tam tikra nurodyta platforma.
- Dalykinės srities. Transformacija išrenka funkcionalumą iš kintamybių.



10 pav. Aukštesnės rikiuotės transformacijos ir pirmos rikiuotės transformacijų procesai.[OH07]

Lyginant su standartinėmis transformacijomis (10 pav.), aukštesnės eilės transformacijos gali būti naudojamos tiek M2C, tiek M2M transformacijų tikslams.

5. Microsoft požiūris į programų sistemų gamyklas

Mircosoft tai pat yra sukūrusi metodą kurti programų sistemų šeimoms, tačiau metodas, nors ir vadinasi panašiai, yra šiek tiek kitoks. Strukturizuotą programų sistemos medžiagų visumą jie vadina programų sistemų gamykla (angl. software factory) [GS03]. Microsoft panaudojo generatyvinio programavimo (Generative programming)[Cza98] ir SPL idėjas ir sukūrė gamybai skirtą karkasą. Kiekviena programų sistemų gamykla savyje talpina tarpusavyje susijusių dalių

rinkinį.

- Gamyklos schema (angl. Factory schema). Schema, iliustruojanti veiklas, sugrupuotas pagal požiūrį ir sisteminę medžiagą.
- Nurodymai. Nurodymai tiekia pavyzdį realistiško, pabaigto produkto kūrimo proceso, kuri programų gamykla padėjo pagaminti.
- Architektūrinės gairės ir šablonai. Architektūrinės gairės ir šablonai padeda paaiškinti programos dizaino pasirinkimą ir motyvuoti tą pasirinkimą.
- Vartotojo žinynai. Vartotojo žinynai teikia nurodymus, kaip atlikti užduotis.
- Receptai. Receptai pasirinktais žingsniais automatizuoja procedūras iš vartotojo žinyno. Jie padeda programuotojui pabaigti monotoninius veiksmus, panaudojant mažiausiai įvesčių.
- Karkasai (angl. templates). Karkasai yra programų sistemos šablonas. Jis turi daug pagamintų elementų ir vietas, kur reikia „prigaminti“ reikiamą funkcionalumą.
- Redaktoriai. Redaktoriai tiekia informaciją, kurią architektas ir/ar programuotojas gali naudoti modeliuojant programą iš aukštesnės abstrakcijos požiūrio. Redaktoriai generuoja kodą pagal architektūrines gaires.
- Pakartotinai naudojamas kodas. Pakartotinai naudojamas kodas – tai kodo komponentai, karkasai ar programų blokas. Tai įgyvendinti kodo blokai su žinomu funkcionalumu ir mechanizmais. Tokio kodo integravimas į programų gamyklą sumažina reikalavimus rankomis rašyti kodą.

Kuriant produktus, atliekamos veiklos:

- Problemos analizė. Nustatoma, ar produktas yra produktų gamyklos apibrėžimo srityje. Gali atsitikti taip, kad tam tikrą dalį arba net viską reikės daryti be programų gamyklos. Galima perfigūruoti produktų gamyklas, kad jos labiau atitiktų ateityje kuriamus produktus.
- Produkto specifikavimas. Aprašo produkto reikalavimus. Aprašymas vyksta aprašant skirtumas tarp programų gamyklos reikalavimų ir produktų reikalavimų.
- Produkto architektūra. Aprašo skirtumus tarp produktų linijos architektūros ir kūrimo proceso, taip pat tarp specializuoto kūrimo proceso ir architektūros.

- Produktų įgyvendinimas. Į tai įeina tokios veiklos kaip komponentų ir testavimo įrankių kūrimas, komponentų kūrimas, komponentų surinkimas (angl. assemble). Daug mechanizmų priklausomai nuo produkto kaip savybių aprašai, vedliai ir modeliai, kurie konfigūruoja komponentus, vizualūs modeliai, kurie sujungia komponentus ir sukuria kitus artefaktus, kodas, konfigūravimo failai, kurie užpildo karkasą. Šis karkasas kuria, modifikuoja, pritaiko komponentus.
- Produkto surinkimas. Į tai įeina kūrimo nuostatus logine fizinių komponentų konfigūravimą paleidimu, sertifikavimą, instaliavimą reikalingu resursų ir t.t.
- Produkto testavimas. Viso produkto testų kūrimas tokių kaip testų situacijos, testų duomenys, skriptai ir įrankiai, matuojantys funkcionalumą.

Lyginant su klasikiniu SPL, šis metodas yra kur kas detalesnis, tačiau idėja ta pati – naudoti programų sistemų šeimos egzemplioriams (programų sistemoms) konstruoti skirtą aplinką (programų sistemų gamyklos dalys Microsoft programų sistemų gamyklose, programų sistemų šeimų analizės, projektavimo, architektūros rezultatai SPL metode), kurti programų sistemas (produktų kūrimas Microsoft programų sistemų gamyklose, programų sistemų inžinerijos veiklos SPL metode).

6. MDSPL metodų analizė

Šiame skyriuje aiškinama, kokias prieš tai aprašyti metodai apima SPL veikas ir kokius MDD sprendimus jie naudoja, kad pasiektų veiklų tikslų. Metodų apimčiai pademonstruoti sudaryta lentelė apačioje. Eilutėse nurodomi autoriai ir straipsnis, o stulpeliuose SPL veiklos. Lentelės elementuose nurodoma, ar straipsnis aprašo būdą pasiekti SPL veiklos rezultatus.

Programų sistemų šeimos inžinerijos veiklos:

	Programų sistemų šeimos analizė	Programų sistemų šeimos projektavimas	Programų sistemų šeimos realizavimas
Sybren Deelstra, Marco Sinnema, Jan Bosch „Product derivation in software product families: a case study“ [DSB03]	Aprašomas etapo tikslo pasiekimo būdas, tačiau metodas yra abstraktus.	Etapo tikslo pasiekimo būdas neaprašomas.	Aprašomas etapo tikslo pasiekimo būdas, tačiau metodas yra abstraktus.
Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Arnor Solberg „An MDA®-based framework for model-driven product derivation“ [HPOS05]	Etapo tikslo pasiekimo būdas aprašomas.	Aprašomas etapo tikslo pasiekimo būdas.	Etapo tikslo pasiekimo būdas aprašomas.
Orlando Avila-García, Antonio Estévez García, E. Victor Sánchez Rebull „Using Software Product Lines to Manage Model Families in Model-Driven Engineering“ [GGR07]	Etapo tikslo pasiekimo būdas aprašomas.	Etapo tikslo pasiekimo būdas aprašomas.	Aprašomas etapo tikslo pasiekimo būdas.

J. Oldevik ir O. Haugen straipsnyje „Higher-Order Transformations for Product Lines" [OH07]	Etapo tikslo pasiekimo būdas aprašomas.	Aprašomas etapo tikslo pasiekimo būdas.	Etapo tikslo pasiekimo būdas aprašomas.
Alexandre Bragança and Ricardo J. Machado straipsnyje „Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines"[BM07]	Aprašomas etapo tikslo pasiekimo būdas.	Aprašomas etapo tikslo pasiekimo būdas.	Aprašomas etapo tikslo pasiekimo būdas.
Krzysztof Czarnecki, Michal Antkiewicz,Chang Hwan Peter Kim, Sean Lau, Krzysztof Pietroszek straipsnyje „Model-Driven Software Product Lines" [CAK+05]	Etapo tikslo pasiekimo būdas aprašomas.	Etapo tikslo pasiekimo būdas neaprašomas.	Etapo tikslo pasiekimo būdas neaprašomas.

Nors kai kurie straipsniuose programų sistemų šeimos inžinerijos veiklos aprašomos, nei viename iš nagrinėtų straipsnių išsamiai neaprašomos programų sistemų inžinerijos veiklos. Lieka neaišku, kaip naudojant kitų autorių metodus gaminti programų sistemas (programų sistemų gamyklų produktus).

Norint sužinoti pateiktų metodų MDD technikų panaudojimo apimtį SPL metodo veiklų rezultatams gauti, aprašoma, ar metodai naudoja MDD technikas SPL veiklų rezultatams gauti.

Programų sistemų šeimos inžinerijos veiklos:

	Programų sistemų šeimos analizė	Programų sistemų šeimos projektavimas	Programų sistemų šeimos realizavimas

Sybren Deelstra, Marco Sinnema, Jan Bosch „Product derivation in software product families: a case study“ [DSB03]	MDD technikos nenaudojamos.	MDD technikos nenaudojamos.	Metodas yra pakankamai senas ir generavimo strategija parenkama kaip vienas iš būdu, naudoja MDD transformacijos.
Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Arnor Solberg „An MDA®-based framework for model-driven product derivation“ [HPOS05]	Naudojamos meta modeliavimo ir modeliavimo MDD technikos.	Naudojamos meta modeliavimo ir modeliavimo MDD technikos.	Naudojami MDA abstrakcijos lygiai MDD M2M ir M2C transformacijos.
Orlando Avila-García, Antonio Estévez García, E. Victor Sánchez Rebull „Using Software Product Lines to Manage Model Families in Model-Driven Engineering“ [GGR07]	Naudojamos meta modeliavimo ir modeliavimo MDD technikos. Iš meta modelio sugeneruojamas redaktorius.	MDD technikos nenaudojamos.	Naudojamos meta modeliavimo ir M2M, M2C transformacijų MDD technikos.
J. Oldevik ir O. Haugen straipsnyje „Higher-Order Transformations for Product Lines” [OH07]	MDD technikos nenaudojamos.	MDD technikos nenaudojamos.	Naudojamos meta modeliavimo ir M2M, M2C transformacijų MDD technikos.
Alexandre Bragança and Ricardo J. Machado straipsnyje „Automating Mappings between	Naudojamos meta modeliavimo ir modeliavimo MDD	MDD technikos nenaudojamos.	MDD technikos nenaudojamos.

Use Case Diagrams and Feature Models for Software Product Lines"[BM07]	technikos.		
Krzysztof Czarnecki, Michal Antkiewicz,Chang Hwan Peter Kim, Sean Lau, Krzysztof Pietroszek straipsnyje „Model-Driven Software Product Lines" [CAK+05]	Naudojamos meta modeliavimo ir modeliavimo MDD technikos. Iš meta modelio sugeneruojamas redaktorius.	MDD technikos nenaudojamos.	MDD technikos nenaudojamos.

Programų sistemų šeimos inžinerijos veiklose tik Alexandre Bragança and Ricardo J. Machado straipsnyje „Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines"[BM07] yra minimos meta modeliavimo ir modeliavimo MDD technikos. Iš meta modelio sugeneruojamas redaktorius.

Iš pateiktų lentelių matoma, kad išnagrinėti metodai nenaudojami visose SPL veiklose ir MDD technikų naudojimas yra pakankamai siauras (technikos naudojamos atskiruose etapuose, o ne visame metode).

7. Galimi modeliais gristų technikų panaudojimai

Kaip parodyta prieš tai esančiame skyriuje, esami MDSPL metodai nenaudojami visose SPL veiklose. Norint panaudoti MDD technikas SPL metode, šiame skyriuje nagrinėjami SPL veiklų rezultatai ir MDD technikų galimybės tuose etapuose.

Programų sistemų šeimos inžinerija:

- Programų sistemų šeimos analizė. Veiklos darbo rezultatas – aprašyta programų sistemų šeima, dažnai vienas darbo produktų yra kintamybių-bendrybių modelis. Kintamybių modelis yra formalus modelis ir jis gali būti aprašytas meta modeliu. Kintamybių modelis gali būti modeliuojamas

modeliavimo įrankiu, kuris būtų sugeneruojamas iš meta modelio.

- Programų sistemų šeimos projektavimas. Šios veiklos produktas yra programų sistemų šeimos architektūra. Kai kuriuose architektūriniuose šablonuose architektūra ar jos dalys yra pakankamai formalus produktas, kad galima būtų aprašyti modeliu turinčiu meta modelį, (pvz., 4+1 architektūros šablone [KRU95] viename iš vaizdų yra naudojamos UML klasių diagramos, jos turi savo meta modelį).

- Programų sistemų šeimos realizavimas. Šios veiklos produktai: programų sistemų šeimos komponentai, generatoriai (skirti automatiniam komponentų jungimui), programų sistemų šeimai specifinės DSL kalbos. Programų sistemų šeimos narių komponentų kodas, įkomponuotas į M2C transformacijų šablonus, gali generuoti programų sistemų šeimos produktus – programų sistemas.

Programų sistemos inžinerija:

- Reikalavimų analizė. Šios veiklos produktas yra reikalavimų specifikacija. Reikalavimų specifikacijoje gali būti „use case“ diagramos. Jos turi meta modelį, yra naudojamos modeliavimui, gali būti panaudotos transformacijoms.

- Programų sistemos konfigūravimas. SPL įrankis – konfigūratorius. Šios veiklos produktas yra sistemos konfigūracija. Kintamybių modelis yra neformali sistemos konfigūracijos meta modelio dalis, nes sistemos konfigūracijos modelyje nurodoma, kokios kintamybės reikalingos produktui iš kintamybių bendrybių modelio. Kadangi kintamybių modelis aprašytas meta modeliu, todėl įmanoma jo M2M transformacija į sistemos konfigūracijos meta modelį. Iš sistemos konfigūracijos meta modelio galima sugeneruoti sistemos konfigūracijos modelio modeliavimo įrankį. Papildžius įrankį modelių tikrinimo įrankiais (taip pat generuojamiems iš ribotumų kalbos, pavyzdžiui, oAW Check), galima gauti SPL konfigūratorių atitinkantį įrankį. Toks įrankis leis parinkti programų sistemai reikalingas savybes ir užtikrins, kad jos viena kitai

nepriešTaraus, ir apskritai užtikrins, kad tokią programų sistemą galima pagaminti.

- Iteracija ir testavimas. Šios veiklos produktas yra programų sistema. Programų sistemų generavimą galima atlikti panaudojus M2C transformacijas. Sugeneravus kodą, ji sukompiliavus ir supakavus į programų sistemų paketus.

Šiame skyriuje nurodoma, kad yra galimybių modeliais grįstas technikas naudoti visuose SPL etapuose. Detaliau apie tai aprašyta sekančiuose skyriuose.

8. MDD technikomis praplėstas SPL procesas

Skyriuje „Galimi modeliais gristų technikų panaudojimai“ trumpai aprašytos galimybės naudoti modeliais grįstas technikas programų sistemų gamyklose. Šiame skyriuje detaliau aprašoma, kaip panaudoti modeliais grįstas technikas, norint pasiekti SPL etapo tikslus.

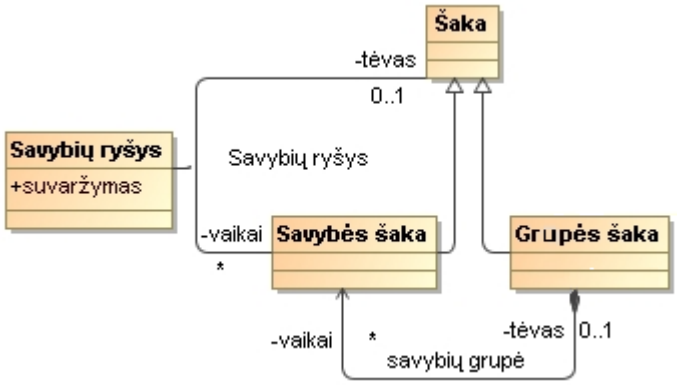
11 Pav. Bendra pasiūlymo schema

Bendra pasiūlymo schema pavaizduota 11 pav. Tolesni skyriai aprašo kiekvieną iš SPL etapų

ir paaiškina schemoje pavaizduotą pasiūlymą.

8.1. Programų sistemų šeimos analizė

Programų sistemų šeimos analizės rezultatas – aprašyta programų sistemų šeima. Šiai veiklai viena iš dalių yra kintamybių-bendrybių modelis. Modelį galima modeliuoti panaudojant modeliavimo įrankį. Dabar egzistuoja keliolika sprendimų, kurie iš meta modelio sugeneruoja modeliavimo įrankį (Pvz. Eclipse su GMF ir EMF papildiniais). Kintamybėms-bendrybėms aprašyti autorius siūlo naudoti modelius, aprašytus meta modeliais. Galima naudoti straipsnyje „Model-Driven Software Product Lines“ [CAK+05] aprašytą savybių kintamybių-bendrybių taksonominę formą (angl. taxonomic form).

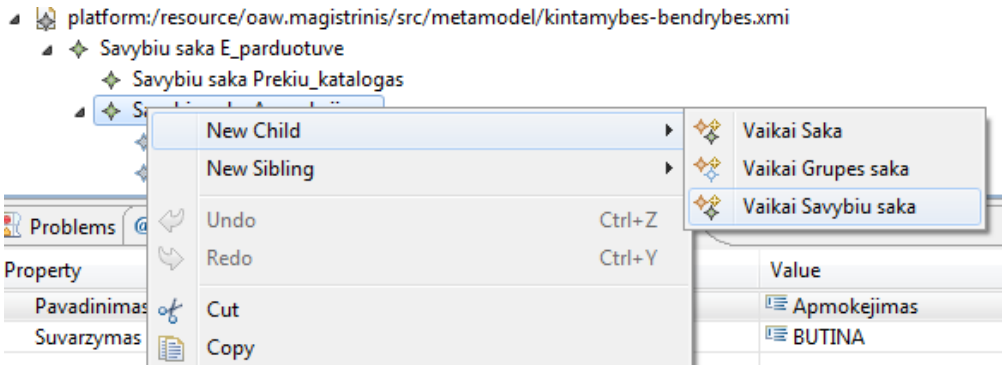


5 pav. Savybių modelio meta modelis [CHU05]

Kitamybių-bendrybių meta modelis aprašo, kad modelis yra medžio pavidalo (šakos turi vaikus šakas). Šakos vaizduoja savybes. Savybės turi ryšio suvaržymus. Suvaržymas gali būti būtinumo arba alternatyvos. Būtinumo suvaržymu surišta šaka nurodo bendrybę, alternatyvos – kintamybę. Kintamybės tai pat yra visos grupės šakos (būtina vieną iš kelių alternatyvų, t.y. sistemos konfigūracijoje būtina parinkti vieną šaką iš grupės).

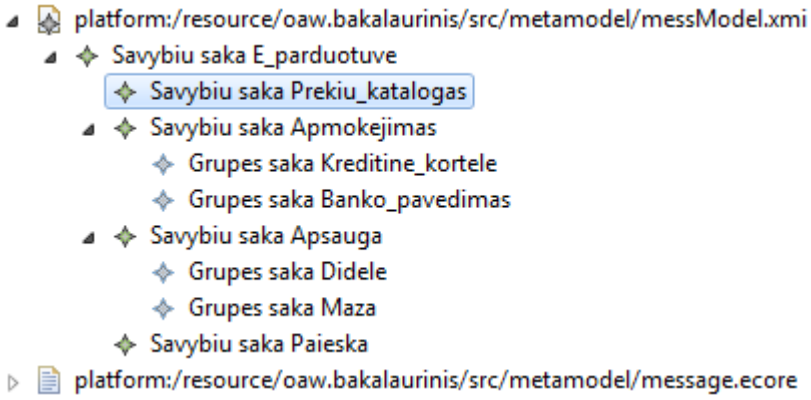
Kintamybes-bendrybes modeliuoti galima pasinaudojus modeliavimo įrankiu sugeneruotu iš meta modelio (pvz. EMF[EF11a] „Sample Reflective Ecore Model Editor“). Kuriant modelį iš meta modelio naudojant šį įrankį automatiškai sukuriamos visos grafinės priemonės, įvedimo laukai,

reikalingi modeliuoti kintamybių-bendrybių modelį. Norint sukurti kintamybių-bendrybių modelį „Sample Reflective Ecore Model Editor“ įrankiu, tereikia sukurti modelį iš kintamybių-bendrybių meta modelio. Norint sukurti bendrybę ant esančios savybės (jei nėra modelyje savybių, reikia sukurti šakninę savybę) turime spragtelti dešinį pelės klavišą ir išsirinkti „Savybės šaka“, o į suvaržymas įvedimo lauką įrašant „BUTINA“. Kuriant kintamybę, reikia savybės šakos suvaržymuose įvedimo lauke įrašyti „ALTERNATYVA“ arba sukurti grupės šaką, jei alternatyva yra vienas iš kelių pasirinkimų.



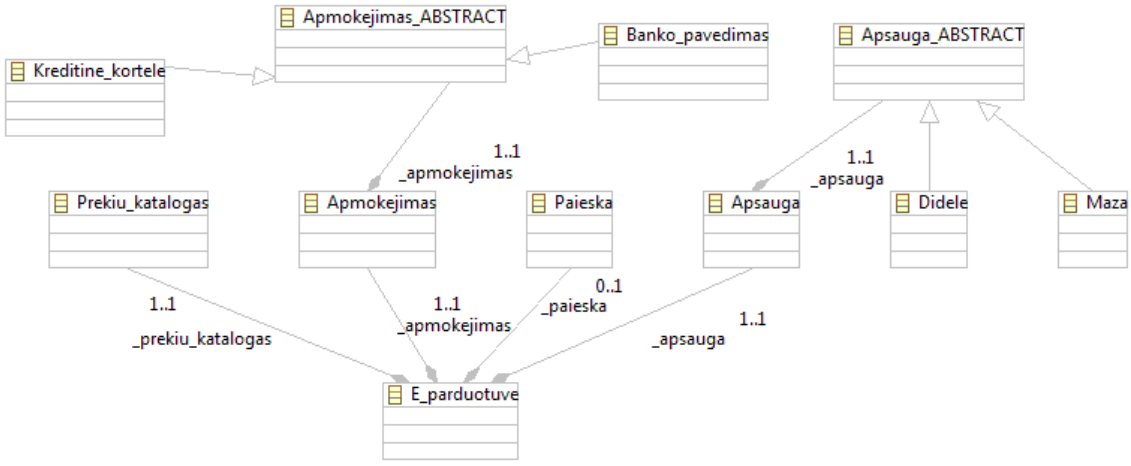
12 pav. modeliavimas „Sample Reflective Ecore Model Editor“ įrankiu

Sumodeliavus kintamybių-bendrybių modelį, iš jo galima sugeneruoti sistemos konfigūracijos meta modelį (jis bus naudojamas kituose SPL etapuose), tam panaudojus M2M transformaciją. Ta transformacija kiekvienai savybių šakai sukurtų atskirą klasę ir ja, su tėvine klase, sujungtų agregaciją. Grupės šakai sukurtų tėvo abstrakčią klasę ir praplėstų ją. Su tėvine klase tokia klasė būtų sujungta agregacija per abstraktų tėvinės klasės atributą. Suvaržymai aprašomi agregacijose (pvz., tėvinė klasė agreguoja 1 su 1, ar 1 su 0 jei agreguoja su kintamybe) Transformaciją iliustruosiu pavyzdžiu. Tarkime sumodeliuojama 13 pav. pavaizduotas kintamybių-bendrybių modelis. Savybė „Paieška“ turi alternatyvos suvaržymą (kintamybė), „Kreditine_kortele“, „Banko_pavedimas“, „Maza“, „Didele“ tai pat yra alternatyvos (kintamybės). Visos kitos savybių šakos yra būtinos (bendrybės).



13 pav. Kintamybių-bendrybių modelis

Transformacijos rezultatas yra pavaizduotas 14 pav. Kiekvienai savybių šakai sukuriama klasė: „E_parduotuve“, „Prekiu_katalogas“, „Apmokejimas“, „Paieska“, „Apsauga“. Visos savybių šakos su vaikinėmis savybių šakomis sujungiamos agregacijomis. Kadangi „Paieska“ yra kintamybė, ji su tėvine šaka sujungiama 0 su 1 agregacija, visos kitos savybių šakos yra bendrybės ir su tėvinėmis šakomis sujungiamos 1 si 1 agregacija. Grupės šakoms sukuriama tėvo abstrakti klasė („Kreditine_kortele“, „Banko_pavedimas“ grupės šakos turi „Apmokejimas“ tėvinę klasę, jai sukuriama „Apmokejimas_ABSTRACT“ klasė). Šią abstrakčią klasę praplečia grupės savybes (kintamybių-bendrybių modelyje) atitinkančios klasės („Kreditine_kortele“, “Banko_pavedimas“ praplečia „Apmokejimas_ABSTRACT“). Abstrakti klasė su tėvine sujungiama agregacija 1 su 1 („Apmokėjimas_ABSTRACT“ su „Apmokejimas“ sujungiama agregacija 1 su 1 „_apmokejimas“).



14 pav. Sistemos konfigūracijos meta modelis

Sistemos konfigūracijos modelis testuojamas, aprašant papildomus ribojimus ribojimų kalba. Naudojant check kalbą ir openArchitectureWare įrankius pvz.:

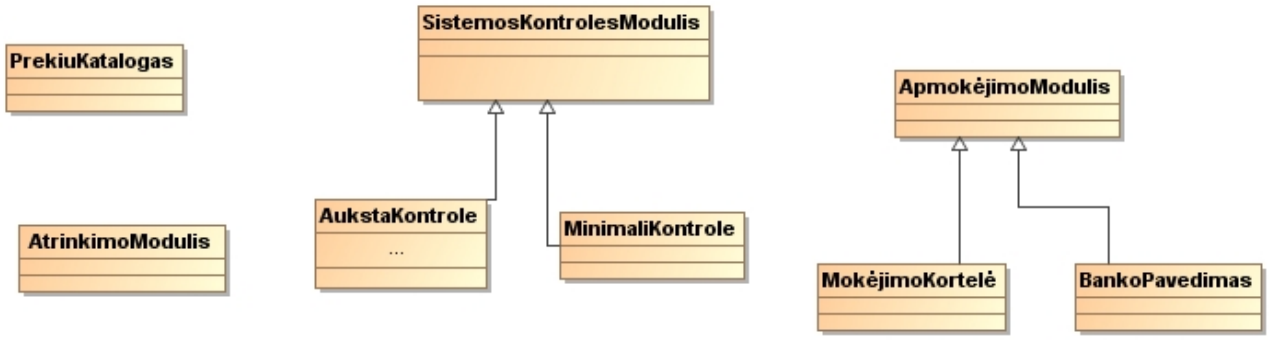
```
import E_parduotuvė;

extension org::openarchitectureware::util::stdlib::naming;

context E_parduotuve ERROR "Paieška gali realizuota būti tik naudojant didelę apsauga":
    ( Apsauga.metaType.toString() == "E_parduotuve::Maza") && ( Paieska!=null);
```

Sistemos konfigūracijos modelio ribojimas tikrina, ar sistemos konfigūracijos modelis turi mažą apsaugą su paieška. Jei sąlyga teisinga – praneša apie klaida. Tokiu būdu taisyklėmis galima aprašyti papildomus ribojimus. Tiesa, reikia šiek tiek programuoti, jei reikalinga ribojimus aprašyti ribojimų kalba, jei to negalima padaryti kintamybių-bendrybių modeliu. MDD technikos neapima visą kintamybių-bendrybių modeliavimo metodą, programų sistemų šeimos apibrėžimą (kaip, pavyzdžiui, jį apima FODA metodas [Cza98]), tačiau pateikia modeliavimo įrankį bei priemonę kintamybėms ir bendrybėms apibrėžti.

8.2. Programų sistemų šeimos projektavimas

Programų sistemų šeimos projektavimo rezultatas – programų sistemų šeimos architektūra. Aprašant architektūrą, autorius siūlo naudoti meta modeliais aprašytus modelius. Modeliais grįstos technikos neišsprendžia visų projektavimų uždavinių, tačiau autoriaus pasiūlytas būdas susieja kintamybes ir bendrybes iš kintamybių-bendrybių modelio su architektūroje aprašomais elementais(klasėmis). Tokiu būdu aprašomas elementų ir savybių (iš kintamybių-bendrybių modelio) ir elementų (iš architektūros) trasuojamumas, be to, šis susiejimas naudingas. Iš jo generuojamos pradinės (nepilnos) transformacijos, naudojamos programų sistemų šeimos realizacijos etape. Nors generuojamos transformacijos yra nepilnos, tačiau jos sutaupo laiko, nes nereikia jų kurti nuo pradžių. 4+1 architektūros šablonu [KRU95] viename iš vaizdų (loginiam vaizde) yra naudojamos UML klasių diagramos. Tokias klasių diagramas papildžius anotacijomis ar kitais elementais (pvz., būdo elementais), galima vykdyti M2C transformacijas ir transformuoti UML klasių diagramas į kitas M2C transformacijas (naudojamas programų sistemų šeimos

realizavimo etape), skirtas kodo generavimui. Kad būtų aiškiau, autorius M2C transformacijas, naudojamas programų sistemų šeimos realizavimo etape, pavadina TR transformacija, o transformacijas, naudojamas projektavimo etape – TP. TP generuoja TR. TR iš sistemos konfigūracijos modelio generuoja programų sistemų kodą. Panašus principas aprašytas J. Oldevik ir O. Haugen straipsnyje „Higher-Order Transformations for Product Lines" [OH07], aprašytoms aukštesnės eilės transformacijoms realizuoti. Autorius iliustruoja transformacijas pavyzdžiu. Tarkime, suprojektuojama klasių diagrama (15 pav.). Kintamybėms ir bendrybėms naudojame ankstesniame skyrelyje aprašytą ir pavaizduota kintamybių-bendrybių modelį (13 pav.). Klasė „AtrinkimoModulis", pažymėta „Paieska", pavadintų būdo elementu (elementas yra vidinis ir klasių diagramoje nesimato). „AukštaKontrole" pažymėta „Didele" (apsaugos), „ŽemaKontrole" pažymėta „Maza" (apsaugos) pavadintų būdo elementu.



15. Pav. UML klasių diagrama.

TP transformacija kiekvienai kintamybei sukuria po taisyklę TR transformacijoje. Tarkime TR transformacijoms aprašyti naudojama Xpand [Ope10] transformacijų kalba (ja galima aprašyti M2C transformacijas). TP sukuria taisykles TR transformacijoje „Paieska", "Apsauga", "Apmokėjimas" savybėms. Kadangi „Apsauga" turi „Maža" arba „didelė" alternatyvas (šios dvi yra grupės šakos), sukuriama žemiau Xpand kalba aprašytos taisyklės:

```
«IF (E_parduotuve._Apsauga._Didele!=null &&
E_parduotuve._Apsauga.metaType.toString() == "E_parduotuve::Didele")»«ENDIF»
«IF (E_parduotuve._Apsauga._Maza!=null &&
E_parduotuve._Apsauga.metaType.toString() == "E_parduotuve::Maža")»«ENDIF»
```

Ši taisyklė TR transformacijoje tikrina, ar kintamybę aprašantis objektas egzistuoja sistemos konfigūracijos modelyje ir koks jis (ar „E_parduotuve::Didele“, ar „E_parduotuve::Didele“). Jei sąlyga teisinga, transformacijos kodas, esantis tarp «IF» «ENDIF», bus įvykdytas arba bus išvestas tekstas. Pavyzdžiui, «IF (E_parduotuve._Apsauga._Didele!=null && E_parduotuve._Apsauga.metaType.toString() == "E_parduotuve::Didele")»System.console.writeln("Vykdomas kodas");«ENDIF». Esant didelei apsaugai sistemos konfigūracijos modelyje, išves: „System.console.writeln ("Vykdomas kodas")“. „Paieska“ kintamybei TP transformacija sukuria žemiau parašytą taisyklę:

```
«IF (E_parduotuve._Paieska!=null)»«ENDIF»
```

Ši taisyklė TR transformacijoje tikrina, ar kintamybę aprašantis objektas egzistuoja sistemos konfigūracijos modelyje ir koks jis (ar „E_parduotuve::Didele“, ar „E_parduotuve::Didele“). Jei sąlyga teisinga, transformacijos kodas, esantis tarp «IF» ir«ENDIF», bus įvykdytas arba bus išvestas tekstas. Kadangi klasės UML diagramos modelyje yra susietos kintamybėmis. Taisyklės TR praplečiamos (naudojantis TP transformacija) sugeneruotu pradiniu klasių kodu (jei klasės turi atributus – sugeneruojami ir atributai). Pavyzdžiui, kadangi „AukštaKontrole“ susieta su „Didele“ (apsaugos) kintamybe ir, tarkime, „AukštaKontrole“ turi „pav“ „string“ tipo atributą. TP sugeneruos žemiau parašytą TR taisyklę:

```
«IF (E_parduotuve._Apsauga._Didele!=null &&
E_parduotuve._Apsauga.metaType.toString() == "E_parduotuve::Didele")»
            «FILE "AukstaKontrole.java" »
            class AukstaKontrole{   String pav;       }
            «ENDFILE»«ENDIF»
Šiame etape
```

Nors architektūros kūrimas nėra pilnai apimamas MDD technikų, tačiau MDD nauda yra - galima gauti pradines, pusiau pilnas (neišbaigtas) M2C transformacijas.

8.3. Programų sistemų šeimos realizavimas

Programų sistemų šeimos realizavimo etapo rezultatas – M2C transformacijos. Šių transformacijų siekis yra sistemos konfigūracijos modelį transformuoti į programų sistemų kodą. Pradinis šių M2C transformacijų išeities kodas gaunamas iš programų sistemų šeimos projektavimo etapo. Tačiau jį reikia papildyti programų sistemų komponentų kodu ir komponentų integravimo kodu, tai daroma papildžius M2C transformacijas (aprašytus transformavimo kalba).

Transformacijos papildomos, papildant taisykles. Iš programų sistemų projektavimo etapo gaunamos visos taisyklės, tikrinančios sistemos konfigūracijos sutiktas bendrybes ar kintamybes. Gali tekti pridėti tokių taisyklių grupavimo variantus, pavyzdžiui, tikrinant, ar yra paieška ir ar apsauga didelė, ir išvedant „System.console.writeln ("Didele apsauga su paieska");" tekstą, jei taisyklės sąlyga yra tenkinama (Xpand kalbos fragmentas[Ope10])):

```
«IF (E_parduotuve._Apsauga._Didele!=null &&
E_parduotuve._Apsauga.metaType.toString() == "E_parduotuve::Didele" &&
E_parduotuve._Paieska!=null))»
   System.console.writeln("Didele apsauga su paieska");
«ENDIF»
```

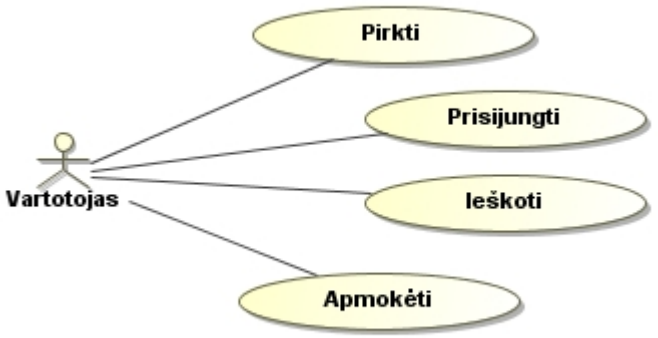
Šiame etape reikia programuoti, realizuoti klases ir komponentus, jų išeities kodo įrašius į taisyklės vidų («IF»«ENDIF»). Tokio kodo fragmentai išvedami, priklausomai nuo to, ar suveikia taisyklė (ar sistemos konfigūracijos modelyje yra aprašyta savybė tikrinama taisyklės). Nors didžiąją dalį programų sistemų elementų reikia programuoti, tačiau, lyginant su paprasto SPL (nepraplėsto MDD technikomis), nereikia programuoti sistemos konfigūracijos modelio notacijos sintaksinio analizatoriaus (angl. parser). Nereikia aprašyti programavimo kalbos kodu sistemos konfigūracijos modelio transformavimų taisyklių (pasiūlytu atveju jos sugeneruojamos programų sistemų šeimos projektavimo etape transformacijų kalba). Nereikia kurti kodo generatoriaus, visą programų sistemos kodą sugeneruoja M2C įrankiai. Autorius pavyzdžiuose naudoja Xpand kalbą, tačiau galima naudoti ir bet kokią kitą M2C transformacijų kalbą.

8.4. Reikalavimų analizė

Reikalavimų analizės etapo rezultatas – reikalavimų specifikacija. Autorius siūlo reikalavimuose naudoti meta modeliais aprašytus modelius ir juos sujungti su kintamybių-bendrybių modeliu. Dalį reikalavimų galima aprašyti naudojant panaudojimo atvejų (angl. use case) UML diagramas. Kita dalis gali būti aprašyta natūralia kalbą (pvz., vartotojo naudojimo scenarijais (ang. user story). Autorius pripažįsta, kad modeliais aprašyti visų reikalavimų gali nepavykti, tačiau jis siūlo dalį reikalavimų susieti su kintamybėmis kintamybių-bendrybių modelyje. Tokiose UML panaudojimo atvejų diagramose galima pažymėti kintamybes, naudojant anotacijas ar kitais elementais (pvz., būdo elementais). Kadangi UML panaudojimo atvejų modeliai turi meta modelius, tokius modelius galima transformuoti į pradinius, tačiau ne visada pilnus sistemų konfigūracijos modelius. Tai padaroma pasinaudojus M2M transformacijomis. Transformacija eina per kintamybių

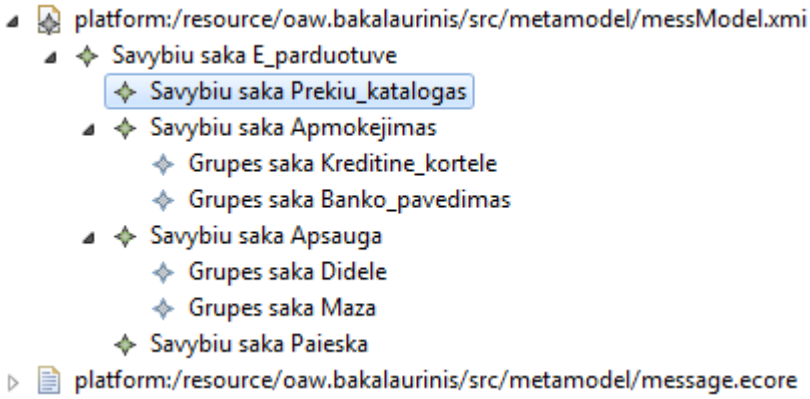
bendrybių modelio šakas ir ieško UML modelyje pažymėtų dalių ir kuria sistemos konfigūraciniame modelyje esančius objektus. Jei panaudojimo atvejų UML diagramos yra labai detalios ir jose yra pažymėtos visos reikalingos kintamybės, tai sugeneruojamas netgi pilnas sistemos konfigūracijos modelis.

Transformaciją autorius iliustruoja pavyzdžiu. Kadangi būdo elementai yra vidiniai, jie dažnai diagramoje nevaizduojami (tiesiog saugomi modelyje). Tarkime, UML panaudojimo atvejų diagrama (16 pav.). „Ieškoti“ panaudojimo atvejo elementas turi „Paieska“ būdo elementą, o „Apmokėti“ panaudojimo atvejo elementas turi (apsaugos) „Didele“ ir „Banko_Pavedimas“ (apmokėjimo būdo) elementus.



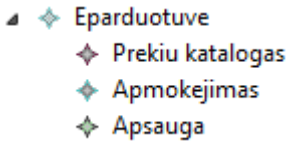
16 pav. UML panaudojimo atvejų (angl. use case) diagrama.

Kintamybių modelis (17 pav.; paveikslėlis kartojamas skaitytojo patogumui) turi „E_parduotuvės“ šaką. Ji savyje turi „Prekiu_katalogas“, “Apmokejimas“, “Apsauga“, “Paieska“ šakas. „Apmokėjimas“ turi „Kreditine_kortele“, “Banko_pavedimas“ šakas. „Apsauga“ turi „Didelė“, “Maza“ šakas. „Kreditine_kortele“, “Banko_pavedimas“, „Didelė“, “Maza“ yra kintamybės, visos kitos – bendrybės.



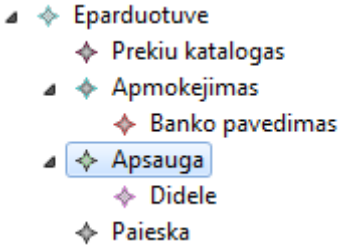
17. pav. Kintamybių-bendrybių modelis

Taigi transformacija pirmu žingsniu eina per šakas ir sistemos konfigūracijos modelyje sukuria bendrybėmis pažymėtus objektus. Po pirmo žingsnio bus sukurti šie objektai: E_parduotuve,_apmokėjimas,_apsauga. Modelis atrodytų taip (18 pav.):



18 pav. Sistemos konfigūracijos modelis

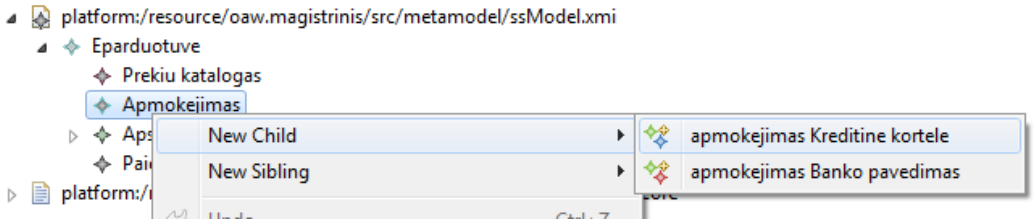
Antru žingsniu transformacija ieško kintamybių vardais pažymėtų elementų UML panaudojimo atveju diagramoje. Šiuo atveju: „Paieška“, „Didelė“, „Banko_Pavedimas“. Jas radus, sistemos konfigūracijos modelis praplečiamas kintamybių objektais. Po antro žingsnio, sistemos konfigūracijos modelis atrodytų taip (19 pav):



19 pav. Kintamybių-bendrybių modelis

8.5. Produkto konfigūravimas

Produkto konfigūravimo rezultatas – sistemos konfigūracijos modelis. Reikalavimų analizės etapo metu gautą pirminį sistemos konfigūracijos modelį galima pataisyti ar papildyti naudojantis sugeneruotu modeliavimo įrankiu. Autorius siūlo jį gauti, naudojant programų sistemų šeimos analizės etapo metu gautą sistemos konfigūracijos meta modelį. Sistemos konfigūracijos modeliui kurti galima pasinaudoti modeliavimo įrankiu, sugeneruotu iš meta modelio (pvz. EMF[EF11a] „Sample Reflective Ecore Model Editor“). Kuriant/modifikuojant modelį iš meta modelio, naudojant šį įrankį, automatiškai sukuriamos visos grafinės priemonės bei įvedimo laukai, reikalingi modeliuoti sistemos konfigūracijos modelį. Norint modifikuoti sistemos konfigūracijos modelį „Sample Reflective Ecore Model Editor“ įrankiu, tereikia atsidaryti modelį (sukurtą reikalavimų analizės etape) ir galima pradėti jį modifikuoti. Norint pridėti bendrybę, reikia ant esančios tėvinės savybės spragtelti dešinį pelės klavišą ir išsirinkti vaikinę savybę (20 pav.). Norint ištrinti savybę, tereikia pažymėti ją ir paspausti trynimo („Delete“) klavišą.



20 pav. Kintamybių-bendrybių modelis

Sistemos konfigūracijos modelis gali neatitikti kitų ribojimų. Tai galima patikrinti naudojant MDD įrankius, paleidžiant programų sistemų šeimos analizės metu aprašytus modelių tikrinimo(ribojimų) kodus (pvz., „Check“ programavimo kalba [Ope10]).

Šio etapo MDD technikų nauda - modelių ribojimų tikrinimas ir modeliavimo įrankiai. Naudojant juos, programuotojui nebereikia rašyti modeliavimo ir ribojimų tikrinimų programinės įrangos nuo pat pradžių.

8.6. Integracija ir testavimas

Integracijos ir testavimo rezultatas – programų sistema. Šis rezultatas pasiekiamas įvykdžius programų sistemų šeimos realizavimo etape gautas M2C transformacijas bei įvykdžius ir sukompiliavus sugeneruotus programų sistemų išeities tekstus. M2C transformacijos tai pat geba generuoti tektus. Juos sugeneravus, galima iš dalies pratestuoti programų sistemą. Išsamus programų sistemų testavimas gali būti atliekamas standartiniais programų sistemų inžinerijos testavimo metodais.

8.7. Naudojami modeliai ir transformacijos

Šiuo poskyriu autorius trumpai aprašo pasiūlyme naudojamus modelius ir transformacijas. Aprašytas pasiūlymas naudoja šiuos modelius:

Modelis	**Aprašymas**
Kintamybių-bendrybių modelis	Aprašo programų sistemų šeimos bendrybes ir kintamybes.
Sistemos konfigūracijos modelis	Aprašo konkretaus produkto įgyvendinamas bendrybes ir kintamybes
UML klasių diagramos modelis	Aprašo programų sistemų šeimos statinę struktūrą. Nurodo, kokios klasės susietos su kokiomis programų sistemų šeimos kintamybėmis ar bendrybėmis
UML panaudojimo atvejų diagramos modelis	Aprašo dalį programų sistemų šeimos reikalavimų. Nurodo, kurie panaudojimo atvejų elementai yra susieti su programų sistemų šeimos kintamybėmis ar bendrybėmis

Aprašytas pasiūlymas naudoja šias transformacijas:

Įvestis	**Išvestis**	**Aprašymas**
Kintamybių-bendrybių modelis	Sistemos konfigūracijos meta modelis	Naudojamas programų sistemų šeimos analizės veikloje. Transformacija sugeneruoja pilną (nereikalaujantį papildymų) sistemos konfigūracijos modelį. Pati transformacija yra ta pati visoms programų sistemų šeimoms.
UML klasių diagramos modelis, Kintamybių-bendrybių modelis	M2C transformacijos išeities kodas	Naudojamas programų sistemų šeimos projektavimo veikloje. Transformacija sugeneruoja nepilną (reikalaujantį papildymų) M2C transformacijų kodą. Jį reikia programuoti praplečiant reikalingomis savybėmis realizuoti skirtu programų sistemų komponentų kodu. Pati eilutėje aprašyta transformacija yra ta pati visoms programų sistemų šeimoms.
Sistemos konfigūracijos modelis	Programų sistemos kodas	Naudojamas integracijos ir testavimo veikloje. Sugeneruoja pilną (nereikalaujantį papildymų) programų sistemos (programų sistemų gamyklos produkto) kodą. Ši transformacija yra skirtinga

		kiekvienai programų sistemų šeimai.

Aprašytas pasiūlymas naudoja ir gamina šiuos artefaktus SPL veiklose:

SPL veikla	**Naudojami artefaktai**	**Gaminami artefaktai**	**Vykdomos veiklos**
Programų sistemų šeimos analizė	Kintamybių-bendrybių meta modelis	Kintamybių-bendrybių modelis, sistemos konfigūracijos meta modelis, papildomi ribojimai (aprašyti ribojimų kalba)	Kintamybių-bendrybių modelio sukūrimas (neautomatizuotas, atliekamas žmogaus), sistemos konfigūracijos meta modelio generavimas (automatizuotas), papildomų ribojimų rašymas (neautomatizuotas)
Programų sistemų šeimos projektavimas	UML klasių diagramos, kintamybių-bendrybių modelis	Nepilnos (reikalaujančios papildymo) M2C transformacijos (naudojamos programų sistemų realizavimo veikloje)	UML klasių diagramos modelio kūrimas (neautomatizuotas), M2C transformacijų generavimas (automatizuotas)
Programų sistemų šeimos realizavimas	M2C transformacijos (nepilnos ir programų sistemų šeimos	M2C transformacijos (pilnos)	M2C transformacijų pildymas (neautomatizuotas)

	projektavimo)		
Reikalavimų analizė	UML panaudojimo atvejų diagramos, kintamybių-bendrybių modelis	Sistemos konfigūracijos meta modelis (gali būti nepilnas, reikalaujantis papildymo ar pataisymų)	UML panaudojimo atvejų diagramos kūrimas (neautomatizuotas), sistemos konfigūracijos modelio generavimas (automatizuotas)
Produkto konfigūravimas	Sistemos konfigūracijos meta modelis, sistemos konfigūracijos modelis (iš reikalavimų analizės, gali būti nepilnas) papildomi ribojimai	Sistemos konfigūracijos modelis (pilnas)	Sistemos konfigūracijos modelio pildymas (neautomatizuotas), ribojimų tikrinimas (automatizuotas)
Integracija ir testavimas	M2C transformacijos (pilnos), Sistemos konfigūracijos modelis (pilnas)	Programų sistema	Programų sistemos kodo generavimas (automatizuotas), programų sistemos testavimas (neautomatizuotas)

Kaip matoma iš viršuje esančios lentelės, pasiūlytas būdas sugeneruoja dalį artefaktų kiekvienai programų sistemų gamyklai.

8.8. Pasiūlymo originalumas

Šiame skyriuje aprašytas būdas nėra visiškai originalus, jame panaudotos kitų autorių idėjos. Toliau detaliau aprašoma, kokias idėjas ir kur autorius naudoja:

Programų sistemų šeimos analizės etape naudojamas Krzysztof Czarnecki, Michał Antkiewicz,Chang Hwan Peter Kim, Sean Lau, Krzysztof Pietroszek straipsnyje „ Model-Driven Software Product Lines“ [CAK+05] aprašytas kintamybių ir bendrybių aprašymo būdas ir meta modelis.

Programų sistemų šeimos projektavimo etape naudojama J. Oldevik ir O. Haugen straipsnio „Higher-Order Transformations for Product Lines”[OH07] idėja apie transformacijas, kurių rezultatas yra kitos transformacijos.

Reikalavimų inžinerijos etape naudojamas Alexandre Bragança, Ricardo J. Machado straipsnyje „Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines”[BM07] pasiūlytas būdas, kuriuo aprašomos kintamybės reikalavimų analizės etape, pasinaudojant UML use case diagramomis.

8.9. Modeliais grįstų technikų nauda kuriant konfigūratorius ir generatorius

Kadangi MDD technikų (transformacijų) realizacijos yra viešai prieinamos ir jau pakankamai brandžios, galima palengvinti generatorių rašymą – vietoj to, kad visą generatorių rašyti nuo nulio, galima pasiremti MDD srityje sukurtomis generatyviomis technikomis. Kaip matoma iš ankstesniu skyrių galima sugeneruoti didelę dali transformavimo taisyklių (iš sumodeliuotu kintamybių). Kaip ir standartiniu SPL atveju komponentus programuoti reikia, reikia suprogramuoti komponentus taisyklių viduje (tarp«`IF`»ir«`ENDIF`» jei naudojama openArchitectureWare Xpand transformavimo kalbą).

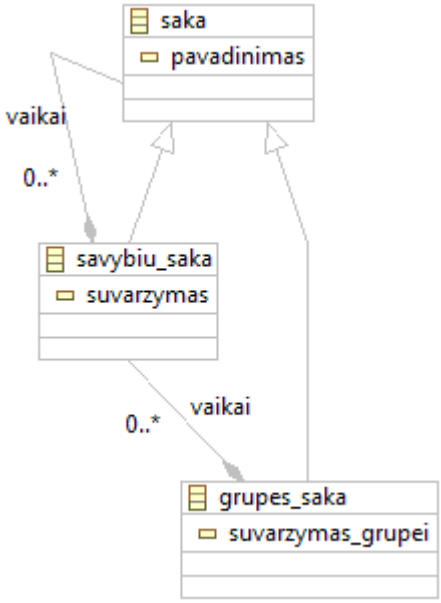
Modelių ribojimų kalbos tai pat yra pakankamai brandžios. Tokia kalba (pvz. openArchitectureWare Check) galima aprašyti ribojimus vartotojo parinktomis kintamybėmis (pvz. renkantis automobilio savybes, išsirinkus elektrini variklį, reikia būtinai išsirinkti ,kad automobyje bus ir elektros akumuliatorius). Konfigūratoriaus modeliavimo įrankį galima sugeneruoti iš sistemos konfigūracijos meta modelio (kuris sugeneruotas iš kintamybių – bendrybių modelio). Taigi programuoti tereikia modelių ribojimus.

8. Eksperimentas

Šis skyrius aprašo eksperimentą, kurio metu „MDD technikomis praplėstas SPL procesas“ skyriuje aprašytą būdą autorius realizuoja praktiškai, pasinaudodamas realiais įrankiais ir technologijomis. Eksperimentu tikrinama, ar pasiūlytą būdą galima realizuoti ir ar yra pakankamai tam reikalingų įrankių.

8.1. Programų sistemų šeimos analizė

Kaip ir minėta pasiūlymo skyriuje, programų sistemų šeimos analizės SPL veikloje kintamybėms aprašyti galima naudoti kintamybių meta modelį. Tokį meta modelį autorius sukuria naudojantis Eclipse EMF įrankiais. Jo principinė schema sugeneruota Ecore Diagram Editor (EMF dalis) pavaizduota 21 pav. Šio meta modelio išeities kodas yra pateiktas priede. Šis meta modelis yra savybių kintamybių bendrybių taksonominės formos (aprašyta [CAK+05]) realizacija. Toks būdas nėra pats geriausias aprašyti kintambes ir savybes, nes, pavyzdžiui, negali aprašyti suvaržymo tarp gretimų (broliškų šakų). Tačiau šis modelis yra gerai aprašytas, aprašytas jo meta modelis.



21 Pav. Kintamybių-bendrybių meta modelis

Meta modelis turi savybių ir grupės šakų klases. Savybių šakos savyje turi kitas savybių ar grupės šakas (vaikus). Tokiu būdu galima modeliuoti medžio tipo struktūrą ir aprašyti suvaržymus tiek pavienėms savybėms, tiek jų grupėms. Aprašomi šie suvaržymai (ecore meta modeliu):

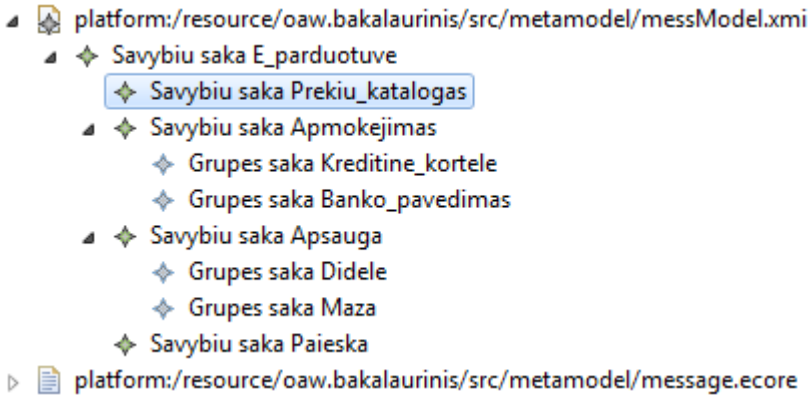
- Būtinumas. Tokia šaka modelyje būtinai turi egzistuoti ir žymi būtiną savybę (pvz., variklio automobilyje).

- Nebūtinumas. Tokia šaka modelyje gali būti, gali ir nebūti ir žymi nebūtiną savybė (pvz., GPS navigacija automobilyje).

- Alternatyva. Šis suvaržymas taikomas grupei, jis žymi būtiną savybę iš kelių variantų (pvz., benzininis ar dyzelinis variklis automobilyje).

Sukūrus meta modelį, autorius sukūrė jį atitinkantį modelį, naudojant „Sample Reflective Ecore Model Editor“ modeliavimo įrankį (EMF dalis). Šis yra modelių redaktorius, jis sugeneruoja modeliavimui reikalingus grafinius elementus iš meta modelio. Šis įrankis pateikia visas kintamybių-bendrybių modeliavimui reikalingas priemones (kurti/redaguoti modelius) naudojantis tik meta modelio informacija. Tokiu būdu nereikia patiems kurti redagavimo įrankio.

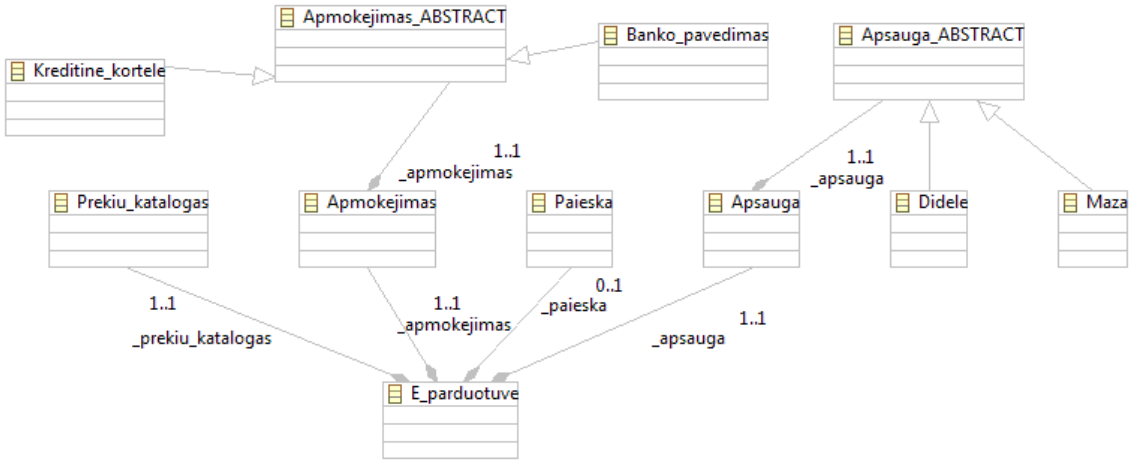
Modelis, pavaizduotas 22 paveiksle, turi E-parduotuvės šakninę savybę, kuri savyje turi prekių katalogo, apmokėjimo, apsaugos ir paieškos vaikines savybes. Paieškos savybė yra nebūtina, kitos – būtinos. Apmokėjimui aprašyti apmokėjimo šaka turi kreditinės kortelės ir banko pavedimo vaikines grupės šakas. Apsaugai aprašyti „apsauga“ turi „didelė“ ir „maža“ vaikines grupės šakas. Toks modelis, keliais žodžiais tariant, aprašo elektroninę parduotuvę su kreditinių kortelių arba banko pavedimo apmokėjimo galimybe, aukštu ar žemu apsaugos lygiu ir galima paieška. Šio modelio išeities kodas yra pateiktas priede.



22. pav. Kintamybių-bendrybių modelis

Kaip aprašyta pasiūlymo įgyvendinimo skyriuje, programų sistemų šeimos analizės metu galima sugeneruoti sistemos konfigūracijos meta modelį. Norint tai pasiekti, autorius parašė M2C transformaciją, naudojantis xpand (openArchitectureWare dalis) transformavimo kalba. Šią kalbą supranta openArchitectureWare įrankiai, kurie meta modeliu aprašytus modelius transformuoja į kodą. Šiuo atveju į sistemos konfigūracijos meta modelį aprašyta ecore (EMF dalis, naudoja openArchitectureWare) formatu. Ši transformacija kiekvienai savybių šakai ir kintamybių-savybių modeliui sukuria po klasę sistemos konfigūracijos meta modelyje. Klasę sujungia su tėvinės šakos sukurta klase, naudojant agregaciją. Grupės šakų tėvui sukuriama abstrakti klasė sistemos konfigūracijos meta modelyje, kiekviena grupės šaka išplečia sukurtą klasę. Abstrakti klasė susiejama su grupės šakos tėvu per agregaciją. Šios transformacijos išeities tekstas pateiktas trečiame priede.

Įvykdžius transformaciją su pateiktu kintamybių modeliu, įrankis gražina sistemos konfigūracijos meta modelį. Šis modelis pavaizduotas 23 paveiksle (paveikslėlis sugeneruotas Ecore Diagram Editor įrankiu). Paveiksle matomi agregacijų ribojimai. Šie ribojimai yra pakankama priemonė užtikrinant, kad konfigūracijos modelyje bus reikalingos bendrybės ir ne daugiau kintamybių nei tai sumodeliuota kintamybių bendrybių modelyje. Šie ribojimai neprieštaraus kintamybių-bendrybių modeliui, nes iš jo jie buvo generuojami.



23 pav. Sistemos konfigūracijos meta modelis

Šis meta modelis bus naudojamas programos konfigūravimo SPL etape. Iš jo generuojamas sistemos konfigūracijos modeliavimo įrankis, modelio ribojimo tikrinimo įrankiai. Tai pat šį meta modelį naudoja programų sistemų šeimos realizavimo etape M2C transformacijos, kuris sistemos konfigūracijos modelius transformuoja į kodą. Sistemos konfigūracijos testuojamas, aprašant papildomus ribojimus, naudojant check kalbą ir openArchitectureWare įrankius. Pavyzdžiui:

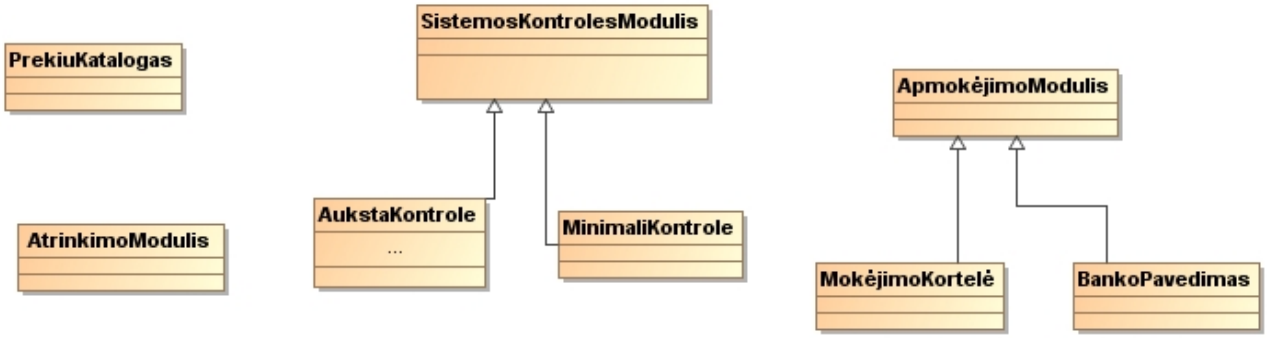
```
import E_parduotuvė;

extension org::openarchitectureware::util::stdlib::naming;

context E_parduotuve ERROR "Paieška gali realizuota būti tik naudojant didelę apsauga":
    ( Apsauga.metaType.toString() == "E_parduotuve::Maza") && ( Paieska!=null);
```

Sistemos konfigūracijos modelio ribojimas parašytas Check kalba. Jis tikrina, ar sistemos konfigūracijos modelis turi mažą apsaugą su paieška. Jei sąlyga teisinga – praneša apie klaida. Taip aprašomi papildomi kintamybių modelio ribojimai.

8.2. Programų sistemų šeimos projektavimas

Kaip ir minėta pasiūlymo įgyvendinimo skyriuje, programų sistemų šeimos projektavimo

etape galima gauti pradines M2C transformacijas, kurios transformuoja sistemos konfigūracijos modelį į programų sistemą. Kai kuriuos programų sistemų šeimos architektūros elementus galima aprašyti UML klasių diagrama. Eksperimento metu autorius detaliai neaprašo architektūros, nes tai neįtakotų eksperimento rezultatų. Autorius aprašo statinę, eksperimento metu kuriamą elektroninės parduotuvės programų sistemų šeimos struktūrą (24 pav.). Klasių diagramos modelyje klasės pažymėtos būdo elementais (ang. Behavior elements). Šie elementai pavadinti kintamybių ar bendrybių vardais.



24. Pav. UML klasių diagrama.

M2C transformacijos, kurios transformuoja sistemos konfigūracijos modelį į programų sistemą, generuotos kitomis M2C transformacijomis. Šią idėją aprašė J. Oldevik ir O. Haugen straipsnyje „Higher-Order Transformations for Product Lines" [OH07]. Šios transformacijos keliauja per kintamybių modelio šakas ir ieško klasių, pažymėtų būdo elementais kitame-klasių diagramos modelyje. Radus tokią klasę, jis sugeneruoja užklausą su sistemos konfigūracijos modelio elementų egzistavimo sąlyga, pavyzdžiui, „ar sistemos konfigūracijos modelyje yra paieška". Tai pat ši transformacija atlieka kitus su MDD ne su SPL susijusius dalykus, pavyzdžiui, generuoja klasių kodą (klasių aprašymus: atributus, tuščius metodus). Šios transformacijos išeities kodas pateiktas ketvirtame priede.

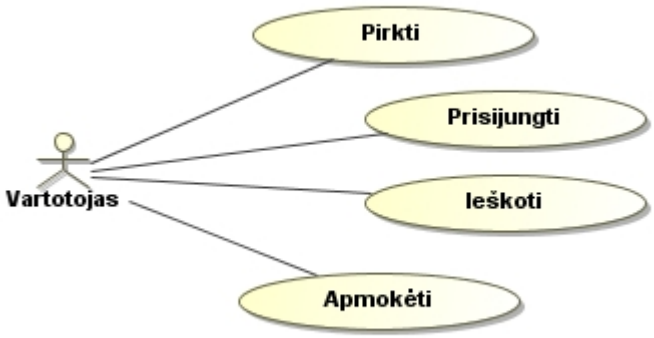
Įvykdžius transformaciją su pateiktu kintamybių ir UML modeliu, įrankis kitą transformaciją, kuri sistemos konfigūracijos modelį transformuoja į programų sistemų kodą. Tokios transformacijos rezultato pavyzdys pateiktas penktame priede.

8.3. Programų sistemų šeimos realizavimas

Programų sistemų šeimos projektavimo etape gautos iš sistemos konfiguracinio modelio į programų sistemos kodą generuojančios transformacijos nėra pilnos, jas dar reikia papildyti programų sistemų kodu. Transformacijų išsamumą galima padidinti naudojant kitas MDD technikas (pvz., aprašyti vartotojo sąsają susigalvota DSL kalba ir iš jos generuoti kodą), tačiau tai nėra šio darbo tikslas. Eksperimente autorius transformacijos taisykles papildo savybe, realizuojančia išeities kodo.

8.4. Reikalavimų analizė

Reikalavimų analizės rezultatas – reikalavimų specifikacija. Tačiau kaip ir programų sistemų šeimos projektavimo etape tam tikrus reikalavimus ar jų aspektus galima aprašyti UML. Autorius tai padaro pasinaudodamas UML panaudojimo atvejų diagramomis (25 pav). Šias diagramas, papildant būdo elementais, kaip ir programų sistemų šeimos projektavimo etape kintamybės sujungiamos su UML panaudojimo atvejų diagramos modeliu.



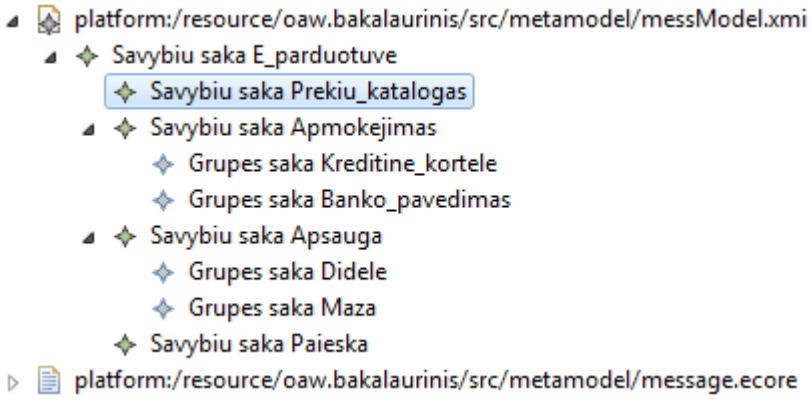
25 pav. UML panaudojimo atvejų (angl. use case) diagrama

Kaip minėta pasiūlymo įgyvendinimo etape, reikalavimų analizės etape galima gauti ir pradinį sistemos konfigūracijos modelį pasinaudojus transformacija. Ši transformacija kaip ir visos kitos M2C transformacijos aprašomos xpand transformacijų kalba. Transformacijos idėja yra aprašyta

Alexandre Bragança and Ricardo J. Machado straipsnyje „Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines"[BM07]. Ji tai pat aprašyta šio darbo literatūros apžvalgoje.

Transformacija įvestimi naudoja UML panaudojimo atvejų diagramas (išsaugotus XMI formatu) ir kintamybių-bendrybių modelį. Ji keliauja per kintamybių-bendrybių modelio šakas ir ieško panaudojimo atvejų elementų, pažymėtų būdo elementais kitame-panaudojimo atvejų UML diagramos modelyje. Radus tokią klasę, ji sistemos konfigūracijos modelyje sukuria elementą.

Transformaciją autorius iliustruoja pavyzdžiu. Kadangi būdo elementai yra vidiniai, jie dažnai diagramoje nevaizduojami (tiesiog saugomi modelyje). Pavyzdžiui, UML panaudojimo atvejų diagrama (25 pav.). „Ieškoti" panaudojimo atvejo elementas turi „Paieška" būdo elementą, o „Apmokėti" panaudojimo atvejo elementas turi (apsaugos) „Didelė" ir „Banko_Pavedimas" (apmokėjimo būdo) elementus. Kintamybių modelis (22 pav.; paveikslėlis kartojamas skaitytojo patogumui) turi „E_parduotuvės" šaką, ji savyje turi „Prekiu_katalogas", "Apmokejimas", "Apsauga", "Paieska" šakas. „Apmokėjimas" turi „Kreditine_kortele", "Banko_pavedimas" šakas. „Apsauga" turi „Didelė", "Maza" šakas. „Kreditine_kortele", "Banko_pavedimas" „„Didelė", "Maza" yra kintamybės, visos kitos – bendrybės.



22. pav. Kintamybių-bendrybių modelis

Taigi transformacija pirmu žingsniu eina per šakas ir sistemos konfigūracijos modelyje sukuria bendrybėmis pažymėtus objektus. Po pirmo žingsnio bus sukurti šie objektai: E_parduotuve,_apmokėjimas,_apsauga. XMI formatu atvaizduotas modelis atrodytų taip:

```
<?xml version="1.0" encoding="ASCII"?>
```

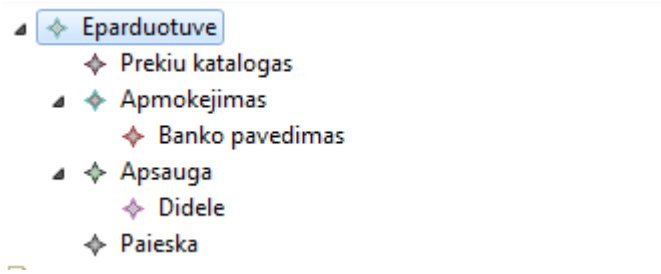
```
<NS:E_parduotuve xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:NS="http://magistrinis/KintamybModel"
xsi:schemaLocation="http://magistrinis/KintamybModel E_parduotuve.ecore">
  <_prekiu_katalogas/>
  <_apmokejimas>
  </_apmokejimas>
  <_apsauga>
  </_apsauga>
</NS:E_parduotuve>
```

Antru žingsniu transformacija ieško kintamybių vardais pažymėtų elementų UML panaudojimo atvejų diagramoje. Šiuo atveju: „Paieška“, „Didelė“, „Banko_Pavedimas“. Jas radus, sistemos konfigūracijos modelis yra praplečiamas kintamybių objektais. Po antro žingsnio, sistemos konfigūracijos modelis XMI formatu atrodytu taip:

```
<?xml version="1.0" encoding="ASCII"?>
<NS:E_parduotuve xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:NS="http://magistrinis/KintamybModel"
xsi:schemaLocation="http://magistrinis/KintamybModel E_parduotuve.ecore">
  <_prekiu_katalogas/>
  <_apmokejimas>
    <_apmokejimas xsi:type="NS:Banko_pavedimas"/>
  </_apmokejimas>
  <_apsauga>
    <_apsauga xsi:type="NS:Didele"/>
  </_apsauga>
  <_paieska/>
</NS:E_parduotuve>
```

8.5. Programų sistemos konfigūravimas

Reikalavimų analizės etapo metu gautas sistemos konfigūracijos modelis, priklausomai nuo panaudojimų atvejų UML diagramos modelio, gali būti nepilnas. Tačiau tam, kad jį papildytų ar modifikuotų, autorius naudoja Sample Reflective Ecore Model Editor (EMF dalis) modeliavimo įrankį ir pasinaudojus sistemos konfigūracijos meta modeliu, sugeneruotu programų sistemų analizės etapo metu. Įrankis puikiai tinka modeliuoti, turi visus modelio savybėms reikalingus įvedimo laukus ir priemones. Tai labai didelė nauda, nes nereikia nuo pat pradžių kurti modelio redaktoriaus.



16 pav. Sistemos konfigūracijos modelis

Autorius sumodeliuotą sistemos konfigūracijos modelį tikrina naudodamas programų sistemų šeimos etape aprašytais ribojimais.

8.6. Integracija ir testavimas

Šiame etape, paleidus programų sistemų šeimos realizavimo etape gautas transformacijas su sistemos konfigūracijos modelio įvestimi, gautas programų sistemų gamyklos produktas – elektroninės parduotuvės programų sistema.

8.7. Pakartotinis panaudojamumas

Vieną kartą atlikus programų sistemų šeimų inžinerijos veiklas, naujas programų sistemas galima generuoti tiesiog parašius reikalavimų specifikaciją ir papildžius sistemų konfigūracijos modelį reikalingomis savybėmis. Todėl šiuo metodu pasiekiamas labai didelis komponentų pakartotinis panaudojamumas.

Kuriant kitą programų sistemų gamyklą ir atliekant programų sistemų šeimų inžinerijos veiklas, tereikia sumodeliuoti kintamybių – bendrybių modelį, sukurti programų sistemų šeimos architektūrą ir papildyti programų sistemų šeimos realizacijos etape konstruojamas M2C

transformacijas. Visos kitos transformacijos gali būti pakartotinai panaudotos jų nekeičiant, nes jos priklauso tik nuo kintamybių bendrybių meta modelio.

Gauti rezultatai ir išvados

Magistrinio darbo metu gauti šie rezultatai:

- Modeliais grįsta SPL metodų apžvalga ir suklasifikavimas pagal SPL etapus
- SPL konfigūratoriaus ir generatoriaus kūrimo pasiūlymas, paremtas MDD technikomis

Magistrinio darbo metu gautos šios išvados:

- Atsiradusios DSL, meta-modelių kūrimo ir apribojimų aprašymo priemonės (pvz., openArchitectureWare „Check“) leidžia palyginti ir lengvai sukurti SPL konfiguratorių.
- Pastaruoju metu išvystytos M2M ir M2T transformacijų vykdymo priemonės leidžia sukurti SPL generatorius. Kai kurios priemonės (pvz. Xtend2) turi dinaminių programavimo kalbų elementų, išplėstinius (angl. extension concept), polimorfizmo tvarkymą (angl. Polymorphic dispatching), tipų apdorojimo mechanizmus (angl. Type inference), aukštesnės eilės funkcijas (higher order functions), sukurtą redaktoriu (eclipse). Tai leidžia lanksčiau kurti SPL generatorių nei su įprastinėmis programavimo kalbomis.

Šaltiniai

[BM07] Alexandre Bragança , Ricardo J. Machado. Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines. The 11th International Conference of Software Product Line Conference. Kyoto, Japonija 2007. [žiurėta 2010.6.20] Prieiga per Internetą: http://www3.dsi.uminho.pt/rmac/privatefiles/papers/2007_SPLC_BragancaMachado-ieee.pdf

[CAK+05] Krzysztof Czarnecki, Michał Antkiewicz,Chang Hwan Peter Kim, Sean Lau, Krzysztof Pietroszek. Model-Driven Software Product Lines. OOPSLA '05 Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM New York, Jungtinės Amerikos Valstijos. 2005 [žiurėta 2010.6.20] Prieiga per Internetą: http://portal.acm.org/citation.cfm?id=1094855.1094896

[CHU05] Krzysztof Czarnecki1, Simon Helsen, Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. Software Process: Improvement and Practice. 2005 [žiurėta 2010.6.20] Prieiga per Internetą:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.4104&rep=rep1&type=pdf

[CN07] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2007, 563 pages

[Cza98] K. Czarnecki. Generative Programming:Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Technical University of Ilmenau. [žiurėta 2010.6.19] Prieiga per Internetą: http://www.prakinf.tu-ilmenau.de/~czarn/diss/diss.pdf

[DSB03] Sybren Deelstra, Marco Sinnema, Jan Bosch. Product derivation in software product families: a case study. The Journal of Systems and Software 74 (2005)

173–194 [žiurėta 2010.6.20] Prieiga per Internetą: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.6246&rep=rep1&type=pdf

[EF10a] Eclipse Foundation. Graphical Modeling Framework. [žiurėta 2010.6.20] Prieiga per Internetą: http://www.eclipse.org/modeling/gmf/

[EF11a] Eclipse Foundation. Graphical Modeling Framework. [žiurėta 2011.5.26] Prieiga per Internetą: http://www.eclipse.org/modeling/emf/

[EF10b] Eclipse Foundation. ATLAS Transformation Language. [žiurėta 2010.6.20] Prieiga per Internetą: http://www.eclipse.org/m2m/atl/

[GGR07] Orlando Avila-García, Antonio Estévez García, E. Victor Sánchez Rebull.Using Software Product Lines to Manage Model Families in Model-Driven Engineering. SAC '07 Proceedings of the 2007 ACM symposium on Applied computing.ACM New York, Jungtinės Amerikos Valstijos 2007 [žiurėta 2010.6.20] Prieiga per Internetą: http://portal.acm.org/citation.cfm?id=1244221

[GS03] Jack Greenfield, Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. Wiley, 2004, 500 pages

[HPOS05] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Arnor Solberg An MDA®-based framework for model-driven product derivation. Software Engineering and Applications. ACTA Press. 709--714 [žiurėta 2010.6.20] Prieiga per Internetą: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.1813&rep=rep1&type=pdf

[MBM09] M. Mezini, D. Beuche, A. Moreira. 1st International Workshop on Model-Driven Product Line Engineering. [žiurėta 2010.1.13] Prieiga per Internetą: http://feasiple.de/public/proceedings-mdple2009.pdf

[JM03] Joaquin Miller, Jishnu Mukerji. MDA Guide Version 1.0.1. Object Management

Group (OMG), Document omg/03-06-01, June 2003 [žiurėta 2010.6.20] Prieiga per Internetą: http://www.enterprise-architecture.info/Images/MDA/MDA%20Guide%20v1-0-1.pdf

[Kar07] M. Karlsch. A model-driven framework for domain specific languages. Hasso-Plattner-Institute of Software Systems Engineering. Magistrinis darbas [žiurėta 2010.6.19] Prieiga per Internetą: http://karlsch.com/frodo.pdf

[OH07] J. Oldevik, O.Haugen. Higher-Order Transformations for Product Lines. SPLC '07 Proceedings of the 11th International Software Product Line Conference IEEE Computer Society Washington, DC, USA ©2007 [žiurėta 2010.6.20] Prieiga per Internetą: http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.11

[Ope10] openArchitectureWare. The leading platform for professional model-driven software development. [žiurėta 2010.6.20] Prieiga per Internetą: http://www.openarchitectureware.org/index.php

[Per88] J. M. Perry. Perspective on Software Reuse. Technical Report. CMU/SEI-88-TR-022 September 1988 [žiurėta 2010.6.19] Prieiga per Internetą: http://www.sei.cmu.edu/reports/88tr022.pdf

[TP08] Rasha Tawhid, Dorina Petriu. Integrating Performance Analysis in the Model Driven Development of Software Product Lines. MoDELS '08 Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems. Springer-Verlag Berlin, Heidelberg 2008 [žiurėta 2010.6.20] Prieiga per Internetą: http://www.sce.carleton.ca/faculty/petriu/papers/TP-MODELS08.pdf

[KRU95] Philippe Kruchten. Architectural Blueprints—The “4+1” View Model of Software Architecture. IEEE Software 12 (6) November 1995, pp. 42-50 [žiurėta 2011.5.09] Prieiga per Internetą: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

Sąvokų apibrėžimai ir santrumpų sąrašas

MDD – (angl. model driven development) modeliais grįstas programų sistemų kūrimas. Programinės įrangos kūrimo metodiką.

SPL – (angl. software product lines) programų sistemų gamykla. Programų sistemų artefaktų rinkinys ir būdas iš jo kurti programų sistemas (programų sistemų gamyklų produktus). Produktai (programų sistemos) orientuoti į tam tikrą dalykinę sritį ar rinkos segmentą ir patenkina to segmento poreikius. Produktai (programų sistemos) kuriamos apibrėžtu būdu, naudojant bendrus pirminius šaltinius (išteklius).

MDD terminai [Kar07]:

- Meta modelis – modelius aprašantis modelis.
- Modeliai – formali sistemos specifikacija, aprašanti funkcijas, elgesį ir/arba struktūrą.
- Transformacija – procesas, kurio metu vienas modelis konvertuojamas į kitą.
- Dalykinei sričiai specifinės kalbos (angl. Domain specific language) (toliau DSL) apibrėžiama kaip modeliavimo arba specifikavimo kalba, specialiai sukuriama tam tikros rūšies problemoms/situacijoms analizuojamoje dalykinėje srityje aprašyti. Pagrindinė DSL paskirtis – išreikšti tam tikrą siaurą problemą ar sprendimą daug lakoniškiau, trumpiau ir aiškiau negu tai įmanoma padaryti naudojant egzistuojančias bendros paskirties modeliavimo kalbas (pvz. UML).

- Ribojimų kalbos. Loginėmis išraiškomis aprašytos taisyklės kurias modelis turi tenkinti Jas realizuoja , Check [Ope10] įrankiai.

MDD technikos:[Kar07]:

- Transformacijos iš modelio į modelį M2M (angl. Model to model). M2M transformacijos yra procesas, kuris modelį, aprašytą meta modeliu, verčia į kitą meta modeliu aprašytą modelį, naudojant modelių transformacijų kalba aprašytus būdus. Jas realizuoja ATL

[EF10b], Xtend [Ope10] įrankiai.

- Transformacijos iš modelio į tekstą M2C (angl. Model to code). M2C transformacijos yra procesas, kuris modelį, aprašytą meta modeliu, verčia į tekstą, naudojant šablonu kalba aprašytus būdus. Jas realizuoja Xpand [Ope10] įrankiai.
- Tekstiniai DSL. Tai programų sistemų šeimai specifinės kalbos, kurių notacija yra tekstinė. Jas realizuoja, pavyzdžiui, xText [Ope10] įrankis.
- Grafiniai DSL. Tai programų sistemų šeimai specifinės kalbos, kurių notacija yra grafinė. Viena iš realizacijų yra GMF [EF10a] įrankis.

Priedai

Šiame skyriuje pateikti ir trumpai aprašomi darbe naudojami priedai.

1.1 Kintamybių-bendrybių meta modelis

Kintamybės ir bendrybės aprašytos XMI formatu. EPackage elementas laiko visus meta modelyje modeliuojamus elementus. eClassifiers šaka nurodo modelio elementą, type atributas nurodo elemento tipą (pvz. klasei - "ecore:EClass") name atributas nurodo elemento vardą. Agregacijos aprašytos eStructuralFeatures elementu, jo kartinalumai aprašyti upperBound ir lowerBound atributais (pvz. 0..1 agregacijos kardinalumas aprašomas upperBound=”-1” lowerBound=“0“ atributais).

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="KintamybModel"
    nsURI="http://magistrinis/KintamybModel" nsPrefix="NS">
  <eClassifiers xsi:type="ecore:EClass" name="saka">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="pavadinimas"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"
        iD="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="grupes_saka"
eSuperTypes="#//saka">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="suvarzymas_grupei"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="vaikai" upperBound="-
1"
        eType="#//savybiu_saka" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="savybiu_saka"
eSuperTypes="#//saka">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="suvarzymas"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="vaikai" upperBound="-
1"
        eType="#//saka" containment="true"/>
  </eClassifiers>
</ecore:EPackage>
```

1.2 Kintamybių-bendrybių modelis

Kintamybių-bendrybių modelis saugojamas XMI formatu. Šis formatas yra atviras ir plačiai taikomas saugant modelius. Kaip matoma iš pavyzdžio šis formatas aiškiai skaitomas, agregacijos parodomos tartom XML šakų vaikai, o agreguotu objektų tipai pažymimi type atributu.

```
<?xml version="1.0" encoding="ASCII"?>
<NS:savybiu_saka xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:NS="http://magistrinis/KintamybModel"
xsi:schemaLocation="http://magistrinis/KintamybModel kintamyb.ecore"
pavadinimas="E_parduotuve">
  <vaikai xsi:type="NS:savybiu_saka" pavadinimas="Prekiu_katalogas"
suvarzymas="BUTINA"/>
  <vaikai xsi:type="NS:savybiu_saka" pavadinimas="Apmokejimas"
suvarzymas="BUTINA">
    <vaikai xsi:type="NS:grupes_saka" pavadinimas="Kreditine_kortele"
suvarzymas_grupei="ARBA"/>
    <vaikai xsi:type="NS:grupes_saka" pavadinimas="Banko_pavedimas"
suvarzymas_grupei="ARBA"/>
  </vaikai>
  <vaikai xsi:type="NS:savybiu_saka" pavadinimas="Apsauga" suvarzymas="BUTINA">
    <vaikai xsi:type="NS:grupes_saka" pavadinimas="Didele"
suvarzymas_grupei="ARBA"/>
    <vaikai xsi:type="NS:grupes_saka" pavadinimas="Maza"
suvarzymas_grupei="ARBA"/>
  </vaikai>
  <vaikai xsi:type="NS:savybiu_saka" pavadinimas="Paieska"
suvarzymas="PASIRINKTINA"/>
</NS:savybiu_saka>
```

1.3 Kintamybių modelio transformacija į sistemos konfigūracijos meta modelį

Ši transformacija kiekvienai savybių šakai ir kintamybių-savybių modelio sukuria po klase sistemos konfigūracijos meta modelyje. Ir sujungia ja su tėvinės šakos sukurta klase naudojant agregaciją. Grupės šakų tėvui sukuriama abstrakti klasė sistemos konfigūracijos meta modelyje, kiekviena grupės šaka išplečia tą sukurtą klase. Abstrakti klasė susiejama su grupės šakos tėvu per agregaciją.

```
«IMPORT KintamybModel»

«DEFINE Root FOR KintamybModel::savybiu_saka»
«FILE this.pavadinimas+".ecore" »<?xml version="1.0" encoding="UTF-8"?>
```

```
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
name="«this.pavadinimas»"
    nsURI="http://magistrinis/KintamybModel" nsPrefix="NS">

  «EXPAND Child FOR this»


</ecore:EPackage>
«ENDFILE»
«ENDDEFINE»

  «DEFINE Child FOR KintamybModel::saka»

  «ENDDEFINE»


  «DEFINE Child FOR KintamybModel::savybiu_saka»
   <eClassifiers xsi:type="ecore:EClass" name="«this.pavadinimas»">

                  «FOREACH this.vaikai AS vaikas»
                        «IF vaikas.metaType.toString() ==
"KintamybModel::savybiu_saka"»
                        «EXPAND Structural FOR vaikas»
                  «ENDIF»
                  «ENDFOREACH»
      «IF this.vaikai.exists(e|e.metaType.toString() ==
"KintamybModel::grupes_saka")»
       <eStructuralFeatures xsi:type="ecore:EReference"
name="_«this.pavadinimas.toFirstLower()»" lowerBound="1" upperBound="1"
                        eType="#//«this.pavadinimas»_ABSTRACT"
containment="true"/>
       </eClassifiers>

       <eClassifiers xsi:type="ecore:EClass" name="«this.pavadinimas»_ABSTRACT">
             </eClassifiers>

      «FOREACH this.vaikai AS vaikas»
            «EXPAND Grupe_child(this.pavadinimas) FOR vaikas»
            «ENDFOREACH»

      «ELSE»
            </eClassifiers>
    «ENDIF»



      «EXPAND Child FOREACH this.vaikai»

  «ENDDEFINE»

  «DEFINE Child FOR KintamybModel::grupes_saka»

  «ENDDEFINE»
```

```
  «DEFINE Structural FOR KintamybModel::savybiu_saka»
      «IF this.suvarzymas == "BUTINA"»
            <eStructuralFeatures xsi:type="ecore:EReference"
name="_«this.pavadinimas.toFirstLower()»" lowerBound="1" upperBound="1"
                        eType="#//«this.pavadinimas»" containment="true"/>
      «ELSE»
                  <eStructuralFeatures xsi:type="ecore:EReference"
name="_«this.pavadinimas.toFirstLower()»" lowerBound="0" upperBound="1"
                        eType="#//«this.pavadinimas»" containment="true"/>
    «ENDIF»
  «ENDDEFINE»

  «DEFINE Structural FOR KintamybModel::grupes_saka»
  «ENDDEFINE»

  «DEFINE Structural FOR KintamybModel::saka»
  «ENDDEFINE»

  «DEFINE Grupe_child(String pavadinimas) FOR KintamybModel::saka»
  «ENDDEFINE»

  «DEFINE Grupe_child(String pavadinimas) FOR KintamybModel::savybiu_saka»
  «ENDDEFINE»

  «DEFINE Grupe_child(String pavadinimas) FOR KintamybModel::grupes_saka»
      <eClassifiers xsi:type="ecore:EClass" name="«this.pavadinimas»"
eSuperTypes="#//«pavadinimas»_ABSTRACT">
                  «FOREACH this.vaikai AS vaikas»
                        «IF vaikas.metaType.toString() ==
"KintamybModel::savybiu_saka"»
                        «EXPAND Structural FOR vaikas»
                  «ENDIF»
                  «ENDFOREACH»
            </eClassifiers>

      «EXPAND Child FOREACH this.vaikai»

       «ENDDEFINE»
```

1.4 UML klasių diagramos ir kintamybių modelio transformavimas į kitą transformaciją

Šiame pavyzdyje pavaizduota UML modelio transformacija į programų sistemų kodą. Ši transformacija aprašyta Xpand kalba. Transformacija eina per kintamybių-bendrybių modelio šakas ir ieško klasių diagramoje būdo elementų tokiais vardais. Rados tokius elementus ji generuoja transformacijos sąlyga. Pvz. IF sakiniu (pvz. „IF (E_parduotuve._Apmokejimas._Kreditine_kortele!=null && E_parduotuve._Apmokejimas.metaType.toString() == "E_parduotuve::Kreditine_kortele" tikrina ar egzistuoja kreditinės korteles kintamybe ir ar jos tipas „Kreditine_kortele“) jei salyga taisinga

rezultate išvedamas tekstas esantis tarp IF ir ENDIF dalių.

```
«IMPORT KintamybModel»

«DEFINE Root(uml::Model model) FOR KintamybModel::savybiu_saka»
«FILE "SistemosKonfiguracijos2Kodas.xpt" »
«"«"»IMPORT «this.pavadinimas»«"»"»
«EXPAND Child(model,this.pavadinimas,this.pavadinimas) FOREACH this.vaikai»
«EXPAND UML_class2code FOR model»
«ENDFILE»
«ENDDEFINE»

«DEFINE Child(uml::Model model,String names,String root) FOR
KintamybModel::saka»«ENDDEFINE»


«DEFINE Child(uml::Model model,String names,String root) FOR
KintamybModel::savybiu_saka»
      «IF this.suvarzymas == "BUTINA"»
                  «EXPAND Child(model,names+"._"+this.pavadinimas,root) FOREACH
this.vaikai»
      «ELSE»
            «"«"»IF («names»._«this.pavadinimas»!=null)«"»"»
                  «EXPAND UML_class4node(this.pavadinimas) FOR model»
                  «EXPAND Child(model,names+"._"+this.pavadinimas,root) FOREACH
this.vaikai»
            «"«"»ENDIF«"»"»
    «ENDIF»
«ENDDEFINE»

«DEFINE Child(uml::Model model,String names,String root) FOR
KintamybModel::grupes_saka»
            «"«"»IF («names»._«this.pavadinimas»!=null &&
«names».metaType.toString() == "«root»::«this.pavadinimas»")«"»"»
                  «EXPAND UML_class4node(this.pavadinimas) FOR model»
                  «EXPAND Child(model,names+"._"+this.pavadinimas,root) FOREACH
this.vaikai»
            «"«"»ENDIF«"»"»
 «ENDDEFINE»

«DEFINE UML_class4node(String pavadinimas) FOR uml::Model»
      «FOREACH this.packagedElement AS element»
        «EXPAND Class(pavadinimas) FOR element»
      «ENDFOREACH»
«ENDDEFINE»

«DEFINE Class(String pavadinimas) FOR uml::PackageableElement»
«ENDDEFINE»

«DEFINE Class(String pavadinimas) FOR uml::Class»
    «IF this.ownedBehavior.exists(e|e.name==pavadinimas) »

            «"«"»FILE "«this.name»+.java" «"»"»
```

```
            class «this.name»
            {
            «FOREACH this.attribute AS att»
                   «att.type.name» «att.name»;
            «ENDFOREACH»

            «FOREACH this.getOperations() AS element»
                  «element.returnResult()» «element.name» ( «element» );
            «ENDFOREACH»
            }


            «"«"»ENDFILE«"»"»

      «ENDIF»
«ENDDEFINE»


«DEFINE UML_class2code FOR uml::Model»
      «FOREACH this.packagedElement AS element»
        «EXPAND Class2code FOR element»
      «ENDFOREACH»
«ENDDEFINE»

«DEFINE Class2code FOR uml::PackageableElement»
«ENDDEFINE»

«DEFINE Class2code FOR uml::Class»

    «IF this.ownedBehavior.size<1 »


            «"«"»FILE "«this.name».java" «"»"»

            class «this.name»
            {
            «FOREACH this.attribute AS att»
                   «att.type.name» «att.name»;
            «ENDFOREACH»

            «FOREACH this.getOperations() AS element»
                  «element.returnResult()» «element.name» ( «element» );
            «ENDFOREACH»
            }

            «"«"»ENDFILE«"»"»

      «ENDIF»


    «ENDDEFINE»
```

1.5 Sistemos konfiguracijos modelio transformacija į programų sistemų kodą

Šiame pavyzdyje pavaizduota konfigūracijos modelio transformacija į programų sistemų kodą. Ši transformacija aprašyta Xpand kalba. Transformacija tikrina ar egzistuoja kintamybės sistemos konfigūracijos modelyje, tai daro IF sakiniu (pvz. „IF (E_parduotuve._Apmokejimas._Kreditine_kortele!=null && E_parduotuve._Apmokejimas.metaType.toString() == "E_parduotuve::Kreditine_kortele" tikrina ar egzistuoja kreditinės korteles kintamybe ir ar jos tipas „Kreditine_kortele“) jei sąlyga teisinga rezultate išvedamas tekstas esantis tarp IF ir ENDIF daliu

```
«IMPORT E_parduotuve»
            «IF (E_parduotuve._Apmokejimas._Kreditine_kortele!=null &&
E_parduotuve._Apmokejimas.metaType.toString() ==
"E_parduotuve::Kreditine_kortele")»
            «ENDIF»
            «IF (E_parduotuve._Apmokejimas._Banko_pavedimas!=null &&
E_parduotuve._Apmokejimas.metaType.toString() ==
"E_parduotuve::Banko_pavedimas")»
            «ENDIF»
            «IF (E_parduotuve._Apsauga._Didele!=null &&
E_parduotuve._Apsauga.metaType.toString() == "E_parduotuve::Didele")»
            «FILE "AukstaKontrole.java" »
            class AukstaKontrole{   String pav;       }
            «ENDFILE»
            «ENDIF»
            «IF (E_parduotuve._Apsauga._Maza!=null &&
E_parduotuve._Apsauga.metaType.toString() == "E_parduotuve::Maza")»
            «ENDIF»
            «IF (E_parduotuve._Paieska!=null)»
            «ENDIF»
            «FILE "AtrinkimoModulis.java" »
            class AtrinkimoModulis
            {}
            «ENDFILE»
            «FILE "SistemosKontrolesModulis.java" »
            class SistemosKontrolesModulis
            {}
            «ENDFILE»
            «FILE "+.java" »
            class
            {}
            «ENDFILE»
            «FILE "MinimaliKontrole.java" »
            class MinimaliKontrole
            {}
            «ENDFILE»
            «FILE "PrekiuKatalogas.java" »
            class PrekiuKatalogas
            {}
```

```
«ENDFILE»
«FILE "Apmok?jimoModulis.java" »
class Apmok?jimoModulis
{}
«ENDFILE»
«FILE "MoejimoKortele.java" »
class MokejimoKortele
{}
«ENDFILE»
«FILE "BankoPavedimas.java" »
class BankoPavedimas
{}
«ENDFILE»
```

1.6 Sistemos konfigūracijos modelis

Sistemos konfigūracijos modelis išsaugotas XMI formatu. Šis formatas yra atviras ir plačiai taikomas saugant modelius. Kaip matoma iš pavyzdžio šis formatas aiškiai skaitomas, agregacijos parodomos tartum XML šakų vaikai, o agreguotu objektų tipai pažymimi type atributu.

```
<?xml version="1.0" encoding="ASCII"?>
<NS:E_parduotuve xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:NS="http://magistrinis/KintamybModel"
xsi:schemaLocation="http://magistrinis/KintamybModel E_parduotuve.ecore">
  <_prekiu_katalogas/>
  <_apmokejimas>
    <_apmokejimas xsi:type="NS:Banko_pavedimas"/>
  </_apmokejimas>
  <_apsauga>
    <_apsauga xsi:type="NS:Didele"/>
  </_apsauga>
  <_paieska/>
</NS:E_parduotuve>
```

1.7 UML panaudojimo atvejų transformavimas į sistemos konfigūracijos modelį

Šiame pavyzdyje pavaizduota UML modelio transformacija į programų sistemos konfigūracijos modelį. Ši transformacija aprašyta Xpand kalba. Transformacija eina per kintamybių-bendrybių modelio šakas, kuria bendrybių objektus ir ieško panaudojimo atvejų diagramoje būdo elementų kintamybių vardais. Rados tokius elementus ji generuoja objektą sistemos konfigūracijos modelyje. Pvz. transformacija sukuria „prekiu_katalogas”,”_apmokejimas”,”_apsauga” bendrybių

objektus sistemos konfigūracijos modelyje, o UML diagramoje radus “paieška” būdo elementa - sistemos konfigūracijos modelyje objekta “_paieška”;

```
«IMPORT KintamybModel»

«DEFINE Root(uml::Model model) FOR KintamybModel::savybiu_saka»
«FILE "SistemosKonfiguracijos2Kodas.xpt" »
«"«"»IMPORT «this.pavadinimas»«"»"»
«EXPAND Child(model,this.pavadinimas,this.pavadinimas) FOREACH this.vaikai»
«EXPAND UML_class2code FOR model»
«ENDFILE»
«ENDDEFINE»

«DEFINE Child(uml::Model model,String names,String root) FOR
KintamybModel::saka»«ENDDEFINE»


«DEFINE Child(uml::Model model,String names,String root) FOR
KintamybModel::savybiu_saka»
      «IF this.suvarzymas == "BUTINA"»
                  «EXPAND Child(model,names+"._"+this.pavadinimas,root) FOREACH
this.vaikai»
      «ELSE»
            «"«"»IF («names»._«this.pavadinimas»!=null)«"»"»
                  «EXPAND UML_class4node(this.pavadinimas) FOR model»
                  «EXPAND Child(model,names+"._"+this.pavadinimas,root) FOREACH
this.vaikai»
            «"«"»ENDIF«"»"»
    «ENDIF»
«ENDDEFINE»

«DEFINE Child(uml::Model model,String names,String root) FOR
KintamybModel::grupes_saka»
            «"«"»IF («names»._«this.pavadinimas»!=null &&
«names».metaType.toString() == "«root»::«this.pavadinimas»")«"»"»
                  «EXPAND UML_class4node(this.pavadinimas) FOR model»
                  «EXPAND Child(model,names+"._"+this.pavadinimas,root) FOREACH
this.vaikai»
            «"«"»ENDIF«"»"»
 «ENDDEFINE»

«DEFINE UML_class4node(String pavadinimas) FOR uml::Model»
      «FOREACH this.packagedElement AS element»
        «EXPAND Class(pavadinimas) FOR element»
      «ENDFOREACH»
«ENDDEFINE»

«DEFINE Class(String pavadinimas) FOR uml::PackageableElement»
«ENDDEFINE»

«DEFINE Class(String pavadinimas) FOR uml::Class»
    «IF this.ownedBehavior.exists(e|e.name==pavadinimas) »

            «"«"»FILE "«this.name»+.java" «"»"»
```

```
            class «this.name»
            {
            «FOREACH this.attribute AS att»
                   «att.type.name» «att.name»;
            «ENDFOREACH»

            «FOREACH this.getOperations() AS element»
                  «element.returnResult()» «element.name» ( «element» );
            «ENDFOREACH»
            }


            «"«"»ENDFILE«"»"»

      «ENDIF»
«ENDDEFINE»


«DEFINE UML_class2code FOR uml::Model»
      «FOREACH this.packagedElement AS element»
        «EXPAND Class2code FOR element»
      «ENDFOREACH»
«ENDDEFINE»

«DEFINE Class2code FOR uml::PackageableElement»
«ENDDEFINE»

«DEFINE Class2code FOR uml::Class»

    «IF this.ownedBehavior.size<1 »


            «"«"»FILE "«this.name».java" «"»"»

            class «this.name»
            {
            «FOREACH this.attribute AS att»
                   «att.type.name» «att.name»;
            «ENDFOREACH»

            «FOREACH this.getOperations() AS element»
                  «element.returnResult()» «element.name» ( «element» );
            «ENDFOREACH»
            }

            «"«"»ENDFILE«"»"»

      «ENDIF»


    «ENDDEFINE»
```