VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Baigiamasis bakalauro darbas

Dalykinės srities modelio transformavimas į UML užduočių diagramas

(Deriving use cases from business process)

Atliko: 4 kurso 1 grupės studentas
Aleksandras Sivkovas (parašas)
Darbo vadovas:
Prof. dr. Saulius Gudas (parašas)
Recenzentas:
prof. dr. Romas Baronas (parašas)

Vilnius
2018

Santrauka

Darbe rašoma apie sistemos funkcinių reikalavimų gavimą užduočių diagramos pavidalu iš dalykinės srities modelio. Parodomas reikalavimų inžinerijos bei įrankio šiam procesui atlikti aktualumas. Įrankis padėtų sumažinti klaidų kiekį, paspartintų procesą, jis leistų dalintis reikalavimais ir sekti jų pasikeitimus. Darbe nagrinėjami BPMN, UML ir SysML standartai bei pasiūlomi jų išplėtimai apibrėžiantys modelius reikalavimų inžinerijos automatizavimui. Taip pat pateikiamas algoritmas užduočių diagramos gavimui iš pasiūlyto BPMN išplėtimo ir aprašoma sukurta programa demonstruojanti jo veikimą.

Raktiniai žodžiai: BPMN, Užduočių diagrama, UML, SySML, DVGM

Summary

The paper describes how to obtain the functional requirements of the system in the form of the use cases from business process. The relevance of requirement engineering and a tool for it is shown. The usage of tool would to reduce the amount of errors, speed up the process, allow sharing of requirements and tracking their changes. The paper analyses the standards of BPMN, UML and SysML, and proposes their extensions that could be used for automation of requirement engineering process. Also the algorythm for deriving use cases from extended BPMN and description of tool to demonstrate algorythm is provided.

Keywords: BPMN, Use cases, UML, SySML, DVCM

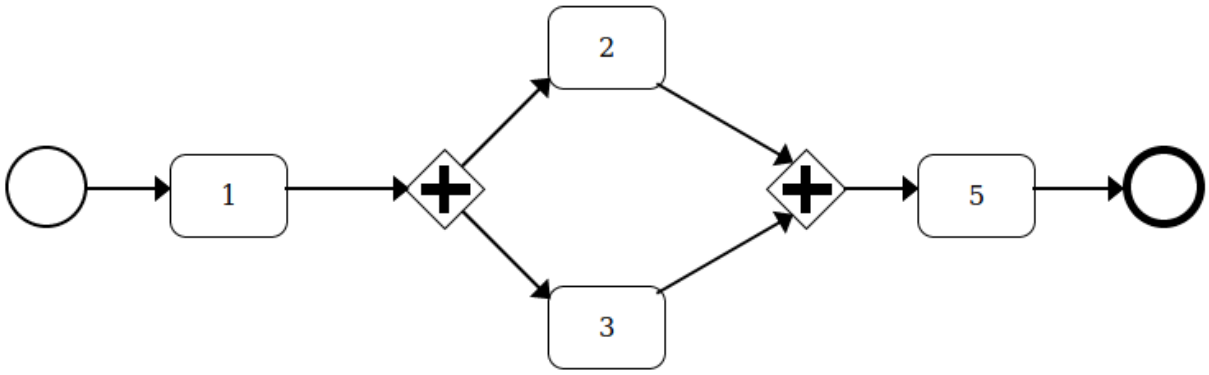
Turinys

Įvadas

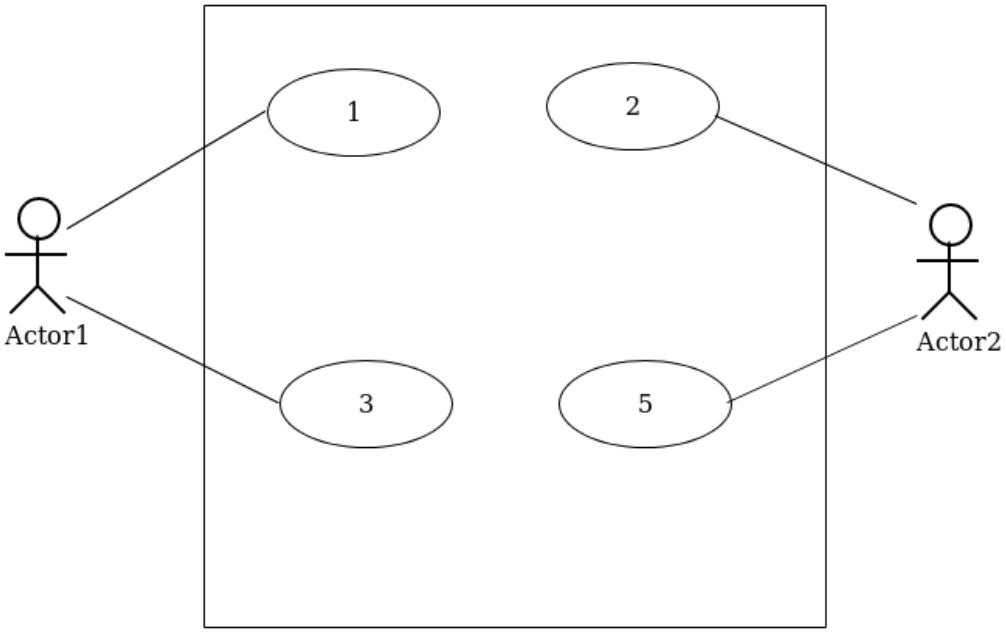
Reikalavimų inžinerija yra svarbi programų kūrimo dalis. Tai yra akivaizdus dalykas naudojant tradicines programų kūrimo metodologijas, bet ir pastarąjį dešimtmetį pripažinimo sulaukusiose judriose metodologijose, svarbu žinoti kokia sistema yra kuriama. Straipsnis [ABS+17] kalba apie reikalavimų inžinerijos ypatumus ir trūkumus judriose metodologijose. [JSB11] parodoma kaip užduočių diagramos galėtų būti naudojamos judriose metodologijose. Taip pat, dėl didesnių galimybių susirasti įgudusių programų sistemų inžinierių, organizacijos vis labiau ima taikyti „globalų programų sistemų kūrimo metodą“. [UHM17] kalba apie reikalavimų inžineriją paskirstytose komandose (distributed teams). Viena iš straipsnio analizės išvadų yra „There Need To Be A Shared Understanding Between Teams About How Requirements Changes Would Be Dealt With“. Autoriai siūlo naudoti įrankį reikalavimų analizei ir sistemiškai tvarkingam reikalavimų specifikavimui.

Reikalavimų inžinerija yra sudėtinga programų kūrimo dalis. Proceso sudėtingumas dažnai tampa klaidų priežastimi. Čia atsiradusios klaidos sunkiai aptinkamos ir sukelia brangiai kainuojančias pasekmes, nes sekančiuose etapuose bus kuriama neteisingai apibrėžta programa. Norint išvengti klaidų galima kai kurias proceso veiklas automatizuoti. Nors yra daug modeliavimo įrankių, juose trūksta automatinio vieno modelio generavimo iš kito.

Projektuojant sistemą gali atsitikti toks atvejis (2 pav.). Čia pavaizduota, kad analitikas aptiko kokias funkcijas reikia įgyvendinti. Analizuodamas 1 pav. pavaizduotą veiklą jis aptiko vartojimo atvejus pažymėtus 1, 2, 3, 5. Taigi nuspręsta kurti sistemą pavaizduotą 2 pav.

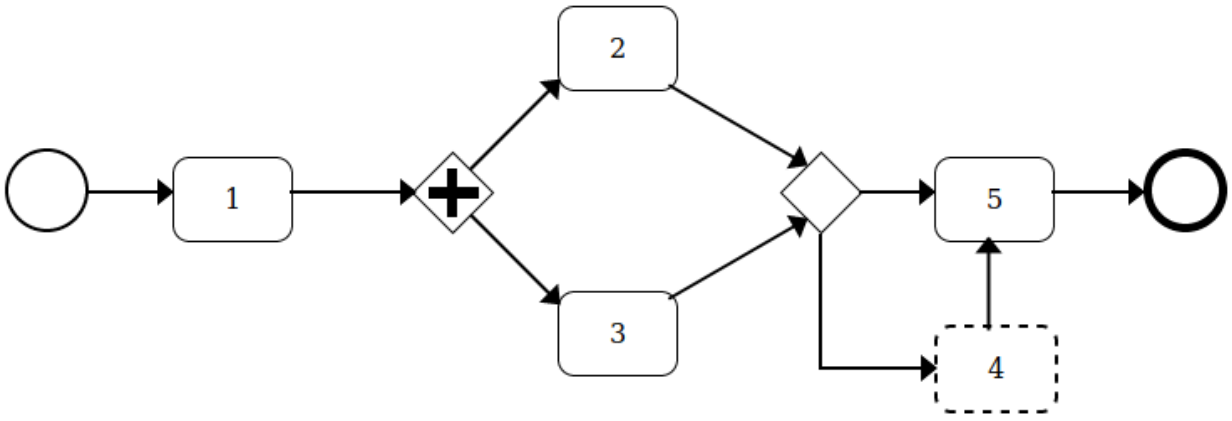


1 pav. Veiklos modelio pavyzdys

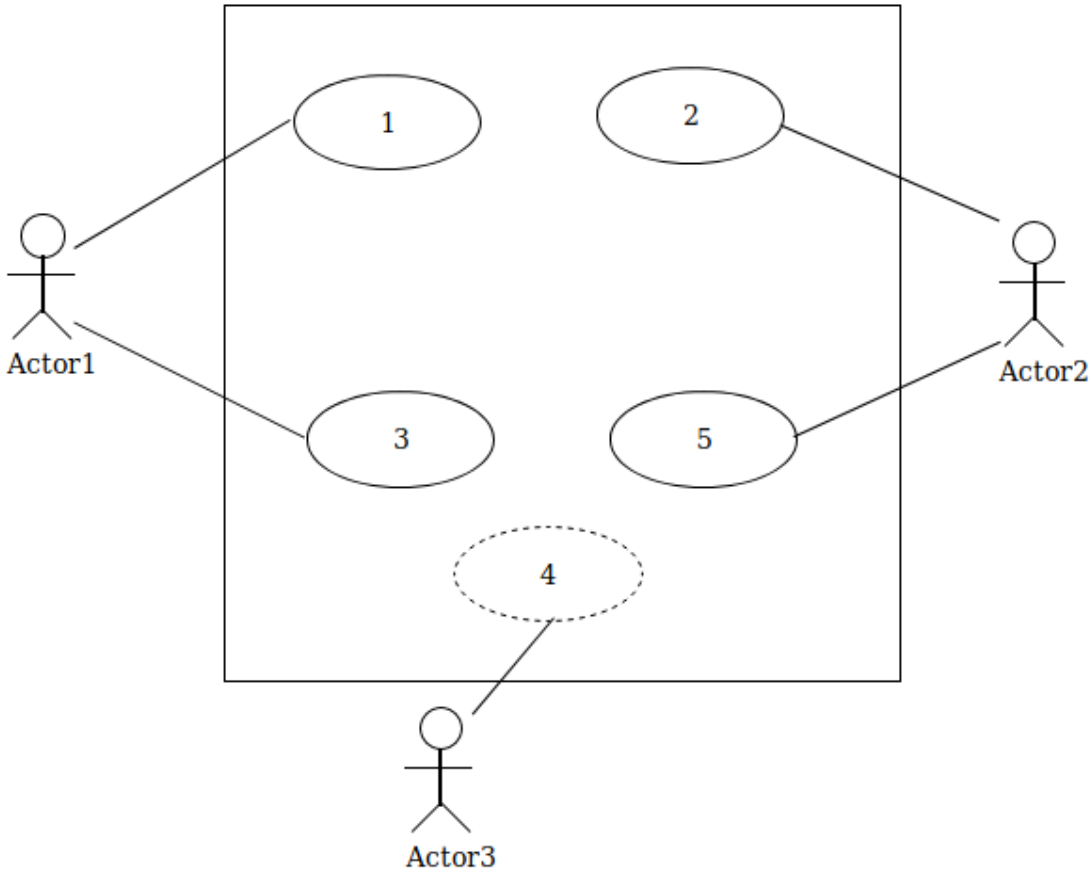


2 pav. Vartojimo atvejų diagramos sukurtos pagal 1 pav. veiklą pavyzdys

Bet vėliau paaiškėjo, kad norint įgyvendinti veiklą pažmėtą 5 kartais reikia duomenų iš 4 veiklos pažymėtos punktyrine linija (3 pav.). Taigi vartojimo atvejų diagramoje trūko 4 vartojimo atvejo (4 pav.). Analitikas to nepastebėjo ir įvyko projektavimo klaida apie kurią niekas nesužinojo.



3 pav. Pataisyto veiklos modelio pavyzdys



4 pav. Vartojimo atvejų diagramos sukurtos pagal 3 pav. pavyzdys

Sumažinti klaidų kiekį galima užrašant turimus duomenis ir automatiškai generuojant modelius. Generavimo įrankis galėtų patikrinti ar įvesties duomenys atitinka keliamus reikalavimus. Taip bus parodyti netikslumai ir analitikas galės pakoreguoti modelį.

Šio darbo tikslas – sukurti algoritmą **BPMN** modelio transformacijai į užduočių diagramas ir įgyvendinti programos prototipą. Užduočių diagramos yra svarbi reikalavimų inžinerijos dalis, kadangi ji apibrėžia naudotojo reikalavimus. Įmonės dažniausiai žino kaip ir kokias veiklas jos vykdo. Verslo procesą galima apibrėžti **BPMN** diagramomis. Bet ne viską, kas yra **BPMN** modelyje, galima perkelti į užduočių diagramą, todėl darbe bus apibrėžtas modifikuotas **BPMN** modelis, kuriame bus vaizduojama tik algoritmui aktuali informacija. Taip pat gali tekti pridėti papildomų atributų, kurie padės pasiekti tikslesnius rezultatus. Čia bus tiriamas **BPMN** modelio transformacijos į užduočių diagramas algoritmas.

Siekiami rezultatai yra:

1. Algoritmas galintis transformuoti **BPMN** modelį į užduočių diagramą(angl. Use case diagram).

2. Programa demonstruojanti algoritmo veikimą.

1. Veiklos srities modeliavimo metodai ir kalbos

1.1. BPMN diagrama

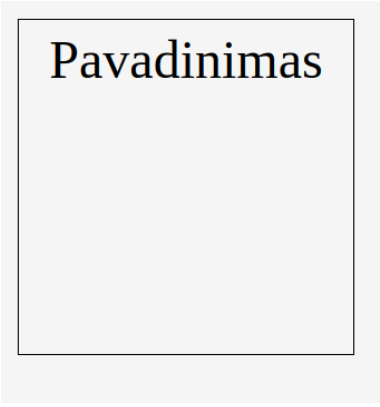
Norint standartizuoti verslo modelių atvaizdavimą 2004 metais organizacija BPMI išleido **BPMN** 1.0 specifikaciją. Ji leido tiek vaizduoti esamus, tiek apsikeisti kuriamų procesų reikalavimais. **BPMN** greitai išpopuliarėjo tarp vadybininkų, verslo analitikų ir programuotojų, nes pasiūlė pažįstamą verslo procesų atvaizdavimą ir turėjo matematinį pagrindą. Vėliau BPMI susijungė su **OMG** ir 2013 metais buvo išleista **BPMN** 2.0 versija, kurioje **BPMN** įgavo geresnį skaitomumą, lankstumą ir išplečiamumą.

1.1.1. BPMN apimtis

BPMN specifikacija standartizuoja verslo procesų modelį, jų atvaizdavimo būdą ir duomenų apsikeitimo formatą perduoti tiek modelį, tiek jo atvaizdavimą. Sutelkti dėmesiui į skirtingas proceso dalis pateikiami procesų ir choreografijos submodeliai. Bendradarbiavimo submodelis, į save įtraukiantis kitus, gali būti naudojamas pateikti iš karto viskam. Specifikacijos apimtis yra verslo procesai taigi dalykai kaip strategija, duomenų struktūros, resursai, taisyklės ir įstatymai neįeina į ją.

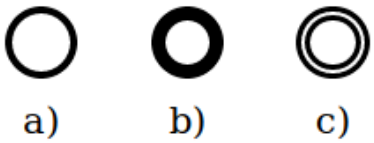
1.1.2. BPMN komponentai

BPMN specifikacija leidžia atvaizduoti gana nemažai verslo proceso atributų [Obj11]. Specifikacijoje jie yra suskirstyti į pagrindinius ir išvestinius. Toliau pateikiami pagrindiniai **BPMN** komponentai.



5 pav. Juostos žymuo

Juosta (pool) žymi diagramos dalyvį. Šio komponento paskirtis yra parodyti už kokias veiklas ir koks vykdytojas yra atsakingas. Juosta gali būti skaidoma smulkiau norint konkrečiau nurodyti vykdytojų grupes ir jų pareigas, tuomet ji gali būti vadinama linija (lane). Juosta žymima apibraukiant tam tikrą sritį (5 pav.). Viduje yra vieta komponentams už kurių atlikimą atsakingas dalyvis.

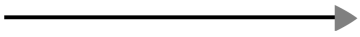


6 pav. Įvykio žymuo

Įvykis (event) žymi, kad įvyko kažkas, kas įtakojo proceso būseną. Šis komponentas dažniausiai turi priežastį (trigger) dėl ko jis įvyko ir pasekmes (result). Specifikacijoje įvykiai skirstomi į tris tipus: pradžios (6 pav. a), pabaigos (6 pav. b) ir tarpinius (6 pav. c). Žymėjimas yra tuščias apskritimas (6 pav.), viduje paliekant vietos tipo konkretizavimui.

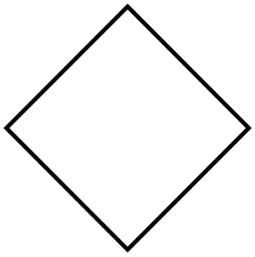
7 pav. Veiklos žymuo

Veikla (Activity) yra darbas atliekamas organizacijos procesuose. Ji gali būti atominė ir turėti tik pavadinimą arba skaidoma labiau, tokiu būdu tapdama subprocesu. Visais atvejais žymėjimas yra stačiakampis su užapvalintais kampais (7 pav.).



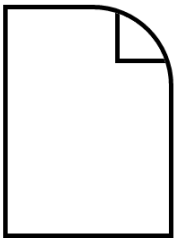
8 pav. Sekos srauto žymuo

Sekos srautas (Sequence Flow) žymi veiklų seką. Jeigu nenurodyta lygiagretumo veiklos modelyje vykdomos iš eilės. Šis komponentas parodo kokia tvarka tai vyks. Jį galima apibūdinti kaip grafo su kryptimis briauna, kryptis parodo kuri veikla turi būti įvykdyta vėliau. Sekos srautas žymimas solidžia linija ir užpildyto trikampio formos rodykle parodančia kryptį (8 pav.).



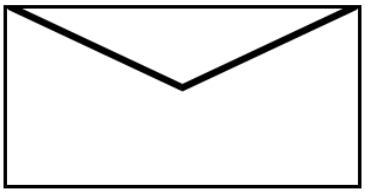
9 pav. Sprendimo žymuo

Sprendimas (gateway) gali būti įterptas sekos srautuose tarp veiklų. Jis žymi srautų išsišakojimą arba susijungimą. Šis komponentas parodo, kad priklausomai nuo konkretaus proceso būsenos bus vykdomos veiklos į kurias eina sekos srautai atitinkantys sprendimo sąlygas. Žymėjimas yra kvadratas pasuktas 45 laipsniu kampu (9 pav.). Viduje yra vieta sprendimo konkretizavimui.



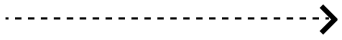
10 pav. Duomenų objekto žymuo

Duomenų objektas (data Object) pateikia informaciją apie tai kokių duomenų reikalauja veiklos vykdytojas norėdamas ją atlikti ir kokie duomenys pagaminami po jos vykdymo. Komponentas gali žymėti tiek neskaidomus tiek sudėtinius duomenis. Jis vaizduojamas (10 pav.) kaip stačiakampis su nukirptu dešiniuoju viršutiniu kampu ir trikampiu prie jo (arba kaip stačiakampis lapas su užlenktu dešiniuoju viršutiniu kampu).



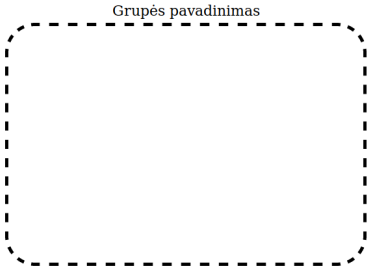
11 pav. Pranešimo žymuo

Pranešimas (message) žymi bendravimą tarp dalyviu. Šis komponentas skirtas apibrėžti perduodamai informacijai tarp jų. Vaizduojamas (11 pav.) stačiakampiu su trikampiu viduje (vokas).



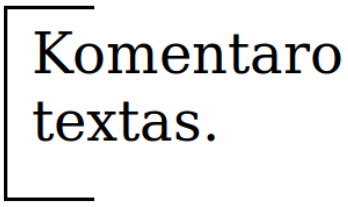
12 pav. Pranešimų srauto žymuo

Pranešimų srautas (Message flow) žymi duomenų perdavimą. Tai yra kryptinė grafo briauna, kuri jungia duomenų objektus ir žinutes su juos sukuriančiomis arba naudojančiomis veiklomis. Duomenų srautas į objektą ar žinutę rodo, kad tai yra veiklos išeiga, priešingu atveju tai yra veiklos įeiga, duomenys reikalingi jai įvykdyti. Vaizduojama (11 pav.) punktyrine linija su rodykle rodančia kryptį.



13 pav. Grupės žymuo

Grupė (group) skirta nurodyti modelio komponentų kategorijas. Ji neturi įtakos sekų ar duomenų srautams, o tik pasako kas priklauso tai pačiai kategorijai. Vaizdavimas diagramoje yra komponentų apibraukimas ir grupės pavadinimo nurodymas (13 pav.).



14 pav. Komentaro žymuo

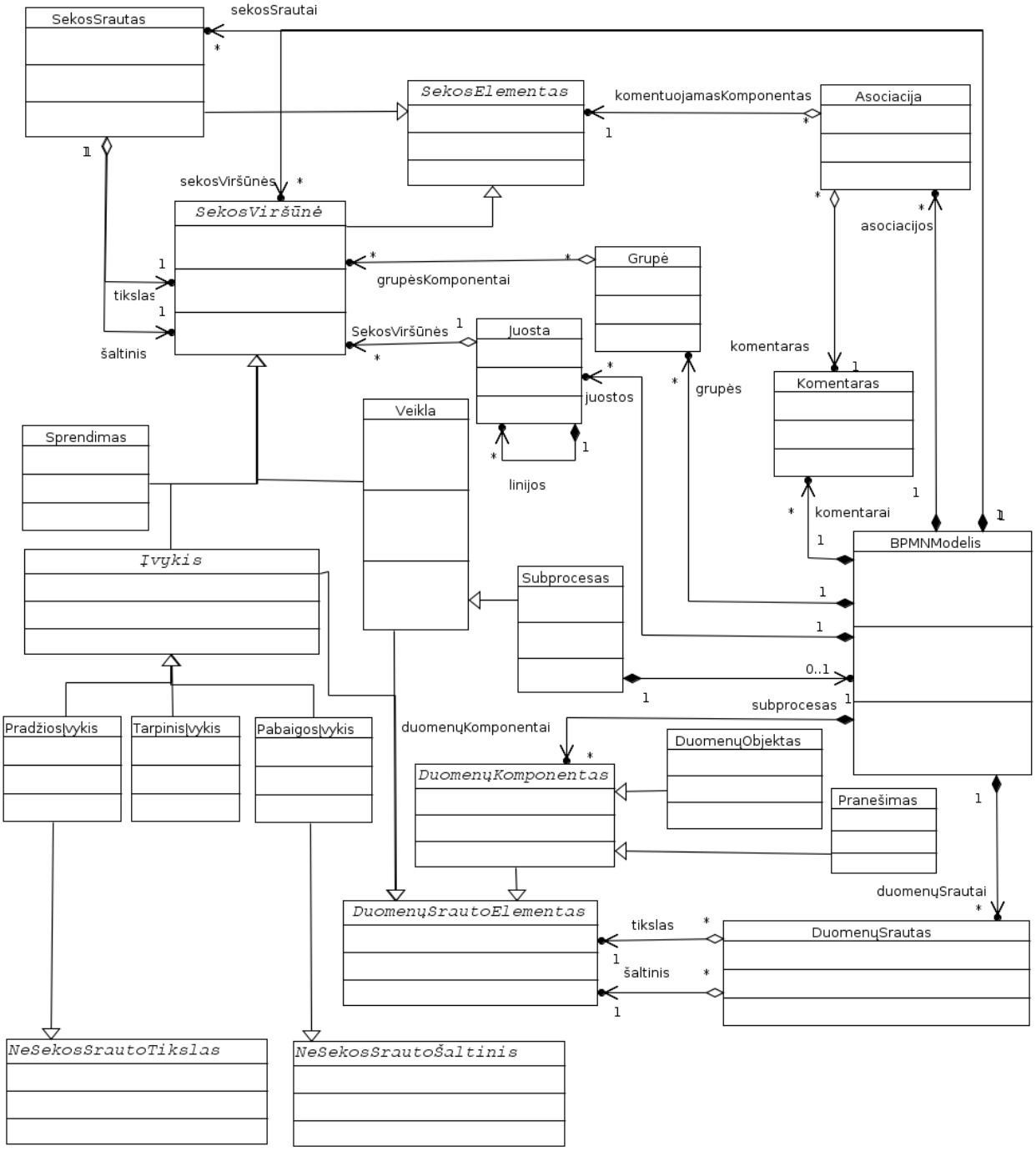
Komentaras (text annotation) yra mechanizmas skirtas pateikti papildomai informacijai diagramos skaitytojams. Šis komponentas neįtakoja sekos ar duomenų srautų, o tik paaiškina kas ir kodėl vyksta. Žymimas skliaustu iš kairės komentaro teksto pusės (14 pav.).

15 pav. Asociacijos žymuo

Asociacija (association) skirta komentarams susieti su modelio komponentais. Ji nurodo kuri diagramos dalis yra komentuojama. Asociacija vaizduojama punktyrine linija (15 pav.).

1.1.3. BPMN komponentų tarpusavio ryšiai

Norint modeliuoti verslo procesus vien komponentų nepakanka, taip pat reikia žinoti ir kaip jie sąveikauja tarpusavyje. Tarpusavio ryšius gana neblogai apibūdina metamodelis. Šiame darbe jis bus dažnai naudojamas tam tikslui. Skyriuje 1.1.2 aprašytų komponentų sąveiką galima pavaizduoti (16 pav.) metamodeliu.



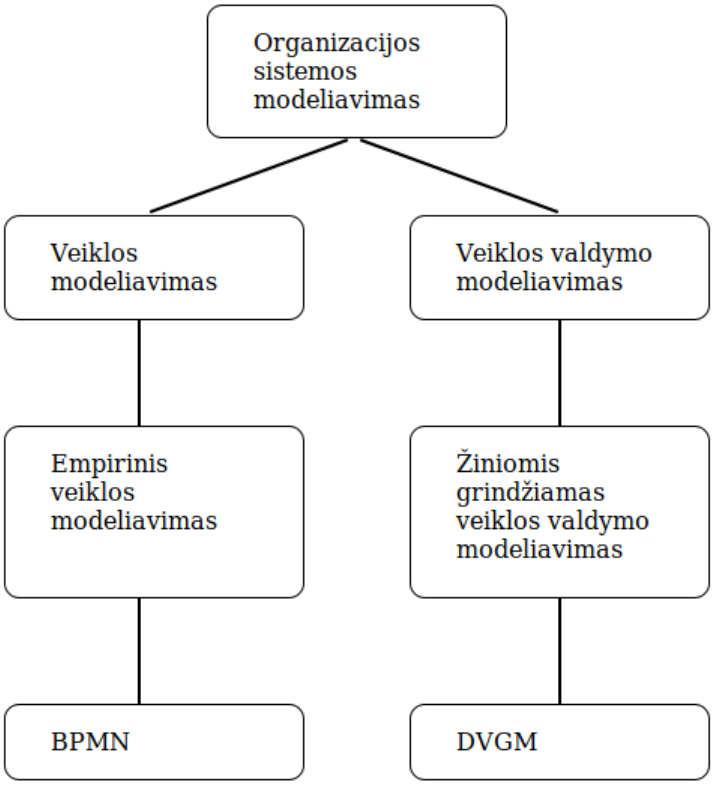
16 pav. **BPMN** pagrindinių komponentų metamodelis

Šiame metamodelyje tipas BPMNModelis yra šakninis komponentas. Jis savyje laiko sekos viršūnes, sekos srautus, juostas, grupes, duomenų komponentus, duomenų srautus, komentarus ir asociacijas. Komponentus metamodelyje galima suskirstyti į tris grupes: sekos komponentus, duomenų komponentus ir komentavimo komponentus. Sekos komponentams priklauso SekosElemento tipų hierarchija, duomenų komponentams priklauso DuomenųKomponento hierarchija ir DuomenųSrautas, komentavimui priklauso Komentaras ir Asociacija. Taigi galima iš eilės apibūdinti šias grupes iš kurių susideda **BPMN** modelis.

Sekos viršūnė yra abstrakcija nurodanti, kad tai yra komponentai jungiami sekos srautais, tokie kaip veikla, sprendimas ir įvykio subtipai. Įvykis yra abstraktus tipas nes modelyje būna kurie nors iš jo subtipų. Sekos srautas turi vieną šaltinį ir vieną tikslą. Autorius įveda 2 abstrakcijas: ne sekos srauto šaltinis ir ne sekos srauto tikslas, kad parodyti **BPMN** apribojimus, kurie draudžia pabaigos įvykiui būti sekos srauto šaltiniu, o pradžios įvykiui – tikslu. Grupės ir Juostos nurodo kurios sekos viršunės joms priklauso. Juosta dar gali turėti linijų. Metamodelyje taip pat parodoma, kad Veikla gali būti subprocesas, tokiu būdu savyje laikydama kitą procesą. Duomenų srauto elementas yra autoriaus įvesta abstrakcija parodanti, kad jo subtipai gali būti jungiami duomenų srautais. Įvesta abstrakcija duomenų komponentas norint apibendrinti praneŠimą ir duomenų objektą. Komentaras jungiamas Asociacija ir gali komentuoti sekos komponentus.

1.2. Veiklos modeliavimas

Veiklos modeliavimo metodologijas galima suskirstyti pagal savybes kuriomis jos pasižymi. Šiuo atveju jos skirstomos pagal tai, kaip naudoja žinias Toks suskirstymas parodys kaip sumodeliuoti daugiau naudingos informacijos ir kodėl šiame darbe naudojamas detalizuotas grandinės vertės modelis (**DVGM**).



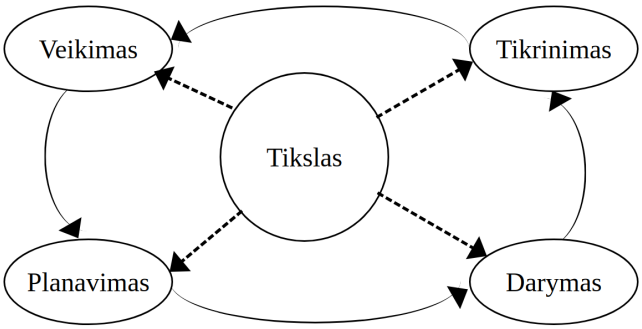
17 pav. Veiklos modeliavimo požymiai pagal žinių naudojimą

Skirstant veiklos modeliavimo metodus pagal žinių naudojimą galima pastebėti dvi šakas (17 pav.). Pirmoje veikla modeliuojama naudojant patirtį iš anksčiau stebėtų veiklų, tokiu atveju analitikas sudaro modelį, kuris jo manymu bus teisingas. Prie tokių metodologijų galima priskirti

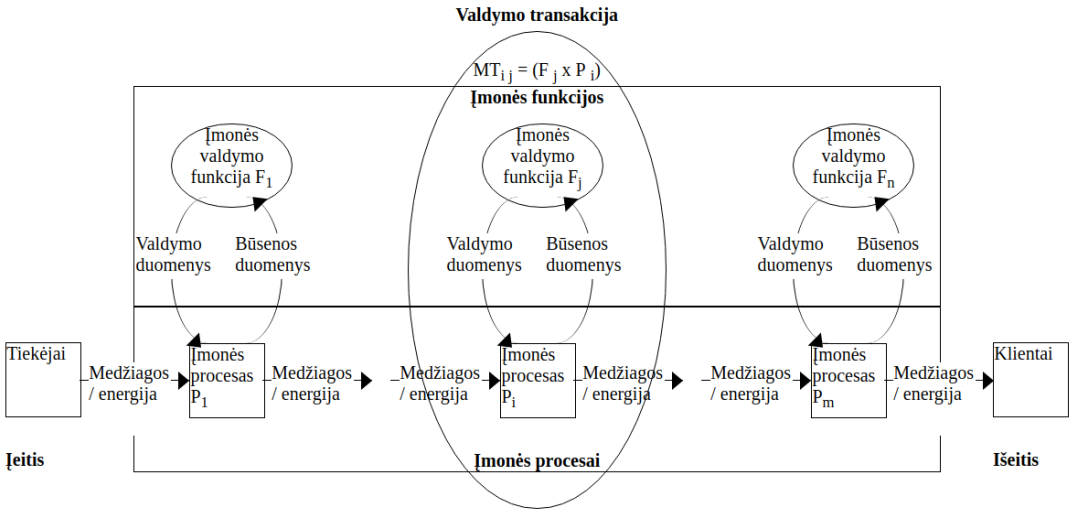
BPMN . Antroje šakoje veiklos valdymas modeliuojamas naudojant žinias apie tai kokį modelį reikia sudaryti. Šiuo atveju yra paimamas teorijoje aprašytas modelis ir veiklos organizuojamos pagal jį. Tuo pasižymi **DVGM**.

1.3. Detalizuotas vertės grandinės modelis – DVGM

BPMN modelyje galima pastebėti organizacijos valdymo veiklų supratimo neapibrėžtumus [GL16]. Taip yra todėl, kad empirinio modeliavimo metodai neparodo informacijos arba resursų transformavimo priežasčių. Tačiau įmonę galima analizuoti ir transakcinių darbų sekų modelio (18 pav.) požiūriu, tokiu būdu organizuojant veiklą kaip teoriškai teisingą.



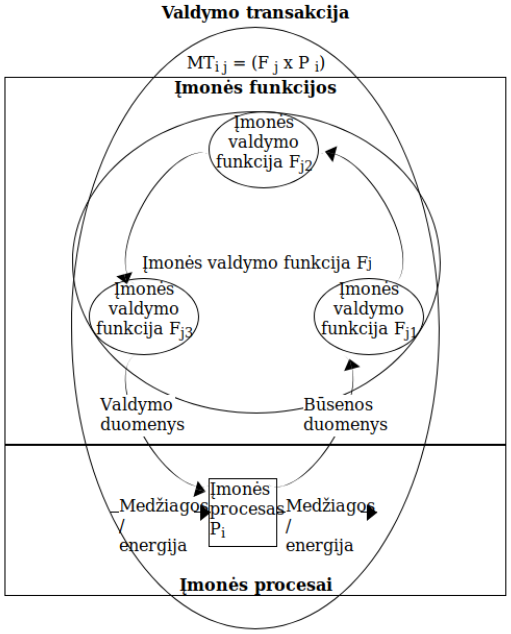
18 pav. Transakcinių darbų sekų modelio pavyzdys



19 pav. Detalizuotas M. Porterio vertės grandinės modelis

Darbe organizacijos procesas bus nagrinėjamas kaip transakcijų visuma. Materialios veiklos atskiriamos nuo valdymo veiklų. P_i žymi veiklos procesą, kuris transformuoja žaliavas, medžiagas, energiją ir formuoja materialią išeigą. F_j yra veiklos valdymo funkcija, informacijos (duomenų, žinių) transformavimo veikla, būtina valdant procesą P_i. Modelis yra suskirstytas į valdymo transakcijas $MT_{ij} = F_j \times P_i$. Tokiu būdu pateikiama daugiau informacijos apie įmonę. Diagrama bus vaizduojama kaip detalizuotas M. Porterio vertės grandinės modelis (19 pav).

Valdymo funkcija F_j gali būti suskaidyta smulkiau. (20 pav.) parodytas pavyzdys kai F_j susideda iš smulkesnių dalių F_{j1}, F_{j2} ir F_{j3}. Kartu visa tai suteikia veiklos procesui P_i valdymo duomenis kuriuos jis panaudoja vykdymui. Vėliau grąžinami būsenos duomenys, jie panaudojami valdymo funkcijoje ir ciklas kartojasi.



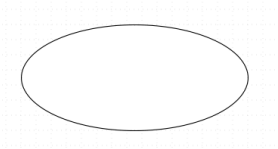
20 pav. Valdymo funkcijos F_j išskaidymo pavyzdys

1.3.1. DVGM apimtis

Detalizuotas vertės grandinės modelis leidžia modeliuoti organizacijos veikimą kaip transakcijų visumą. Jis atskiria vadybos veiklas nuo produkcijos veiklų. Parodomi valdymo informacijos ciklai, kurie kontroliuoja produkcijos veiklą ir padeda siekti organizacijos užsibrėžtų tikslų. Taip pat parodomi informacijos srautų tipai, produkcijos veiklų įeigos ir išeigos tipai. Visa tai leidžia suprasti kodėl įmonėje egzistuoja viena ar kita valdymo transakcija ir kaip jos įtakoja organizacijos tikslų pasiekimą.

1.3.2. DVGM komponentai

Pateikiami pagrindiniai **DVGM** komponentai. Jie leidžia veiklą modeliuoti kaip transakcinių darbų seką, kurioje procesai yra atskirti nuo valdymo funkcijų.

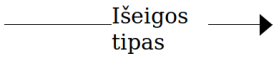


21 pav. Valdymo funkcijos žymuo

Valdymo funkcija (management function) darbas atliekamas organizacijoje, kuris prisideda į valdymo duomenų generavimą. Šio darbo įeiga paprastai savyje turi jo valdomo veiklos proceso proceso būseną. Atlikus darbą generuojami valdymo duomenys. Žymimas ovalu viduje paliekant vietos pavadinimui (21 pav.).

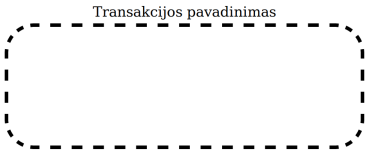
22 pav. Veiklos proceso žymuo

Veiklos procesas arba tiesiog procesas (process) yra darbas atliekamas organizacijoje, kuris tiesiogiai prisideda prie organizacijos gaminamos išeigos sukūrimo. Šis komponentas valdomas valdymo funkcijų per valdymo transakcijose keliaujančius duomenis. Joms, atlikęs darbą, per interakcijų sekos srautus jis pateikia savo būsenos duomenis ir gauna valdymo duomenis. Žymimas stačiakampiu viduje paliekant vietos pavadinimui (22 pav.).



23 pav. Interakcijų sekos srauto žymuo

Interakcijų sekos srautas (interactions sequence flow) parodo valdymo funkcijų ir veiklos procesų seką, taip pat kas pagaminama atliekant darbus. Kadangi organizacijos veikla vaizduojama ciklais vieno darbo išeiga tampa kito įeiga. Šis komponentas vaizduojamas solidžia linija su užpildyto trikampio formos rodykle gale ir išeigos (kuri tampa įeiga) tipo vardu prie pavaizduotos linijos (23 pav.).

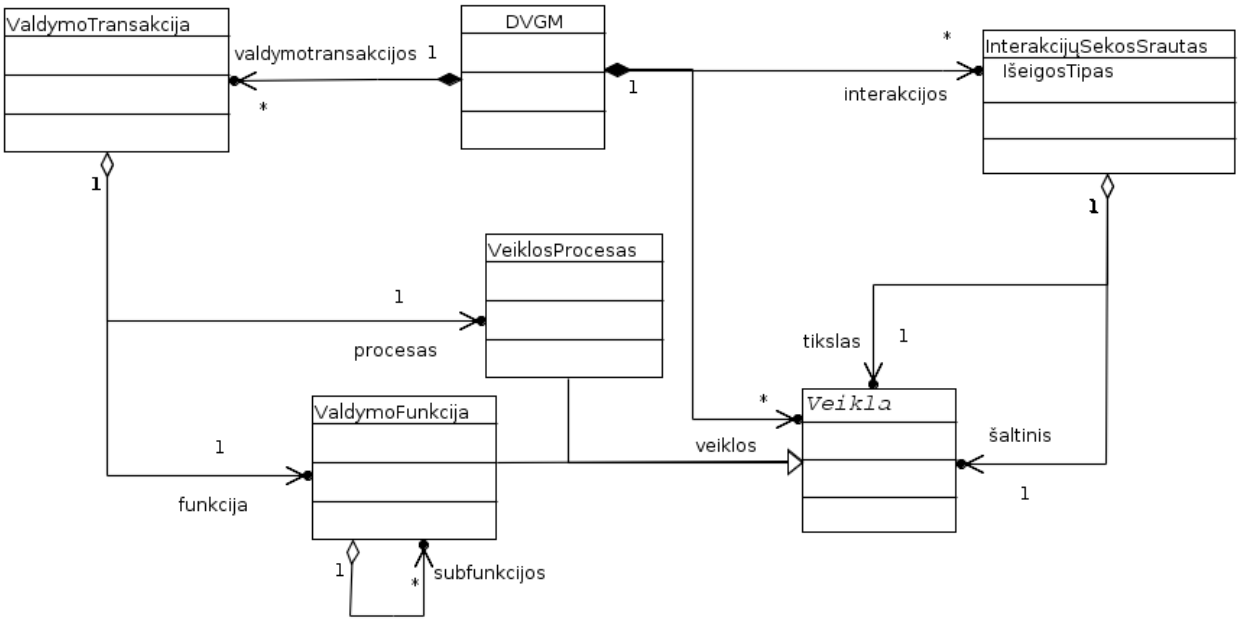


24 pav. Valdymo transakcijos žymuo

Valdymo transakcija (management transaction) parodo, kaip valdymo funkcijos kontroliuoja veiklos procesą. Šis komponentas žymi valdymo duomenų transformacijų ciklą. Jis vaizduojamas punktyrine linija apibraukiant transakcijai priklausantį veiklos procesą ir valdymo funkciją bei parašant transakcijos pavadinimą (24 pav.).

1.3.3. DVGM komponentų tarpusavio ryšiai

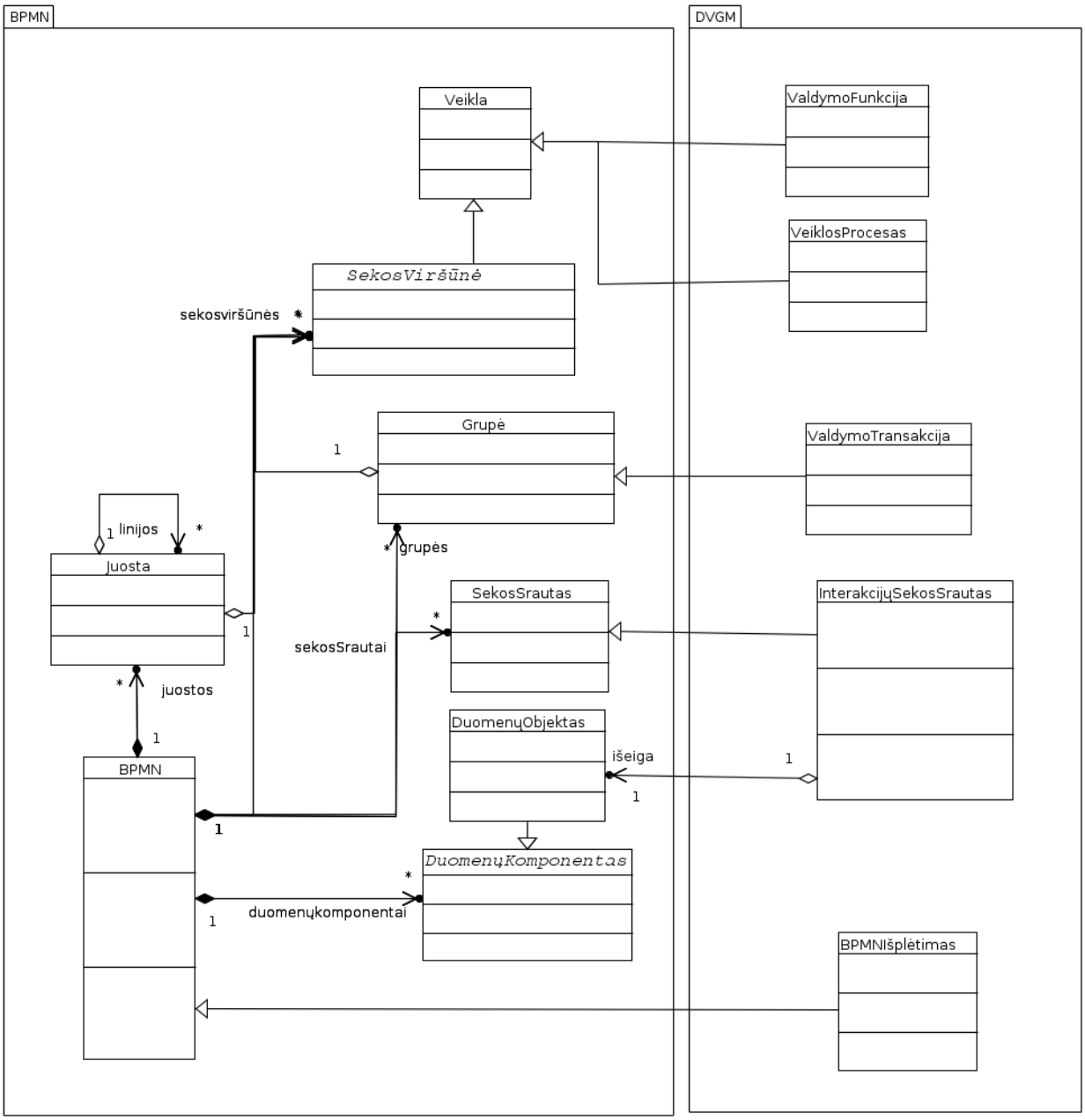
DVGM komponentų tarpusavio ryšiai pavaizduoti metamodeliu (25 pav.). Šakninis komponentas DVGM savyje laiko valdymo transakcijas, interakcijų sekos srautus ir veiklas. Veiklos jungiamos interakcijų sekos srautais, jie taip pat nurodo kokią išeigą šiai interakcijai generuoja veikla. Veiklos skirstomos į veiklos procesus ir valdymo funkcijas. Valdymo transakcija turi nuoroda į procesą ir jį valdančią funkciją, kuri gali susidaryti iš smulkesnių dalių, taip pat sujungtų interakcijų srautais.



25 pav. **DVGM** metamodelis

1.3.4. DVGM kaip BPMN praplėtimas

DVGM modelį galima apibrėžti ir kaip **BPMN** praplėtimą. Paveikslas 26 vaizduoja tokį praplėtimo pavyzdį, tai bus įvesties duomenų formatas kuriamam algoritmui. Tokiu būdu apibrėžtą modelį galima nagrinėti kaip **BPMN**.



26 pav. **DVGM** kaip **BPMN** praplėtimo metamodelis

26 paveiksle **BPMN** komponentai vaizduojami iš kairės, iš dešinės yra komponentai išplečiantys modelį. Veikla išplečiama valdymo funkcija ir veiklos procesu. Valdymo transakcija yra grupė, o interakcijų srautas rodo veiklų sekos srautą ir jų perduodamus duomenis. Visa Tai sudaro **BPMN** išplėtimą.

1.4. Užduočių diagrama

1992 metais Ivar Jacobson apibrėžė metodologiją specifikuoti vartojimo atvejus [ICJ+92]. Jis pateikė būdą apibrėžti atvejus tiek tekstu (1 lentelė), tiek diagrama (27 pav.). Tarp programų sistemų kūrėjų ši metodologija išpopuliarėjo kaip funkcinių reikalavimų apibrėžimo technika. **OMG**

savo specifikacijose **UML** [Obj15] ir **SysML** [OMG17] standartizuoja vartojimo atvejų modelį ir priskiria jį prie elgsenos apibūdinimo diagramų. 2011 metais Ivar Jacobson paskelbė užduočių modelio 2.0 versiją [JSB11], kuri pritaikyta prie judrių metodologijų projektų įgyvendinimo.

1 lentelė. Tekstinio naudojimo atvejo pavyzdys

ID	NA1
Pavadinimas	Registruoto naudotojo autentifikavimas

Aktoriai

1. Registruotas naudotojas.
2. Išorinė autentifikavimo sistema.

Aprašas

Registruotas naudotojas autentifikuojamas sistemoje.

Prieš sąlygos

1. Naudotojas nėra autentifikuotas sistemoje.

Priežastys

1. Naudotojas pareikalavo būti autentifikuotas.

Po sąlygos

1. Naudotojas yra autentifikuotas sistemoje.

Pagrindinė užduočių seka

1. Sistema pateikia autentifikavimo variantus.
2. Naudotojas autentifikuojamas sistemos turimu autentifikavimo būdu.
3. Sistema pateikia patvirtinimą, kad naudotojas autentifikuotas.

Alternatyvios užduočių sekos

2.1. Naudotojas autentifikuojamas išorine autentifikavimo sistema.

Išimtinės užduočių sekos

*.1. Naudotojas atsisako autentifikavimo.

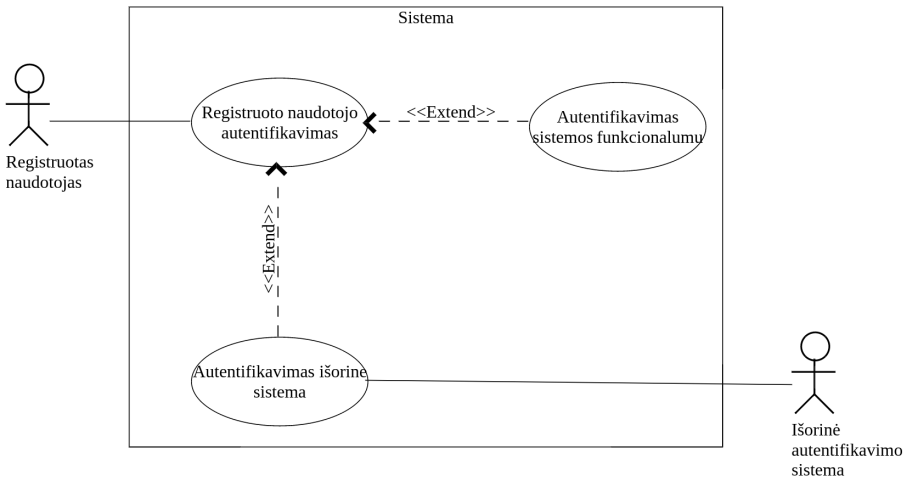
*.1.1. Naudotojas pateikia autentifikavimo atsisakymą.

*.1.2. Sistema nebereikalauja autentifikavimo duomenų.

3.1. Nepavyksta autentifikuoti naudotojo pateiktais duomenimis.

3.1.1. Sistema naudotojui pateikia pranešimą apie tai, kad jis negali būti autentifikuotas pateiktais duomenimis.

Lentelėje nr. 1 pavaizduotas autentifikavimo sistemoje tekstinio naudojimo atvejo pavyzdys. Naudojimo atvejai paprastai turi identifikacijos numerį pagal kurį nurodomi reikalavimų specifikacijoje. Taip pat pavadinimą, kad būtų patogiau apie jį kalbėti. Išvardinami aktoriai dalyvaujantys vykdyme. Naudojimo atvejis trumpai ir aiškiai aprašomas. Nurodomos kokios aktualios sąlygos būna prieš vykdant, kas įtakojo vykdymą ir rezultatai. Tuomet išvardinamos užduotys reikalingos pasiekti rezultatui. Jeigu naudojimo atvejis gali būti įvykdytas keliais būdais, jie nurodomi alternatyviose užduočių sekose. Jeigu vykdant užduočių sekas gali atsitikti kažkas, dėl ko nepavyksta pasiekti sėkmingo rezultato sąlygų, tai nurodoma išimtinėse užduočių sekose.



27 pav. Užduočių diagrama pagal 1 lentelę

Naudojimo atvejus galima atvaizduoti užduočių diagrama. 27 paveikslas vaizduoja 1 lentelėje aprašytą naudojimo atvejo užduočių diagramą. Pagrindinė užduočių seka pavaizduota naudojimo

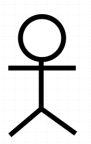
atveju „Autentifikavimas sistemos funkcionalumu”, alternatyvi – „Autentifikavimas išorine sistema”, išimtinės sekos nepavaizduotos.

1.4.1. Užduočių diagramos apimtis

Užduočių diagrama skirta apibūdinti kaip naudotojai siekia savo tikslų pasitelkdami sistemą. Ji parodo kokie yra funkciniai reikalavimai, naudotojų taksonomiją, duomenų srautus tarp naudotojų ir sistemos. Taip pat pateikia funkcijų hierarchijos modeliavimą ir leidžia jas suskaidyti (naudojant įtraukimo ryšį). Užduočių diagrama apibūdina funkcinius reikalavimus, ji nepateikia nei funkcijų atlikimo tvarkos, nei duomenų struktūrų naudojamų sistemoje.

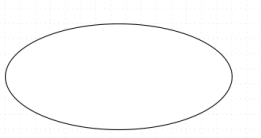
1.4.2. Užduočių diagramos komponentai

Skirtingi šaltiniai pateikia kiek skirtingus komponentus ir jų apibrėžimus. Šiame darbe daugiausia taikomi **OMG** standartuose pateikti apibrėžimai. Toliau bus pateikiami **UML** standarte apibrėžtos užduočių diagramos komponentai.



28 pav. Aktoriaus žymuo

Aktorius (actor) žymi naudotojo rolę. Ją gali atlikti tiek žmogus, tiek išorinė programų sistema. Aktoriai sąveikauja su bendravimo kanalais prijungtais naudojimo atvejais. Šis komponentas žymimas žmogumi iš pagaliukų (28 pav.).

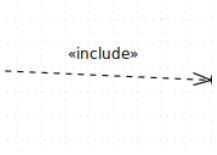


29 pav. Naudojimo atvejo žymuo

Naudojimo atvejis (use case) žymi veiklų kurias atlikus gaunamas naudingas rezultatas aibę. Rezultatas gali būti naudingas tiek vykdytojui, tiek kitiems suinteresuotiems žmonėms. Šis komponentas vaizduojamas elipse (29 pav.), naudojimo atvejo pavadinimas gali būti tiek elipsėje, tiek po ja.

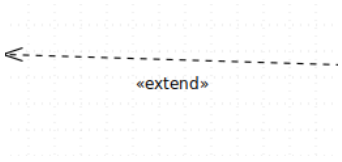
30 pav. Bendravimo kanalo žymuo

Bendravimo kanalas (communication path) žymi sąveika tarp aktoriaus ir sistemos. Šis komponentas diagramoje jungia aktorių su naudojimo atveju, taip parodydamas, kad prieš atlikdamas veiklas (sistemos funkcijas) naudotojas pateikia įvestį o po jų atlikimo gaunamas rezultatas. Jeigu bendravimo kanalo ryšio su aktoriumi gausa yra daugiau nei 1 reiškia naudojimo atvejui atlikti reikalingi keli vykdytojai, jeigu bendravimo kanalo ryšio su naudojimo atveju gausa yra daugiau nei 1 reiškia aktorius gali atlikti tas pačias veiklas daugiau nei vieną kartą. Asosijacija žymima solidžia linija (30 pav.).



31 pav. Įtraukimo žymuo

Įtraukimas (includes) žymi naudojimo atvejo suskaidymą. Šis komponentas modeliuoja sąryši tarp dviejų vartojimo atvejų, taip parodydamas, kad įtraukiamo naudojimo atvejo veiklos yra atliekamos įtraukiančiame naudojimo atvejyje. Įtraukimą numatyta naudoti tuomet, kai tos pačios veiklos pasikartoja keliuose naudojimo atvejuose. Tos veiklos įdedamos į atskirą naudojimo atvejį ir prijungiamos šiuo ryšiu, tokiu būdu iškeliamas pasikartojantis funkcionalumas. Įtraukimas vaizduojamas punktyrine linija su rodykle prie įtraukiamo naudojimo atvejo ir patikslinimu dvigubuose kampiniuose skliaustuose (31 pav.).



32 pav. Išplėtimo žymuo

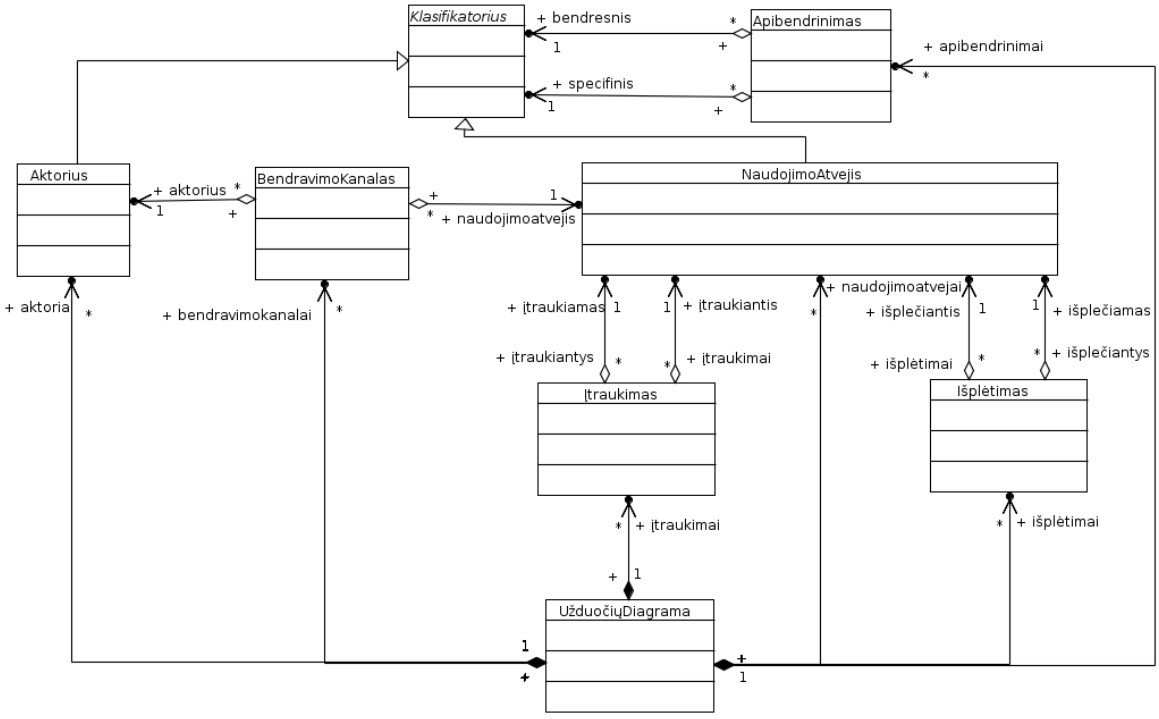
Išplėtimas (extends) žymi, kad esant tam tikroms sąlygoms naudojimo atvejis įtraukia veiklas iš kitų naudojimo atvejų. Šis komponentas numatytas naudoti bendram funkcionalumui iškelti, bet kitaip nei ryšys „Įtraukia“, parodo, kad veiklos įtraukiamos ne visada. Jis vaizduojamas punktyrine linija su rodykle prie išplečiamo naudojimo atvejo ir patikslinimu dvigubuose kampiniuose skliaustuose (32 pav.).

33 pav. Apibendrinimo žymuo

Apibendrinimas (generalization) žymi, kad elementas yra bendresnio elemento variantas. Nuo išplėtimo naudojimo atvejo apibendrinimas skiriasi tuo, kad pasakoma jog bent vienas iš apibendrinamų naudojimo atvejų funkcionalumų įtraukiamas į apibendrinančio naudojimo atvejo funkcionalumą. Žymimas solidžia linija su rodykle prie apibendrinamo naudojimo atvejo (33 pav.).

1.4.3. Užduočių diagramos komponentų tarpusavio ryšiai

Komponentų aprašytų 1.4.2 skyriuje tarpusavio ryšiai pavaizduoti metamodeliu (34 pav.). Jis sudarytas pagal **UML** ir **SysML** standartuose pateiktą informaciją.



34 pav. Užduočių diagramos metamodelis

Užduočių diagrama yra paketas savyje laikantis aktorius, naudojimo atvejus, bendravimo kanalus, įtraukimus, išplėtimus ir apibendrinimus. Bendravimo kanalas jungia vieną aktorių su vienu naudojimo atveju. Įtraukimas yra ryšys tarp įtraukiamo ir įtraukiančio naudojimo atvejo. Išplėtimas jungia išplečiamą naudojimo atvejį su išplečiančiu. **UML** modelis leidžia abstraktaus tipo klasifikatoriaus apibendrinimą. Aktorius ir naudojimo atvejis yra klasifikatoriaus subtipai, todėl gali būti apibendrinti.

2. Veiklos modelių konvertavimas

2.1. UML diagramų transformavimo algoritmai

2.2. Algoritmas BPMN modeliui transformuoti į Užduočių diagramą

Literatūroje yra parašyta apie **vartojimo atvejų diagramos** išvedimą iš **BPMN** modelio [DJ02]. Straipsnyje aprašytas algoritmas atlieka (3) transformaciją. Imamas modifikuotas **BPMN** modelis (1) ir tie **vartojimo atvejų diagramos** komponentai, kurie gali būti iš jo išvesti (2).

$$BPMNElements = \{Start,End,Role,Branch,Task,Transition\}; \quad (1)$$

$$UseCasesElements = \{Actor, Generalization, Association,UseCase, Include, ExtensionPoint, Extend\}; \quad (2)$$

$$BPMN(BPMNModelElements) \Rightarrow UseCases(UseCasesElements); \quad (3)$$

Autoriai pateikia suprastintus jų naudojamų modelių metamodelius (priedai Nr. 3 ir Nr. 4). Kadangi daugybė **BPMN** komponentų neturi jokios įtakos algoritmo rezultatui jie nėra minimi straipsnyje. Modelis taip pat praplečiamas sekos srautui pridedant laiko trukmę, kuri naudojama empirinėse algoritmo taisyklėse.

Algoritmo autoriai pirmiausia siūlo surasti ryšius tarp modelių. Juostos atitinka aktorius. Užduotys tuo tarpu grupuojamos, kol nepasiekia maksimalaus skaičiaus vykdomų be pertraukos, priklausančių tai pačiai juostai ir pagaminančių rezultatą. Tokia grupė pavadinama žingsniu ir yra laikoma atitinkančia vartojimo atvejį. Tokiu būdu galima gauti integruotą metamodelį apimantį savyje anksčiau minėtus metamodelius. Straipsnyje pateiktas suprastintas jo variantas (priedas Nr. 5) Tuomet lieka surasti kaip dar galima būtų panaudoti informaciją, patikslinti ir suprastinti gautoms diagramoms.

Vėliau pristatomas algoritmas. Jis Pirmiausia sudėlioja užduotis į proceso žingsnius. Vėliau juostos tampa aktoriais, o žingsniai jose – vartojimo atvejais. Galiausiai pasikartojančios užduotis išimamos iš žingsnių ir prijungiamos bendravimo kanalu įtraukia arba išplečia pagal situaciją.

2.3. Užduočių diagramos išvedimas iš BPMN modelio

Šio darbo tikslas, algoritmas galintis gauti užduočių diagramas iš **BPMN** modelio, bus kuriamas pagal 2.2 skyriuje aprašytą algoritmo sukūrimo pavyzdį. Pirmiausia bus rasti ryšiai tarp diagramų, vėliau sukurtas būdas juos panaudoti, galiausiai panaudota likusi modelio informacija patikslinti ir suprastinti diagramoms.

Nuo anksčiau minėto algoritmo jis skirsis tuo, kad naudos **BPMN** sukonstruota pagal **DVGM** metodologiją kaip įvesties duomenis. 2.2 skyriuje aprašytas algoritmas įvestiems duomenims aprašyti siūlo modifikuotą **BPMN** modelį ir užduočių diagramas sudaro vadovaudamasis empirinė-

mis taisyklėmis pasakančiomis kada geriau konstruoti užduotį. Šiame darbe kuriamas algoritmas užduotis konstruos remdamasis **DVGM** apibrėžta veiklos organizavimo teorija. Tai leis tiksliau apibrėžti užduočių diagramos naudojimo atvejų tarpusavio ryšius. 2.2 algoritmo autorių siūlomas būdas apibrėžti veiklos žingsnius atitinka **DVGM** transakciją, tokiu būdu nereikia taikyti empirinių taisyklių ieškant žingsnio pradžios ir pabaigos.

2.3.1. Ryšiai tarp BPMN ir užduočių diagramų

Norint duomenis iš vieno modelio perkelti į kitą galima pasinaudoti ryšiais esančiais tarp jų.

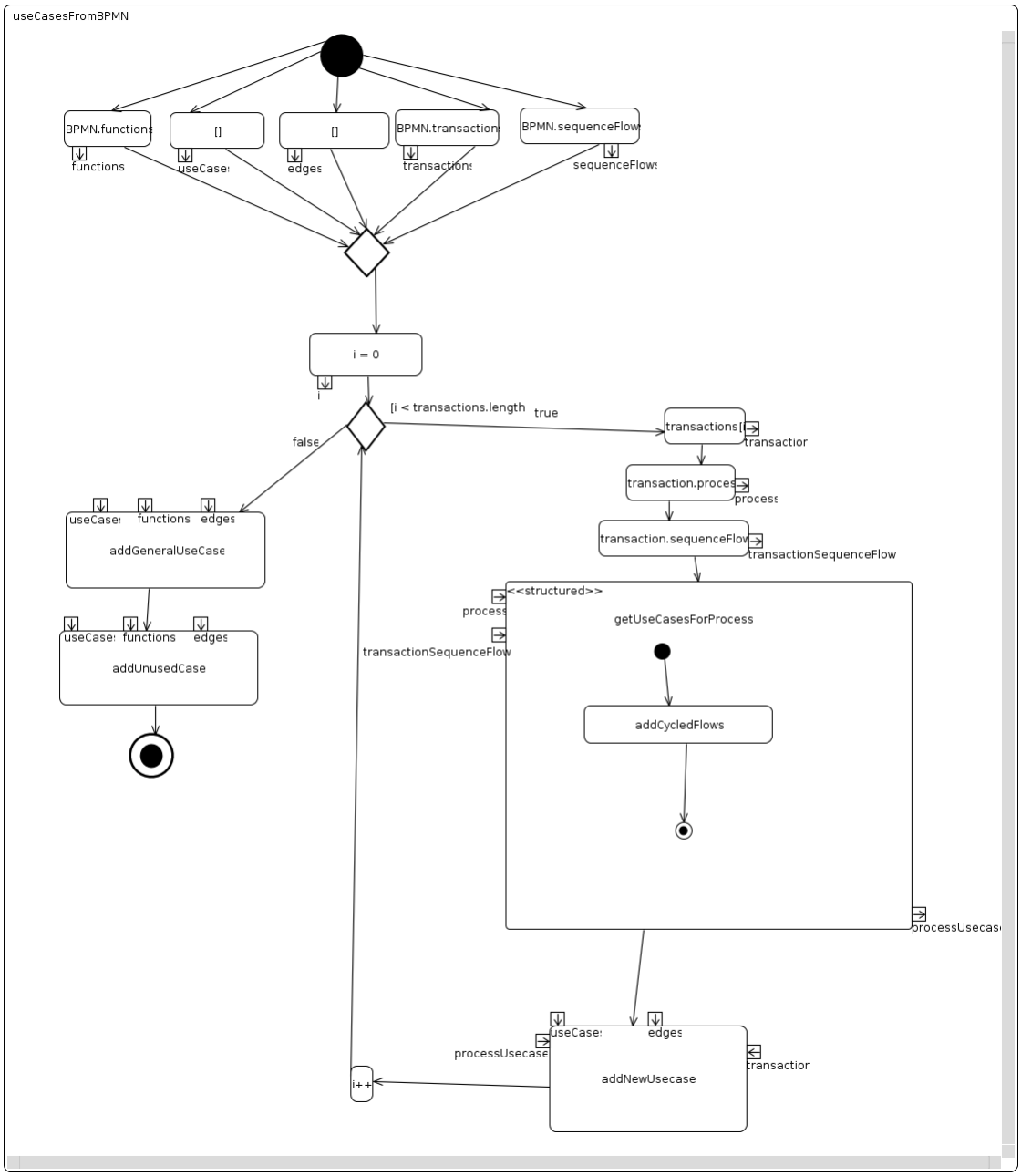
2 lentelė. Ryšiai tarp **DVGM** kaip **BPMN** praplėtimo ir užduočių diagramų

	Aktorius	Vartojimo atvejis	Bendravimo kanalas	Įtraukia	Išplečia
Juosta	+				
Valdymo funkcija		+		+	+
Įmonės procesas	+				
Valdymo transakcija		+		+	+
Interakcijų sekos srautas			+		

Iš juostos galima gauti Aktorius. Valdymo funkcija turi informacijos apie tai kokie vartojimo atvejai yra sistemoje ir kaip jie susiję vienas su kitu. Įmonės procesas taip pat yra aktoriai naudojantys sistemą. Valdymo transakcija yra vartojimo atvejis. Interakcijų sekos srautas parodo kokie duomenys keliauja bendravimo kanalais. Rasti ryšiai taip pat parodo kurie komponentai bus imami ir kurie gaunami. Taigi galima apibrėžti algoritmo įvesties ir išvesties duomenis.

2.3.2. Ryšių tarp diagramų panaudojimas transformacijai

Rasti ryšiai parodo į kokius **DVGM** komponentus reikia žiūrėti išvedant užduočių diagramos dalis. Toliau peržiūrimi diagramos variantai ir sudėliojami konkretūs žingsniai kuriuos reikia atlikti. Galiausiai gaunamas pseudokodas (Pseudokodas 1). Jo veikimas pažingsniui aprašomas, taip pat pavaizduotas (pav. 35).



35 pav. Algoritmo diagrama

1. Iškviečiama funkcija useCasesFromBPMN (Pseudokodas 1) paduodant jai **BPMN** modelį.

Pseudokodas 1: **UML** Užduočių diagramos gavimo iš **BPMN** modelio algoritmo pseudokodas

```
1 useCasesFromBPMN = function(BPMN){
2   transactions = BPMN.transactions;
3   sequenceFlows = BPMN.sequenceFlows;
4   useCases = [];
5   edges = [];
6   for(transaction in transactions){
```

```
7     process = transaction.process;
8     transactionSequenceFlows = transaction.sequenceFlows;
9     processUsecases = getUseCasesForProcess(process,
10        transactionSequenceFlows);
11    addNewUsecases(edges,useCases,processUsecases,transaction);
12  }
13  functions = BPMN.functions;
14  addGeneralUseCases(edges,useCases,functions);
15  addUnusedCases(useCases,functions);
16
17  UseCaseDiagram = {useCases,edges};
18  return UseCaseDiagram;
19 }
```

2. Po duomenų inicializavimo pirmiausia imamas transakcijos procesas (eil. 7) ir gaunami sekos srautai jungiantys komponentus transakcijoje (eil. 8).

3. Vėliau sukuriami vartojimo atvejai apibrėžiantys proceso valdymą (eil. 9) funkcija getUseCasesForProcess (Pseudokodas 2).

Pseudokodas 2: Funkcija getUseCasesForProcess

```
1  getUseCasesForProcess = function(process,sequenceFlows){
2    processFlows = sequenceFlows.filter(flow => (
3      flow.source == process
4    ));
5    cycledFlows = [];
6    for(flow in processFlows){
7      addCycledFlows(cycledFlows,[],
8          flow,process,sequenceFlows)
9    }
10   processUsecases = [];
11   for(flow in cycledFlows){
12     usecase = createUseCase(flow);
13     processUsecases.add(usecase);
14   }
15   return processUsecases;
16 }
```

4. Joje randami sekos srautai išeinantys iš proceso (eil. 2) ir kiekvienam iš jų rekursijos bū-

du surandami ciklai su procesu funkcija addCycledFlows (Pseudokodas 3). Kiekvienam iš cikle esančių sekos srautų sukuriamas vartojimo atvejis (eil. 10 - 14). Jų kolekcija ir yra (Pseudokodas 2) grąžinamas rezultatas.

Pseudokodas 3: Funkcija addCycledFlows

```
1  addCycledFlows = function(cycledFlows,currentPath,
2      newFlow,process,sequenceFlows){
3    if(newFlow.target == process){
4      return true;
5    }
6    if(currentPath.contains(newFlow)){
7      return false;
8    }
9    if(cycledFlows.contains(newFlow)){
10     return true; // We already found path this way
11   }
12   nextFlows = sequenceFlows.filter(flow => (
13     flow.source == newFlow.target
14   ));
15   if(nextFlows.length == 0){
16     return false;
17   }
18   cycledFlows.add(newFlow);
19   currentPath.add(newFlow);
20   pathFound = false;
21   for(flow in nextFlows){
22     if(addCycledFlows(cycledFlows,currentPath,
23         flow,process,sequenceFlows)){
24       pathFound = true;
25       // No brake because we need to recursivelly add all paths
26     }
27   }
28   if(!pathFound){
29     cycledFlows.remove(newFlow);
30   }
31   currentPath.remove(newFlow);
32   return pathFound;
33 }
```

Minėta Funkcija (Pseudokodas 3) pirmiausia patikrina ar jau nėra ciklo su procesu ir jei taip patvirtina, kad parametras currentPath turi savyje kelia į procesą (eil. 3 - 5). Jei kelias užsiciklino grąžinamas neigiamas atsakymas (eil. 6 - 8), tai reiškia grįžimą atgal. Jei žingsnis veda į jau išsaugotą sėkmingą kelio atkarpą patvirtinamas jo teisingumas (eil. 9 - 11). Jei nei viena iš šių sąlygų nepasitvirtino paimami sekantys žingsniai (eil. 12). Jų neradus pranešama apie aklavietę (eil. 15 - 17). Kol kas pažymima, kad žingsnis yra sėkmingas (eil. 18) ir žengtas (eil. 19). Toliau ieškoma ciklų su procesu einant sekančiais sekos srautais (eil. 21 - 27). Neradus nei vieno kelio į procesą ištrinamas pažymėjimas apie žingsnio teisingumą (eil. 28 - 30). Kadangi keliai žengus šį žingsnį ištyrinėti grįžtama atgal (eil. 31).

5. (Pseudokodas 2) sukurti vartojimo atvejai pridedami prie jau gautų vartojimo atvejų kartu su ryšiais tarp jų funkcija addNewUsecases (Pseudokodas 4).

Pseudokodas 4: Funkcija addNewUsecases

```
1  addNewUsecases = function(edges,useCases,newUseCases,transaction){
2    if(newUseCases.length < 1){
3      return;
4    }
5    mainUseCase = createUseCase(transaction);
6    useCases.add(mainUseCase);
7    if(newUseCases.length == 1){
8      useCase = newUseCases[0];
9      mainUseCase.createdFromFlow = useCase.createdFromFlow;
10     mainUseCase.usedBy = useCase.usedBy;
11     return;
12   }
13   for(useCase in newUseCases){
14     useCases.push(useCase);
15     edge = createIncludeEdge(mainUseCase,useCase);
16     edges.push(edge);
17   }
18 }
```

Joje sukuriamas vartojimo atvejis visai transakcijai (eil. 5), pažiūrima kiek vartojimo atvejų rasta, jei vienas tai jo informacija išsaugoma į pagrindinį vartojimo atvejį (eil. 7 - 12). Radus daugiau, jie išsaugomi kaip įeinantys į transakciją (eil. 13 - 17).

6. Galiausiai randamos bendros funkcijos tarp transakcijų ir sukuriami apibendrinantys vartojimo atvejai (Pseudokodas 5).

Pseudokodas 5: Funkcija addGeneralUseCases

```
addGeneralUseCases = function(edges,useCases,functions){
  for(f in functions){
    useCasesUsingThisFunction = useCases.filter(useCase => (
      useCase.createdFromFlow.to == f
    ));
    if(useCasesUsingThisFunction.length < 2){
      continue;
    }
    generalUseCase = createGeneraluseCase(f);
    useCases.push(generalUseCase);
    for(useCase in useCasesUsingThisFunction){
      edge = createExtendEdge(generalUseCase,useCase);
      edges.push(edge);
    }
  }
}
```

7. Jeigu liko nepanaudotų funkcijų, iš jų sukuriami vartojimo atvejai su perspėjimais (Pseudokodas 6).

Pseudokodas 6: Funkcija addUnusedCases

```
addUnusedCases = function(useCases,functions){
  for(function in functions){
    if(useCases.includes(useCase =>(
      useCase.createdFromFlow.target == f
    ))){
      continue;
    }
    useCase = createUnusedUsecase(function);
    useCases.push(useCase);
  }
}
```

Sudėtingiausia algoritmo vieta yra transakcijos grafo viršūnių radimas (Pseudokodas 3), kuriam naudojamas paieškos į gylį algoritmas, taigi sudėtingumas yra tiesinis. Atlikusi šio algoritmo veiksmus programa iš Nr. 1 priede pavaizduoto vertės grandinės modelio gauna Nr. 2 priede pavaizduotas užduočių diagramas.

3. Programa BPMN transformacijai į užduočių diagramą

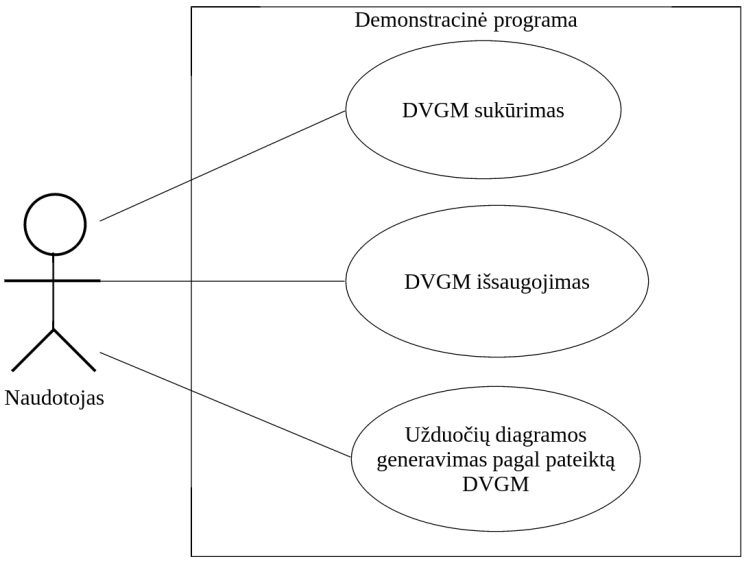
Vienas iš šio darbo tikslų yra programa, demonstruojanti algoritmo **BPMN** transformacijai į užduočių diagramą, veikimą. Programos įeigai pasirinktas **BPMN** išplėtimas **DVGM** modelis, išeiga – užduočių diagrama. Algoritmas aprašytas 2.1 skyriuje. Programos realizacija yra „github“ sistemoje. Ten galima rasti:

1. Programos kodą: https://github.com/Aleksandras-Sivkovas/diagrams-editor-app.
2. Sutransliuotą demo versiją: https://aleksandras-sivkovas.github.io/diagrams-editor-app/.
3. Pavyzdinį **JSON** formato duomenų failą, kurį galima importuoti programoje arba generuoti iš jo užduočių diagramas: https://aleksandras-sivkovas.github.io/diagrams-editor-app/dvcm.json.

Algoritmo aprašyto 2.1 skyriuje realizaciją „Javascript“ kalba pateikiama Nr. 10 priede.

3.1. Funkciniai reikalavimai programai

Čia aprašomi demonstracinės programos funkciniai reikalavimai. Jie pateikiami užduočių diagramos pavidalu (36 pav.) bei aprašomi 3, 4 ir 5 lentelėse. Šie reikalavimai parodys kokiomis funkcijos galės pasinaudoti naudotojas norėdamas pasižiūrėti algoritmo veikimą.



36 pav. Demonstracinės programos funkciniai reikalavimai.

3 lentelė. **DVGM** kūrimo naudojimo atvejis

ID	NA1
Pavadinimas	**DVGM** sukūrimas

Aktoriai

1. Naudotojas.

Aprašas

Naudotojas pasinaudojęs demonstracine programa sukuria **DVGM**.

Prieš sąlygos

1. Naudotojas yra atidaręs demonstracinės programos langą.

Priežastys

1. Naudotojas pareikalavo sukurti **DVGM**.

Po sąlygos

1. Naudotojas yra sukūręs **DVGM**.
2. Naudotojas mato savo sukurtą **DVGM** demonstracinės programos lange.

Pagrindinė užduočių seka

1. Demonstracinė programa pateikia interfeisą **DVGM** kūrimui.
2. Naudotojas kuria **DVGM**.

Alternatyvios užduočių sekos

Išimtinės užduočių sekos

4 lentelė. **DVGM** saugojimo naudojimo atvejis

ID	NA2
Pavadinimas	**DVGM** išsaugojimas

Aktoriai

1. Naudotojas.

Aprašas
Naudotojas išsaugo demonstracine programa sukurtą **DVGM**.

Prieš sąlygos

1. Naudotojas yra atidaręs **DVGM** kūrimo interfeisą.

Priežastys

1. Naudotojas pareikalavo išsaugoti **DVGM**.

Po sąlygos

1. Naudotojas yra išsaugojęs **DVGM**.
2. Naudotojas žino, kad **DVGM** išsaugotas.

Pagrindinė užduočių seka

1. Demonstracinė programa pateikia išsaugojimo formatus.
2. Naudotojas pasirenka kaip saugoti **DVGM**.
3. Demonstracinė programa pareikalauja išsaugojimo informacijos.
4. Naudotojas įveda išsaugojimo informaciją.
5. Demonstracinė programa išsaugo **DVGM**.
6. Demonstracinė programa patvirtina, kad **DVGM** išsaugotas.

Alternatyvios užduočių sekos

Išimtinės užduočių sekos

*.1. Naudotojas atsisako išsaugoti **DVGM**.

*.1.1. Naudotojas pateikia išsaugoti atsisakymą.

*.1.2. Demonstracinė programa nebereikalauja duomenų.

5 lentelė. Užduočių diagramos generavimo pagal pateiktą **DVGM** modelį naudojimo atvejis

ID NA3

Pavadinimas Užduočių diagramos generavimas pagal pateiktą **DVGM**

Aktoriai

1. Naudotojas.

Aprašas

Naudotojas demonstracinei programai programa pateikia **DVGM** ir programa sugeneruoja užduočių diagramą.

Prieš sąlygos

1. Naudotojas yra atidaręs programos langą.

Priežastys

1. Naudotojas pareikalavo sugeneruoti užduočių diagramą iš **DVGM**.

Po sąlygos

1. Demonstracinėje programoje yra užduočių diagramos duomenys.
2. Naudotojas mato sugeneruotą užduočių diagramą Demonstracinės programos lange.

Pagrindinė užduočių seka

1. Demonstracinė programa pateikia užduočių diagramos generavimo iš **DVGM** variantus.
2. Naudotojas pasirenka generuoti bendrą diagramą iš **DVGM**.
3. Demonstracinė programa pateikia interfeisą **DVGM** įvedimui.
4. Naudotojas įveda **DVGM**.
5. Demonstracinė programa sugeneruoja užduočių diagramos modelį.
6. Demonstracinė programa pavaizduoja užduočių diagramą lange.

Alternatyvios užduočių sekos

2.1. Naudotojas Naudotojas pasirenka iš **DVGM** sugeneruotą diagramą rodyti po vieną transakciją.

Išimtinės užduočių sekos

*.1. Naudotojas atsisako generavimo.

*.1.1. Naudotojas pateikia generavimo atsisakymą.

*.1.2. Demonstracinė programa nebereikalauja duomenų.

3.2. Technologijos pasirinktos programos įgyvendinimui

3.2.1. Javascript

Programai įgyvendinti pasirinkta „Javascript“ programavimo kalba. Kadangi reikės atvaizduoti diagramas buvo nuspręsta atlikti tai panaudojant HTML galimybes. „Javascript“ yra aukšto lygio programavimo kalba. Naujausios jos versijos yra objektiškai orientuotos ir pateikia nemažai kitų aukšto lygio abstrakcijų. Ši kalba yra silpnai tipizuota, objektų išplėtimams naudoja prototipus. „Javascript“ suteikia naršyklių rodomiems puslapiams dinamiškumą. Ilgą laiką ji buvo naudojama tik tam, bet kalbai išpopuliarėjus, ją imta taikyti ir srityse kaip serverinės ar net darbastalio programos. „Javascript“ naudojama kaip interpretuojama programavimo kalba. Interpretavimo standartas „ECMAScript“ buvo išleistas 1997 metais, nuo to laiko jis smarkiai pasikeitė. 2018 metais labiausiai naršyklių pilnai palaikomas yra 5 standartas, šiame darbe rašomas kodas bus transliuojamas į jį.

Programos „Javascript“ kodui rašyti pasirinkta 6 „ECMAScript“ versija [ECM15].

3.2.2. Node.js

Pasirinkta atviro kodo „Node.js“ programavimo aplinka [Dah09]. Ši aplinka pritaikyta dirbti su „Javascript“ projektais. Išoriniams paketams tvarkyti pasirinkta atviro kodo „NPM“ paketų tvarkyklė [RM10]. Ši tvarkyklė padeda tvarkyti „Javascript“ paketų versijas. Ji buvo pasirinkta nes yra pagrindinė „Node.js“ aplinkos paketų tvarkyklė.

3.2.3. Webpack

Programos modelių surinkimas ir transliavimas aprašomas naudojant atviro kodo „Javascript“ modulių surinkimo bibliotekos „Webpack“ konfiguraciją [KLE10]. Ji leidžia pasirinkti kokie moduliai ir kokiu būdu turi būti surinkti ir transliuoti. „Webpack“ naudojamas surinkti „Javascript“ projektą iš daugybės modulių ir transliuoti juos taip, kad palaikytų norimą versiją. Šiuo atveju „Javascript“ 6 standarto kodas transliavimas į 5, kad veiktų naršyklėse.

3.2.4. Babel

Programos kodas transliuojamas atviro kodo transliatoriumi „Babel“, kuris naudojamas kaip „Webpack“ prijungimas (plugin).

3.2.5. MobX

Programa modeliuojama naudojant MVC metodologiją. Modeliui ir kontroleriui pasirinkta atviro kodo „MobX“ modeliavimo biblioteka [Wes15]. Ji leidžia kurti programos duomenų modelius ir kontroliuoja jų būsenos klausymąsi naudodama „Javascript“ objekto duomenų priskyrimo ir gavimo metodus.

3.2.6. React

Programos „MVC“ vaizdavimo dalis kuriama naudojant atviro kodo „React“ biblioteką [Wal13]. Ji pateikia patogų virtualaus dokumento objekto modelio interfeisą kuris pritaikytas veikti vienodai visose naršyklėse ir optimizuoja dokumento objekto modelio atnaujinimus. Taip pat turi patogią sintaksę vaizdo apibrėžimui – „JSX“. „React“ labai gerai integruojasi su „MobX“.

3.3. Programos veikimo pavyzdžiai

Atidarius programą matomas trumpas jos aprašymas. Programą galima lokalizuoti į konsolę įvedus komandą `editorModel.locale = "lt_LT"` (priedas Nr. 6). Programa pateikia grafinę diagramų atvaizdavimo sąsają, bet jų kūrimui pateikiamas komandinis interfeisas. Komandomis vadinami „window“ objektui priskirti programos objektai (naudojimo pavyzdys: kodas 1). Paspaudus mygtuką „Naujas“ matomas diagramos kūrimo langas (priedas Nr. 7). Pasirinkus „**DVGM**“ rodomas **DVGM** kūrimo sąsaja (priedas Nr. 8). Joje galimos **DVGM** redagavimo komandos. Suvedus komandas iš 1 kodo gaunamas **DVGM** modelis. Šiek tiek pakoreguotas grafine sąsaja jis pavaizduotas priede Nr. 9. Sudėtingesnio **DVGM** (priedas Nr. 1) **JSON** formato duomenų failas yra pateiktas https://aleksandras-sivkovas.github.io/diagrams-editor-app/dvcm.json.

Kodas 1: Komandos gauti Nr. 9 priede pavaizduotą **DVGM**

```
1 // Nodes
2 pp = editorModel.diagram.root.processesPool
3 fp = editorModel.diagram.root.functionsPool;
4 start = new dvcm.StartEvent()
5 start.position.y = 20
6 start.position.x = 100
7 pp.append(start)
8 p1 = new dvcm.Activity("P1")
9 p1.position.y = 160
```

```
p1.position.x = 85
pp.append(p1);
f1 = new dvcm.Activity("F1")
f1.position.y = 160
f1.position.x = 85
fp.append(f1)
end = new dvcm.EndEvent()
end.position.y = 340
end.position.x = 100
pp.append(end)
tr = new dvcm.Transaction("MT1")
editorModel.diagram.addComponent(tr)


// Edges
sf = new dvcm.SequenceFlow()
sf.source = start
sf.target = p1
editorModel.diagram.addEdge(sf)

sf = new dvcm.SequenceFlow()
sf.source = p1
sf.target = f1
editorModel.diagram.addEdge(sf)

sf = new dvcm.SequenceFlow()
sf.source = f1
sf.target = p1
editorModel.diagram.addEdge(sf)

sf = new dvcm.SequenceFlow()
sf.source = p1
sf.target = end
editorModel.diagram.addEdge(sf)

sf = new dvcm.InTransaction()
sf.source = tr
sf.target = p1
editorModel.diagram.addEdge(sf)
```

```
49
50 sf = new dvcm.InTransaction()
51 sf.source = tr
52 sf.target = f1
53 editorModel.diagram.addEdge(sf)
```

Rezultatai ir išvados

1. Iš literatūros analizės apie reikalavimų inžineriją nustatytas reikalavimų inžinerijos aktualumas. Taip pat nustatytas įrankių reikalavimų inžinerijai aktualumas. Įrankis leistų:
   (a) Sumažinti žmogiškąjį klaidos faktorių. Automatizavus procesą žmogus gautų teoriškai teisingus reikalavimus.
   (b) Paspartinti reikalavimų inžinerijos procesą.
   (c) Padaryti reikalavimus prieinamus visiem komandos nariam. Visi žinotų koks yra projekto statusas.
   (d) Sekti reikalavimų pasikeitimus.
2. Pagal modeliavimo kalbų standartų ir transformavimo algoritmų analizę nustatyta, kad trūksta **BPMN** transformavimo į užduočių diagramą algoritmo. Rastas algoritmas, kuris atlieka transformaciją remdamasis empirinėmis taisyklėmis.
3. Kadangi daugelis **BPMN** komponentų neturi įtakos užduočių diagramos generavimo rezultatui išanalizavus **BPMN** standartą sudarytas **BPMN** pagrindinių komponentų metamodelis. Jis apibūdina algoritmo įvesties duomenis.
4. Išanalizavus **UML** ir **SysML** standartus sudarytas užduočių diagramos metamodelis. Jis apibūdina algoritmo rezultatą.
5. Pasiūlytas **BPMN** standarto praplėtimas. Parodyta kokios anksčiau aprašyto algoritmo problemos išsisprendžia generuojant užduočių diagramą pagal **BPMN** sukurtą pagal **DVGM** teoriją.
6. Išanalizavus modeliavimo kalbų standartus ir egzistuojančius algoritmus pasiūlytas **BPMN** transformavimo į užduočių diagramą algoritmas, kuris sukonstruoja užduočių diagramos modelį. Pateikiamas algoritmo pseudokodas bei jį apibūdinanti veiklos diagrama. Sukurtas algoritmas galėtų būti pritaikytas reikalavimų inžinerijos įrankyje.
7. Sukurta demonstracinė programa leidžianti:
   (a) Kurti **BPMN** pagal **DVGM** teoriją.
   (b) Iš sukurto modelio generuoti užduočių diagramą.

Literatūra

[ABS+17] Sehrish Alam, Shahid Nazir Bhatti, S. Asim Ali Shah, and Amr Mohsen Jadi. Impact and challenges of requirement engineering in agile methodologies: a systematic review. *International Journal of Advanced Computer Science and Applications*, 8:411–420, 2017-11.

[Dah09] Ryan Dahl. Node.js, 2009. URL: `https://nodejs.org`.

[DJ02] R.M. Dijkman and Stef M.M. Joosten. An algorithm to derive use cases from business processes. 08:679–684, 2002-03.

[ECM15] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 6 leid., 2015-06. URL: `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf`.

[GL16] Saulius Gudas ir Audrius Lopata. Towards internal modelling of the information systems application domain. *Informatica, Lith. Acad. Sci.*, 27(1):1–29, 2016. URL: `http://content.iospress.com/articles/informatica/inf1085`.

[ICJ+92] Jacobson Ivar, Magnus Christerson, Patrik Jonsson ir Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, 1992.

[JSB11] Ivar Jacobson, Ian Spence ir Kurt Bittner. *USE-CASE 2.0 The Guide to Succeeding with Use Cases*. Ivar Jacobson International SA., 2011. URL: `file:///C:/Users/bergerma/AppData/Local/Temp/Use-Case+2_0_Jan11.pdf`.

[KLE10] Tobias Koppers, Sean Larkin ir Johannes Ewald. Webpack, 2010-03. URL: `https://webpack.js.org/`.

[Obj11] Object Management Group (OMG). Business process model and notation (BPMN). OMG Document Number formal/2011-01-03 (`http://www.omg.org/spec/BPMN/2.0`), 2011. Version 2.0.

[Obj15] Object Management Group (OMG). Omg unified modeling language (OMG UML). OMG Document Number formal/2015-03-01 (`http://www.omg.org/spec/UML/2.5`), 2015. Version 2.5.

[OMG17] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.5, Object Management Group, 2017. URL: `https://www.omg.org/spec/SysML/1.5`.

[RM10] Isaac Z. Schlueter and Rebecca Turner ir Kat Marchán. Node package manager, 2010. URL: `https://www.npmjs.com/`.

[UHM17] Nabiha Usmani, Rabbia Hassan, and Waqas Mahmood. Impediments to requirement engineering in distributed team. *International Journal of Information Engineering and Electronic Business(IJIEEB)*, 9:10–18, 2017-11.

[Wal13] Jordan Walke. React, 2013. URL: https://reactjs.org/.

[Wes15] Michel Weststrate. Mobx, 2015. URL: https://github.com/mobxjs/mobx.

Santrumpos

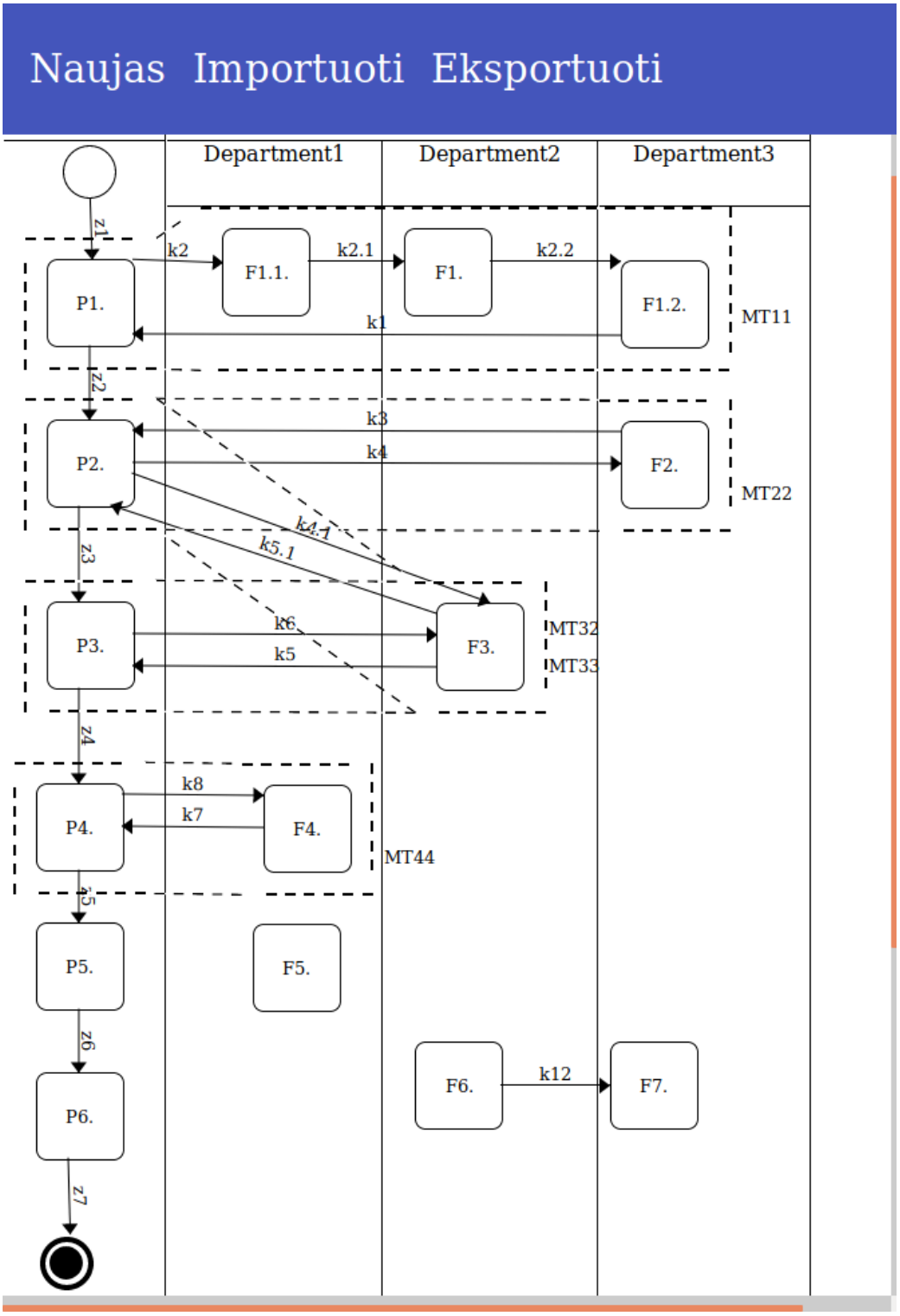
Šiame darbe naudojami žymėjimai:

1. **UML** – modeliavimo kalba, skirta suteikti standartinį sistemos analizės, architektūros, veikimo ir kūrimo pavaizdavimą [Obj15].
2. **SysML – UML** išplėtimas skirtas modeliuoti programų sistemas. [OMG17]
3. **OMG** (angl. Object Management Group) – atviras, tarptautinis ne pelno siekiantis technologijų standartų konsorciumas (https://www.omg.org/).
4. **JSON** (angl. JavaScript Object Notation) – atviras failo formato standartas naudojantis žmogui perskaitomą tekstą duomenims koduoti.

Priedas Nr. 1

Vertės grandinės modelis programos lange

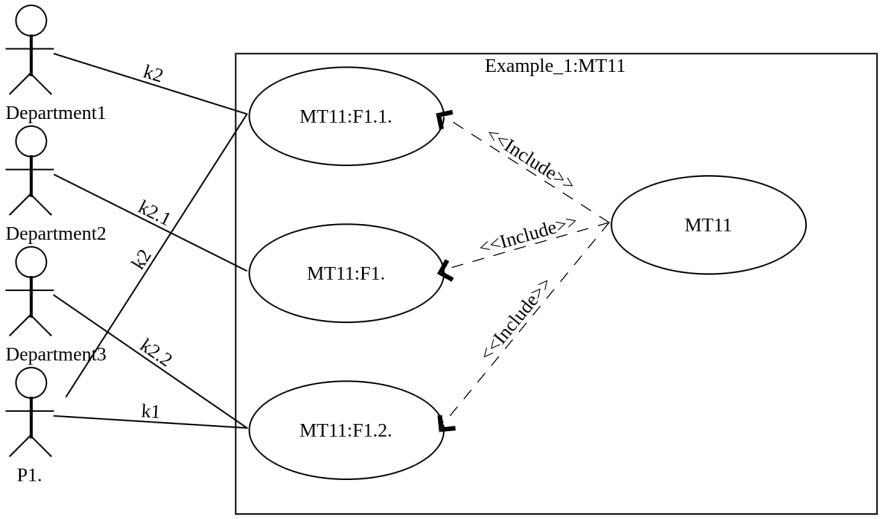
JSON formato duomenų failas yra pateiktas https://aleksandras-sivkovas.github.io/diagrams-editor-app/dvcm.json.

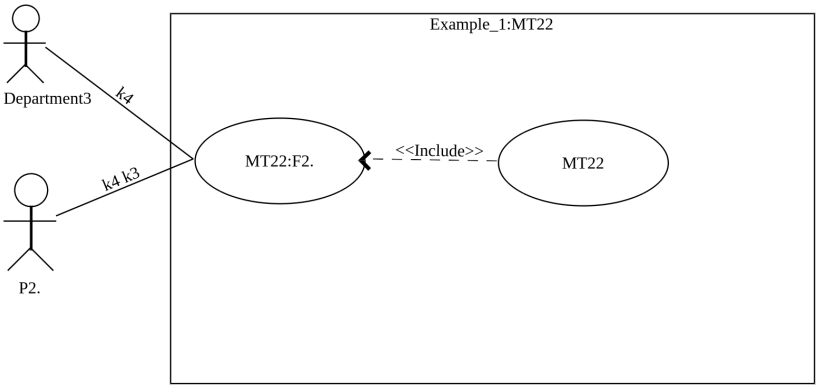


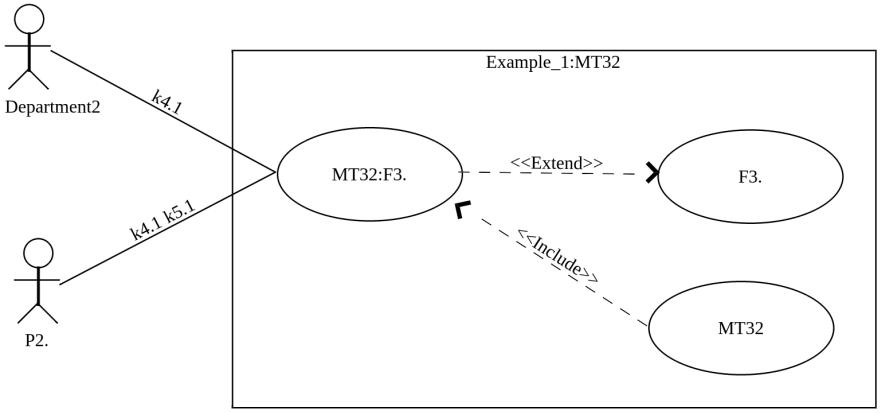
Priedas Nr. 2

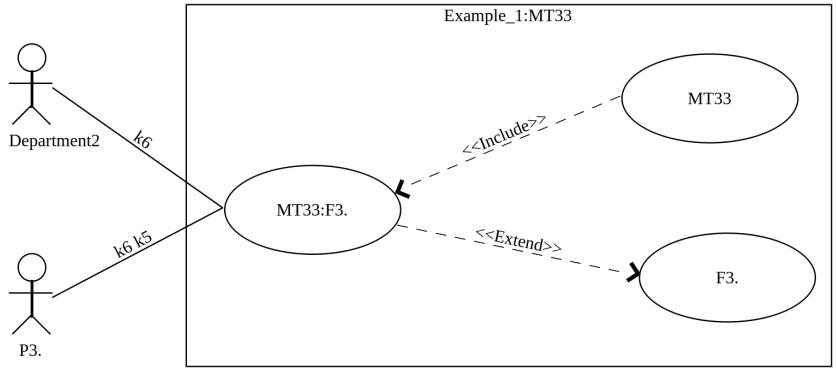
Nr. 1 priedo užduočių diagramos

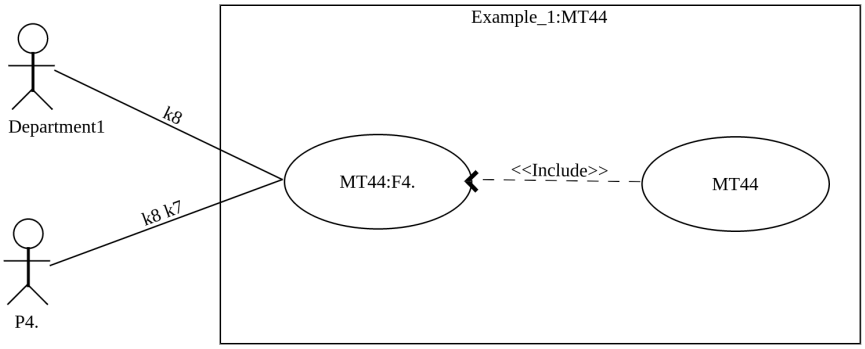
Iš Nr. 1 priedo **DVGM** transformuotos užduočių diagramos.

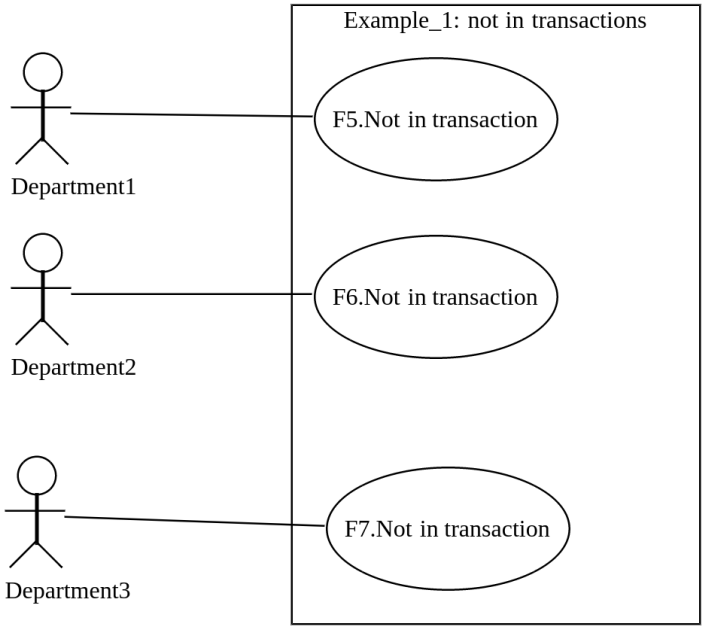







Example_1:MT33
Department2
k6
MT33
<<Include>>
MT33:F3.
k6 k5
<<Extend>>
F3.
P3.

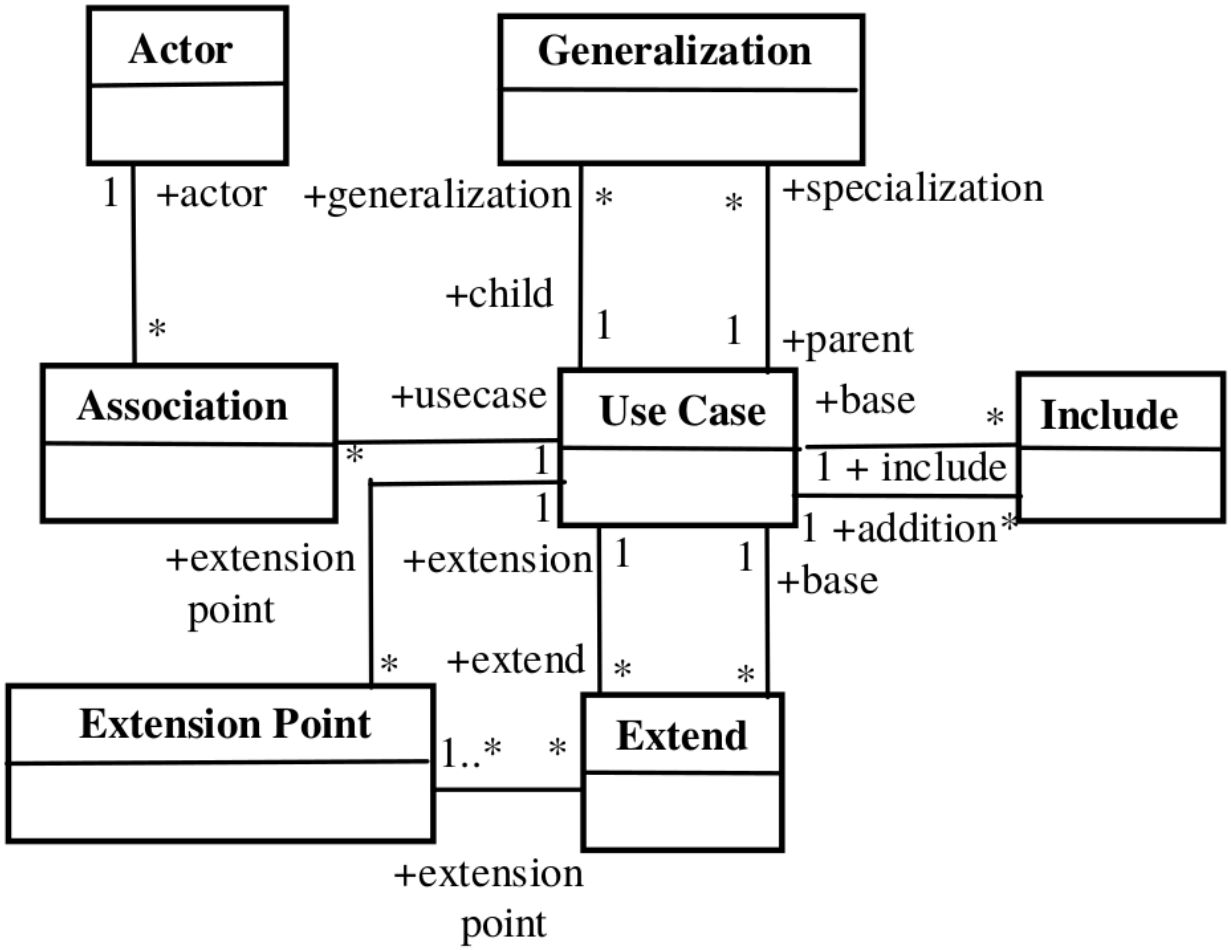

Example_1:MT44
Department1
k8
MT44:F4.
<<Include>>
MT44
k8 k7
P4.


Example_1: not in transactions
F5.Not in transaction
Department1
F6.Not in transaction
Department2
F7.Not in transaction
Department3

Priedas Nr. 3

Užduočių diagramos metamodelis

Čia pateikiamas literatūroje rasto algoritmo straipsnio [DJ02] pateiktas Užduočių diagramos (Use case) metamodelis.

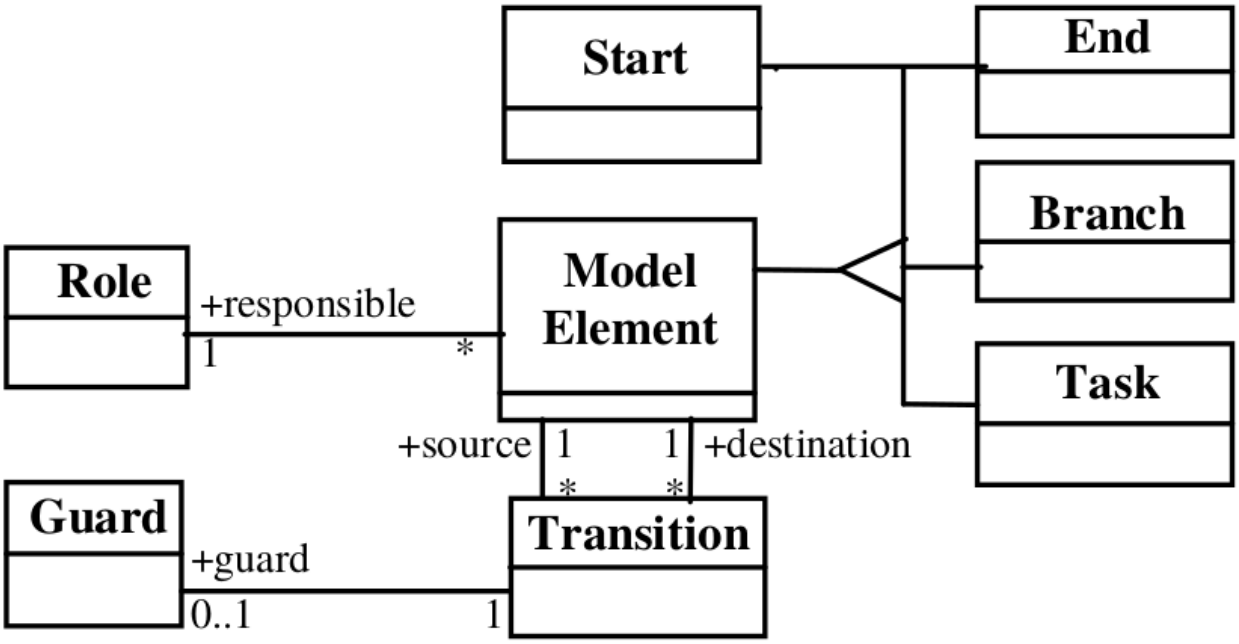


37 pav. Užduočių diagramos metamodelis

Priedas Nr. 4

BPMN metamodelis

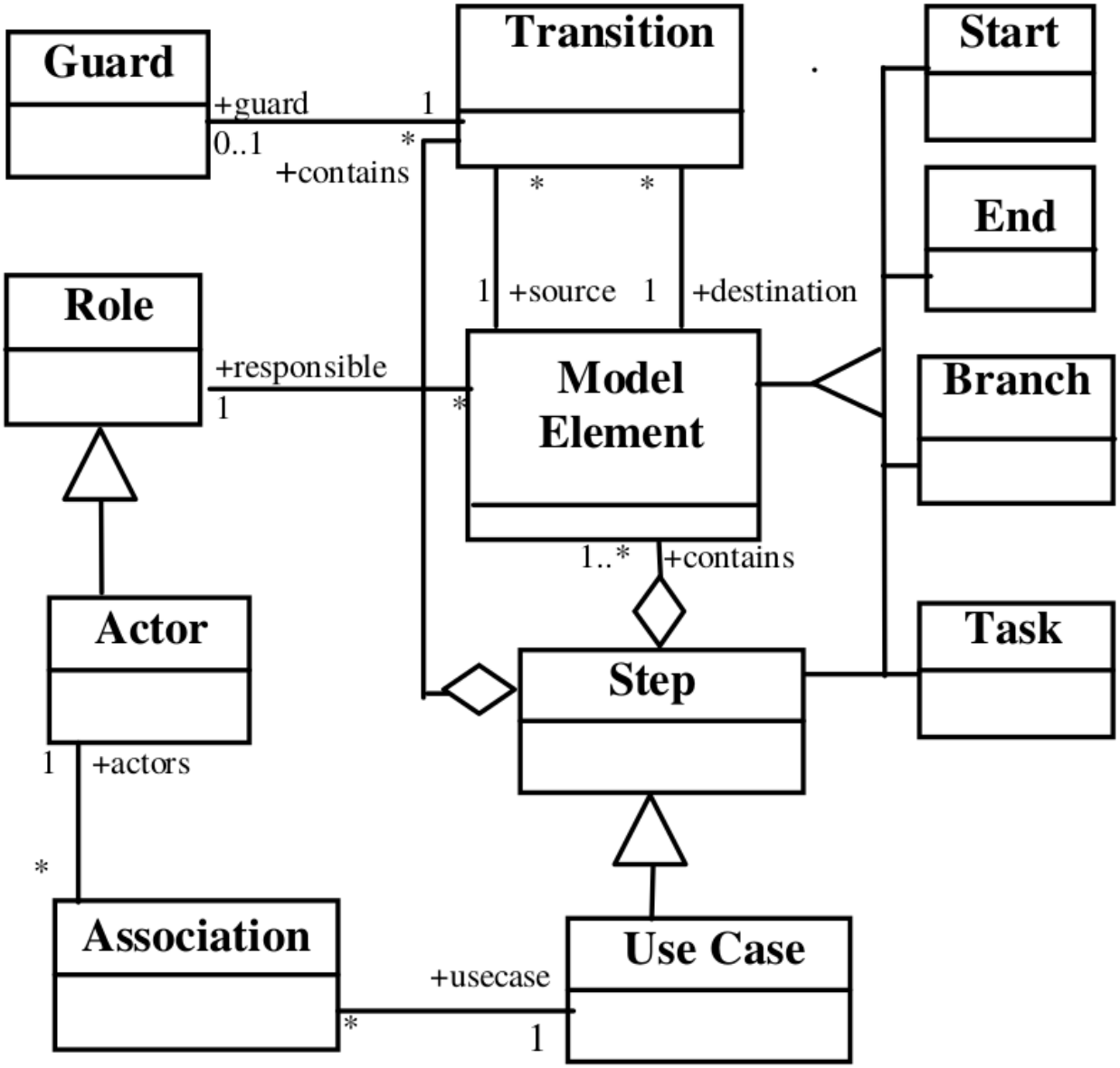
Čia pateikiamas literatūroje rasto algoritmo straipsnio [DJ02] pateiktas **BPMN** metamodelis.



Priedas Nr. 5

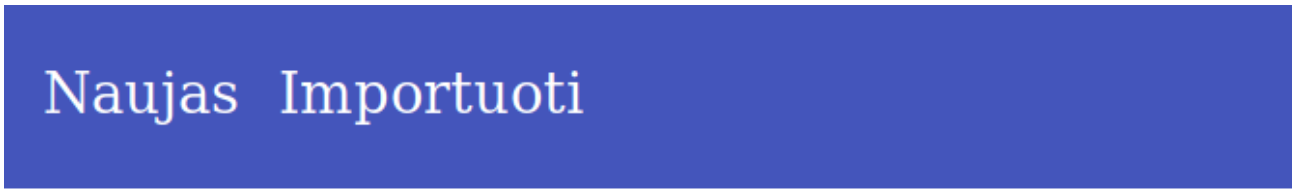
Inttegruotas metamodelis

Čia pateikiamas literatūroje rasto algoritmo straipsnio [DJ02] pateiktas integruotas **BPMN** ir užduočių diagramos (Use case) metamodelis.



Priedas Nr. 6

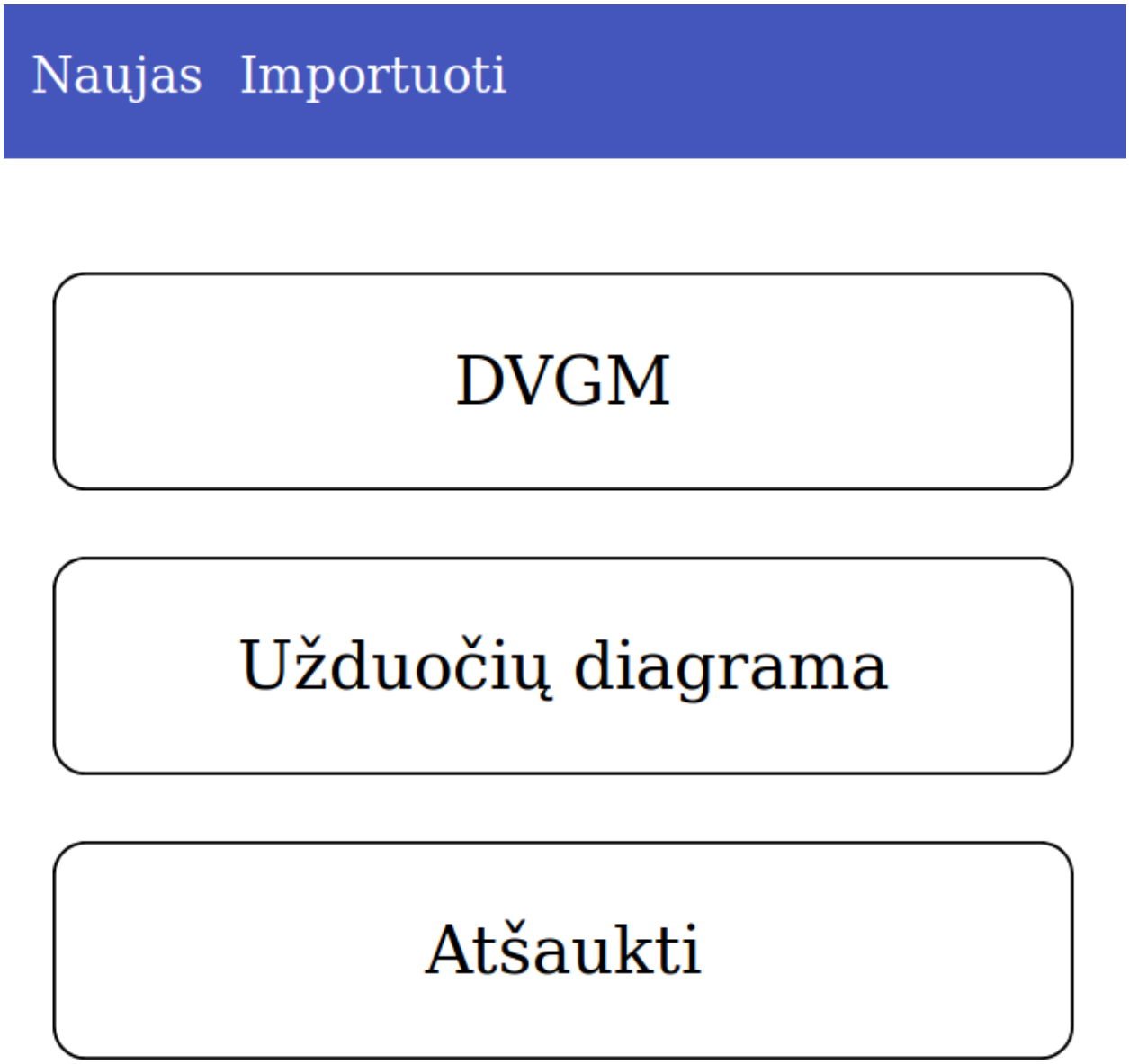
Langas matomas atidarius programą



Diagramų tvarkyklė 0.0.0
Grafinė sąsaja:
Importuoti - Importuoti diagramos failą ir parodyti diagramą..
Eksportuoti - Eksportuoti atidarytą diagramą į failą ir pasiūlyti jį išsaugoti failų sistemoje.
Naujas->DVGM -Sukuria naują DVGM ir parodo jį.
Naujas->Užduočių diagrama->Nauja diagrama -Sukuria naują užduočių diagramą ir parodo ją.
Naujas->Užduočių diagrama->Generuoti iš DVGM -Sukuria užduočių diagramą iš DVGM ir parodo ją.

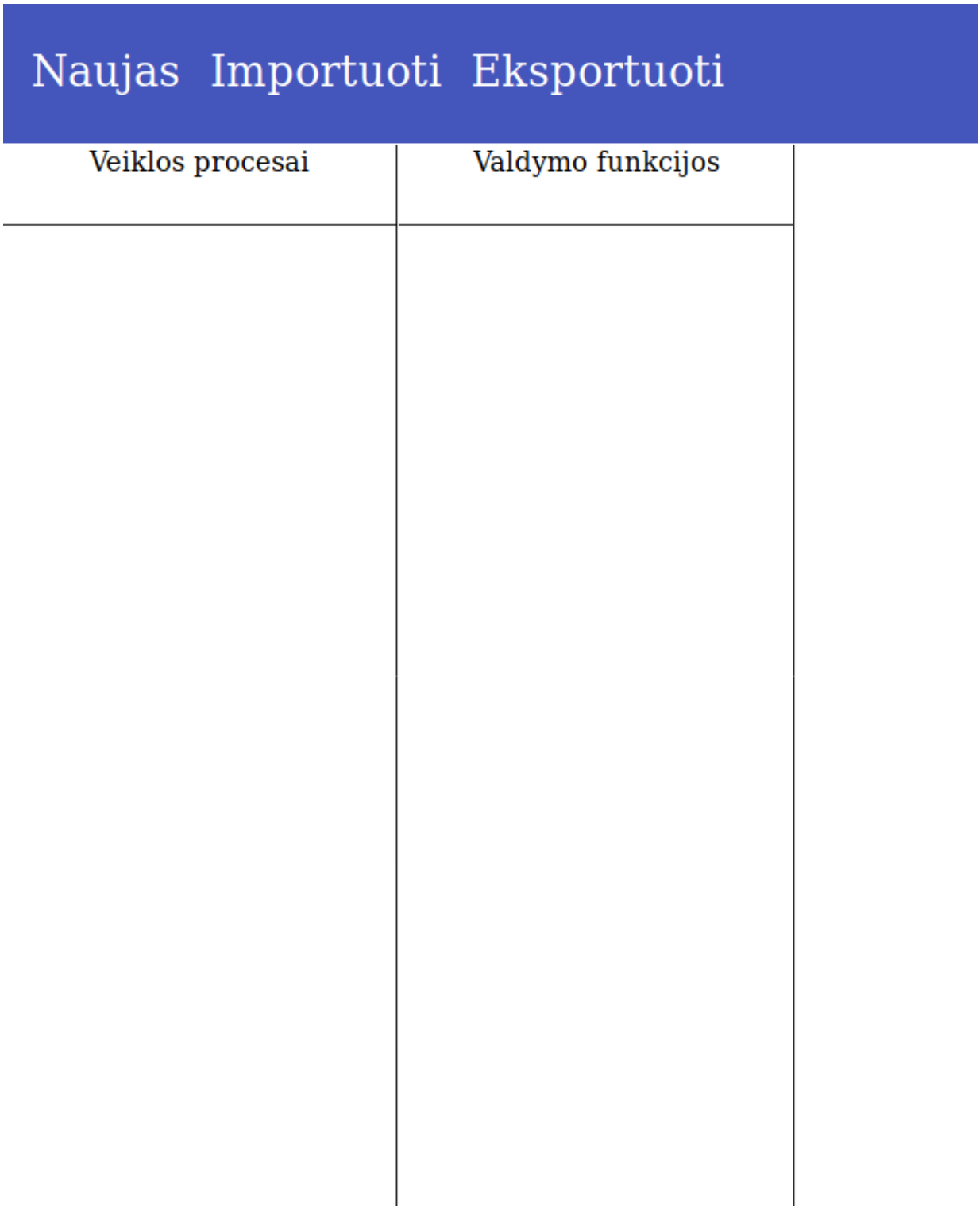
Priedas Nr. 7

Sukūrimo sąsaja



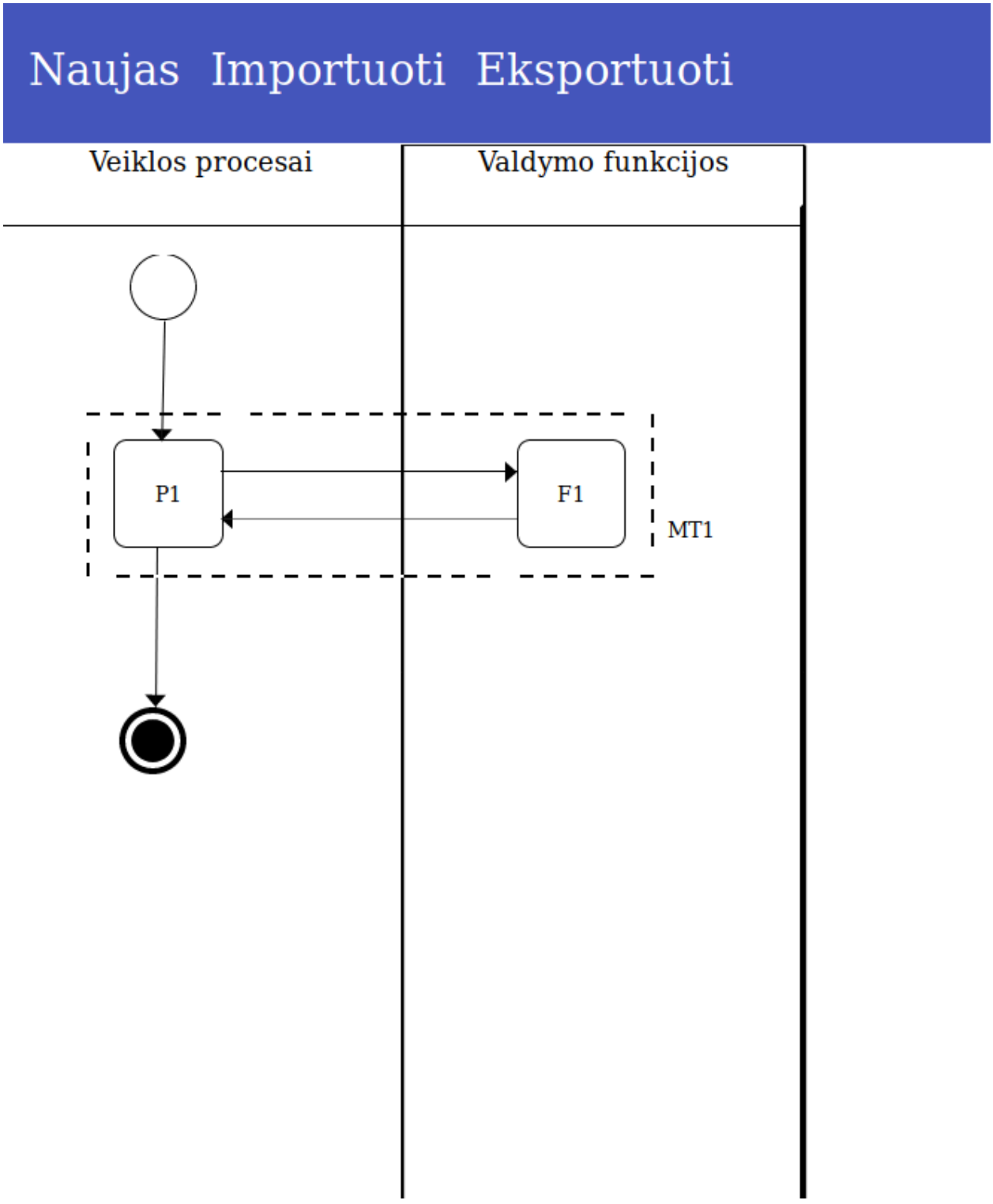
Priedas Nr. 8

DVGM Kūrimo sąsaja



Priedas Nr. 9

Sukurto DVGM pavyzdys



Priedas Nr. 10

Pseudokodo realizacija Javascript kalba

Algoritmo aprašyto 2.1 skyriuje realizaciją „Javascript“ kalba.

```
import {DVCMModel,Activity,SequenceFlow,Transaction} from "dvcm";
import {UseCasesModel,UseCases,UseCase,Association,Inclusion,
    Extension,System,Actor} from "use-cases"

export default class UseCasesFromDvcmDataGenerator {

  generateDVCMData(dvcm){
    const sequenceFlows = dvcm.sequenceFlows;
    const transactions = dvcm.transactions;

    const useCases = this.createUseCasesObject();
    const edges = [];
    const actors = new Map();
    const mappings = {
      useCasesFunctionsmap: new Map(),
      useCasesTransactionsMap: new Map()
    }

    for(let transaction of transactions){
      if(transaction.processes.length < 1){
        continue;
      }
      const process = transaction.processes[0];
      const transactionSequenceFlows = this.
          _getTransactionSequenceFlows(transaction,sequenceFlows);
      const cycle = this.
          _getCycleForProcess(process,transactionSequenceFlows);
      if(!cycle){
        continue;
      }
      this._addUseCases(edges,useCases,actors,cycle,transaction,mappings);
    }

    // getFunctions
    const functions = dvcm.activities.filter(activity =>
```

```
      activity.isFunction);

    this._addGeneralUseCases(edges,useCases,functions,mappings);
    this._addUnusedCases(edges,useCases,functions,actors,mappings);

    return {
      useCases:useCases,
      edges:edges,
      actors:actors,
      mappings:mappings,
    }
  }
  createUseCasesObject(){
    return {
      transactionuseCases:[],
      functionUseCases:[],
      subfunctionUseCases:[],
      generalUseCases:[],
      unusedUseCases:[]
    };
  }
  _getTransactionSequenceFlows(transaction,sequenceFlows){
    const activities = transaction.activities;
    return sequenceFlows.filter(flow => (
      (activities.includes(flow.source)) &&
         (activities.includes(flow.target))
    ));
  }
  _getCycleForProcess(process,sequenceFlows){
    const processFlows = sequenceFlows.filter(flow => (flow.source ==
       process));
    if(processFlows.length == 0){
      return null;
    }
    const processFlow = processFlows[0];
    const cycledFlows = [];
    this._addCycledFlows(cycledFlows,[],processFlow,process,sequenceFlows);
    const starts = cycledFlows.filter(flow => (flow.source == process));
    if(starts.length == 0){
```

```
72        return null;
73      }
74      const ends = cycledFlows.filter(flow => (flow.target == process));
75      if(ends.length == 0){
76        return null;
77      }
78      return {
79        start: starts[0],
80        ends:  ends,
81        cycledFlows: cycledFlows,
82        process: process
83      };
84    }
85    _addUseCases(edges,useCases,actors,cycle,transaction,mappings){
86      const mainUseCase = new UseCase(transaction.name);
87      useCases.transactionuseCases.push(mainUseCase);
88
89      let useCaseTransactions = new Set();
90      mappings.useCasesTransactionsMap.set(mainUseCase,useCaseTransactions);
91      useCaseTransactions.add(transaction);
92
93      const mainPrefix = transaction.name + ":";
94      const fuMap = new Map();
95      for(let flow of cycle.cycledFlows){
96        if(flow.target == cycle.process){
97          continue;
98        }
99        const name = mainPrefix + flow.target.name;
100       const useCase = new UseCase(name);
101       useCases.subfunctionUseCases.push(useCase);
102       const include = new Inclusion();
103       include.source = mainUseCase;
104       include.target = useCase;
105       edges.push(include);
106
107       const actorName = flow.target.parent.name;
108       let actor = actors.get(actorName);
109       if(!actor){
110         actor = new Actor(actorName);
```

```
111         actors.set(actorName,actor);
112       }
113
114       const association = new Association();
115       edges.push(association);
116       association.source = actor;
117       association.target = useCase;
118       let associationName = flow.name;
119       association.name = associationName;
120
121       fuMap.set(flow.target,useCase);
122
123       let functionUseCases = mappings.useCasesFunctionsmap.get(flow.target);
124       if(!functionUseCases){
125         functionUseCases = [];
126         mappings.useCasesFunctionsmap.set(flow.target,functionUseCases);
127       }
128       functionUseCases.push(useCase);
129
130       let useCaseTransactions = new Set();
131       mappings.useCasesTransactionsMap.set(useCase,useCaseTransactions);
132       useCaseTransactions.add(transaction);
133     }
134     this._addProcessAsActor(edges,actors,cycle,mainUseCase,fuMap);
135
136   }
137   _addProcessAsActor(edges,actors,cycle,useCase,fuMap){
138     const actorName = cycle.start.source.name;
139     let actor = actors.get(actorName);
140     if(!actor){
141       actor = new Actor(actorName);
142       actors.set(actorName,actor);
143     }
144
145     const nameMap = new Map();
146     for(let end of cycle.ends){
147       const fu = fuMap.get(end.source);
148       let name = nameMap.get(fu);
149       if(!name){
```

```
150         name = "";
151       }
152       nameMap.set(fu,name + " " + end.name);
153     }
154
155     let fu = fuMap.get(cycle.start.target);
156     let name = nameMap.get(fu);
157     if(!name){
158       name = "";
159     }
160     nameMap.set(fu,cycle.start.name + " " + name);
161
162     for(fu of nameMap.keys()){
163       const association = new Association();
164       edges.push(association);
165       association.source = actor;
166       association.target = fu;
167       association.name = nameMap.get(fu);
168     }
169   }
170   _addCycledFlows = function(cycledFlows,currentPath,
171       newFlow,process,sequenceFlows){
172     if(currentPath.includes(newFlow)){
173       return false;
174     }
175     if(cycledFlows.includes(newFlow)){
176       return true; // We already found path this way
177     }
178     if(newFlow.target == process){
179       cycledFlows.push(newFlow);
180       return true;
181     }
182     const nextFlows = sequenceFlows.filter(flow => (
183       flow.source == newFlow.target
184     ));
185     if(nextFlows.length == 0){
186       return false;
187     }
188     cycledFlows.push(newFlow);
```

```
189     currentPath.push(newFlow);
190     let pathFound = false;
191     for(let flow of nextFlows){
192       if(this._addCycledFlows(cycledFlows,currentPath,flow,
193           process,sequenceFlows)){
194         pathFound = true;
195         // No brake because we need to recursivelly add all paths
196       }
197     }
198     if(!pathFound){
199       // Remove newFlow
200       cycledFlows.splice(cycledFlows.indexOf(newFlow),1);
201     }
202     // Remove newFlow
203     currentPath.splice(currentPath.indexOf(newFlow),1);
204     return pathFound;
205   }
206   _addGeneralUseCases(edges,useCases,functions,mappings){
207     for(let f of functions){
208       const useCasesUsingThisFunction =
              mappings.useCasesFunctionsmap.get(f);
209       if(!useCasesUsingThisFunction || (useCasesUsingThisFunction.length <
              2)){
210         continue;
211       }
212       const generalUseCase = new UseCase(f.name);
213       useCases.generalUseCases.push(generalUseCase);
214       for(let useCase of useCasesUsingThisFunction){
215         const extend = new Extension();
216         extend.source = useCase;
217         extend.target = generalUseCase;
218         edges.push(extend);
219
220         let useCaseTransactions =
              mappings.useCasesTransactionsMap.get(useCase);
221         if(useCaseTransactions){
222           let newUseCaseTransactions = mappings.useCasesTransactionsMap
223               .get(generalUseCase);
224           if(!newUseCaseTransactions){
```

```
225             newUseCaseTransactions = new Set();
226             mappings.useCasesTransactionsMap.set(generalUseCase,
227                 newUseCaseTransactions);
228           }
229           for(let transaction of useCaseTransactions){
230             newUseCaseTransactions.add(transaction);
231           }
232         }
233       }
234     }
235   }
236   _addUnusedCases(edges,useCases,functions,actors,mappings){
237     for(let f of functions){
238       if(mappings.useCasesFunctionsmap.get(f)){
239         continue;
240       }
241       const useCase = new UseCase(f.name + "Not in transaction");
242       useCase.errors = true;
243       useCases.unusedUseCases.push(useCase);
244
245       const actorName = f.parent.name;
246       let actor = actors.get(actorName);
247       if(!actor){
248         actor = new Actor(actorName);
249         actors.set(actorName,actor);
250       }
251       const association = new Association();
252       edges.push(association);
253       association.source = actor;
254       association.target = useCase;
255     }
256   }
257 };
```