KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
KOMPIUTERIŲ KATEDRA

Maksim Kartašov

Veiklos diagramų modeliavimo sistemos sudarymas ir tyrimas

Magistro darbas

Darbo vadovas
prof. E. Kazanavičius

Kaunas, 2006

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

KOMPIUTERIŲ KATEDRA

Maksim Kartašov

Veiklos diagramų modeliavimo sistemos sudarymas ir tyrimas

Magistro darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
dr. I. Mickienė
2006-05

Vadovas
prof. E. Kazanavičius
2006-05

Recenzentas
doc. P. Kanapeckas
2006-05

Atliko
IFM0/1 gr. stud.
Maksim Kartašov
2006-05

Kaunas, 2006

TURINYS

SUMMARY ... 5
1. ĮVADAS ... 6
2. SISTEMOS ANALIZĖ ... 8
2.1 Sistemos UML kalbos veiklų diagramos projektavimo galimybės ... 8
2.2 Petri tinklų galimybės analizė ... 11
2.3 Petri tinklų elementai ir veikimas ... 13
2.4 Formalus modeliavimo sistemos spalvotų Petri tinklų aprašymas ... 14
2.5 Petri tinklo savybės ... 15
2.6 Petri tinklo analizė ... 16
2.7 Sistemos Petri tinklų imitatorių prototipų analizė ... 17
2.8 Veiklos diagramų pertvarkymo į Petri tinklus taisyklių analizė ... 21
3. VEIKLOS DIAGRAMOS PERTVARKYMAS Į PETRI TINKLĄ: PERTVARKYMO TAISKYKLĖS IR ELEMENTŲ PERKĖLIMAS ... 23
3.1 Veiklos diagramos pertvarkymo fazės ... 23
3.2 Petri tinklų posistemės aprašymas ... 23
3.3 Veiklos diagramos elementų perkėlimo aprašymas ... 25
3.3.1 Veiksmo būsena ... 25
3.3.2 Poveiksmio būsena ... 27
3.3.3 Iškvietimo būsena ... 29
3.3.4 Sprendimai ... 31
3.3.5 Susiliejimai ... 31
3.3.6 Lygiagretumo palaikymo elementai ... 32
3.3.7 Pradinės ir galinės būsenos ... 33
3.4 Veiklos diagramų pertvarkymo į Petri tinklus metodika ... 33
3.4.1 Veiklos diagramos pritaikymo koncepcija Petri tinkluose ... 34
3.4.2 Transformacijos iš veiklos diagramų į Petri tinklus taisyklės ... 35
3.4.2.1 Veiksmo mazgai ... 35
3.4.2.2 Kontrolės mazgai ... 35
3.4.2.3 Objekto mazgai ... 37
3.4.2.4 Veiklos pakraščiai ... 37
3.4.2.5 Veiklų suskirstymas ... 39
3.4.3 Veiklų diagramų supaprastinimas prieš darant pasikeitimą ... 39
3.4.3.1 Veiklos mazgas prieš išsišakojimo mazgą ... 39
3.4.3.2 Iššakojimo mazgas prieš iššakojimo mazgą ... 40
3.4.3.3 Sujungimo mazgas prieš iššakojimo mazgą ... 41
3.4.3.4 Sujungimo mazgas prieš sujungimo mazgą ... 42
3.4.3.5 Sprendimo mazgas prieš sprendimo mazgą ... 43
3.4.3.6 Susiliejimo mazgas prieš sprendimo mazgą ... 44
3.4.3.7 Susiliejimo mazgas prieš susiliejimo mazgą ... 45
3.4.4 Pertvarkymo pavyzdžiai ... 46
3.4.5 Nežymimi elementai ... 50
3.4.6 Santrauka ... 50

4. VEIKLOS DIAGRAMŲ MODELIAVIMO SISTEMOS SUDARYMAS IR TYRIMAS 51
4.1 Modeliavimo sistemos sudarymas 51
4.2 Modeliavimo sistemos tyrimas 60
5. IŠVADOS 68
5.1 Bendos išvados 68
5.2 Numatoma plėtra 69
6. LITERATŪROS SĄRAŠAS 70
7. TERMINŲ IR SANTRUMPŲ ŽODYNAS 71
8. PRIEDAI 72
8.1 Programos tekstas 72

Paveikslų sąrašas

2.1 pav. Rational Rose 9
2.2 pav. Petri tinklų panaudojimas modeliavimui ir analizei 12
2.3 pav. The Petri Net Kernel 18
2.4 pav. Petri tinklas Centaurus programoje 20
2.5 pav. Petri tinklas Simula 1.0 programoje 21
3.1 pav. Veiklos diagramų bendras perėjimo metodas 26
3.2 pav. ŽASPT veiksmas, poveikla ir iškvietimo būsenos 29
3.3 pav. ŽASPT sprendimas (prieš pasikeitimus) 31
3.4 pav. ŽASPT iššakojimas, sujungimas, suliejimas, galinė būsena 32
3.5 pav. Veiklos mazgas prieš iššakojimo mazgą. 40
3.6 pav. Iššakojimo mazgas prieš iššakojimo mazgą 41
3.7 pav. Sujungimo mazgas prieš iššakojimo mazgą 42
3.8 pav. Sujungimo mazgas prieš sujungimo mazgą 43
3.9 pav. Sprendimo mazgas prieš sprendimo mazgą 44
3.10 pav. Susiliejimo mazgas prieš sprendimo mazgą 45
3.11 pav. Susiliejimo mazgas prieš susiliejimo mazgą 46
3.12 pav. Užsakymo apdorojimo veiklos diagrama 47
3.13 pav. Veiklos diagrama po užsakymo apdorojimo supaprastinimo 47
3.14 pav. Petri tinklas po užsakymo apdorojimo pertvarkymo 48
3.15 pav. Veiklos diagrama probleminiam bilietui 48
3.16 pav. Veiklos diagrama po probleminio bilieto supaprastinimo 49
3.17 pav. Petri tinklas po probleminio bilieto pertvarkymo 49
4.1 pav. Veiklos diagrama Magic Draw UML redaktoriuje 52
4.2 pav. Modeliavimo sistemos formos vaizdas 54
4.3 pav. UML veiklos diagramos vaizdas modeliavimo sistemoje 56
4.4 pav. Petri tinklo vaizdas modeliavimo programoje 57
4.5 pav. Petri tinklo modeliavimo tarpiniai rezultatai 59
4.6 pav. Modeliavimo sistemos klasių diagrama 60
4.7 pav. Gėrimo išrinkimo proceso veiklos diagrama Magic Draw UML redaktoriuje 62
4.8 pav. Gėrimo išrinkimo proceso veiklos diagrama modeliavimo sistemoje 63
4.9 pav. Gėrimo pasirinkimo proceso Petri tinklas 64
4.10 pav. Gėrimo pasirinkimo proceso simuliavimas 65
4.11 pav. Gėrimo pasirinkimo proceso modeliavimo pabaiga 66
4.12 pav. Petri tinklo visų šakų modeliavimas 66
4.13 pav. Petri tinklo modeliavimo procesas 67

ANALYSIS AND DESIGN OF MODELING SYSTEM FOR ACTIVITY DIAGRAMS

SUMMARY

Making electronic business systems, products of information technologies or solving real time tasks more and more IT specialists their designing work connect to UML language. The usage of this language provides opportunities to arrange documentation more simply, allows to communicate different position employees more easily, guarantee IT systems compatibility with business needs and requirements. One of the diagrams types used is activity diagram, which could be used to describe business processes, firms or its unit, compose official, activity or users instructions, show systems behaviour.

Any investigative, designed or management activity at some cases is connected to modelling. Commonly designing systems, where parallel events could take place or, if necessary, to give an opportunity gradually, taking in account the reality, show events or state proceeding with the help of Petri nets. It is both graphical and mathematical design method.

Designing activity diagrams of IT systems UML editors are used. To research and show events proceeding Petri nets are used. So the problem area is to compose a design system, which could transfer the elements of activity diagrams into Petri nets and model the functionality of the project. The major purpose of this study paper is to compose UML language activity diagrams modelling system, examine the functionality and design opportunities of system's activity diagrams into Petri nets.

1. ĮVADAS

Kuriant elektroninio verslo sistemas, informacinių technologijų produktus arba sprendžiant realaus laiko uždavinius, vis daugiau IT specialistų savo projektavimo darbą sieja su UML kalba. Modeliavimo ir specifikacijų kūrimo kalba, skirta specifikuoti, atvaizduoti ir konstruoti į objektus orientuotų programų dokumentus. UML kalbos naudojimas suteikia galimybių paprasčiau tvarkyti dokumentaciją, leidžia lengviau susikalbėti skirtingų pareigybių žmonėms, laiduoja paprastesnį IT sistemų suderinimą su verslo poreikiais. UML apibrėžia 9 rūšių diagramas, kurios leidžia specifikuoti įvairius architektūros aspektus. Viena iš naudojamų diagramų tipų yra veiklos diagrama, kurią galima naudoti verslo procesų, firmos arba padalinio atliekamų funkcijų aprašymui, pareigybinėms, veiklos, vartotojų instrukcijoms. Ji skirta vaizduoti elgseną sistemos viduje, nors gali būti naudojama modeliuoti visos organizacijos veiklą. Veiklos diagramose specifikuojami reikalavimai atsižvelgiant į sistemos tikslus bei sistemos teikiamas paslaugas. Mažinama IT projektų biudžeto viršijimo rizika. Tačiau projektuotojas, naudodamasis veiklos diagramomis, kad aprašytų procesus struktūriškai, neturi galimybės realiame laike modeliuoti sistemos veikimą, ištirti projekto elgseną, surasti veikimo aklavietes.

Bet kuri tiriamoji, projektinė ar vadybinė veikla tam tikrais atvejais siejama su modeliavimu. Detalės brėžinys, įrenginio projektas, lėktuvo ar pastato maketas, valdymo sistemos, apibūdinančios technologinį procesą arba kinematinio mechanizmo judėjimą – visa tai yra gamybos, statybos ir valdymo objektų modeliai.

Objektų, tikslų ir uždavinių įvairumas sąlygoja įvairių modelių tipų aibę. Modelis – tai matematiniais terminais pristatymas to, kas yra būdinga tiriamajam objektui arba sistemai. Manoma, kad juo manipuliuojant galima gauti naujų žinių apie modeliuojamą reiškinį, ir taip išvengti pavojų, aukštos kainos ar nepatogumų, palyginus, jei būtų manipuliuojama pačiu realiu reiškiniu.

Dažniausiai modeliuojant sistemas, kuriose vienu metu gali vykti keletas įvykių, arba, esant poreikiui, sudaryti galimybę tolygiai, atsižvelgiant į tikrovę, parodyti įvykių ir būsenų eigą, naudojami Petri tinklai. Tai grafinis bei matematinis modeliavimo būdas, kuris gali būti naudojamas pačių įvairiausių tipų sistemose, perspektyvus multiprograminių, asinchroninių,

skirstomųjų, lygiagrečių, nedeterminuotų ir/arba stochastinių informacijos apdorojimo sistemų aprašymo ir tyrimo instrumentas.

Kaip *grafinis* būdas Petri tinklai panašūs į blokines schemas, struktūrines schemas ir tinklinius grafikus, gali būti naudojami modeliuojamai sistemai vaizdžiai įsivaizduoti. Šiuose tinkluose įvesta žymeklių samprata leidžia modeliuoti sistemų funkcionavimo dinamiką ir lygiagrečius procesus.

Kaip *matematinis* būdas analitinis Petri tinklų įsivaizdavimas leidžia sudaryti būsenos lygybes, algebrines lygybes ir kitus sistemų dinamiką aprašančius matematinius santykius. Petri tinklais gali sėkmingai pasinaudoti ir teoretikai, ir praktikai. Tai padaroma efektyvios tarpusavio komunikacijos būdu: praktikai gali perimti iš teoretikų tobulesnę modelių sudarymo metodologiją, o teoretikai – pasimokyti iš praktikų, kaip priartinti savo modelius prie realybės.

Projektuojant IT sistemų veiklos diagramas naudojami redaktoriai (Rational Rose, Magic Draw UML), o tyrinėti arba parodyti įvykių eigą taikomi Petri tinklai, taigi probleminė sritis šiuo atveju yra sudaryti modeliavimo sistemą, kuri veiklos diagramos elementus pervestų į Petri tinklus ir modeliuotų projekto funkcionalumą. Taigi pagrindinis magistro darbo tikslas – sudaryti UML kalbos veiklos diagramų modeliavimo sistemą, ištirti šios sistemos veiklos diagramos pertvarkymo į Petri tinklus funkcionalumą bei modeliavimo galimybes. Remiantis svarbiausiu magistro darbo tikslu, galima išvardinti pagrindinius uždavinius:

- pasirinkti ir išnagrinėti kuriančią modeliavimo sistemą;
- išanalizuoti Petri tinklų modeliavimo galimybes;
- sukurti veiklos diagramos pertvarkymo algoritmą į Petri tinklą;
- suprogramuoti modeliavimo sistemos grafinę sąsają;
- ištirti modeliavimo sistemos funkcionalumą naudojantis pavyzdžiui;

2. SISTEMOS ANALIZĖ

2.1 Sistemos UML kalbos veiklų diagramos projektavimo galimybės

Pramoninis programinių sprendimų projektavimas šiuolaikinėmis sąlygomis jaučia spaudimą dėl nuolat didėjančių patikimumo, spartos ir produktyvumo reikalavimų projektavimo procesui. Programiniai sprendimai greitai evoliucionuoja: nuo didelių ir sudėtingų iki ypač didelių ir ypač sudėtingų. Spartus informacinių technologijų srities panaudojimo ir technologinės įrangos plėtimasis nustato vis griežtesnius reikalavimus projektavimo proceso organizavimui, kuris vis dažniau yra kolektyvinis. Projektavimo proceso organizavimo užduotyje galima išskirti tokias pagrindines dalis:

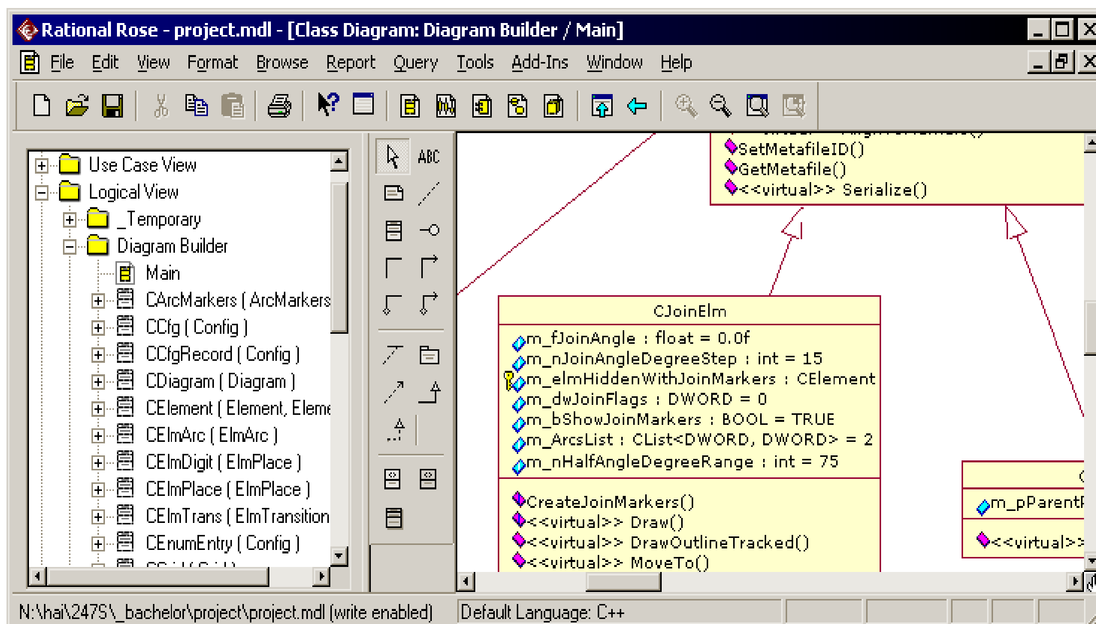
- saugus programinių komponentų ir projektinių sprendimų, kuriamų darbo metu, saugojimas;
- pakartotinis programinių komponentų ir projektinių sprendimų, sukurtų konkrečiame projekte arba gautų iš išorės, panaudojimas;
- projektuotojų naudojamų instrumentų integracija, unifikacija ir tarpusavio sąveika;
- kuriamos sistemos versijų valdymas ir t.t.

Šių užduočių sprendimas reikalauja rimto techninio palaikymo. Neužtenka tik pateikti medžiagos elektroninę versiją ir užtikrinti priėjimą prie jos – šiuo atveju labai didelis rutininio darbo krūvis atitenka programuotojams, ir projektavimo procesas tampa nepatikimas, nevaldomas ir nenuspėjamas. Viena vertus, technologinis aprūpinimas turi palaikyti visą spektrą būtinų operacijų, antra vertus, turi būti pakankamai intelektualus, kad įgalintų tipinių procedūrų automatizaciją, be to, būti tarpusavyje suderintas ir plečiamas.

Apgalvotas pasirinkimas ir technologinių instrumentinių projektavimo naudojimas leidžia pereiti nuo priimtų viduramžių manufaktūrose darbo organizavimo metodų prie šiuolaikinės lanksčios automatizuotos gamybos.

Sudėtingos programinės sistemos modelio projektavimas, prieš jį realizuojant, yra neatskiriama viso projekto dalis, kaip ir brėžinys yra naujo pastato statybos pagrindas. Geras modelis yra savotiškas sklandaus kūrėjų bendradarbiavimo pagrindas ir garantuoja bendrą projekto sėkmę. Modelio kūrimas neišvengiamas todėl, kad iš pirmo žvilgsnio neįmanoma aprėpti ne tik visos sistemos bendrai, bet ir jos atskirų funkcinių dalių.

Dėl projektuojamų sistemų sudėtingumo atsiranda vis didesnis būtinumas turėti gerą modeliavimo priemonę. Egzistuoja dideli faktorių kiekiai, kurie turi įtakos bendrai projektavimo sėkmei, bet griežtas programavimo kalbos standartas lieka svarbiausias iš jų. Tai paaiškina didelį susidomėjimą pramoniniu objektu, orientuotu modeliavimo kalbos standartu, kuris yra unifikuota modeliavimo kalba – UML [1].



2.1 pav. Rational Rose

Rational Rose (2.1. pav.) – informacinių sistemų vizualinio projektavimo CASE-priemonė, kuri leidžia modeliuoti ir programinės įrangos komponentus, ir verslo procesus. Rational Rose palaiko įvairias objektų orientuotas metodologijas: OMT, UML kalbą, Buco notaciją. Rose leidžia pagal sudarytus modelius automatiškai generuoti programinį kodą ir, atvirkščiai, konstruoti iš pradinių tekstų grafinius objektus ir modelius. Šį produktą naudoti gali ne tik kūrėjai-programuotojai, bei ir, pavyzdžiui, verslo analitikai ir konsultantai. Iš esmės Rational Rose – tai atskirų CASE-priemonių šeima, orientuota į skirtingas programavimo kalbas arba projektavimo priemones [2]. Rational Rose tankiai integruotas su MS Visual Studio.

Šis produktas skirtas analitikams, verslo analitikams, kūrėjams, kompanijoms, kurios užsiima verslo procesų modeliavimu ir programinės įrangos projektavimu.

Toliau paminėti kai kurie Rational Rose privalumai:

- tai pagrindinė į objektus orientuota projektavimo ir analizės priemonė;

- palaiko UML kalbą;
- palaiko C++, Visual C++, Visual Basic ir Java kalbų kodo generavimą;
- užtikrina komandinį projekto projektavimą.

Dar viena projektavimo priemonė iš daugelio kitų yra UML Studio. Vienas iš didžiausių ir pagrindinių privalumų yra didelis generuojamų kalbų pasirinkimas. Palaikoma pagrindinių techninių dalių programavimo VHDL kalbos generavimo galimybė. Šios programos aprašymas sutampa su Rational Rose paketo aprašymu.

Kitas labai populiarus UML redaktorius yra Magic Draw UML, kuris atitinka naujausius Java bei UML technologijų standartus, turi vieną iš patikimiausių išeities kodų inžinerijos mechanizmų Java, C#, C++ ir CORBA IDL programavimo kalboms bei gali vykdyti šių kalbų kodo atvirkštinę inžineriją, duomenų bazių schemų atvirkštinę inžineriją, kodo bei duomenų bazių schemų generavimą. MagicDraw UML panaudoja "roundtrip" technologiją, leidžiančią keisti tiek OO modelį, tiek programos kodą bet kokiu eiliškumu, juos nuolat sinchronizuojant. MagicDraw UML yra vienas iš nedaugelio rinkoje esančių paketų, leidžiančių braižyti visas devynias UML diagramas bei turintis UML diagramų semantinio teisingumo tikrinimo mechanizmą. MagicDraw UML Teamwork Server programinė įranga turi komandinio darbo galimybę, leidžiančią daugeliui programinės įrangos kūrimo inžinierių dirbti su tuo pačiu OO modeliu vienu metu. Labai didelis šios sistemos privalumas yra duomenų išsaugojimas XML formatu, tai suteikia labai didelę galimybę duomenų apsikeitimui. Tai pat palaikoma UML 2.0 notacija, visiškai sudarytas UML 1.4 metamodelis. Galima patiems susikurti programinius plėtinius (plugins) ir išplėsti MagicDraw UML funkcionalumą per atvirąją programinę sąsają (Open API). Yra numatyta galimybė apsikeisti diagramomis per Unisys XMI išplėtimą su kitomis UML priemonėmis (IBM Rational Rose, Borland Together ControlCenter). Šio produkto kūrime dalyvauja Lietuvos UAB „Baltijos Programinė Įranga“.
Taigi magistro darbe sudarant veiklos diagramos modeliavimo sistemą bus naudojamas Magic Draw UML redaktorius, kuris gali veiklos diagramas išsaugoti XML formatu, kad duomenys būtų toliau apdorojami, tvarkomi.

2.2 Petri tinklų galimybės analizė

1962 metais pasirodė pirmasis darbas apie Petri tinklus. Jo autorius K. A. Petri aprašė naują informacinių srautų sistemos modelį. Modelis buvo sudarytas remiantis prielaida, kad atskiros sistemos dalys funkcionuoja asinchroniškai ir konkuruoja tarpusavyje. Ryšiai tarp atskirų dalių buvo vaizduojami grafu arba tinklu. Dažniausiai Petri tinklai naudojami modeliuojant sistemas, kuriose vienu metu gali vykti keletas įvykių, ir yra proceso apribojimų dažnumui, sekai. Įvykiams yra suteikiami prioriteto įvertinimai. Tai labai patogus būdas informaciniams procesams aprašyti, kai valdymo sistemose vyksta konfliktinės situacijos, lygiagretūs atsitiktiniai ir nedeterminuoti procesai.

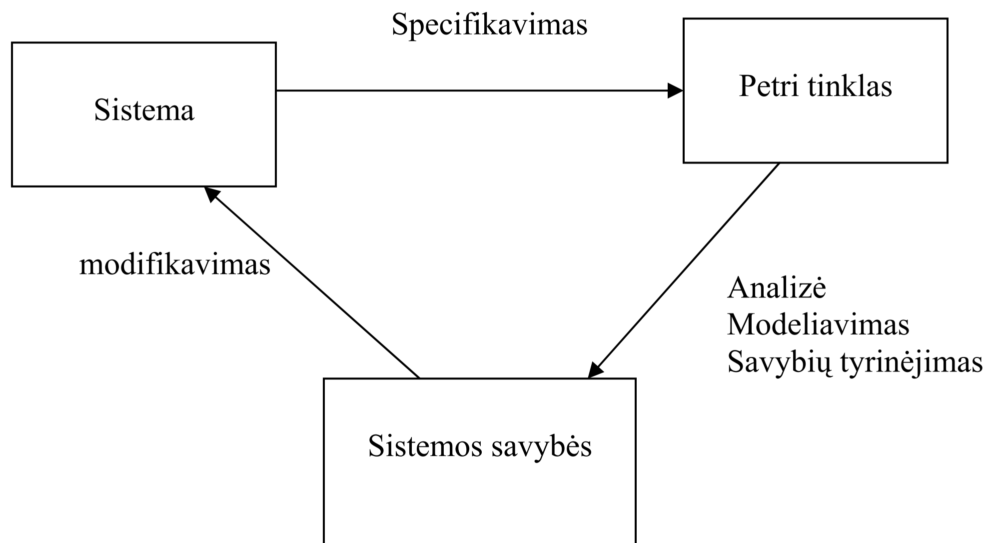
Jau dabar Petri tinklai įgavo didelį populiarumą visame pasaulyje, sudomino daugybę žmonių savo tyrimais ir praktiniu naudojimu [12]. Pavyzdžiui, informacinių technologijų srityje Petri tinklai naudojami kompiuterių techninės ir programinės įrangos modeliavimui. Techninę įrangą galima nagrinėti įvairiais lygiais, ir Petri tinklai gali modeliuoti kiekvieną iš šių lygių:

- pirmame lygyje kompiuteriai sudaryti iš paprastų atminties įrenginių ir tranzistorių;
- aukštesniame lygyje kaip pagrindiniai komponentai naudojami funkciniai blokai ir registrai;
- dar aukštesniame lygyje ištisos skaičiavimo sistemos tampa kompiuterinių tinklų komponentais.

Petri tinklų savybė – galimybė modeliuoti kiekvieną iš minėtų lygių. Žinoma, Petri tinklų naudojimas neapsiriboja vien informacinių technologijų sritimi [5]. Šiuo atveju ryškios ribos faktiškai neegzistuoja. Galima užsiimti GPS modeliavimu karinėse Pentagono operacijoje ir kitur. Tai įrodo galimybę visapusiškai naudoti tinklus ir šio modeliavimo būdo universalumą.

Yra keletas projektavimo ir sistemų analizės būdų, kurie atliekami pasitelkus įvairių programų kompleksus. Petri tinklai vertinami kaip pagalbinė analizės priemonė. Čia sistemos modeliavimui naudojami visuotinai priimti projektavimo metodai. Toliau suprojektuota sistema modeliuojama Petri tinklais ir modelis analizuojamas. Bet kurie analizės metu išsiaiškinti sunkumai atskleidžia projekto defektus. Norint juos pataisyti privaloma modifikuoti projektą. Modifikuotas projektas toliau vėl modeliuojamas ir analizuojamas. Šis ciklas kartojamas tol, kol atliekama analizė nesibaigia sėkmingai. Tai iliustruoja 2.3 pav. Verta

pažymėti, kad juo galima pasinaudoti, analizuojant jau egzistuojančias ir dabartiniu metu veikiančias sistemas.



2.2 pav. Petri tinklų panaudojimas modeliavimui ir analizei

Iš pradžių sistema modeliuojama pasitelkus Petri tinklus, toliau modelis analizuojamas. Analizės rezultatas – sistemos įvertinimas, kuris lemia sistemos tobulinimą.

Kitas būdas pasireiškia tuo, kad visas projektavimo ir charakteristikų nustatymo procesas vyksta Petri tinklų terminuose. Analizės metodai naudojami tik laisvam nuo klaidų Petri tinklų projektui sudaryti. Pagrindinis uždavinys – Petri tinklų pertvarkymas į realią dirbančią sistemą [6].

Bendra projekto užduotis – Petri tinklų sudarymo automatizacija, kuri, pasitelkus kompiuterį, laiduoja jų išsaugojimą, darbą realiu laiku, redagavimą bet kuriame projektavimo etape.

Žemiau pateikiama automatizuoto proceso pagrindinių elementų struktūra:

- redaktorius;
- klasikinio Petri tinklų elementų papildymas;
- diagramos turinio manipuliacijos;
- mastų panaudojimas;

- papildomos galimybės;
- apkrautų Petri tinklų funkcionavimas.

Petri tinklų taikymo sritys:

- paskirstytos duomenų bazės;
- lygiagretus programavimas;
- lanksti gamyba;
- diskrečių įvykiu sistemos;
- daugiaprocesorinės sistemos;
- didelių duomenų srauto apdorojimas;
- kompiliatoriai ir operacines sistemos;
- loginis programavimas;
- neuroniniai tinklai;
- formaliosios kalbos;
- asinchronines grandinės ir struktūros.

2.3 Petri tinklų elementai ir veikimas

Paprastuose Petri tinkluose atsisakoma laiko įvedimo, t. y. būsenos kitimo laike [7]. Tai pakeičiama priežasties – pasekmes ryšiais tarp įvykių. Jei reikia įvesti laiką, tai momentai arba laiko intervalai išreiškiami kaip įvykiai. Grafą sudaro dviejų tipų viršunės:
1) vietų p (apskritimai);
2) perėjimų t (atkarpos).

Viršūnės tarpusavyje yra sujungtos rodyklėmis. Jei rodyklė nukreipta iš viršūnės i į viršūnę j (iš vietos p į perėjimą t arba iš perėjimo t į vieta p), tai i yra j įėjimas, o j yra i išėjimas. Tipiniai vietų ir perėjimų interpretavimo atvejai:

Įėjimo vietos	Perėjimai	Išėjimo vietos
Priežastys Įėjimo duomenys	Įvykis Skaičiavimai	Pasekmė Išėjimo duomenys

Įėjimo signalas	Signalinis procesorius	Išėjimo signalas
Resursų poreikis	Darbas, operacija	Resursų išlaisvinimas
Sąlygos	Loginis sakinys	Išvada
Buferis	Procesorius	Buferis

Dinaminių sistemų modeliavimui naudojami žymėtieji Petri tinklai. Sakysime, kad Petri tinklų viršūnės modeliuoja programos vykdymo žingsnius (arba sąlygas), o žyme (juodu tašku) pažymimos vietos *p*, kurios atitinka tuo metu vykdomą programos žingsnį (arba patenkintą sąlygą). Vykdant programą, žymės juda tinkle iš vienos viršūnės į kitą. Yra tokios žymių perkėlimo iš vienos vietos p_i į kita vieta p_j taisyklės :

1) Žymės perkeliamos į kitą vietą , atsidarius atitinkamiems perėjimams.
2) Perėjimas atsidaro tada, kai visos duoto perėjimo vietos yra pažymėtos.
3) Atsidarius perėjimui, žymės perkeliamos į visas duoto perėjimo išėjimo vietas.
4) Vienu laiko momentu atsidaro ir įvykdomas tik vienas perėjimas. Perėjimo atsidarymo ir vykdymo trukmė lygi nuliui ir neužima laiko.

2.4 Formalus modeliavimo sistemos spalvotų Petri tinklų aprašymas

Pagrindiniai Petri tinklų teorijos teiginiai yra šie:

1. Petri tinklai susideda iš dviejų pagrindinių komponenčių: vietų aibės *P* ir perėjimų aibės *T*.
2. Ryšį tarp vietų ir perėjimų nusako dvi funkcijos: įėjimo funkcija *I* ir išėjimo funkcija *O*.
3. Įėjimo funkcija *I* kiekvienam perėjimui *tj* nusako aibę įėjimo vietų *I(tj).*
4. Išėjimo funkcija *O* kiekvienam perėjimui *tj* nusako aibę išėjimo vietų *O(tj).*

Formaliai Petri tinklas aprašomas ketvertu : $C = (P, T , I, O)$.

Petri tinklo markiruotė (M) yra atliekama patalpinant tinklo viršūnėse (vietose) žymes. Žymių skaičius ir išdėstymas kinta modeliavimo metu. Įvesime markiravimo funkciją

$M: P\text{->}N,$

čia: P- vietų aibė, N- natūrinių skaičių aibė.

Todėl žymėjimas *m(pi)* reiškia vietoje *pi* esančių žymių skaičių. Trumpai *m(pi) =mi*. Vektorius *m =(m1, m2, ... ,mn)* aprašo Petri tinklų žymių aibę. Formaliai pažymėti Petri tinklai aprašomi taip :

$M = (P, T, I, O, m)$.

Kadangi žymių skaičius vienoje tinklo viršūnėje yra neribojamas, tai egzistuoja neribotas Petri tinklų markiruočių skaičius.

2.5 Petri tinklo savybės

Pirma iš savybių yra susijusi su ribota atminties talpa realiomis sąlygomis realizuojant įvykius. Dirbant Petri tinklams, kai kurios tinklo vietos gali sukaupti neribotą žymių skaičių. Jeigu vieta interpretuojama kaip kaupiklis (duomenų, signalų ar kitokios informacijos buferis), tai reikia pareikalauti, kad bet kokiu sistemos funkcionavimo atveju neįvyktų kaupiklių, kurie realiomis sąlygomis turi realią talpą, perpildymas.

Vieta p Petri tinkluose $M = (P, T, I, O, m)$ vadinama ribota, jei egzistuoja toks skaičius n, kad esant bet kokiam atvejui žymių skaičius vietoje $m(p)=< n$.

Tinklas vadinamas ribotu, jei visos vietos jame yra ribotos. Vieta p vadinama nepavojinga, jei joje gali būti ne daugiau kaip viena žymė, t.y. 0 arba 1-na. Atitinkamai tinklas vadinamas nepavojingu, jei kiekviena jo vieta nepavojinga.

Konservuoti tinklai – tai tinklai, kuriuose žymių suma yra visą laiką pastovi, t.y. kiekvieno perėjimo suveikimas nepakeičia žymių skaičiaus tinkle.

Perėjimas tinkle gali pradėti veikti esant tam tikroms sąlygoms, atsiradusioms dėl išsidėstymo jo įėjimo vietose. Gali taip atsitikti, kad kuriame nors perėjime jo veikimo sąlyga negali būti išpildyta, nes neveiktų tinklas. Toks perėjimas vadinamas mirusiu. Jis yra nereikalingas tinkle, ji galima išmesti iš tinklo be žalos.

Taipogi gali atsitikti taip, kad po tam tikro perėjimo veikimo tinkle, kuris labai pakeičia išsidėstymą, kai kurie iš perėjimų suveikė, niekada daugiau neveiks, nepaisant kokie būtų išsidėstymo variantai. Tokie perėjimai vadinami potencialiai mirusiais. Priešingos sąvokos – gyvi ir potencialiai gyvi perėjimai. Gyvu perėjimu vadinamas toks, kuris turi galimybę pradėti veikti esant bet kokiam išsidėstymui tinkle. Potencialiai gyvas perėjimas yra toks, kuris turi galimybę pradėti veikti esant tam tikram žymių išsidėstymui tinkle. Vadinasi, jei visi perėjimai tinkle gyvi – tinklas vadinamas gyvu, jei visi mirę - mirusiu. Tinklas vadinamas potencialiai gyvu, jei jame yra potencialiai gyvų perėjimų, o potencialiai mirusių, jei yra potencialiai mirusių vietų.

2.6 Petri tinklo analizė

Yra žinomi trys Petri tinklų analizės metodai:

1) padengiantis medis;

2) matricinės lygtys;

3) redukcijos ir dekompozicijos technika.

Padengiantis medis - tai paskaičiavimas visu galimu markiravimu, tačiau tai įmanoma tik nedidelės apimties tinkluose.

Kiti du metodai yra „galingesni“, tačiau jie gali būti naudojami tik specifinių situacijų atveju.

Lygiagretūs veiksmai:

Pagal virsmo sužadinimo taisyklę suprantama, kad kiekviena padėtis gali priimti (sutalpinti) neribotą kiekį identiškų ženklų (požymių). Toks Petri tinklas būtų begalinės talpos. Tačiau, modeliuojant realias fizikines sistemas, galima kalbėti tik apie baigtinės talpos tinklą. Tokiame tinkle *(N, M0)* kiekvienai padėčiai *p* priskiriama talpa *K(p)* , t.y. maksimalus požymių (signalu) skaičius, kuris įėjime gali būti priimtas bet kuriuo laiko momentu. Be to, virsmui *t*, kuris gali būti sužadintas, galioja sąlyga, kad po griūties išėjime negali būti viršyta jo talpa *K(p).* Ši taisyklė vadinama griežta virsmo taisykle, o be tokio apribojimo - silpna virsmo taisykle. Duotam baigtinės talpos tinklui *(N, M0)* galima taikyti griežtą arba silpną virsmo taisyklę, prieš tai atlikus papildomą padėčių transformaciją, esant prielaidai priimant, kad tinklas *N* yra taisyklingas.

1. Įvesti papildoma padėtį *p'* kiekvienam *p*, kur *p'* pradinis markiravimas nustatomas: *M0'(p') = K(p) - M0(p)*

2. Tarp kiekvieno virsmo ir bet kurios papildomos padėties brėžiamas lankas *(t, p')* arba *(p', t)*, kur *W(t, p') = W(p, t)* ir *W(p', t) = W(t, p),* taip kad signalų suma padėtyje būtų lygi talpai *K(p)* prieš ir po virsmo griūties.

2.7 Sistemos Petri tinklų imitatorių prototipų analizė

Petri tinklų modeliavimo problema šiuo metu labai aktuali. Dėl to projektuojamas didelis programinių produktų (PP) kiekis, susijusių su įvairiais Petri tinklų panaudojimo aspektais. Kompiuterinėje rinkoje komercinių ir laisvai platinamų šios srities programų santykis apytiksliai lygus. Jų kiekis pakankamai didelis ir pastoviai augantis.

Kokybinių nemokamų Petri tinklų programų Lietuvos Interneto resursuose rasti praktiškai neįmanoma. Šia problema rimtai domisi atskiros žmonių grupės. Bet Europoje ir Amerikoje galima rasti ištisus tokių programų, kurias galima laisvai parsisiųsti, katalogus. Prieš pradedant projektuoti šią sistemą, buvo atlikta kelių užsienio programinių produktų analizė:

▪ *Petri NetWork 2.0* – programa suprojektuota Rusijoje, Valstybiniame naftos technologiniame universitete. Programa parašyta Delphi kalba ir leidžia sukonstruoti Petri tinklą ekrane ir su peles atlikti tinklo sintezę.

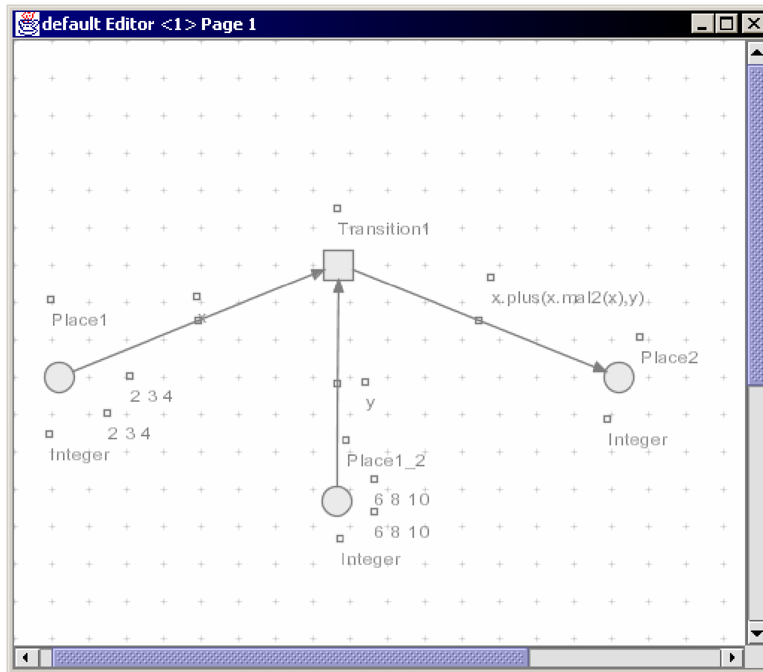
Privalumai: galimybė analizuoti platų suprojektuoto Petri tinklo savybių spektrą.

Trūkumai: nepatogus redaktorius, nėra galimybių paleisti modeliavimo procesą.

▪ *The Petri Net Kernel (PNK) version 2.0 (2.3 pav.)* – tai infrastruktūra Petri tinklų utilitų sudarymui. Suprojektuota Petri Net Kernel Team, Humboldt-University Berlyne, Vokietijoje. Parašyta Java kalba, 1999-2000 m.

Privalumai: originalus redaktorius, patogi bazė rimtiems projektams.

Trūkumai: nėra konkrečios naudos paprastiems vartotojams.



2.3 pav. The Petri Net Kernel

- HPSim – suprojektuota Henryk Anschuetz. Parašyta Visual C++ kalba, palaiko klasikinius Petri tinklus yra modeliavimo su laikinu parametru galimybė.

Privalumai: kokybiškas šios srities produktas, gerai apgalvotas, galingas redaktorius yra tinklo projektavimo animacija.

Šiuolaikinės valdymo sistemos vis labiau kompiuterizuojama, t.y. vis svaresnis tampa valdymo algoritmas. Netgi modernūs elektronikos prietaisai ar įrenginiai turi mikrokompiuterius, kurie valdo prietaisų darbą, t.y. tokie įrenginiai yra valdymo sistemos [8]. Valdymo kompiuterių, valdiklių, programų, algoritmai tampa vis sudėtingesni, ir jų patikimumo, stabilumo, patikrinimas tampa sudėtingu uždaviniu. Čia turima galvoje ne tik programavimo, algoritmo realizavimo klaidų aptikimas, bet ir funkcionavimo logikos sutikimas su valdymo tikslais. Ji nėra fiksuota, nes bendrais atvejais realias sistemas veikia atsitiktiniai trikdžiai, ir valdymo sistemos reakcijos pobūdis dažnai priklauso nuo trikdžių ar įvykio sistemoje pobūdžio. Bene dažniausiai pasitaikantis uždavinys, kurį tenka spręsti lanksčią valdymo logiką turinčioms sistemoms, – pasirinkimas iš kelių galimybių, siekiant optimalaus rezultato. Optimalus rezultatas gali būti ne vietinio ar momentinio pobūdžio, o strateginis, kurį kiekybiškai įvertinti galima tik praėjus gana ilgam sistemos darbo laiko intervalui. Todėl kuriant tokias sistemas būtinas modeliavimo etapas, kurio metu ir sukuriamas

valdymo algoritmas, tenkinantis keliamus reikalavimus. Valdiklių emuliatoriai netinka šiam tikslui, nes jie imituoja jau parengtos programos veikimą ir algoritmo abstraktesnė analizė čia sudėtinga. Minėtam tikslui geriau tiktų modeliavimo sistema, veikianti su abstrakčiu valdymo algoritmu, nesusietu su konkrečiu programos kodu. Be to, tokia valdymo sistema turi gebėti modeliuoti ir valdymo objektą bei jo ryšius su valdikliu. Kita vertus, modeliavimo sistemos gautas rezultatas, algoritmo struktūra, turi būti nesunkiai verčiamas į konkretaus valdiklio programą. Todėl anksčiau minėtų tikslų įgyvendinimui galima pasirinkti spalvotuosius Petri tinklus. Šie tinklai puikiai tinka konkurenciniams (pasirinkimo) algoritmams modeliuoti . Tinklo spalvų aprašams bei šakų ir pereigų reiškiniams užrašyti, galima pasirinkti SML programavimo kalba. Pagrindiniai jos privalumai yra notacijų kompaktiškumas ir griežta sintaksė, neleidžianti atsirasti neapibrėžtoms situacijoms. Visa tai yra patyrimo, įgyto kuriant ir taikant programų paketą CENTAURUS-C, apibendrinimas. Šitą programos paketą sukūrė Kauno technologijos universiteto Elektronikos ir automatikos fakulteto Valdymo technologijų katedros Hibridinių sistemų modeliavimo mokslinė grupė [14].

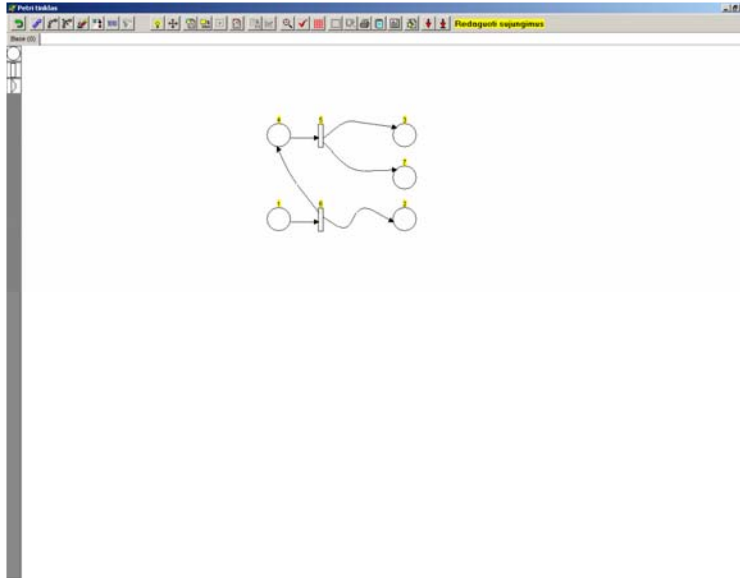
CENTAURUS programinė įranga skirta heterogeninių (mišrios struktūros) ir netiesinių valdymo sistemų analizei bei sintezei; tai specializuota programinė įranga, leidžianti modeliuoti sistemas, turinčias algoritminę (loginę) dalį, aprašomą spalvotų Petri tinklų elementų, ir analoginę dalį, vaizduojamą struktūrine schema. Analoginėje dalyje yra ne tik analoginiai elementai, apjungiantys netiesines diferencines lygtis, bet ir netiesiniai, diskretiniai ir specializuoti elementai. Tokiu būdu heterogeninės sistemos modelyje yra abi sistemos: ir spalvoti Petri tinklai, ir analoginė dalis. Modeliuoti galima trijų rūšių modelius: tiek Petri tinklą, tiek analoginį modelį, tiek abu kartu.

Programų paketo analoginė dalis turi kelis uždavinio sprendimo kelius:

1. Modeliavimas arba analizė.
2. Optimizavimas arba sintezė.
3. Variacijos.
4. Iteracijos arba identifikavimo uždavinys.

Modeliuojant analoginius modelius yra suskaičiuojamas tiriamos sistemos pereinamasis procesas, o taip pat gali būti atlikta tiriamos sistemos parametrinė optimizacija. Modeliavimas spalvotais Petri tinklais vyksta laikantis SML standarto. Yra naujų funkcijų ir kintamųjų, modeliavimo rezultatai vaizduojami grafiškai.

CENTAURUS veikia ne žemesnėje kaip WINDOWS 95 operacinėje sistemoje.



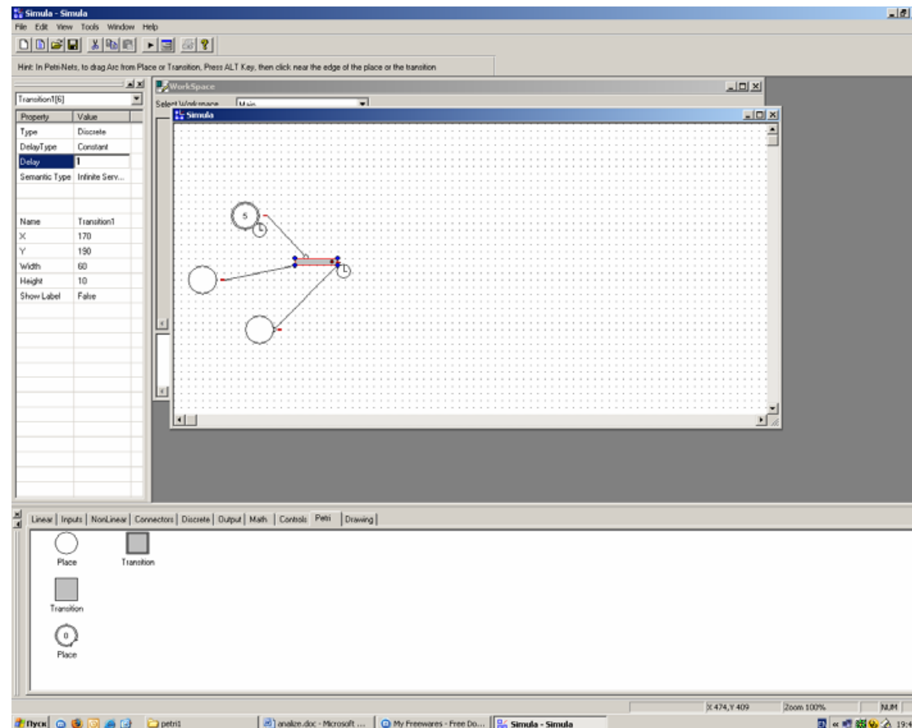
2.4 pav. Petri tinklas Centaurus programoje

Iš mokamų pagamintų Petri tinklų simuliatorių galima išskirti MathTools gamintojo Petri Net Simulator paketą, kuris kainuoja 99 dolerius. Šitas paketas aprūpina bendrą Petri tinklo simuliaciją, taipogi vartotojui suteikiama galimybė sukonstruoti ir simuliuoti skirtingų tipų Petri tinklus. Palaikomi Petri tinklai:

- Diskretūs Petri tinklai
- Autonominiai Petri tinklai
- T-Timed ir P-Timed Petri tinklai
- Stochastinės Petri tinklai
- Pastovių greičių Petri tinklai
- Kintamų greičių Petri tinklai
- Hibridiniai Petri tinklai

Viena iš geriausių Petri tinklų simuliatorių bei modeliavimų programų laikoma Petri Net Simulator Simula 1.0 programa. Kaip ir prieš tai nagrinėta programa, Simula palaiko tuos pačius tinklus, aprūpina bendrą Petri tinklo simuliaciją, taip pat vartotojui suteikiama galimybė konstruoti ir simuliuoti skirtingų tipų Petri tinklus. Be to, šitoje programoje galima nustatyti Petri tinklo vėlinimus. Programa teikia galimybę modeliuoti bei verifikuoti Petri tinklus.

Galima modeliuoti diskretines sistemas, konstruoti norimus sujungimus, yra daug ir kitų privalumų.



2.5 pav. Petri tinklas Simula 1.0 programoje

Šiame darbe stengiamasi pasinaudoti visais gerais aptartų programų sumanymais, vengiama jų trūkumų bei pristatomi kiti ypatumai. Buvo pastebėta, kad projekte realaus laiko sistemos realizavimui bei tyrinėjimui geriausiai naudoti UML kalbos būsenų diagramas bei laiko Petri tinklus (T-Timed Petri Net).

2.8 Veiklos diagramų pertvarkymo į Petri tinklus taisyklių analizė

Šio darbo dalyje nagrinėjamos pasaulyje egzistuojančių veiklos diagramos pertvarkymo pasiūlymų taisyklės, kai kurių užsienio mokslininkų veiklos diagramos perkėlimo į Petrį tinklus pasiūlymai.

Profesorius E. Kazanavičius (KAZANAVIČIUS, E. *The Evaluation and Desingn Methodology for Real Time System.* INFORMATICA, 2004, Vol. 15, No. 1, 45-62) savo straipsnyje teikia siūlymą naudoti Petri tinklus realaus laiko sistemų modeliavimui [3]. Be to, jis pastebėjo, kad realaus laiko sistemų projektavimui bei sistemos elgsenai aprašyti galima panaudoti UML kalbos veiklos diagramas. Straipsnyje aprašoma realaus laiko sistemos

paraiškos modelio kūrimo metodika, kuri turėtų galimybę patikrinti algoritmo teisingumą ir laiko apribojimo atlikimą, projektavimo metu naudojant UML kalbą ir Petri tinklą. Pasiūlytos veiklos diagramos transformacijos į Petrį tinklus žymi keturis scenarijus, kuriais remiantis galima sukurti magistro darbe modeliavimo sistemos pertvarkymo taisykles.

Mokslininkai Rik Eshuis ir Roel Sieringa straipsnyje „*A Comparison of Petri Net and Activity Diagram Variants*" atlieka veiklų diagramų ir Petri tinklų lyginamąjį analizę. Jie ne tik palygina, bet ir randa panašumų tarp diagramos ir Petrį tinklų, o tai suteikia galimybę, kuriant modeliavimo sistemą, supaprastinti veiklos diagramas pertvarkymo metu arba perkelti diagramą dalimis.

Mokslininkai Lopez, Grao J.P., Merseguer J, Campos J savo darbe „*From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering*." (Proceedings of the Fourth International Workshop on Software and Performance (WOSP'04), ACM, Redwood City, California, USA, pages 25-36. January 2004.) išvardina veiklos diagramos pertvarkymo galimybes į Petrį tinklus, bet jie nepasiūlė pertvarkymo metodikos arba transformacijos taisyklių. Tačiau jie išvardino visus veiklos diagramos atvejus, kurių reikia sudarant pertvarkymo taisykles.

Galima pasakyti, kad kol kas nebuvo sudarytos veiklos diagramos pertvarkymo į Petri tinklus taisyklės ir elementų perkėlimo algoritmas. Bet buvo pastebėta, kad mokslininkai ir Lietuvoje, ir užsienyje pripažino šios problemos svarbą bei sudarė veiklos diagramos ir Petri tinklų lyginamąją analizę bei išvardino veiklos diagramos pertvarkymo galimybes. Magistro darbe bandoma suformuoti transformacijos taisykles, veiklos diagramos elementų perkėlimą į Petri tinklus. Tai bus viena iš pagrindinių dalių sudarant veiklos diagramos modeliavimo sistemą.

3. VEIKLOS DIAGRAMOS PERTVARKYMAS Į PETRI TINKLĄ: PERTVARKYMO TAISKYKLĖS IR ELEMENTŲ PERKĖLIMAS

3.1 Veiklos diagramos pertvarkymo fazės

Trumpas kiekvieno veiklos diagramų elemento aprašymas ir jų pertvarkymai į žymeklio apibendrintus stochastinius Petri tinklus (ŽASPT) bus pateikiami toliau. Bendrą metodą iliustruoja 3.1 paveikslas. Dažniausiai kiekvieno iš veiklos diagramų elemento pertvarkymas gali būti apibendrintas žemiau pateiktų trijų fazių proceso:

1 fazė. Kiekvieno išvesties ir rekursyvinio perėjimo pertvarkymas. Pritaikomas veiklai, poveiklai ir iškvietimo būsenai ir pseudobūsenos iššakojimui. Priklausomai nuo perėjimo pobūdžio, yra taikomos skirtingos taisyklės. 3.2 ir 3.4 paveikslai pavaizduoja potinklius, į kuriuos pervedami kiekvienos rūšies perėjimai.

2 fazė. ŽASPT struktūra derinama prie viso kiekvienos rūšies perėjimo rinkinio, kaip buvo apžvelgta 1 fazėje. Pritaikomas veiklai, poveiklai ir iškvietimo būsenai ir pseudobūsenos iššakojimas. Ši struktūra bus aprašyta toliau.

3 fazė. Sudarant ŽASPT iš elementų sudarymas panaudojant ŽASPT superpozicijos metodą, kuris yra gaunamas paskutinėje fazėje ir retkarčiais atitinkantis papildomą ŽASPT įėjimą į jo jungiančią fazę (taip vadinama 'pagrindinė' poveiklės potinklio būsena ir iššakojimo būsenos, žr. 3.2 ir 3.4 paveikslėlius).

3.2 Petri tinklų posistemės aprašymas

3.2 paveikslėlyje parodytas formalus vienos iš ŽASPT sistemos apibrėžimas išdėstomas žemiau. Visi likę 3.2 ir 3.4 paveikslėlyje pristatyti atvejai gali būti nesudėtingai išvesti iš šio pavyzdžio, todėl jie detaliau nekomentuojami.

Nuo šiol bus naudojama OCL kalbos sintaksė tam, kad būtų nurodytas elemento tam tikrais ryšiais priklausantis duomenims vaizdas (arba elementų rinkinio). Šios dalies perkėlimo aprašymas pagristas mokslininkų Lopez, Grao J.P., Merseguer J., Campos J. darbu „*From*

UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering" [4]. Pirmiausia apsvarstysime ryšį tarp *Perėjimo* ir *Veiklos* klasių ir *effekto* vaidmenį tarp jų, toliau atskiro *Veiklos* klasės atvejo vaizdą per parodytą kaip *Action.effect* ryšį *effektą.* Taip pat A klasės atributai, tarkim, *at1,* ir *at2* parodomi panaudojus taško užrašymą *A.atl,* ir *A.at2.*

Verta pažymėti, kad toliau manoma, jog kiekvienas parodytas objektas iš *ModelElement* metaklasės turi unikalų vardą su jo vardo vieta, net jei tai nebus detaliai parodyta modelyje.

Veiklos būsenos *AS* (žr. 3.2 paveikslą, 1.a atvejį) sistema su laiko išvesties perėjimais *ott* yra ŽASPT $LS_{AS}^{ott} = (S_{AS}^{ott}, \psi_{AS}^{ott}, \lambda_{AS}^{ott})$, kuri apibūdinama kaip perėjimų rinkinys $T_{AS}^{ott} = \{t_1, t_2\}$ ir vietų rinkinys $P_{AS}^{ott} = \{p_1, p_2, p_3\}$. Įvesties ir išvesties funkcijos atitinkamai lygios:

$$I_{AS}^{ott}(t) = \begin{matrix} \{p_1\}, if\ t = t_1 \\ \{p_2\}, if\ t = t_2 \end{matrix} \qquad O_{AS}^{ott}(t) = \begin{matrix} \{p_2\}, if\ t = t_1 \\ \{p_3\}, if\ t = t_3 \end{matrix}$$

Kadangi nėra lankų inhibitoriaus, tai $H_{AS}^{ott}(t) = 0$. Pirmenybės ir svorio funkcijos atitinkamai lygios:

$$\Pi_{AS}^{ott}(t) = \begin{matrix} 0, if\ t = t_2 \\ 1, if\ t = t_1 \end{matrix} \qquad W_{AS}^{ott}(t) = > \begin{matrix} \geq r_{ott}, if \quad \Pi_{AS}^{ott}(t) = 0 \\ p_{cond}, if \quad \lambda_{AS}^{ott}(t) = cond_ev \\ 1 \quad kitaip \end{matrix}$$

kur šiuo atveju r_{ott} yra laiko perėjimo parametro dažnis t_2 ir p_{cond} yra nedelsiamo perėjimo svoris t_1.

Svoris P_{cond} nustato tikėtinumo pastabos reikšmę, kuri yra prisegama prie veiklos diagramos perėjimo *ott* su formatu PAprob = *Pcond.* Jei tokios pastabos nėra, *Pcond* yra lygus *1/nt,* kur *nt* yra elementų skaičius rinkinyje *AS.outgoing.*

Dažnis r_{ott} yra lygus *1/n,* kai laiko pastaba prisegama prie veiklos diagramos perėjimo, kuris yra išreikštas formatu *PArespTime = (<source-modifier>,max,(n,'s.'))*, kai jis yra išreikštas formatu *PArespTime = (<sourcemodifier>,' dist',(n-m,'s.')),* tai r_{ott} yra lygus *2/(n + m).*

Pirminė ženklinimo funkcija yra apibrėžiama kaip $\forall p \in P_{AS}^{ott} : M_{AS}^{ott0}(p) = 0$. Galiausia, ženklinimo funkcijos yra lygios:

$$\psi_{AS}^{ott}(p) = \begin{cases} ini_AS, & if \quad p = p_1 \\ execute, & if \quad p = p_2 \\ ini_nextx, & if \quad p = p_3 \end{cases} \quad \lambda_{AS}^{ott}(t) = \begin{cases} cond_ev, & if \quad t = t_1 \\ out_lambda, & if \quad t = t_2 \end{cases}$$

kur, užrašymas pertvarkomas: *AS = AS.name* ir *nextx =oft.target.name.*

kadangi jie yra gausiai naudojami kitame skyriuje, *AG* taip pat apibrėžiama kaip veikos diagrama su $Lstvertex^P$ būsenos viršūnių žymeklių rinkiniu joje, $Lstvertex^P$ = *{ini_target,* $\forall$ *target* $\in$ *E AG.transitions* $\rightarrow$ *target.name}* ir Lev^p kaip įvykių sistemoje rinkinys Lev^P = *{e_evx,* $\forall$ *evx* $\in$ *Ev}* $\cup$ *{ack_evx,* $\forall$ *Vevx* $\in$ *Ev}.*

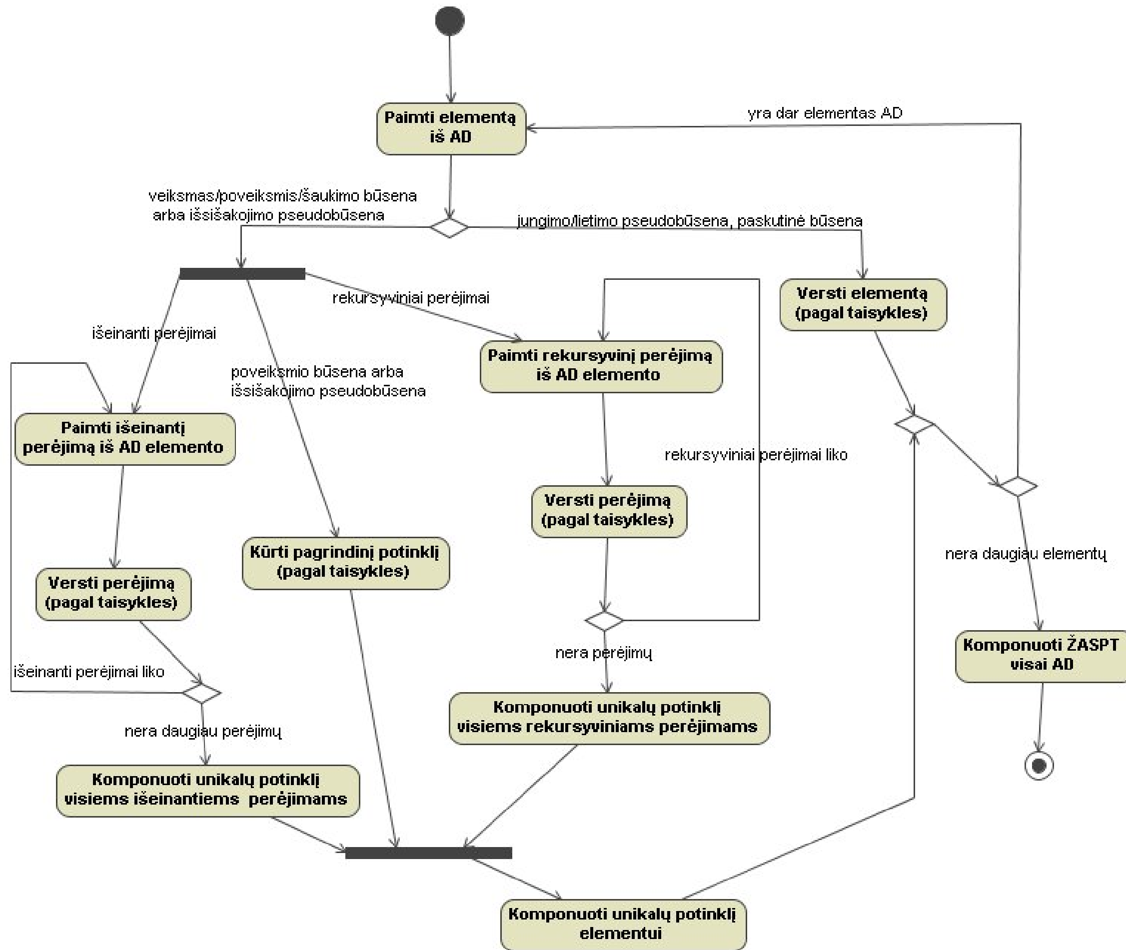
3.3 Veiklos diagramos elementų perkėlimo aprašymas

Tolimesni skyreliai skirti kiekvieno veiklos diagramos elemento perkėlimui į ŽASPT. Šios veiklos rezultatas yra stochastinių Petri tinklų sistema, kuri bus naudojama siekiant vykdyti modeliuotų elementų parametrus.

3.3.1 Veiksmo būsena

Veiksmo būsena yra būsenos su įėjimo veikla sutrumpinimas ir bent vieno perėjimo, įtraukiančio numatomo užbaigiančio veiksmą įvykio. Pagal šį apibrėžimą ir paprastų būsenų perėjimą SMs reikia interpretuoti kaip atominį veiksmą ir pristatyti jį nedelsiamu perėjimu per ŽASPT derinimą su būsena.

Tačiau, siekiant patogaus modeliavimo vykdymo, čia numatomi laiko veiksmai (tai yra veiksmai su reikšminga trukme) [9]. Tam, kad tai būtų padaryta, atskirsim laiko išvesties perėjimus nuo nelaiko išvesties perėjimų. Kaip jau buvo minėta anksčiau, pastabos yra prisegamos, kad būtų sudaryta galimybė skirtingų veiksmo trukmių pavedimoms priklausyti nuo nutraukimo sąlygų (svarbu pažymėti, kad veiksmų diagramose išvesties perėjimai iš veiksmo būsenos modeliuoja sprendimo sritis). Laiko perėjimai veiklos diagramose sukelia laiko perėjimų įtraukimą (su susijusiu su funkcijos atlikimo pastaba dažniu) gaunamame ŽASPT. Nelaiko perėjimas pasireikš nedelsiamu perėjimu į ŽASPT modelį.



3.1 pav. Veiklos diagramų bendras perėjimo metodas

Verčiant veiksmo būseną į ŽASPT formaliai reikia vadovautis trijomis fazėmis, kurios buvo pristatytos anksčiau. Jei veiksmo būsena yra *AS,* tai q – būsenos išvesties laiko perėjimų (*OTi*), kurie nesibaigia sujungtoje pseudobūsenoje, skaičius; q' – išvesties nelaiko perėjimų (ONj), kurie nesibaigia sujungtoje pseudobūsenoje, skaičius; r - išvesties laiko perėjimų (*OTJm),* kurie baigiasi sujungtoje pseudobūsenoje, skaičius; r' - išvesties nelaiko perėjimų (*ONJ~),* kurie baigiasi sujungtoje pseudobūsenoje, skaičius; s – rekursyviniių laiko perėjimų *STk* skaičius; s ' - rekursyvinių laiko perėjimų *SNi* skaičius.

Toliau kiekvienam išvesties rekursyviniam perėjimui t, yra ŽASPT $LS^t_{AS} = (S^t_{AS}, \psi^t_{AS}, \lambda^t_{AS})$, kaip parodyta 3.2 paveikslėlio 1.a-l.f atvejais. Gaunamas rezultatas - q

+ q' + r + r' + s + s' rinkinys ŽASPT modelyje, kurį reikia sujungti tam, kad būtų gautas modelis su būsena *AS,* $LS_{AS} = (S_{AS}, \psi_{AS}, \lambda_{AS})$.

Pirmiausia reikia sudaryti vienos rūšies perėjimų pomodelius, naudojant superpozicijų operatorius, kuriuos aprašo šios lygybės:

$$LS_{AS}^{OT} = \underset{Lstvertex^P}{\overset{i=1,\dots,q}{||}} LS_{AS}^{OT_i} \qquad LS_{AS}^{ON} = \underset{Lstvertex^P}{\overset{i=1,\dots,q'}{||}} LS_{AS}^{OT_J}$$

$$LS_{AS}^{ST} = \underset{ini_AS}{\overset{k=1,\dots,s}{||}} LS_{AS}^{ST_K} \qquad LS_{AS}^{SN} = \underset{ini_AS}{\overset{i=1,\dots,s'}{||}} LS_{AS}^{ST_i}$$

$$LS_{AS}^{OTJ} = \underset{AS}{\overset{m=1,\dots,T}{||}} LS_{AS}^{OTJ_M} \qquad LS_{AS}^{ONJ} = \underset{ini_AS}{\overset{i=1,\dots,r'}{||}} LS_{AS}^{ONJ_N}$$

Sudarant posistemes kaip yra parodyta, ŽASPT modelio LS_{AS} dabar aprašoma taip:

$$LS_{AS} = ((((LS_{AS}^{SN} \underset{ini_AS}{||} LS_{AS}^{ST}) \underset{ini_AS}{||} LS_{AS}^{ON}) \underset{Lstvertex^P}{||} LS_{AS}^{OT}) \underset{ini_AS}{||} LS_{AS}^{OTJ}) \underset{ini_AS}{||} LS_{AS}^{ONJ}$$

Galiausiai, reikia prisiminti, kad UML leidžia bet kokios rūšies veiksmą įvykdyti veiksmo būsenos viduje. Vadinasi reikia surasti *CallAction* arba *SendAction*. Tačiau UML sintaksė suteikia du specialius elementus šio tipo būsenoms: iškvietimo būsenų ir signalo siuntimo ženkliukus, kuriuos galima naudoti, bet jei vietoj jų yra naudojama veiksmo būsena, tai reikia taikyti ekvivalentinių elementų (iškvietimo būsenos arba signalo siuntimo kontrolės ženkliukus) perėjimo metodą.

3.3.2 Poveiksmio būsena

Poveiksmio būsena visada remiasi vienodų veiksmų diagramų rinkiniu. Jos išvesties perėjimai neturi prisegtų pastabų, nes veiklos trukmė gali būti nulemta veiklos diagramos vertimo ir pilnos sistemos sudarymo (tai bus parodyta toliau). Verčiant poveiklos būseną į ŽASPT, formaliai reikia vadovautis trijomis fazėmis, kurios buvo pristatytos anksčiau. Svarbu pažymėti, kad yra papildomas ŽASPT, kuris derinamas su įėjimu į būseną, vadinamą *basic.*

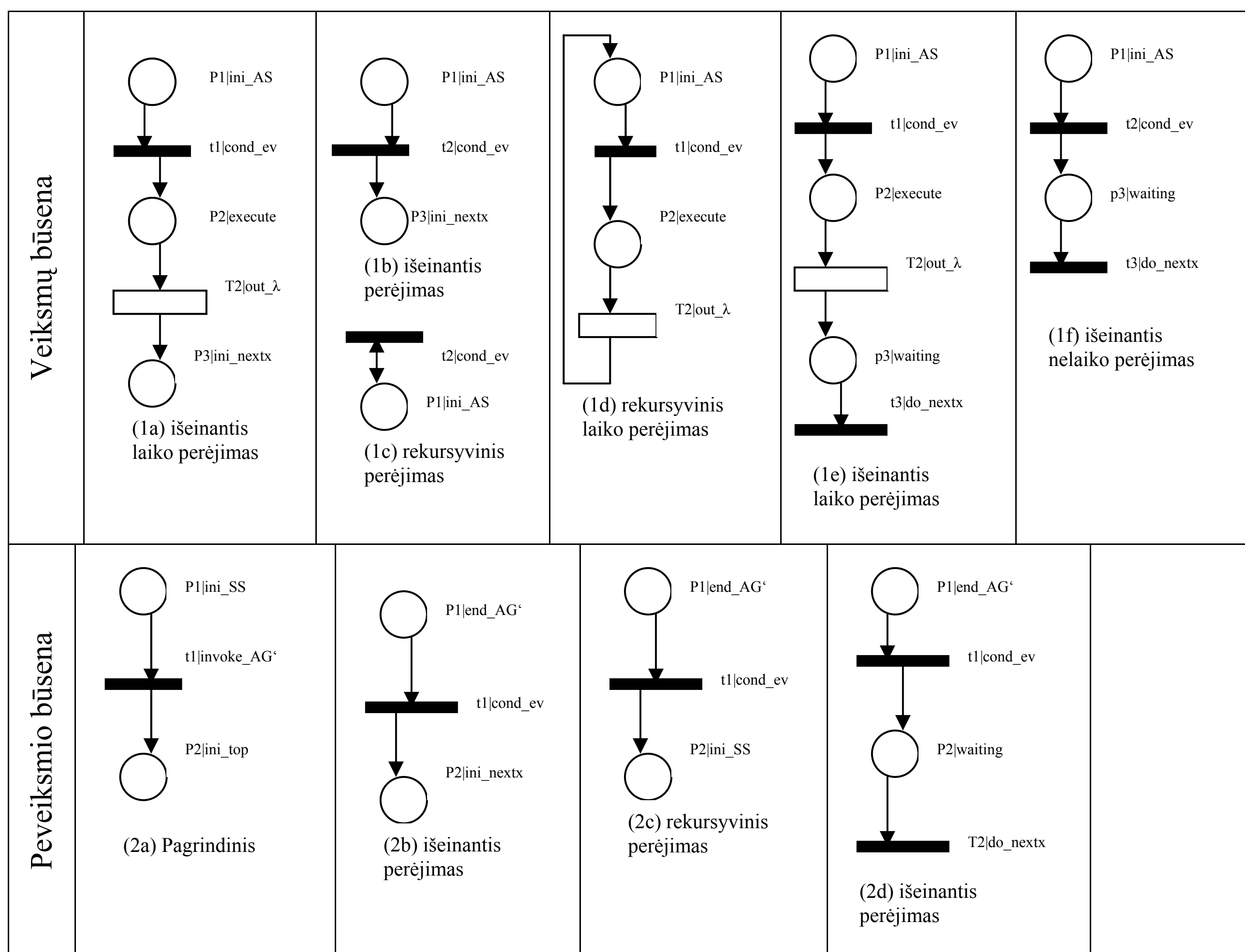
Toliau, jei poveiklos būsena yra *SS,* tai q – būsenos išvesties laiko perėjimų (*Oi*), kurie nesibaigia sujungtoje pseudobūsenoje, skaičius; r – išvesties laiko perėjimų (*OJk),* kurie baigiasi sujungtoje pseudobūsenoje, skaičius; s – rekursyviniių laiko perėjimų *Sj* skaičius. Taip pat *AG'* yra vienodų veiksmų diagrama ir *top* – pirmas *AG'* elementas*,* kur *top = AG'.top.*

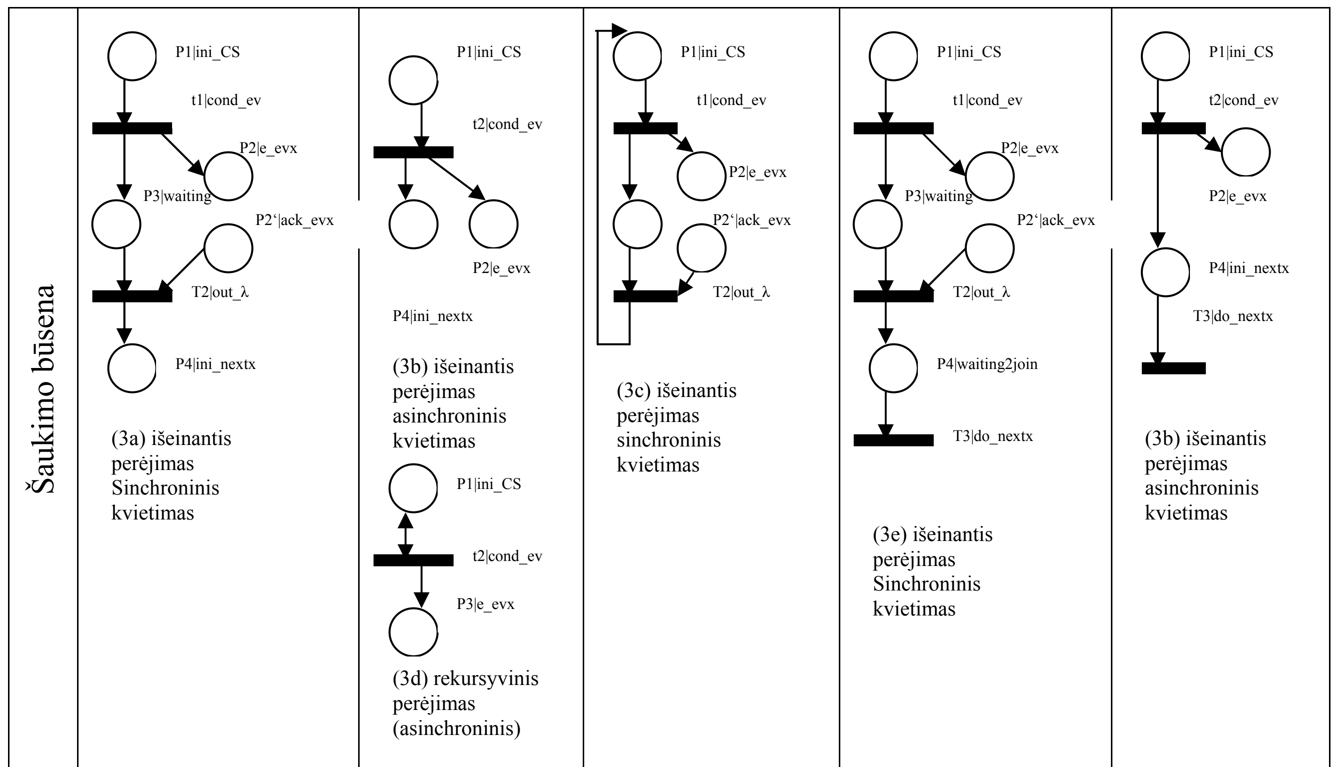
Pagal perėjimus, parodytus 3.2 paveikslėlio 2.a-2.d atvejais, pagrindinis ŽASPT $LS_{SS}^{B} = (S_{SS}^{B}, \psi_{SS}^{B}, \lambda_{SS}^{B})$ ir vienas ŽASPT kiekvienam rekursyviniam išvesties perėjimui *t*, $LS_{SS}^{t} = (S_{SS}^{t}, \psi_{SS}^{t}, \lambda_{SS}^{t})$. Taigi q + r + s + 1 yra ŽASPT modeliai, kuriuos reikia sujungti, siekiant gauti būsenos *SS* modelį, $LS_{SS} = (S_{SS}, \psi_{SS}, \lambda_{SS})$. TŽASPT derinamas su kiekvienos rūšies perėjimais, pasiekiamais pagal superpozicijas:

$$LS_{SS}^{O} = \underset{Lstvertex^{P}}{\overset{i=1,...q}{\|}} LS_{SS}^{O_I} \qquad LS_{SS}^{OJ} = \underset{end_AG}{\overset{k=1,...r}{\|}} LS_{SS}^{OJ_K} \qquad LS_{SS}^{S} = \underset{end_AG}{\overset{j=1,...q}{\|}} LS_{SS}^{S_J}$$

Ir pagaliau ŽASPT modelis LS_{ss} poveiklos būsenai dabar aprašomas taip:

$$LS_{SS} = ((LS_{SS}^{OJ} \underset{end_AG}{\|} LS_{SS}^{S}) \underset{end_AG}{\|} LS_{SS}^{O}) \underset{ini_SS}{\|} LS_{SS}^{B}$$





3.2 pav. ŽASPT veiksmas, poveikla ir iškvietimo būsenos

3.3.3 Iškvietimo būsena

Iškvietimo būsenos yra tam tikras veiksmo būsenų atvejis, kuriame įėjimo veiksmas yra CallAction (iškvietimo veiksmas), todėl šių elementų vertimas yra labai panašus. Būtina paminėti, kad kai CallAction yra atliktas, gali būti sugeneruotas CallEvents (iškvietimo įvykiai) rinkinys. Siekdami paprastumo, laikysimės prielaidos, kad dauguma atvejų sugeneruojamas tik vienas įvykis, bet apibrėžimas gali būti tęsiamas, papildant jį naujomis ŽASPT vietomis.

Be to, CallAction gali būti sinchroninis arba nepriklausomas nuo jo atributo *isAsynchronous* vertės. Čia *sinchroninis* reiškia, kad veiksmas nebus užbaigtas tol, kol galutinai nebus sugeneruotas įvykis, o veiksmas nebus paleistas per gaviklį. Šiuo atveju reikia naujos vietos ir perėjimo atitinkamame ŽASPT, kad sumodeliuotumėm sinchronizaciją (žr. 3.2 paveikslėlio 3.a, 3.c ir 3.e atvejus).

Verčiant iškvietimo būseną, atlikimo fazės labai panašios į anksčiau aprašytas. Tarkim, į *CS* iškvietimo būseną,

• jei patikrinamas *S.entry.IsAsynchronous = false* (tai yra susijęs iškvietimo veiksmas CallAction yra sinchroninis iškvietimas), mes apibrėžiam u kaip būsenos išvesties perėjimų (OS_i), kurie nesibaigia apjungtoje pseudobūsenoje, skaičių; v - išvesties laiko perėjimų (OJS_k), kurie baigiasi apjungtoje pseudobūsenoje, skaičių; w rekursyvinių perėjimų SS_m skaičių.

• jei patikrinamas *S.entry.IsAsynchronous = true* (tai yra susijęs iškvietimo veiksmas CallAction yra nesinchroninis iškvietimas), mes apibrėžiam u' kaip būsenos išvesties perėjimų (OA_i), kurie nesibaigia apjungtoje pseudobūsenoje, skaičių; v' - išvesties laiko perėjimų (OJA_z), kurie baigiasi apjungtoje pseudobūsenoje, skaičių; w' rekursyvinių perėjimų SA_m skaičių.

Taip pat tarkim, kad *evx* yra iškvietimo veiksmo sugeneruotas įvykis, *evx = S.entry.operation->occurrence.* Laikantis šios prielaidos, mes turime vieną ŽASPT kiekvienam rekursyviniam išvesties perėjimui *t,* $LS^t_{CS} = (S^t_{CS}, \psi^t_{CS}, \lambda^t_{CS})$, kaip parodyta 3.2 paveikslėlio 3.a-3.f atvejais. Todėl mes turime aibę $u + v + w$ arba $u' + v' + w'$ ŽASPT modelių, kuriuos reikia sujungti tam, kad gautume būsenos *CS* modelį, $LS_{CS} = (S_{CS}, \psi_{CS}, \lambda_{CS})$. ŽASPT atitinkantys kiekvienos rūšies perėjimus, dabar atitinka superpozicijas:

$$LS^{OS}_{CS} = \mathop{\|}_{Lstvertex^P}^{i=1,\dots u} LS^{OS_I}_{CS} \qquad LS^{OA}_{CS} = \mathop{\|}_{Lstvertex^P}^{j=1,\dots u'} LS^{OA_J}_{CS}$$

$$LS^{OJS}_{CS} = \mathop{\|}_{ini_CS, Lev^P}^{k=1,\dots v} LS^{OJS_K}_{CS} \qquad LS^{OJA}_{CS} = \mathop{\|}_{ini_CS, Lev^P}^{l=1,\dots v'} LS^{OJA_L}_{CS}$$

$$LS^{SS}_{CS} = \mathop{\|}_{ini_CS, Lev^P}^{m=1,\dots w} LS^{SS_M}_{CS} \qquad LS^{SA}_{CS} = \mathop{\|}_{ini_CS, Lev^P}^{n=1,\dots w'} LS^{SA_N}_{CS}$$

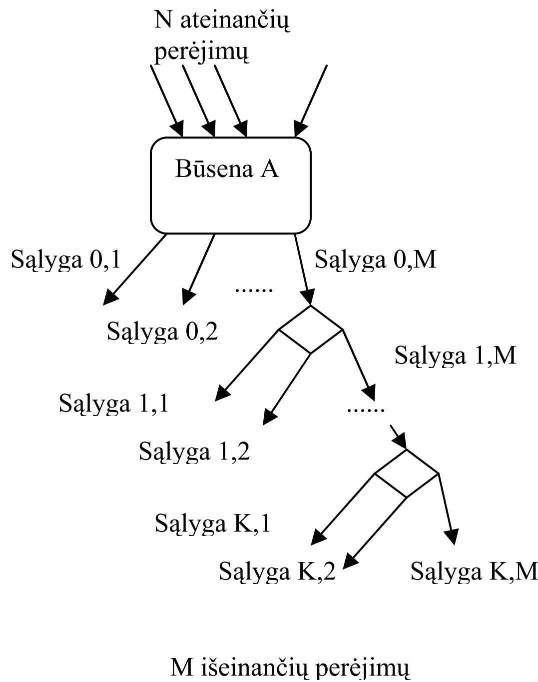
Galutinis ŽASPT LS_{CS} būsenai apibrėžiamas kaip viena iš dviejų žemiau pateikiamų lygybių, priklausomai nuo to, ar veiksmas yra sinchroninis *(a),* ar nesinchroninis *(b):*

$$LS_{CS} = (LS^{SS}_{CS} \mathop{\|}_{ini_CS, Lev^P} LS^{OS}_{CS}) \mathop{\|}_{ini_CS, Lev^P} LS^{OJS}_{CS} \text{ (a)}$$

$$LS_{CS} = (LS^{SA}_{CS} \mathop{\|}_{ini_CS, Lev^P} LS^{OA}_{CS}) \mathop{\|}_{ini_CS, Lev^P} LS^{OJA}_{CS} \text{ (b)}$$

3.3.4 Sprendimai

Sprendimai yra apdorojami prieš veiksmų diagramos vertimą. Jie yra pakeičiami lygiais veiksmo būsenos išvesties perėjimais (kaip parodyta 3.3 paveikslėlyje), išlaikant savybes neatskiriamas atlikimo pastabų savybes. Tačiau sprendimai nebūtinai turi būti verčiami. Pažymėtina, kad pastabos 3.3 paveikslėlyje griežtai neatitinka UML profilio, tam, kad būtų išlaikytas kompaktiškas užrašymas (kitaip paveikslėlis būtų per daug apkrautas).



3.3 pav. ŽASPT sprendimas (prieš pasikeitimus)

3.3.5 Susiliejimai

Susiliejimai yra naudojami kontrolės srautams, kuriuos sprendimai suskaidė į skirtingas sritis (arba kuriuos būsenų išvesties perėjimus pažymėjo apsaugos) sujungti. Dažnai jie yra tik užrašymo formalumas, nes sujungimas gali būti modeliuojamas kaip būsenos įvesties perėjimai.

Pseudobūsenos M susiliejimo vertimas priklauso nuo jos išvesties perėjimo tikslinių elementų. 3.4 paveikslėlio 5.a ir 5.b atvejai parodo tiesioginį modelio LS_M vertimą pagal apačioje išreikštas sąlygas:

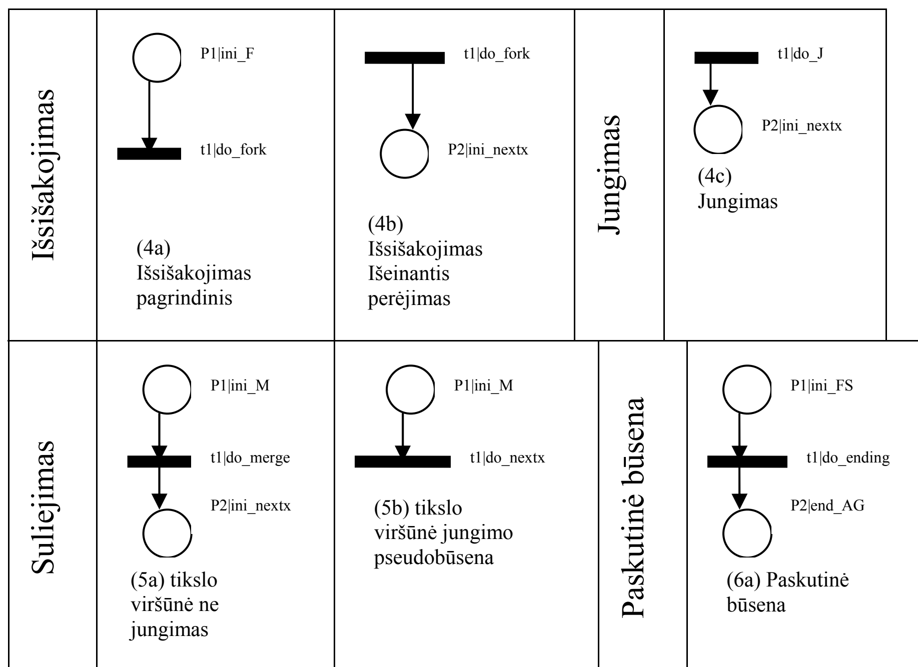
(a) $LS_M = LS'_M \Leftrightarrow (PS.outgoing.tarrget \notin Pseudostate \vee Ps.outgoing.tarrget.kind \neq join$ (sujungti)

(b)

$LS_M = LS''_M \Leftrightarrow (PS.outgoing.tarrget \in Pseudostate \wedge Ps.outgoing.tarrget.kind = join$ (sujungti)

3.3.6 Lygiagretumo palaikymo elementai

UML suteikia du elementus lygiagretumo modeliavimui veiklos diagramose: iššakojimai ir sujungimai. Gerai žinoma tai, kad veiklos diagramos buvo sukurtos kaip įvairių resursų charakteristikų derinys: Odell įvykio diagrama, SDL ir Petri tinklai. Išsišakojimai ir sujungimai buvo tiesiogiai paveldėti neseniai (nors tam tikras lygiagretumo palaikymas jau buvo ir Odell įvykio diagramose). Vertimas į ŽASPT modelius labai paprastas abiem atvejais.



3.4 pav. ŽASPT iššakojimas, sujungimas, suliejimas, galinė būsena

Tarkim, sujungimo pseudobūsena J verčiama į pažymėtą sistemą LS_j , kaip parodyta 3.4 paveikslėlio 4.c atveju.

Verčiant išsišakojimą, reikia laikytis trijų žingsnių:

Jei yra išsišakojimo pseudostadija *F*, tai *q* apibrėžiame kaip išvesties perėjimų O_i skaičių. Toliau, pagal parodytus 3.4 paveikslėlyje vertimus, gaunamas pagrindinis ŽASPT $LS_F^B = (S_F^B, \psi_F^B, \lambda_F^B)$ (minėto paveikslėlio 4.a atvejis) ir po vieną ŽASPT (4.b atvejis) kiekvienam išvesties perėjimui *t*, $LS_F^t = (S_F^t, \psi_F^t, \lambda_F^t)$ Tačiau gauname *q + 1* ŽASPT modelių, kuriuos reikia sujungti, kad gautumėt pseudobūsenos modelį $LS_F = (S_F, \psi_F, \lambda_F)$. ŽASPT atitinkantys kiekvienos rūšies perėjimus, dabar atitinka superpoziciją:

$$LS_F^O = \underset{do_fork, Lstvertex^P}{\overset{i=1,\dots q}{\|}} LS_F^{O_I}$$

Galutinis ŽASPT LS_F sudaromas pagal žemiau pateikiamą išraišką:

$$LS_F = \underset{do_fork}{\|} LS_F^O$$

3.3.7 Pradinės ir galinės būsenos

Pradinės pseudobūsenos ir galinės būsenos yra paveldėtos iš UML būsenos semantikos mechanizmų. Tačiau, skirtingai nei tai vyksta UML SMs, pradinė pseudobūsena neverčiama į ŽASPT modelį, verčiant veiklos diagramas, nes jokio veiksmo negalima prisegti prie išvesties perėjimų. Kita vertus, galinės būsenos verčiamos, bet gaunamas ŽASPT yra skirtingas nuo parodyto.

Jei galutinė būsena yra *FS,* tai ŽASPT modelis $LS_{FS} = (S_{FS}, \psi_{FS}, \lambda_{FS})$ pagal parodytus 3.4 paveikslėlio 6.a atvejo perėjimus lygus apibrėžtai būsenai.

3.4 Veiklos diagramų pertvarkymo į Petri tinklus metodika

Specifikacinių reikalavimų patvirtinimas yra be abejo neatskiriama inžinerinių reikalavimų dalis. Patvirtinimas tai procesas, kurio metu tikrinama, ar specifikaciniai reikalavimai atitinka tarpininkų tikslus bei lūkesčius. Yra įvairių požiūrių į reikalavimų patvirtinimą. Vienas iš jų grindžiamas simuliacijų/atlikimo modeliu, kuris yra kilęs iš pradinių specifikacinių reikalavimų [10]. Petri tinklai yra formali kalba, kuri gali būti naudojama tikrinti reikalavimus pasitelkus simuliacijų/atlikimo modelį [11]. Toliau bus paaiškinama, kaip pristatytos Veiklos diagramų UML 2.0 koncepcijos gali būti pritaikytos Petri tinkluose. Taip

pat toliau bus pristatytos ir specifinės veiklos diagramos koncepcijos, kurios negali būti pritaikytos Petri tinkluose. Šios taisyklės gali būti naudojamos automatiniam pertvarkymui iš veiklos diagramų į Petri tinklus, panaudojus UML diagramų XMI sintaksę. Pabaigoje bus paaiškinamos veiklos diagramų supaprastinimo galimybės prieš pertvarkant į Petri tinklus, o tai leistų sumažinti perėjimų, vietų ir lankų skaičių galutiniame tinkle.

3.4.1 Veiklos diagramos pritaikymo koncepcija Petri tinkluose

Lentelė apačioje paaiškina veiklos diagramų pritaikymo koncepcijas Petri tinkluose.

Koncepcija	**Veiklos diagrama**	**Petri tinklai**
1. Scenarijaus pateikimas	Veikla	CP tinklas
2. Objektyviosios realybės	Suskirstymas	Bus modeliuojamas kaip vieta
3.Funkcija ir veiklos atlikimas	Veiksmas	Perėjimas
4. Scenarijaus pradžia ir pabaiga	Pradinis mazgas & Galinis mazgas (srauto galinis)	Vieta be jokių įvesties pakraščių ir atitinkamai vieta be jokių išvesties pakraščių
5. Alternatyvus scenarijus	Poveikla	Popuslapis
6. Sutapimo srautas	Išsišakojimo mazgas	Bus modeliuojamas kaip perėjimas
7. Alternatyvus srautas	Sprendimo mazgas	Bus modeliuojamas kaip vieta
8. Sekos srautas	Veiklos pakraščiai	Lankas
9. Alternatyvus sujungimas	Susiliejimo mazgas	Bus modeliuojamas kaip vieta

10. Sinchronizuotas sutampantis srautas	Sujungimo mazgas	Bus modeliuojamas kaip perėjimas
11. Tikslai	Tikslo mazgas	Bus modeliuojamas kaip vieta

Lentelė 1 Veiklos diagramų ir Petri tinklų koncepcijų palyginimas.

3.4.2 Transformacijos iš veiklos diagramų į Petri tinklus taisyklės

3.4.2.1 Veiksmo mazgai

Veiklos diagramose veikla yra fundamentalus elgsenos specifikacijos vienetas. Veiklos procese įvesties duomenų rinkinys transformuojamas į išvesties duomenų rinkinį, nors bet kuris iš šių rinkinių gali būti tuščias. Išeities duomenys vienoje veikloje gali būti naudojami kaip kitų veiklų įvesties duomenys, pasitelkiant veiklos srauto modelį.

Kaip ir veiksmai veiklos diagramose, perėjimai yra aktyvūs Petri tinklų komponentai. Jie modeliuoja veiklas, kurios gali atsirasti, keisdami sistemos būseną. Perėjimams leidžiama užsidegti tik tada jei jie yra įgalinti, vadinasi, kad visos prielaidos, kurios yra panašios į UML 2.0 veiklas, yra įvykdytos.

Siūloma naudoti vykdomąjį / poveikio mazgą transformuojant veiklos diagramas į Petri tinklus, nes tai yra natūralus panaudojimas.

3.4.2.2 Kontrolės mazgai

Veiklos diagramose kontrolės mazgai naudojami kontrolės ir duomenų siuntimui per veiklos kreivę. UML2.0 turi septynias koncepcijas ir penkis skirtingus užrašymus kontrolių mazgams. Bet Petri tinklus sudaro tik trys elementų tipai: perėjimai, vietos ir lankai. Todėl yra būtina naudoti daugiau nei vieną veiklos diagramų koncepciją vienai Petri tinklų koncepcijai.

- ✓ *Pradiniai mazgai.* Pradinis mazgas gauna kontrolę, kai veikla prasideda ir pereina jį neuždelsdama per išvesties pakraščius. Jokios kitos elgsenos UML kalboje su pradiniu mazgu nėra. Pradinis mazgas negali turėti į juos įvesties

pakraščių. Natūralus pradinio mazgo panaudojimas yra vieta be jokių įvesties pakraščių, kurie dar vadinami Petri tinklų ištakomis.

- ✓ *Sprendimo mazgai.* Sprendimo mazgas palydi srautą viena ar kita linkme, bet tiksliai ta linkme, kuri yra nustatyta pakraščių, išeinančių iš mazgo. Siūloma talpinti sprendimo mazgą į Petri tinklų vietą.
- ✓ *Susiliejimo mazgai.* Susiliejimo mazgai sujungia daugialypius sudėtinius srautus. Visa kontrolė ir duomenys, ateinantys į susiliejimo mazgą nedelsdami pereina į iš mazgo išvedančius pakraščius. Susiliejimo mazgų toks pat užrašymas, kaip ir sprendimo mazgų, bet susiliejimo mazgai turi daugialypius įvesties pakraščius ir vieną išvestį, kuri yra priešinga sprendimo mazgui. Srautai, įeinantys į susiliejimą, yra dažniausia alternatyvos iš esančio prieš srovę sprendimo mazgo. Siūloma talpinti susiliejimo mazgą į Petri tinklų vietą.
- ✓ *Išsišakojimo mazgai.* Išsišakojimo mazgas veiklos diagramose naudojamas sinchronizuotų sutampančių srautų modeliavimui. Kontrolė ir duomenys, kurie ateina prie išsišakojimo mazgo, dubliuojami per išvesties pakraščius. Kitas veiksmas su iššakojimo mazgu neasocijuojama. UML2.0 pristatė įsagų, kurios yra asocijuojamos su veiklomis, koncepciją. Jei veiksmas turi daugiau nei vieną asocijuotą išvesties įsagą, jis gali būti panaudotas sinchronizuotų sutampančių srautų modeliavimui. Bet siūloma iššakojimo mazgą talpinti į perėjimą. Jei išsišakojimo mazgas veiklos diagramoje užima ankstesnę vietą nei veiklos mazgas, jį galima supaprastinti. Supaprastinimo taisyklės bus paaiškinamos vėliau.
- ✓ *Sujungimo mazgai.* Sujungimo mazgas sinchronizuoja daugialypius srautus. Bendru atveju kontrolė ir duomenys turi būti pasiekiami bet kuriame įvesties pakraštyje tam, kad būtų toliau siunčiami į išvesties pakraščius. Veiksmas su daugiau nei viena asocijuota įvesties įsagą pavaizduoja tą pačią elgseną kaip ir sujungimo mazgas. Siūloma talpinti sujungimo mazgą į perėjimą.
- ✓ *Galinis mazgas.* Yra du galinio mazgo tipai veiklos diagramose besiskiriantys tuo, kad srauto galinis mazgas apriboja vietinius srautus, o veiklos galinis mazgas apriboja ir užbaigia visą veiklą. Petri tinkluose vieta, kuri neturi išvesties pakraščių, vadinama įduba ir yra panaši į srauto galinį mazgą veiklos

diagramoje. Veiklos galinio mazgo koncepcija negali būti uždėta ant įdubos, nes nėra tikslo ardyti leksemas. Ši elgsena gali būti užfiksuota programiškai kuriant Petri tinklus.

3.4.2.3 Objekto mazgai

Yra keturi skirtingi objekto mazgai UML2.0 veiklos diagramose. Objekto mazgai naudojami laikinai sulaikyti duomenis, kol jie laukia tolimesnio judėjimo per kreivę. Jie nusprendžia ir vertės kurią jie gali išlaikyti, tipą.

Petri tinkluose vietos yra leksemų laikymo vieta. Spalvotuose Petri tinkluose vietos nustato specifinį vertės tipą, kurį gali išlaikyti vieta.

Siūloma talpinti veiklos diagramos objekto mazgą į vietą Petri tinkluose.

3.4.2.4 Veiklos pakraščiai

Veiklos diagramose UML2.0 yra dviejų rūšių veiklos pakraščiai: kontrolės srauto pakraščiai ir tikslo srauto pakraščiai [13]. Spalvotuose Petri tinkluose pakraščiai yra skirstomi pagal tai, kokios rūšies leksemos gali praeiti per pakraščius. Abi veiklos diagramų pakraščių rūšys bus uždedamos ant tipizuotų pakraščių Petri tinkluose.

Petri tinklai yra du kartus perskirsti kreivių. Vadinasi, vieta negali eiti anksčiau už kitą vietą, tas pats galioja ir perėjimams. Veiklos diagramose nėra tokio pobūdžio apribojimų. Tokios taisyklės pradeda galioti pakraščių transformacijos metu iš veiklos diagramų į Petri tinklus metu.

1) Jei veiklos pakraščio ištakos mazgas yra pradinis mazgas ir tiksliniai mazgai yra veiksmo, išsišakojimo arba sujungimo, tai veiklos pakraščiai yra pakeičiami lanku.

2) Jei veiklos pakraščio ištakos mazgas yra pradinis mazgas ir tiksliniai mazgai yra sprendimo, susiliejimo arba galutinis, tai veiklos pakraščiai yra pakeičiami lanku, dirbtiniu perėjimu arba dirbtiniu lanku.

3) Jei veiklos pakraščio ištakos mazgas yra veiklos mazgas ir tiksliniai mazgai yra veiksmo, išsišakojimo arba sujungimo, tai veiklos pakraščiai yra pakeičiami lanku, dirbtine vieta arba dirbtiniu lanku.

4) Jei veiklos pakraščio ištakos mazgas yra veiksmo mazgas ir tiksliniai mazgai yra sprendimo, susiliejimo, tikslo arba galutinis, tai veiklos pakraščiai yra pakeičiami lanku.

5) Jei veiklos pakraščio ištakos mazgas yra išsišakojimo arba sujungimo mazgas ir tiksliniai mazgai yra veiksmo, išsišakojimo arba sujungimo, tai veiklos pakraščiai yra pakeičiami lanku, dirbtine vieta arba dirbtiniu lanku.
6) Jei veiklos pakraščio ištakos mazgas yra išsišakojimo arba sujungimo mazgas ir tiksliniai mazgai yra sprendimo, susiliejimo arba galutinis, tai veiklos pakraščiai yra pakeičiami lanku.
7) Jei veiklos pakraščio ištakos mazgas yra sprendimo, tikslo arba susivienijimo mazgas ir tiksliniai mazgai yra veiksmo, išsišakojimo arba sujungimo, tai veiklos pakraščiai yra pakeičiami lanku.
8) Jei veiklos pakraščio ištakos mazgas yra sprendimo, tikslo arba susiliejimo mazgas ir tiksliniai mazgai yra sprendimo, susiliejimo arba galutinis, tai veiklos pakraščiai yra pakeičiami lanku, dirbtine vieta arba dirbtiniu lanku.

Šias taisykles taip pat pristato ir žemiau pateikiama lentelė.

Pakraščio ištakos mazgas (ai)	**Pakraščio tikslinis (iai) mazgas (ai)**	**Pakeitimai**
• Pradinis mazgas arba • Sprendimo mazgas arba • Susiliejimo mazgas arba • Tikslo mazgas	• Veiksmo mazgas arba • Išsišakojimo mazgas arba • Sujungimo mazgas	Lankas
• Pradinis mazgas arba • Sprendimo mazgas arba • Susiliejimo mazgas arba • Tikslo mazgas	• Sprendimo mazgas arba • Susiliejimo mazgas arba • Tikslo mazgas • Galutinis mazgas	Lankas, dirbtinis perėjimas, dirbtinis lankas
• Veiksmo mazgas arba • Išsišakojimo mazgas arba • Sujungimo mazgas	• Veiksmo mazgas arba • Išsišakojimo mazgas arba • Sujungimo mazgas	Lankas, dirbtinė vieta, dirbtinis lankas
• Veiksmo mazgas arba • Iššakojimo mazgas arba • Sujungimo mazgas	• Sprendimo mazgas arba • Susiliejimo mazgas arba • Tikslo mazgas • Galutinis mazgas	Lankas

Lentelė 2Veiklos pakraščių į Petri tinklus perkėlimo taisyklės

3.4.2.5 Veiklų suskirstymas

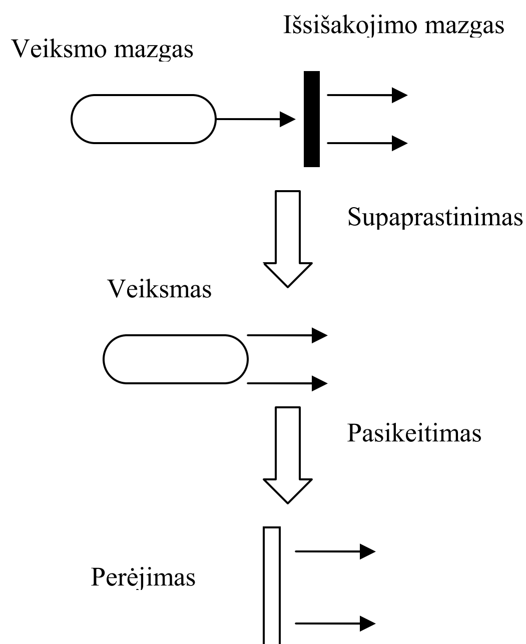
Suskirstymai yra naudojami identifikuoti, kas yra atsakingas už veiklų grupavimą pagal suskirstymus. Nėra prieinamo suskirstymų užrašymo Petri tinkluose, bet jie gali būti naudojami kaip leksmos išteklių vietos.

3.4.3 Veiklų diagramų supaprastinimas prieš darant pasikeitimą

Supaprastinimo taisyklės skirtos sumažinti vietų ir pakeitimų skaičių generuojamuose Petri tinkluose. Įmanoma, kad transformuojant veiklos diagramas, modeliuoja sistemą taip, kad supaprastinimo gali neprireikti. Bet jei geresniam vizualiniam sistemos supratimui modeliuotojas renkasi pasinaudoti žemiau aprašyta seka, mes galime supaprastinti aprašymą taikydami kitame skyrelyje apibrėžtas taisykles. Skyriaus pabaigoje bus pristatytas veiklos diagramų pavyzdys, kuris buvo paimtas iš UML 2.0 specifikacijų ir transformuotas į Petri tinklus po veiklos diagramos supaprastinimo.

3.4.3.1 Veiklos mazgas prieš išsišakojimo mazgą

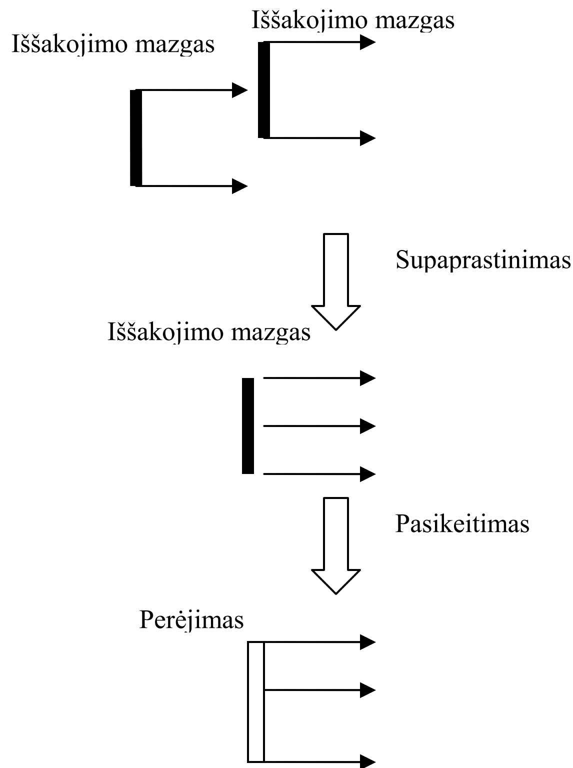
Veiklos diagramoje išsišakojimo mazgas naudojamas leksemų dubliavimui sinchronizuotame sutampančiame sraute ir jokios kitos elgsenos jam nėra numatyta. Siūlomose transformacijos taisyklėse išsišakojimo mazgas buvo uždėtas ant perėjimo. Vadinasi, jei veiksmo mazgas eina prieš iššakojimo mazgą, jis gal būti pakeistas veiksmo mazgu su daugiau nei vienu išvesties pakraščiu kaip parodyta paveikslėlyje apačioje.



3.5 pav. Veiklos mazgas prieš iššakojimo mazgą.

3.4.3.2 Iššakojimo mazgas prieš iššakojimo mazgą

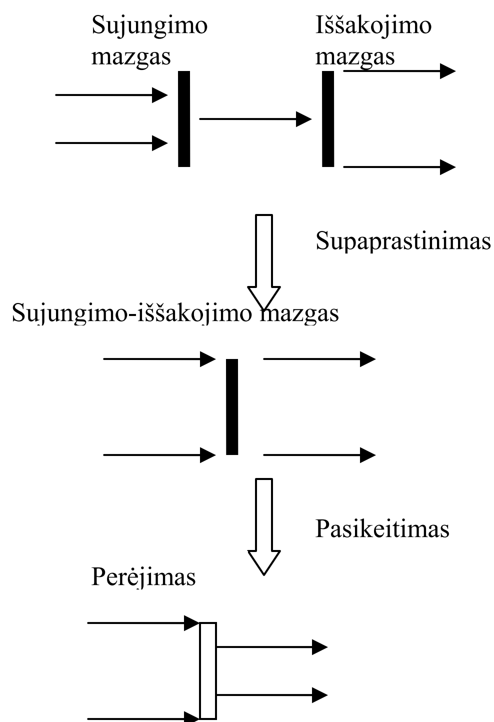
Jei iššakojimo mazgas eina prieš iššakojimo mazgą, tai veiklos diagramoje jis gali būti supaprastintas kaip vienas iššakojimo mazgas, nes iššakojimo mazgo tikslas yra dubliuoti leksemas.



3.6 pav. Iššakojimo mazgas prieš iššakojimo mazgą

3.4.3.3 Sujungimo mazgas prieš iššakojimo mazgą

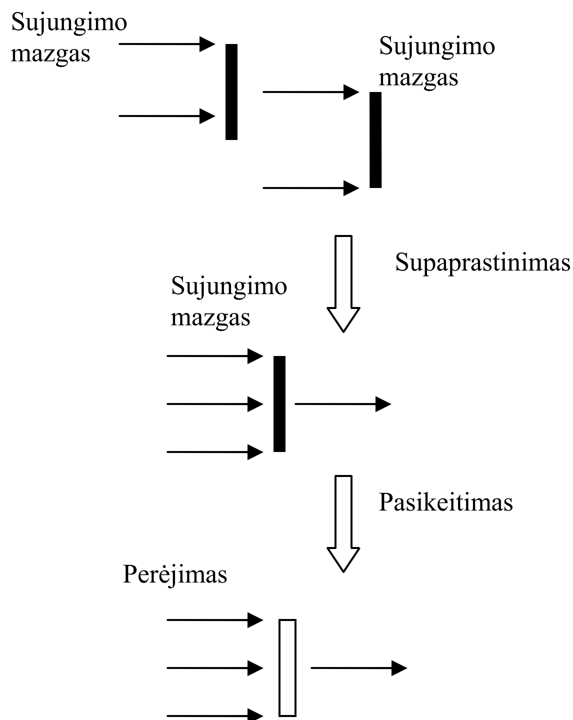
Iššakojimo mazgas naudojamas sinchronizuoti daugialypius srautus veiklos diagramose. Jei sujungimo mazgas eina prieš iššakojimo mazgą, tai supaprastinimo stadijoje juos galima pakeisti veiksmo mazgu, kaip parodyta 3.7 paveikslėlyje.



3.7 pav. Sujungimo mazgas prieš iššakojimo mazgą

3.4.3.4 Sujungimo mazgas prieš sujungimo mazgą

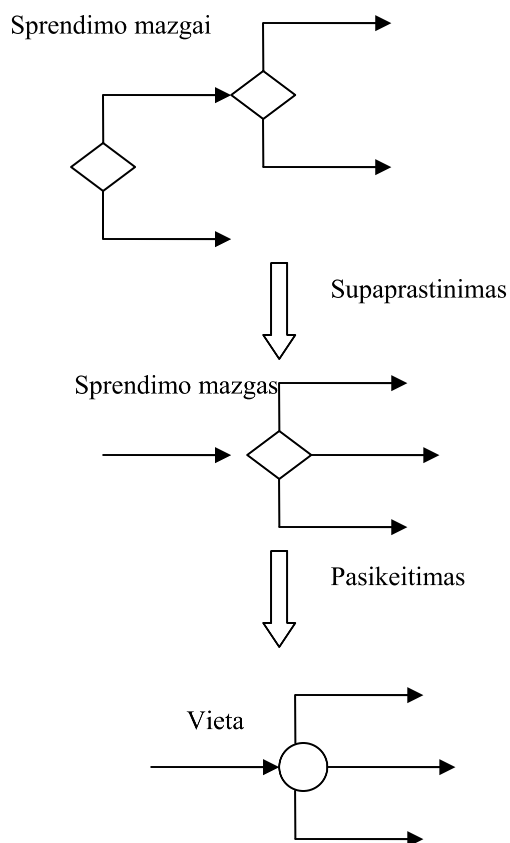
Iššakojimo mazgas naudojamas sinchronizuoti daugialypius srautus. Sujungimo mazgas ima vieną leksemą iš kiekvieno įvesties pakraščio ir sujungia jas. Paskui šios leksemos perduodamos išvesties pakraščiams. Jei iš karto po sujungimo mazgo eina kitas sujungimo mazgas, vadinasi, kad ateinančios prie pirmojo mazgo leksemos bus sujungtos, tada šios sujungtos leksemos bus dar kartą sujungtos su leksemomis, ateinančiomis prie antrojo sujungimo mazgo. Todėl šis procesas gali būti pakeistas vienu sujungimo mazgu, kuris sujungs visas leksemas kaip parodyta paveikslėlyje žemiau.



3.8 pav. Sujungimo mazgas prieš sujungimo mazgą

3.4.3.5 Sprendimo mazgas prieš sprendimo mazgą

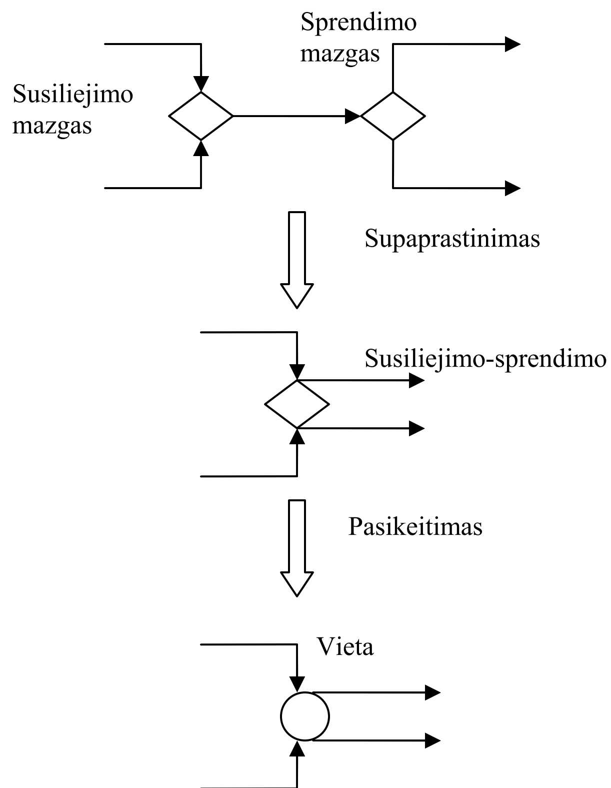
Sprendimo mazgai naudojami palydėti srautą viena ar kita linkme priklausomai nuo išvesties mazgų pakraščių nustatytos apsaugos. Pakraščių apsauga nustatoma individualiai kiekvienai kontrolei, ir duomenų leksemos ateinančios prie sprendimo mazgo, yra priverčiamos pakraščius kirsti skersai. Jei sprendimo mazgas eina prieš kitą sprendimo mazgą, vadinasi, apsauga pirmiausia bus patikrinta pirmo sprendimo mazgo išvesties pakraščiuose ir ta pati leksema bus vėl bus tikrinama kitos apsaugos per antrojo sprendimo mazgo pakraščius. Kadangi galima patikrinti apsaugas vienoje vietoje, sprendimo mazgas einantis prieš kitą sprendimo mazgą gali būti pakeistas vienu sprendimo mazgu.



3.9 pav. Sprendimo mazgas prieš sprendimo mazgą

3.4.3.6 Susiliejimo mazgas prieš sprendimo mazgą

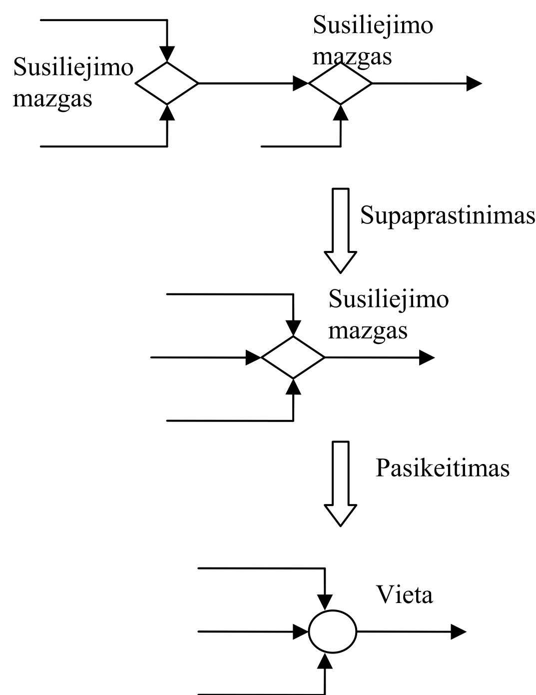
Susiliejimo mazgai naudojami daugialypių srautų sujungimui. Kontrolė ir duomenys, ateinantys prie susiliejimo mazgo, nedelsiant perkeliami į susiliejimo mazgo išvesties pakraščius. Jei po susiliejimo mazgo eina sprendimo mazgas, jis gali būti pakeistas vieta, nes ateinanti leksema pirmiausia bus vertinama pagal pakraščių apsaugas, ir šita leksemos atėjimo operacija ir apsaugos vertinimas gali būti atlikti viename mazge.



3.10 pav. Susiliejimo mazgas prieš sprendimo mazgą

3.4.3.7 Susiliejimo mazgas prieš susiliejimo mazgą

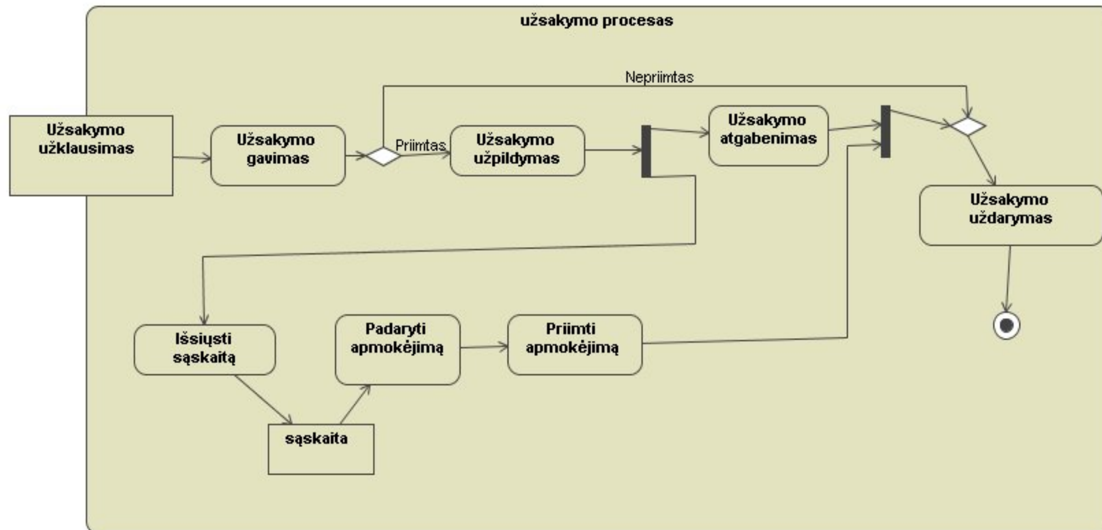
Jei po susiliejimo mazgo eina kitas susiliejimo mazgas, tai juos galima pakeisti vienu susiliejimo mazgu, nes ateinanti leksema pirmiausia bus nedelsiant pastumta pirmyn prie išvesties pakraščių, jei laikysimės prielaidos, kad mazgų apdorojimo laikas yra nereikšmingas.



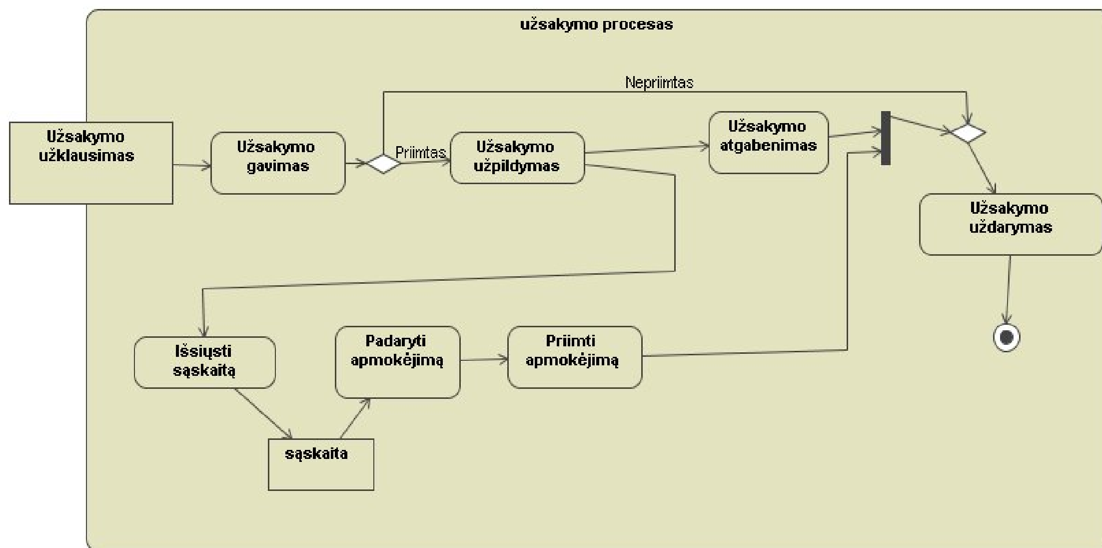
3.11 pav. Susiliejimo mazgas prieš susiliejimo mazgą

3.4.4 Pertvarkymo pavyzdžiai

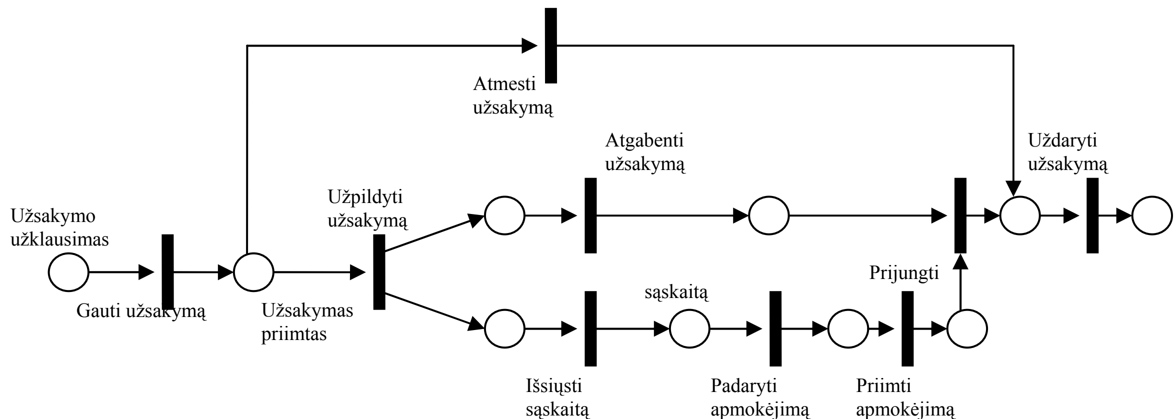
Veiklos diagrama 3.12 paveikslėlyje pristato verslo užsakymo apdorojimo apibrėžimą. Kriterijų mazgas, kuris gauna užsakymą, pavaizduotas kairėje diagramos pusėje. Užsakymo gavimas tuoj pat sąlygoja užsakymo gavimo veiksmą. Sprendimo mazgas po užsakymo gavimo vaizduoja užsakymo priėmimo arba užsakymo atmetimo sričių sąlygas. Po užpildyto užsakymo eina iššakojimo mazgas, kuris siunčia kontrolę sąskaitos išsiuntimui bei užsakymo gabenimui. Sujungimo mazgas parodo, kad kontrolė bus nusiųsta susiliejimo mazgui, kuriame yra užpildomas užsakymo gabenimas ir apmokėjimo priėmimas. Kadangi susiliejimo mazgas tik praleidžia leksemas tolyn, bus iškviesta užsakymo uždarymo veikla. (Kontrolė irgi siunčiama į užsakymo uždarymą, net jeigu užsakymas yra atmestas.) Kai užsakymo uždarymas įvykdytas, kontrolė siunčiama į veiklos pabaigą. 3.13 paveikslėlis parodo veiklos diagramas po supaprastinimo taisyklių panaudojimo. Užsakymo užpildymo veikla ir po jos einantis iššakojimo mazgas (pavaizduotas stačiakampio formos rėmais 3.12 paveikslėlyje) yra atstovaujami užsakymo užpildymo veiklos 3.13 paveikslėlyje.



3.12 pav. Užsakymo apdorojimo veiklos diagrama

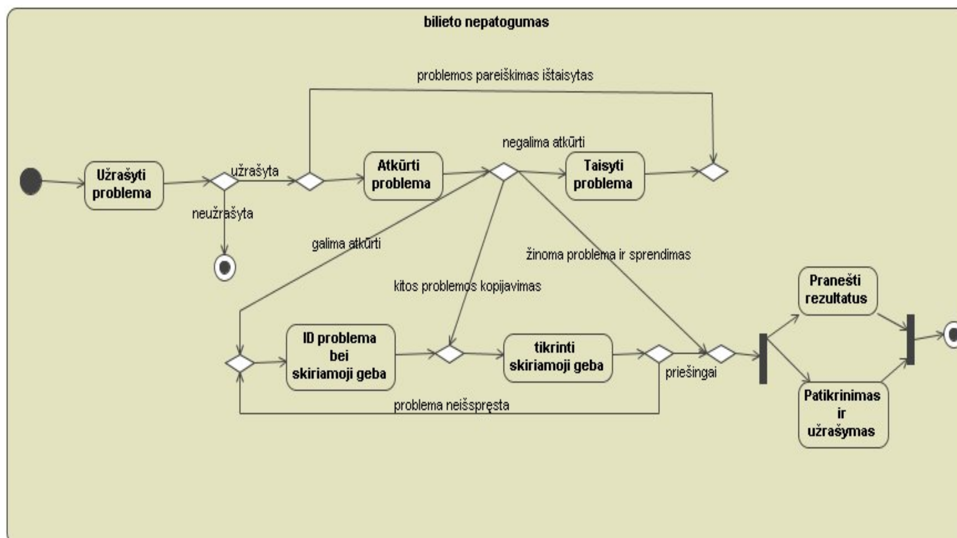


3.13 pav. Veiklos diagrama po užsakymo apdorojimo supaprastinimo

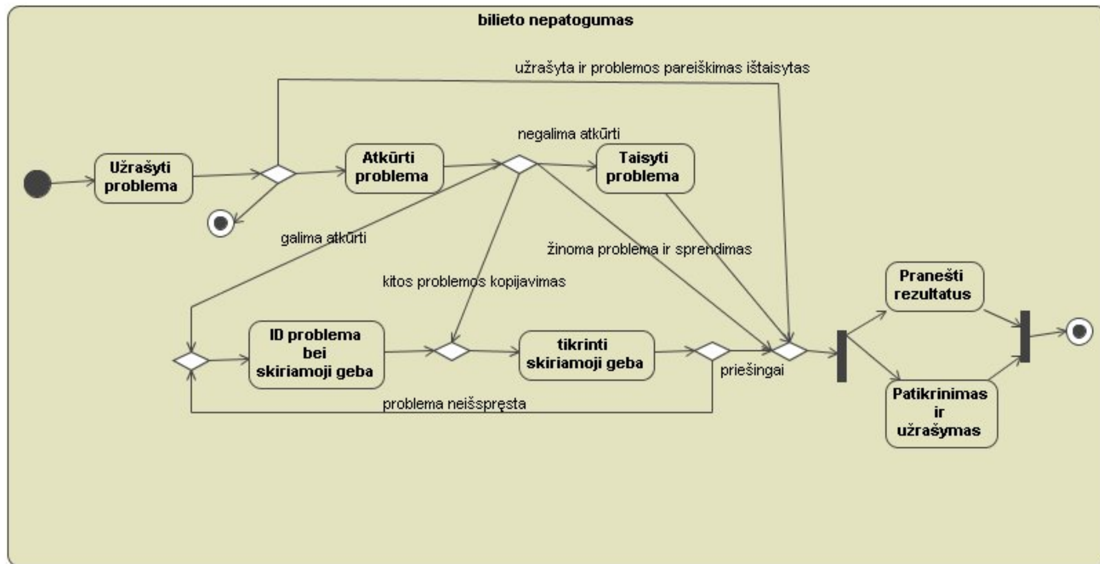


3.14 pav. Petri tinklas po užsakymo apdorojimo pertvarkymo

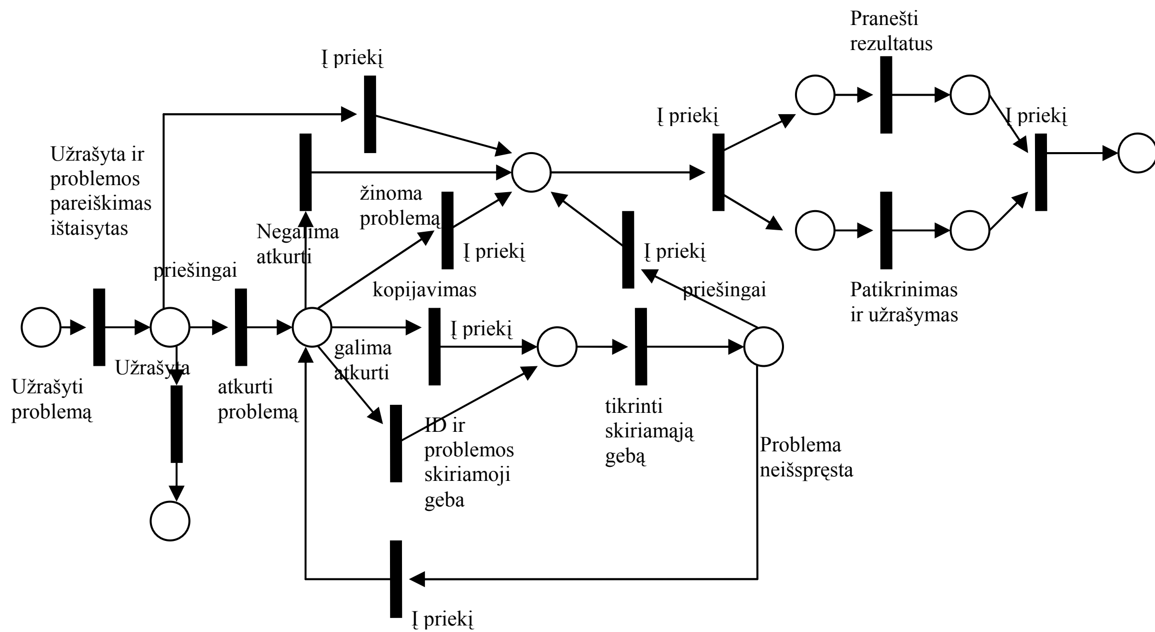
Veiklos diagrama 3.15 paveikslėlyje skirta bilietų nepatogumo problemai ir taip pat yra paimta iš UML2.0 specifikacijų. Bilietų nepatogumo scenarijus dengia kokybės patikrinimo ir klientų palaikymo grupes. Jei „infekcija“ arba „problema“ yra nustatoma, ji turi būti užregistruota, įrašas turi būti kruopščiai patikrintas; net ir mažiausios problemos atveju nustatoma pagrindinė priežastis, priimamas sprendimas, kuris turi būti nusiųstas atgal į problemos atsiradimo vietą. 3.16 paveikslėlis parodo veiklos diagramą po supaprastinimo taisyklių taikymo. Sprendimo mazgas po kurio eina dar vienas sprendimo mazgas (kaip parodyta stačiakampio formos rėmais 3.15 paveikslėlyje), buvo pakeisti vienu sprendimo mazgu.



3.15 pav. Veiklos diagrama probleminiam bilietui



3.16 pav. Veiklos diagrama po probleminio bilieto supaprastinimo



3.17 pav. Petri tinklas po probleminio bilieto pertvarkymo

3.4.5 Nežymimi elementai

Reikia pabrėžti, kad Petri tinklas nesiekia dubliuoti visos veiklos diagramose rastos informacijos. To pasekmė yra kelios koncepcijos, kurios yra unikalios veiksmų diagramose, bet jas nelengva pažymėti Petri tinkluose.
Tokių elementų sąrašas pateikiamas žemiau.

Veiklos galutinis mazgas: tai mazgas, kuris sustabdo visus veiklos srautus ir nedelsdamas nutraukia visą veiklą. Petri tinkluose nėra leksemų naikinimo užrašymo.

Veiklos pertraukimo regionas: skirtas palaikyti leksemų srautų nutraukimą į veiklos dalis. Kadangi Petri tinkluose nėra leksemų naikinimo užrašymo, šis regionas negali būti pažymėtas.

Išimčių prižiūrėtojas: yra apibrėžiamas kaip dalis, kuri veiks esant tiksliai nustatytoms išimtims. Petri tinkluose nėra išimčių priežiūros užrašymo.

3.4.6 Santrauka

Trečiame skyriuje buvo kalbėta apie perėjimo procesą iš veiklos diagramų į Petri tinklus. Buvo pristatytos veiklos diagramų pertvarkymo metodika į Petri tinklus. Po pertvarkymo iš veiklos diagramų į Petri tinklus taisyklių pristatymo buvo apibrėžtos veiklos diagramų supaprastinimo taisyklės. Šios taisyklės yra suderinamos su UML2.0 veiklos diagramų specifikacijomis. Pabaigoje buvo aptartos kelios koncepcijos, kurios yra unikalios veiklos diagramose ir Petri tinkluose nežymimos.

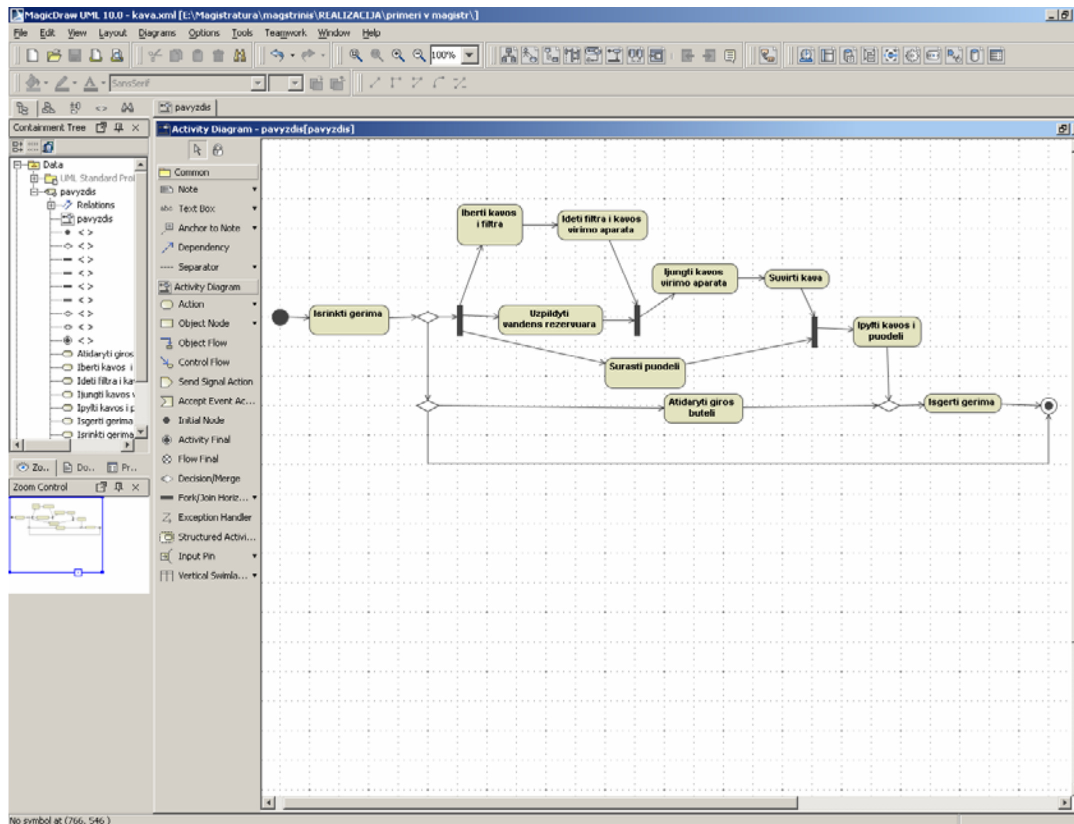
4. VEIKLOS DIAGRAMŲ MODELIAVIMO SISTEMOS SUDARYMAS IR TYRIMAS

4.1 Modeliavimo sistemos sudarymas

Atlikus modeliavimo sistemos analizę ir sudarant veiklos diagramos pertvarkymo taisykles į Petri tinklus, buvo nuspręsta padalinti sistemos sudarymą į tris dalis:

- ✓ Veiklos diagramos kūrimas, naudojant Magic Draw UML paketą;
- ✓ Veiklos diagramos pertvarkymas į Petri tinklus;
- ✓ Petri tinklų procesų simuliavimas;

Pirma sistemos sudarymo dalis yra kuriančio projekto veiklos diagramos braižymas, naudojant Magic Draw UML paketą. UML veiklos diagramoje galima panaudoti tik standartinius elementus, tokius kaip pradinis ir galutinis mazgai, veiksmas, objekto mazgas, objekto ir valdymo lankai, susijungimo/ išsišakojimo, sprendimo/susilietimo elementai. Veiklos diagramos braižymą, naudojant jau minėtą UML paketą, galima pamatyti 4.1 paveiksle.



4.1 pav. Veiklos diagrama Magic Draw UML redaktoriuje

Kaip buvo minėta 2 punkte, vienas iš didelių Magic Draw UML paketo privalumų yra nubraižytos diagramos išsaugojimas XML formatu, tai suteikia galimybę visą sistemą išskleisti elementais ir panaudoti juos nuskaitant Petri tinklo duomenis. Veiklos diagramos pertvarkymui ir Pertri tinklo simuliavimui sukurta modeliavimo sistema, naudojant C++ kalbą ir Borland C++ Builder 5 programavimo paketą bei panaudota XML Parser biblioteka. Šią biblioteką galima nemokai parsisiųsti iš ICOM firmos interneto svetainės (http://www.icom-dv.de/download/uk_download.php3).

XML Parser – tai XML komponentas procesoriaus, kuris analizuoja požymius ir nustato struktūrą bei kitas duomenų savybes. Šitoje bibliotekoje yra daug klasių, bet savo sistemoje mes naudojam tik keletą iš jų tik duomenų nuskaitymui ir jų sutvarkymui į duomenų medį. Pagrindiniai XML Parser bibliotekos panaudotos klases:

- ✓ *TIcXMLElement* – Klasė, kuri atstovauja XML elementui. Elemento tipas – tai elementų klasė su vienodomis savybėmis, įskaitant jų struktūrą ir elemento tipo vardą.
- ✓ *TIcXMLText* – klasė, kuri atstovauja tekstiniams duomenims XML dokumente.

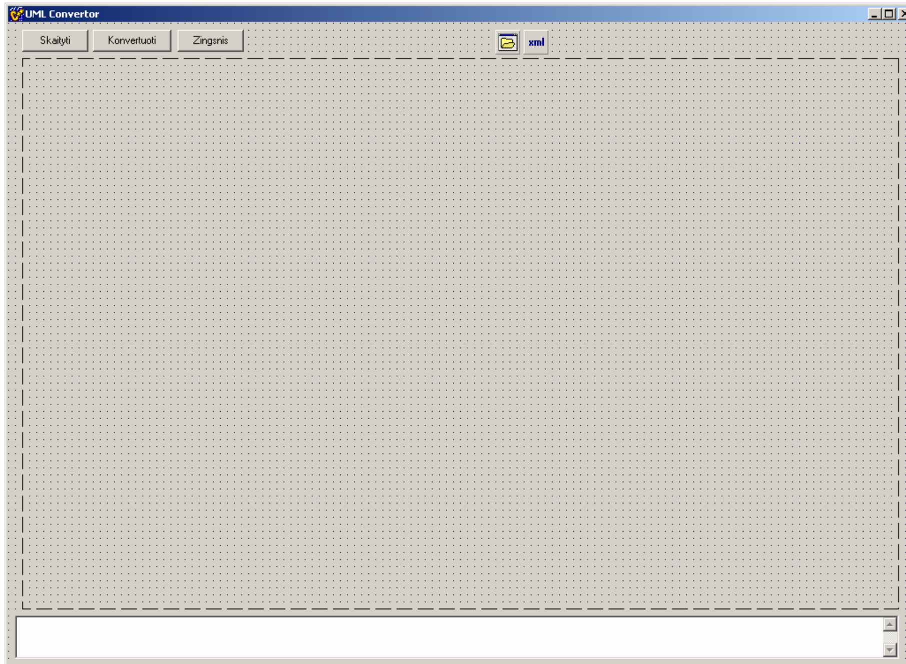
Tada galima išreikšti kiekvienos UML elemento duomenų saugojimo struktūrą:

```
struct UMLElement
{
  int type;
  int id;
  int fromid;
  int toid;
  char xmiid[255];
  char name[1000];
  char xmifrom[255];
  char xmito[255];
  int goesinto;
  int goesfrom;
  int geomcount;
  Geometry *geometries;
  UMLElement *next;
};
```

Duomenų struktūrą sudarančių kintamųjų aprašymas:

- ✓ `Type` – UML elemento tipas;
- ✓ `Id` – elemento identifikavimo numeris;
- ✓ `Fromid` – iš kokio elemento identifikavimo numerio ateina lankas;
- ✓ `Toid` – į kokį lelemento identifikavimo numerį išeina lankas;
- ✓ `Xmiid` – XMI identifikavimo numeris;
- ✓ `Name` – elemento vardas;
- ✓ `Xmifrom` – iš kokio XMI ateina ryšys;
- ✓ `Xmito` – į kokį xmi išeina ryšys;
- ✓ `Goesinto` – su kokiu elementu yra ryšys;
- ✓ `Goesfrom` – iš kokio elemento ateina ryšys;
- ✓ `Geometry` – elemento geometrinės koordinatės duomenų struktūra;

Sukurtos modeliavimo sistemos formos vaizdas parodytas 4.2 paveiksle.



4.2 pav. Modeliavimo sistemos formos vaizdas

Formoje panaudoti tokie pagrindiniai elementai:

TButton – skirtas mygtukams sudaryti;

TImage – skirtas duomenims bei rezultatam parodyti;

TOpenDialog – skirtas duomenų failui išsirinkti;

TIcXMLParser – skirtas UML failui nuskaityti ir išsaugoti;

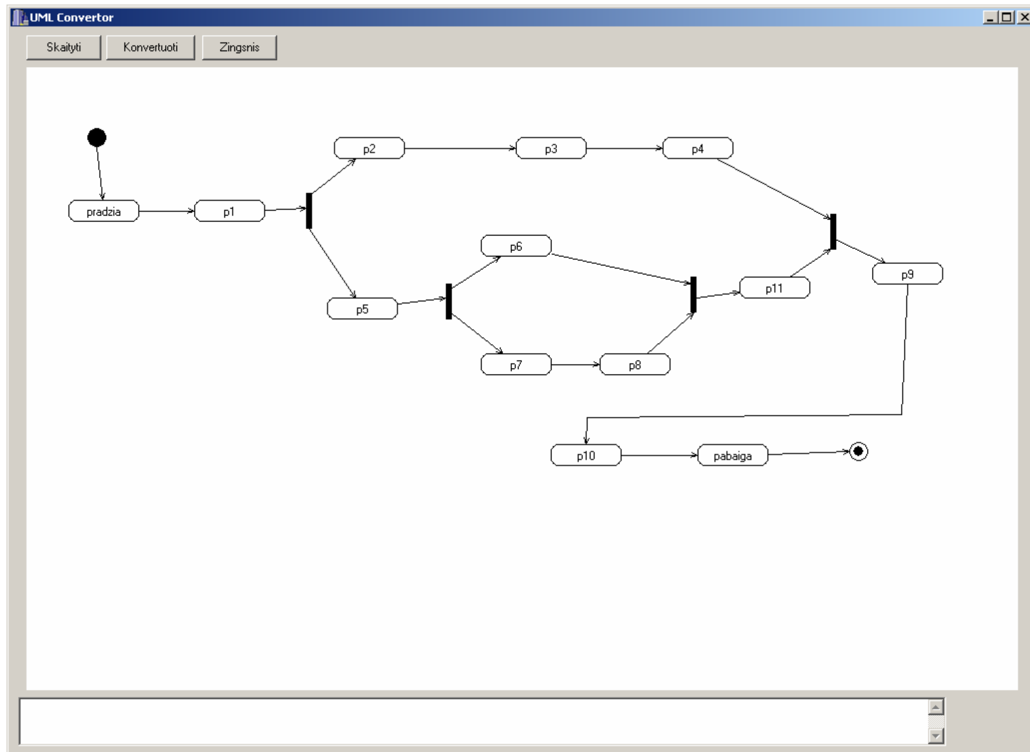
Sukurtoje modeliavimo sistemoje yra trys mygtukai (*Skaityti, Konvertuoti, Žingsnis*), kurie atstovauja tam tikroms procedūroms ir atlieka programos veiksmus.

Mygtuko *Skaityti* procedūra bei veiksmas:

```
void __fastcall TForm1::bnSkaitytiClick(TObject *Sender)
{
  TIcXMLElement *el1;
  TXMLUMLTree *temp;
  if (OpenDialog1->Execute())
  {
```

```
    placing = false;
    XMLParser->Parse(OpenDialog1->FileName, doc);
    el1 = doc->GetDocumentElement();
    Memo1->Clear();
    bnSkaityti->Enabled = false;
    //Actions
    XMLTreeParser->BuildTree(el1);
    temp   =   XMLTreeParser->FindByXMIType("uml:Activity",   XMLTreeParser-
>XMLTree);
    XMLTreeParser->BuildTree(temp->CharName); /* TODO : redo */
    Drawer->Draw(Image1, XMLTreeParser->result); /* TODO : getResult */
    //End of actions
    bnSkaityti->Enabled = true;
    Konvertuoti->Enabled = true;
  }
}
```

Procedūroje *bnSkaitytiClick* registruojami du kintamieji – tai *el1* – XML elemento kintamasis ir *temp* – XML elemento vieta medyje. Paspaudus mygtuką *Skaityti*, parodomas langas, kuriame reikia išrinkti XML duomenų failą, ir kuriame saugomi veiklos diagramos elementai. Procedūra nuskaito failą, perskaito XML dokumentą, išrenka UML elementus, kuriuo patalpina į medį, ir išveda UML veiklos diagramą į programos langą. Šio darbo rezultatą galima pamatyti 4.3 paveiksle.



4.3 pav. UML veiklos diagramos vaizdas modeliavimo sistemoje

Paspaudus mygtuką *Skaityti* ir pasirinkus duomenų failą, programa išveda veiklos diagramos brėžinį į programos langą, ir mygtukas *Konvertuoti* tampa aktyvus. Dabar modeliavimo sistema gali konvertuoti pagal pertvarkymo taisykles, kurios buvo aprašytos 3 punkte, į Petri tinklus.

Mygtukas *Konvertuoti* iškviečia procedūrą *KonvertuotiClick*:

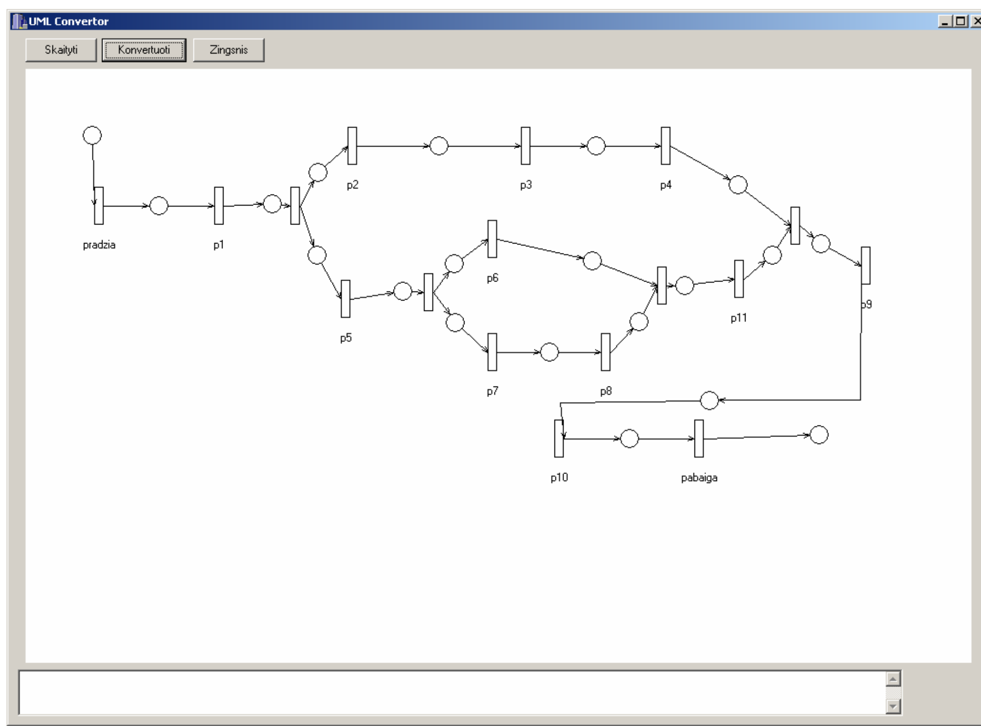
```
void __fastcall TForm1::KonvertuotiClick(TObject *Sender)
{
  PConv->Convert(XMLTreeParser->result);
  Drawer->Draw(Image1, PConv->result);
  placing = true;
  bnZingsnis->Enabled = true;
}
```

Procedūra *KonvertuotiClick* iškviečia procedūrą *Convert*, kuri yra *PetriConverter* klasės pagrindinė funkcija ir naudoja šios klasės kitas procedūras. Jos atlieka veiklos diagramos

konvertavimą į Petri tinklus. Taigi galima išryškinti procedūros *Convert* pagrindinius veiksmus:

- ✓ tikrina UML elemento tipą;
- ✓ jei elementas yra susilietimo/sprendimo, pradinės būsenos, baigtinės būsenos ar buferio tipo, tai konvertuojamas į paprastą Petri tinklo vietą;
- ✓ jei elementas yra veiksmo, sujungimo, išsišakojimo tipo, tai konvertuojamas į Petri tinklo perėjimą;
- ✓ randami visi UML lankai;
- ✓ jei du UML veiksmai buvo konvertuojami į paprastas Petri tinklo vietas, tai lankas tarp šių dviejų elementų paprastai išsaugomas;
- ✓ jei tarp dviejų UML veiksmų konvertavimo metu nuspręsta Petri tinkle padaryti perėjimą, tai apskaičiuojamas lanko ilgis arba lanko laužiamojo didesnė atkarpa ir dalinamas iš dviejų. Į gautą trūkumą įtraukiamas Petri tinklo perėjimo elementas;
- ✓ trinami nereikalingi po konvertavimo gauti elementai;

Atliekant pertvarkymą, procedūra *KonvertuotiClick* parodo gautą Petri tinklą programos lange, naudodama klasės *Drawer* procedūrą *Draw* (paveiklas 4.4).



4.4 pav. Petri tinklo vaizdas modeliavimo programoje

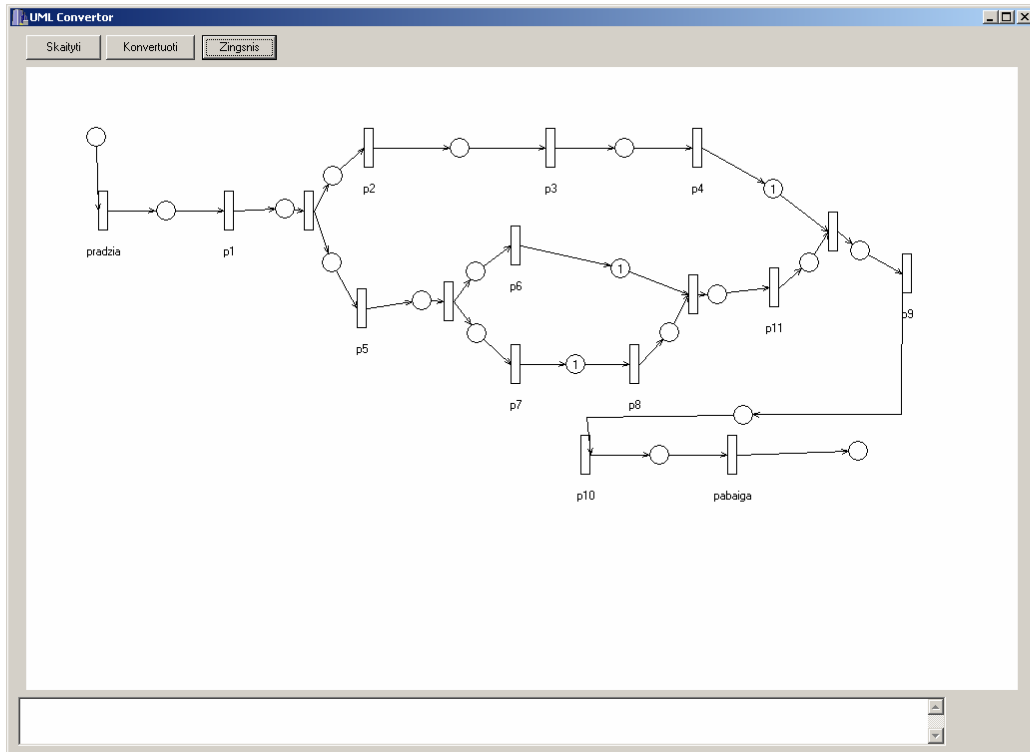
Dabar jau galima pamatyti Petri tinklo darbo simuliaciją, pažymėję mus dominančią vietą pelės kairiuoju mygtukų, mes nustatom tos vietos žymės skaičių. Paspaudę mygtuką *Žingsnis*, mes matysime žymių judėjimą žingsniais. Mygtuko *Žingsnis* iškviečiama procedūra *bnZingsnisClick*:

```
void __fastcall TForm1::bnZingsnisClick(TObject *Sender)
{
  Tester->RunStep(PConv->result);
  Drawer->Draw(Image1, PConv->result);
}
```

RunStep procedūra priklauso *TesterUnit* klasei bei atlieka šiuos veiksmus:

- ✓ apskaičiuoja kiekvienos tinklo vietos žymės skaičius dabartinėje būsenoje;
- ✓ išrenka tinklo vietas, kuriose kitu momentu turi pasirodyti žymės, ir apskaičiuoja jų kiekį;
- ✓ ištrina iš atitinkamų tinklo vietų žymės skaičius bei surašo naujus;

Procedūra *Draw* pavaizduoja kiekviename tinklo simuliacijos žingsnyje Petri tinklą. Tarpinės simuliacijos rezultatus galima pamatyti 4.5 paveiksle.



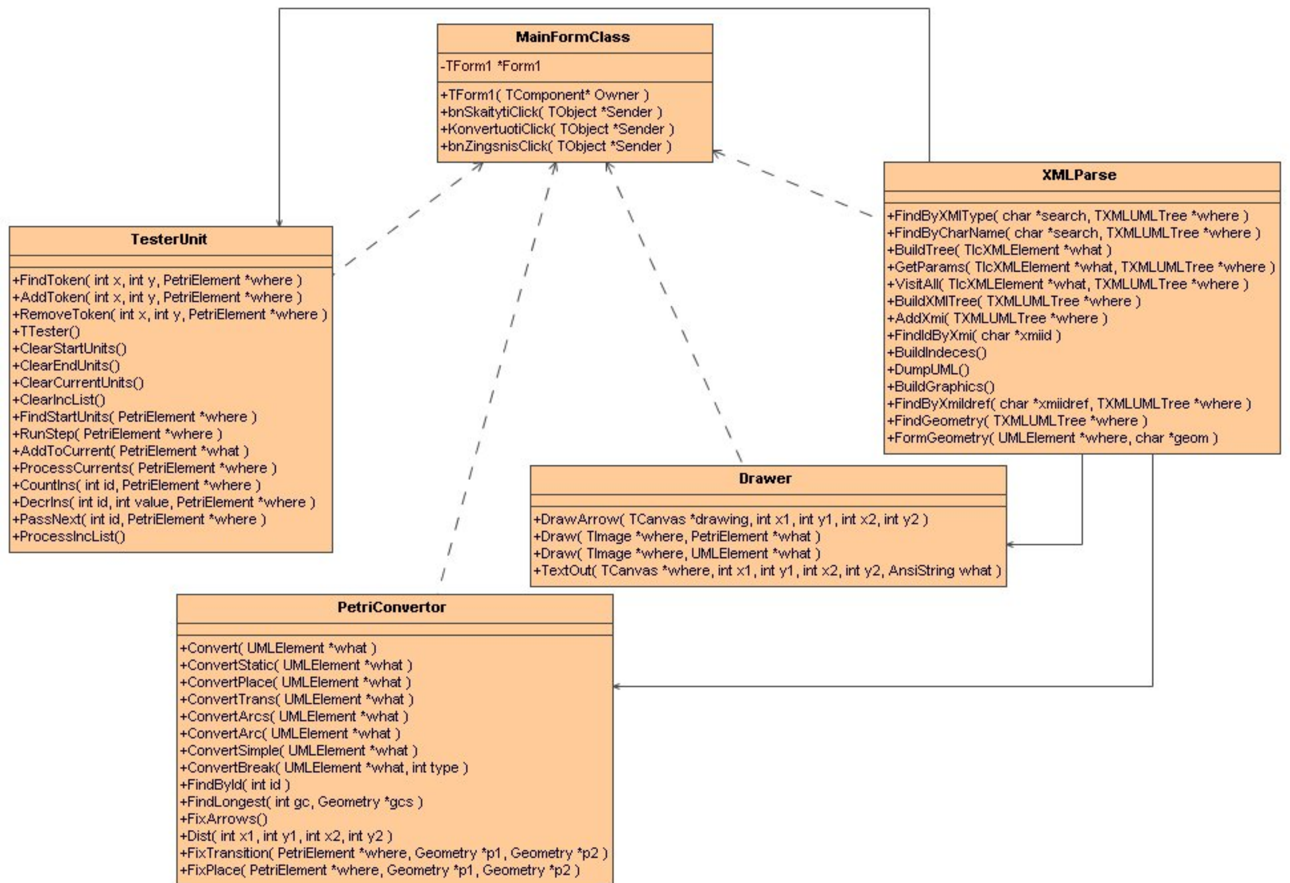
4.5 pav. Petri tinklo modeliavimo tarpiniai rezultatai

Spaudžiant ant Petri tinklo vietas su dešiniu pelės mygtuku, vietos žymės skaičius mažinamas vienetu. Taigi esant reikalui, bet kokiu simuliavimo momentu galima pridėti prie žymos skaičių prie bet kurios Petri tinklo vietos bei atimti esant reikalui. Tai suteikia galimybę simuliuoti ne tik visą Petri tinklą, bet ir, esant reikalui, tyrinėti bet kurią tinklo šaką.

Kad geriau būtų suprasta Petri tinklo sukurto simuliatoriaus veikimo ir sudarymo metodika, buvo sudaryta programos pagrindinių klasių diagrama (4.6 paveikslas), kur rodoma pagrindinių programoje sukurtos klasės tarpusavyje priklausomybes bei šioms klasėms priklausančios procedūros. *MainFormClass* yra pagrindinė klasė, kuri valdo programos veiksmus. Jai priklauso keturios pagrindinės klasės:

- ✓ *XMLParse* – saugo XML failo UML elementų duomenis medyje;
- ✓ *PetriConverter* – konvertuoja UML veiklos diagramą į Petri tinklą;
- ✓ *TesterUnit* – simuliuoja Petri tinklo veikimą;
- ✓ *Drawer* – braižo atitinkamus grafikus programos lange;

Bet klasės *TesterUnit, Drawer, PetriConverter* turi ryšius su *XMLParse* klase, iš kur jos naudoja UML elementų bei Petri tinklo duomenis.



4.6 pav. Modeliavimo sistemos klasių diagrama

4.2 Modeliavimo sistemos tyrimas

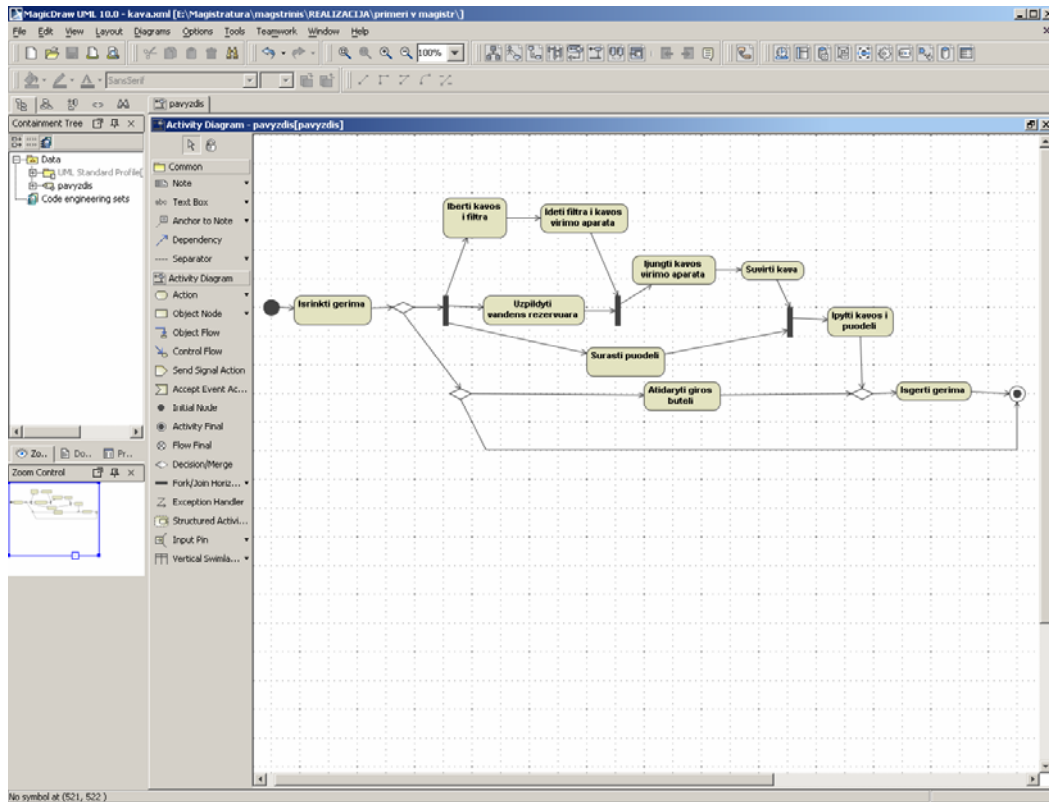
Veiklos diagramos modeliavimo sistemos tyrimas apima:

- ✓ sistemos galimybę nuskaityti Magic Draw UML paketo sugeneruotą XML failą;
- ✓ modeliavimo sistemos pritaikymas nuskaitytos veiklos diagramos parodymui;
- ✓ programos ergonomiškumo lygio tyrimas;
- ✓ sistemos veiklos diagramos konvertavimo į Petri tinklą galimybes;
- ✓ gauto Petri tinklo sudarančių elementų konvertavimo tikslumą;
- ✓ modeliavimo galimybes;

Visi tyrimą sudarantys punktai tikrinami su jau tampančiu klasikiniu pavyzdžiu „gėrimo išrinkimo procesas“, kuris apima gėrimo pasirinkimą ir ruošimą. Vienas iš didžiausių šio pavyzdžio privalumų yra lengvas supratimas, nes nereikalauja papildomo aiškinimo.

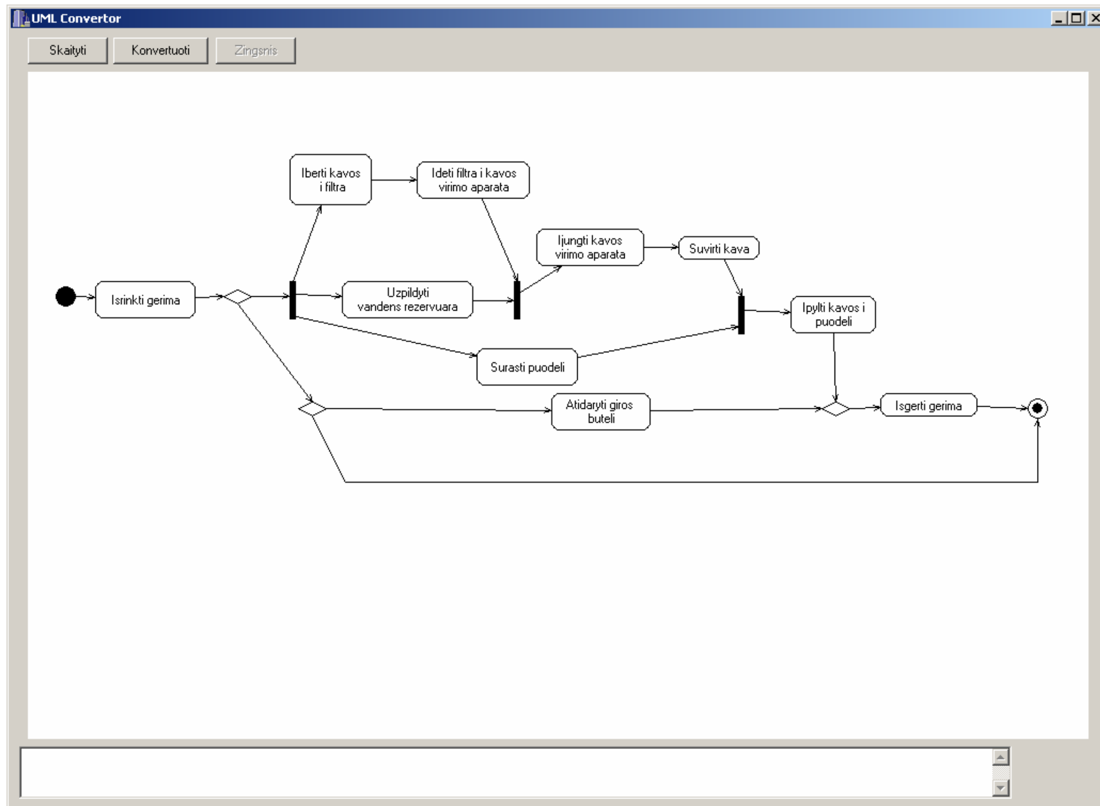
Veiklos diagramos modeliavimo sistema naudoja biblioteką XMLParser XML failui skaitymui, kurį generuoja Magic Draw UML paketas. Kaip buvo minėta sistemos sudarymo aprašyme, veiklos diagramos duomenys ne tik nuskaitomi iš sugeneruoto XML failo, bet ir iš karto saugomi medyje, tai palengvina bet kokio duomenų elemento priėmimą ir paiešką. Sudaryta programa nereikalauja kokių nors papildomų veiksnių veiklos diagramos išsaugojimui Magic Draw pakete. Tai suteikia galimybę braižyti bet kokio sudėtingumo veiklos diagramas, procesų lygiagretumo lygis irgi neapibrėžtas. Taigi projektuotojui, kuriančiam bet kokią projekto veiklos diagramą ir norinčiam sukonvertuoti ją į Petri tinklą modeliavimui, nereikia galvoti apie projekto modeliavimo sistemos nuskaitymo galimybių apribojimus, jis gali galvoti tik apie veiklos diagramos nubraižymą ir projekto aktualumą.

Paspaudus mygtuką Skaityti, programa išveda OpenDialog langą, kuriame reikia pasirinkti išsaugoti su Magic Draw UML paketu veiklos diagramos duomenų failą XML formatu. Atidarius failą, programos lange nubraižoma veiklos diagrama, kaip ir šaltinyje, tai yra išlaikoma veiklos diagramos visa struktūra, visi elementai turi tas pačias geometrines koordinates, kaip ir projektavimo programoje. Projektuotojui nereikia iš naujo analizuoti gautus veiklos diagramos elementus bei nagrinėti jų išdėstymą. Šio punkto patikrinimui buvo nubraižyta jau minėto pavyzdžio „gėrimo išrinkimo proceso“ veiklos diagramą Magic Draw UML programa (4.7 paveikslas).



4.7 pav. Gėrimo išrinkimo proceso veiklos diagrama Magic Draw UML redaktoriuje

4.7 paveiksle galima pamatyti, kad veiklos diagramos schema neperžengia iš programos lango ribų, dabar šios schemos išsaugotą duomenų failą XML formatu nuskaitome naudodamiesi sukurta modeliavimo sistema (4.8 paveikslas).

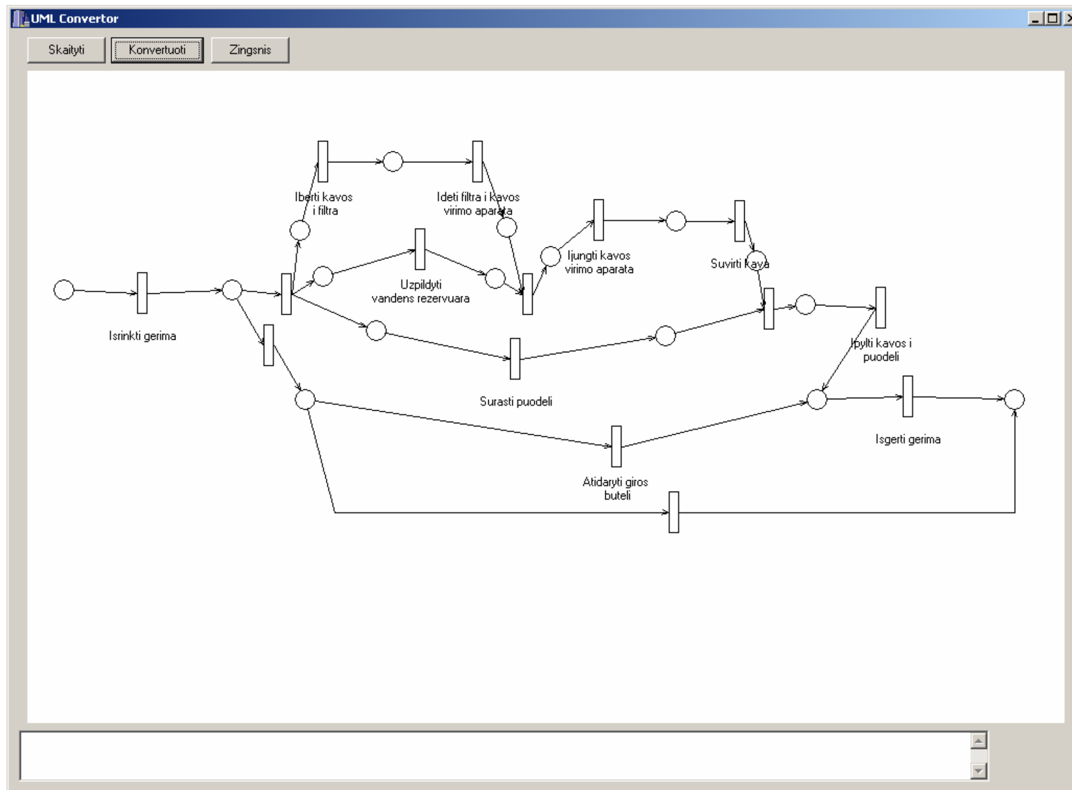


4.8 pav. Gėrimo išrinkimo proceso veiklos diagrama modeliavimo sistemoje

Palyginus veiklos diagramą Magic Draw UML programoje ir grafiškai pavaizduotą modeliavimo sistemoje, galima pasakyti, kad visa veiklos diagramos išdėstymo tvarka liko ta pati, visi elementai geometriškai atitinka savo vietas, ir projektuotojas, esant reikalui, lengvai suras jį dominančią veiklos diagramos dalį.

Dėl programos ergonomiškumo galima pasakyti, kad vartotojas lengvai gali suprasti mygtukų atliekamus darbus net jų pavadinimą. Visi sistemos vartotojo veiksmai apsaugoti nuo darbo eilės tvarkos sumaišymo, to mygtukas Konvertuoti tampa aktyvus tik po duomenų XML failo nuskaitymo. Tik pereinant prie Petri tinklo mygtukas Žingsnis tampa aktyvus bei suteikia galimybę sudėti visas reikalingas vietos žymes ir pradėti simuliuoti tinklą.

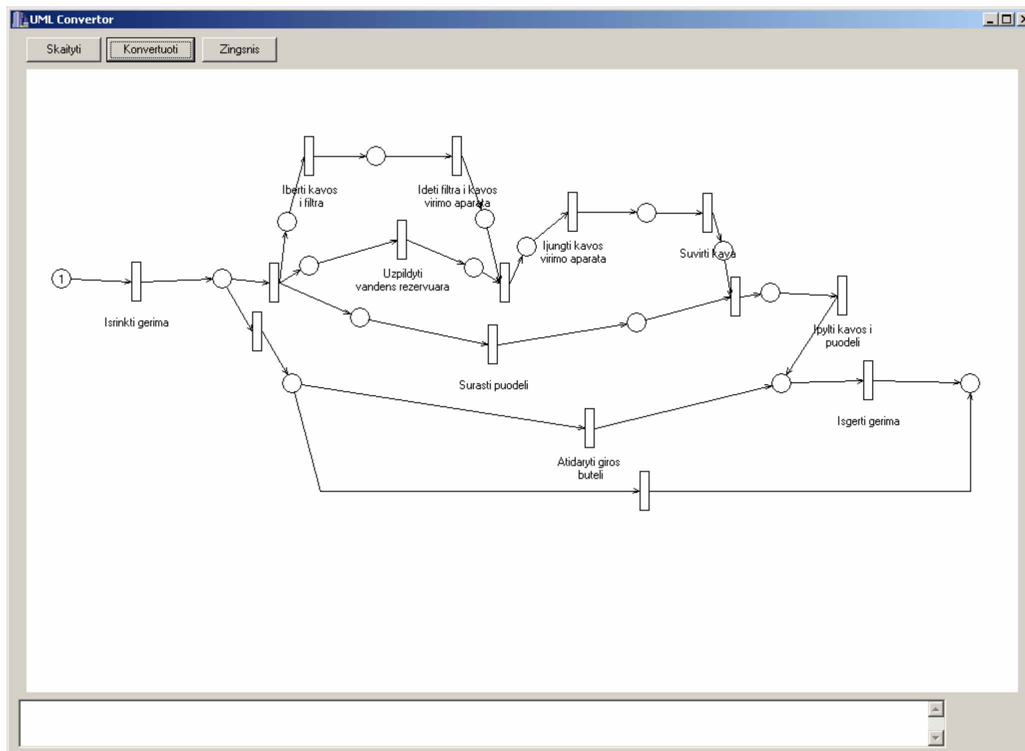
Konvertavimo galimybės neapribojamos veiklos diagramos išdėstymo ir jos sudėtingumo. Gėrimo pasirinkimo proceso gautą Petri tinklą galima pasižiūrėti 4.9 paveiksle.



4.9 pav. Gėrimo pasirinkimo proceso Petri tinklas

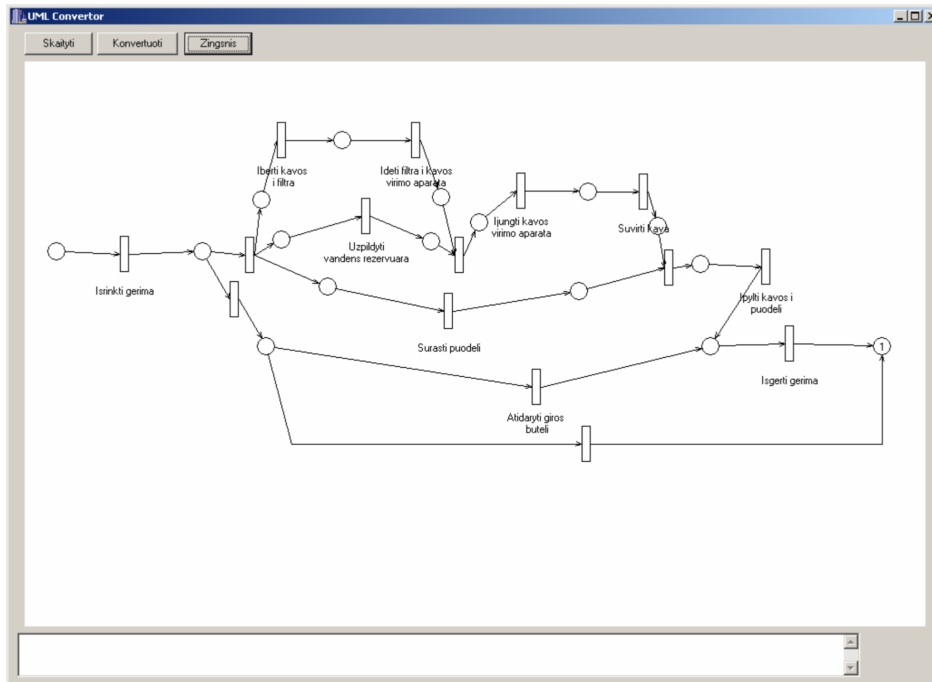
Gėrimo išrinkimo proceso gautas Petri tinklas atitinka norimą rezultatą, visi veiksmai sukonvertuoti teisingai pagal sudarytas konvertavimo taisykles bei parodomi programos išvedimo lange Petri tinkle. Kiekvienas veiklos diagramos elementas atsispindi gautame rezultate, taigi dėl veiklos diagramos sudėtingumo ir panaudotų skirtingų elementų konvertavimo procesas neapribojamas. Galima pasakyti, kad gautą Petri tinklą reikia supaprastinti, bet tai nebuvo pagrindinius šio darbo tikslas, nes tinklo pirminis variantas turi atitikti veiklos diagramos struktūrą, kad projektuotojas pažvelgęs į programos rezultatą, galėtų suprasti, kokie ir kur sukonvertuoti elementai, bei suprasti modeliuojamos sistemos kiekvienos šakos paskirtį. Iš tikrųjų tokiame pavyzdyje, kaip gėrimo pasirinkimo procesas yra per mažai elementų, kad būtų suprastas Petri tinklo perteklius bei supaprastinimo galimybės. Bet didinti nagrinėjamo pavyzdžio elementus nėra prasmės, nes pagrindinis šio darbo tikslas – parodyti sistemos modeliavimą, kad projektuotojas galėtų išnagrinėti sistemą bei surasti sistemos trūkumus. Norint Petri tinklą supaprastinti, galima panaudoti bet kokius jau esančius rinkoje Petri tinklo projektavimo paketus bei simuliatorius, kurie suteikia galimybę ne tik konstruoti tinklus, bet juos supaprastinti bei optimizuoti.

Modeliavimui pirmoje tinklo vietoje padaryta žymė, kad būtų patikrintas žymės kelias per atidarytos giros butelio šaką (paveikslas 4.10).



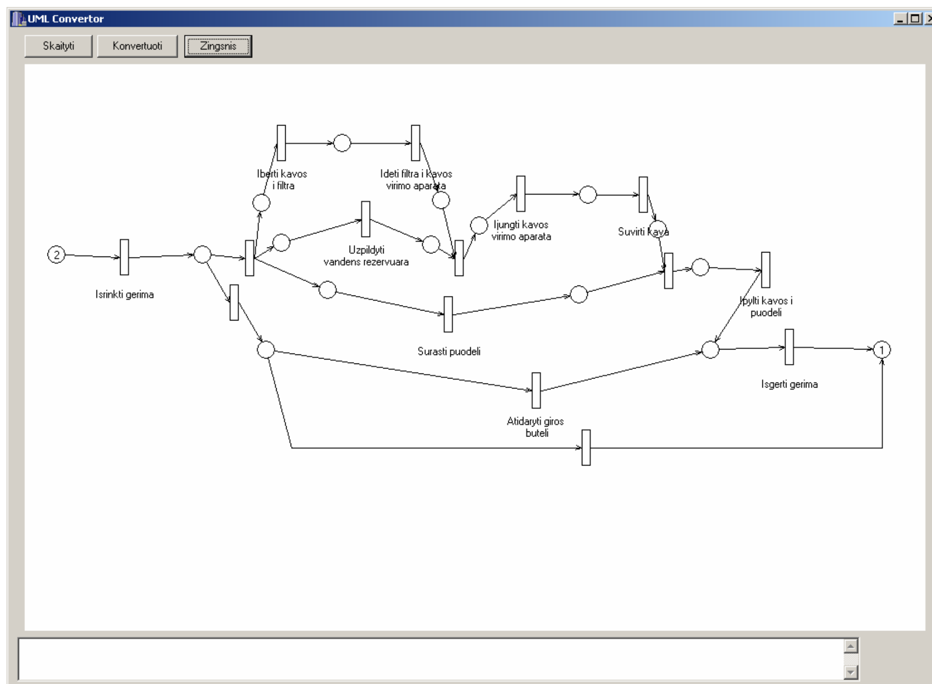
4.10 pav. Gėrimo pasirinkimo proceso simuliavimas

Paspaudę mygtuką Žingsnis, matome vienos Petri tinklo šakos darbą, nes, kaip pamename veiklos diagramos pradžioje buvo padarytas sprendimo mazgas. Modeliavimo paskutinę būseną galima pamatyti 4.11 paveiksle.



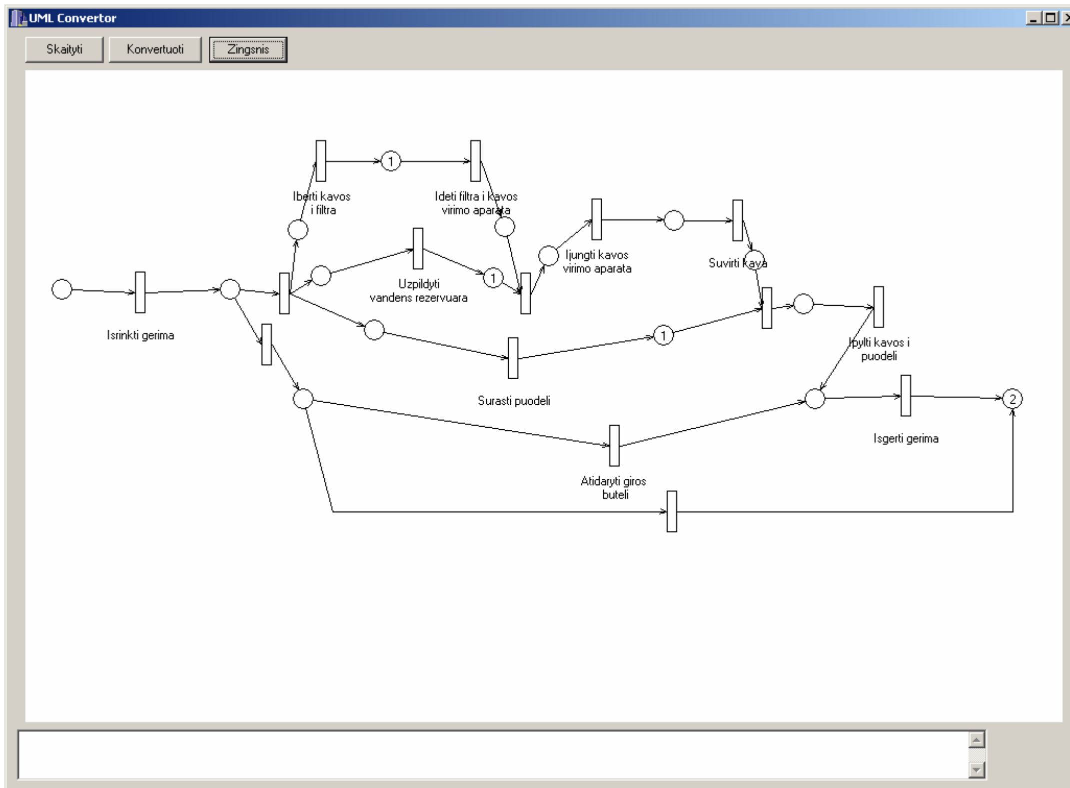
4.11 pav. Gėrimo pasirinkimo proceso modeliavimo pabaiga

Dabar pabaigoje jau yra viena žymė, bet norint paveikti visų Petri tinklo šakų darbą, reikia pradžioje sudėti dvi žymes (4.12 paveikslas). Be to, programa suteikia galimybę tai padaryti neištrynus senų duomenų.



4.12 pav. Petri tinklo visų šakų modeliavimas

Modeliuojant galima jau pamatyti kiekvienos šakos atlikimo greitį (4.13 paveikslas).



4.13 pav. Petri tinklo modeliavimo procesas

4.13 paveiksle yra tokia situacija, pagal kurią mes galime apibūdinti kiekvienos operacijos svarbumą bei greitį. Taigi užpildžius vandens rezervuarą, reikia laukti kol kitoje šakoje praeis procesas *įdėti filtrą į kavos virinimo aparatą*, bet tuo pat metu kitoje vietoje nebus padarytas procesas *įpilti kavos į pudelį*, kol nepraeis 5 iteracijos, bet, nepasirinkus nė vienos gėrimo rūšies, modeliavimas baigtas. Norint nagrinėti bet kokią tinklo šaką, galima bet kokiu metu pridėti žymių skaičių arba, priešingai, atimti.

Tyrimas parodė, kad visi reikalavimai, kurie buvo keliami modeliavimo sistemos sudarymui, buvo įvykdyti, sistema yra lengvai suprantama programos vartotojui, modeliavimo bei konvertavimo galimybės neapribojamos veiklos diagramos sudėtingumo ir jos elementų įvairovių. Visas modeliavimo procesas rodomas programos lange dinamiškai, bet vienas iš didžiausių šios modeliavimo sistemos trūkumų yra sukonvertuoto Petri tinklo optimizacijos apribojimas.

5. IŠVADOS

5.1 Bendos išvados

Šis darbas dar vienu žingsniu priartina projektuojamos sistemos modeliavimo procesą prie lengvai suprantamo, realiame laike dinamiškai kintančio modelio tyrimo. Buvo išnagrinėti UML kalbos redaktorių trūkumai, kurie neleidžia nagrinėti kuriamo projekto veikimą laike. Pasiūlyta naują sistemos veiklos diagramos modeliavimo sistema, kuri apima veiklos diagramos Magic Draw UML redaktorių ir modeliavimo programą. Realizuota nauja veiklos diagramos į Petri tinklą konvertavimo metodika. Viso darbo sudedamųjų dalių rezultatai yra tokie:

- ✓ Atlikta UML redaktorių galimybių ir funkcionalumo analizė, kurios metu buvo išrinktas Magic Draw UML redaktorius tolesniam modeliavimo sistemos sudarymui. Pagrindinė šio produkto paskirtis yra galimybė saugoti duomenis XML formatu, kur veiklos diagrama jau išskaidyta elementais. Be to, šis redaktorius pasižymi dideliu funkcionalumu bei paklausą Europos Sąjungos teritorijoje, o atskiri jo komponentai programuojami Lietuvoje. Visa tai suteikia galimybę tikėtis, kad šį produktą naudoja daug projektuotojų, kurie kurdami naudoja UML kalbą. Jiems nereikės, naudojant modeliavimo sistemą, prisitaikyti prie naujo UML redaktoriaus.
- ✓ Sistemos modeliavimui buvo pasirinktas Petri tinklas. Tai perspektyvus multiprograminių, asinchroninių, skirstomųjų, lygiagrečių, ir stochastinių informacijos apdorojimo sistemų aprašymo ir tyrimo instrumentas. Petri tinklas gali lengvai grafiškai pavaizduoti sistemos kitimą laike. Bet kokios projektuojamos sistemos modeliavimas, naudojant Petri tinklą, tampa lengvai suprantamas bei informatyvus.
- ✓ Darbe buvo pasiūlytos ne tik veiklos diagramos pertvarkymo taisyklės į Petri tinklą, bet ir konvertavimo metodika. Kiekvienas perkėlimo atvejis buvo nagrinėtas atskirai ir jam buvo užrašytos taisykles. Visos elementų transformacijos yra sugrupuotos ir aprašytos. Pateiktas perėjimo taisyklių teorinis pagrindimas.
- ✓ Veiklos diagramos modeliavimo sistemos sudaryme buvo suprogramuota vartotojo grafinė sąsaja, naudojant C++ kalbą. Konvertavimo programą suteikia galimybę pateikti duomenis XML formatu iš Magic Draw UML redaktoriaus be tarpinių veiksmų.

- ✓ Modeliavimo sistema leidžia ne tik konvertuoti veiklos diagramą į Petri tinklą bet ir simuliuoti modelį laike. Programoje duomenys pavaizduojami pradiniu formatu, tai yra parodoma veiklos diagrama. Konvertavimo metu į programos langą išvedamas Petri tinklo vaizdas. Yra galimybė simuliuoti visą tinklą po žingsnį arba tyrinėti bet kurią tinklo šaką. Bet kurio momentu prie bet kurios tinklo vietos žymės skaičių galima ir pridėti, ir atimti.
- ✓ Buvo ištirtas gėrimo pasirinkimo proceso pavyzdys. Pasitelkus jį buvo nustatytas sistemos elementų veikimas, išnagrinėtos atskiros šio projekto šakos. Pastebėti kai kurių šakų atlikimo laike trūkumai. Be to, išaiškintas didžiausias modeliavimo sistemos trūkumas – tai sukonvertuoto Petri tinklo optimizavimo apribojimas.

5.2 Numatoma plėtra

Veiklos diagramos modeliavimo sistemos tyrimo metu buvo pastebėti trūkumai, todėl jų eliminavimui siūlomi tokie darbai:

- ✓ Suprogramuoti galimybę išsaugoti bet kurio simuliavimo momentu Petri tinklą tolimesniam naudojimui. Tai gali būti naudinga perkeliant sukonvertuotą Petri tinklą į kitą kompiuterį arba modeliavimo sistemą.
- ✓ Sukurti Petri tinklo optimizavimo algoritmą bei jį realizuoti jau egzistuojančioje modeliavimo sistemoje. Esant labai dideliems projektams konvertavimo metu Petri tinklas tampa perteklinis, tai atima laiko jį simuliuojant.
- ✓ Suprogramuoti modeliavimo sistemoje galimybę bet kokiu momentų keisti Petri tinklo struktūrą, koreguoti elementus. Pridėdamas arba, atvirkščiai, ištrindamas tinklo elementus, projektuotojas turi galimybę analizuoti pakeistą projektą Petri tinkle, negrįždamas prie veiklos diagramos braižymo.
- ✓ Sukurti ir realizuoti Petri tinklo konvertavimo taisykles ir konvertavimo sistemą. Pakeitus Petri tinklo elementus arba tinklo struktūrą, projektuotojui nereikės perbraižyti veiklos diagramą, jis iš karto galės importuoti į UML redaktorių gautą veiklos diagramą.

6. LITERATŪROS SĄRAŠAS

[1] *Introduction to OMG UML* [interaktyvus]. – [Žiūrėta 2005 m. lapkričio 9 d.]. Prieiga per internetą: <http://www.omg.org/gettingstarted/what_is_uml.htm>

[2] *Published Modeling Specifications* [interaktyvus]. – [Žiūrėta 2005 m. lapkričio 9 d.]. Prieiga per internetą: <http://www.omg.org/technology/documents/spec_catalog.htm>

[3] Kazanavičius, E. *The Evaluation and Desingn Methodology for Real Time System.* Informatica, 2004, Vol. 15, No. 1, 45-62

[4] Lopez, Grao J.P., Merseguer J., Campos J. „*From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering*." (Proceedings of the Fourth International Workshop on Software and Performance (WOSP'04), ACM, Redwood City, California, USA, pages 25-36. January 2004

[5] Reutenauer, C. 1990. *The Mathematics of Petri Nets*. Masson and Prentice-Hall.

[6] Sathaye, A. S., and Krogh, B. H. Synthesis of real-time supervisors for controlled *IEEE Conf. on Decision and Control*, vol. 1, San Antonio, pp. 235–238.

[7] Sifakis, J. 1978. Structural properties of Petri nets. *Mathematical Foundations of Computer Science*, (Winkowski, J., ed.), Springer, pp. 474–483.

[8] Sreenivas, R. S., and Krogh, B. H. 1992. On Petri net models of infinite state supervisory. *IEEE Trans. On Automatic Control* 37(2): 274–277.

[9] Ushio, T. 1989. On the controllability of controlled Petri nets. *Control-Theory and Advanced Technology* 5(3): 265–275.

[10] Luis Alejandro Cortes, 2001. A Petri Net based Modeling and Verification Technique for Real-Time Embedded Systems

[11] Luis Alejandro Corts, Petru Eles, 2001 Verification of Real-Time Embedded Systems using Petri Net Models and Timed Automata

[12] S. Bartkevičius, 2003, Spalvotųjų Petri tinklų taikymas valdymo sistemoms modeliuoti

[13] Jensen Kurt. Springer, 1997, Coloured Petri Nets, basic concepts.

[14] Jančaitis E., Mačerauskas V., Šarkauskas K. Petri tinklai ir jų modeliavimas paketu CENTAURUS. // Konferencijos Automatika ir valdymo technologijos – 2001 pranešimų medžiaga. – Kaunas: Technologija. – 2001. – P. 186-190.

7. TERMINŲ IR SANTRUMPŲ ŽODYNAS

IT – informacinės technologijos

UML (Unified Modeling Language, *Vieninga modeliavimo kalba*) – modeliavimo ir specifikacijų kūrimo kalba, skirta specifikuoti, atvaizduoti ir konstruoti objektiškai orientuotų programų dokumentus.

CASE įrankiai (*Computer-aided software engineering* "kompiuterio padedama programų inžinerija") – programinė įranga ar programų paketai, skirti supaprastinti programų sistemų kūrimą ir palaikymą.

VHDL (VHSIC Hardware Description Language) – techninės įrangos aprašymo kalba.

OCL (Object Constraint Language) – objektų ryšio kalba.

XML (ang. eXtensible Markup Language) – W3C rekomenduojama bendros paskirties duomenų struktūrų bei jų turinio aprašomoji kalba.

ŽASPT – žymeklio apibendrintas stochastinis Petri tinklas.

GPS (*Global Positioning System*) - Visuotinė padėties nustatymo sistema, arba Globali pozicionavimo sistema. Leidžia nustatyti objekto koordinates bet kurioje pasaulio vietoje. Sistemos pagrindas - IT technologijų sąveika su planetą gaubiančiu GPS palydovų tinklu. Tai viena iš palydovinių navigacijos sistemų.

8. PRIEDAI

8.1 Programos tekstas

MainFormClass.cpp

```
#include <vcl.h>
#pragma hdrstop

#include "MainFormClass.h"

//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "IcXMLParser"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
  placing = running = false;
  XMLTreeParser = new TXMLTreeParser(Memo1);
  Drawer = new DiaDrawer();
  PConv = new TPetriConvertor();
  Tester = new TTester();
}
//---------------------------------------------------------------------------
void __fastcall TForm1::bnSkaitytiClick(TObject *Sender)
{
  TIcXMLElement *el1;
  TXMLUMLTree *temp;
  if (OpenDialog1->Execute())
  {
    placing = false;
    XMLParser->Parse(OpenDialog1->FileName, doc);
    el1 = doc->GetDocumentElement();
    Memo1->Clear();
    bnSkaityti->Enabled = false;
    //Actions
    XMLTreeParser->BuildTree(el1);
    temp = XMLTreeParser->FindByXMIType("uml:Activity", XMLTreeParser->XMLTree);
    XMLTreeParser->BuildTree(temp->CharName); /* TODO : redo */
    Drawer->Draw(Image1, XMLTreeParser->result); /* TODO : getResult */
    //End of actions
    bnSkaityti->Enabled = true;
    Konvertuoti->Enabled = true;
  }
}
//--------------------------------------------------------------------------

void __fastcall TForm1::Image1MouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if (placing)
  {
    if (Button == mbLeft)
    {
      Tester->AddToken(X, Y, PConv->result);
      Drawer->Draw(Image1, PConv->result);
    }
    if (Button == mbRight)
    {
      Tester->RemoveToken(X, Y, PConv->result);
      Drawer->Draw(Image1, PConv->result);
    }
  }
  else
    Memo1->Lines->Add("X: " + IntToStr(X) + AnsiString(" Y: ") + IntToStr(Y));
```

```
}
//---------------------------------------------------------------------------

void __fastcall TForm1::KonvertuotiClick(TObject *Sender)
{
  PConv->Convert(XMLTreeParser->result);
  Drawer->Draw(Image1, PConv->result);
  placing = true;
  bnZingsnis->Enabled = true;
  bnCiklas->Enabled = true;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::bnZingsnisClick(TObject *Sender)
{
  Tester->RunStep(PConv->result);
  Drawer->Draw(Image1, PConv->result);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::TimerTimer(TObject *Sender)
{
  if (running)
  {
    Tester->RunStep(PConv->result);
    Drawer->Draw(Image1, PConv->result);
  }
  if (!Tester->NotAtEnd(PConv->result) || Tester->SamePosition(PConv->result))
  {
    running = false;
    Timer->Enabled = false;
    Konvertuoti->Enabled = true;
    bnSkaityti->Enabled = true;
    bnZingsnis->Enabled = true;
    bnCiklas->Enabled = true;
  }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::bnCiklasClick(TObject *Sender)
{
  Konvertuoti->Enabled = false;
  bnSkaityti->Enabled = false;
  bnZingsnis->Enabled = false;
  bnCiklas->Enabled = false;
  Timer->Enabled = true;
  running = true;
}
//---------------------------------------------------------------------------
```

PetriConverter.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <math.h>
#pragma hdrstop

#include "PetriConvertor.h"
//---------------------------------------------------------------------------
void TPetriConvertor::Convert(UMLElement *what)
{
  result = NULL;
  maxid = 0;
  ConvertStatic(what);
  ConvertArcs(what);
  FixArrows();
}
//---------------------------------------------------------------------------
void TPetriConvertor::ConvertStatic(UMLElement *what)
{
  UMLElement *a = what;
  while (a != NULL)
```

```
  {
    if (a->geomcount > 0)
    {
      if (a->type == BUFFER || a->type == MERGE || a->type == DECISION || a->type ==
STARTNODE || a->type == FINALNODE && (a->goesinto > 0 || a->goesfrom > 0))
        ConvertPlace(a);
      if (a->type == ACTION || a->type == FORK || a->type == JOIN && (a->goesinto > 0 || a-
>goesfrom > 0))
        ConvertTrans(a);
    }
    a = a->next;
  }
}
//---------------------------------------------------------------------------
void TPetriConvertor::ConvertPlace(UMLElement *what)
{
  PetriElement *temp = new PetriElement;
  temp->type = PLACE;
  temp->tokens = 0;
  /* TODO : module */
  temp->id = what->id;
  if (temp->id > maxid) maxid = temp->id + 1;
  temp->inputs = what->goesinto;
  temp->outputs = what->goesfrom;
  temp->uni = 0;
  temp->uno = 0;
  temp->x = (what->geometries[0].x * 2 + what->geometries[1].x) >> 1;
  temp->y = (what->geometries[0].y * 2 + what->geometries[1].y) >> 1;
  strcpy(temp->name, what->name);
  temp->next = result;
  result = temp;
}
//---------------------------------------------------------------------------
void TPetriConvertor::ConvertTrans(UMLElement *what)
{
  PetriElement *temp = new PetriElement;
  temp->type = TRANSITION;
//  if (what->geometries[1].x < what->geometries[1].y)
    temp->position = VERTICAL;
//  else
//    temp->position = HORIZONTAL;
  /* TODO : module */
  temp->id = what->id;
  if (temp->id > maxid) maxid = temp->id + 1;
  temp->inputs = what->goesinto;
  temp->outputs = what->goesfrom;
  temp->uni = 0;
  temp->uno = 0;
  temp->x = (what->geometries[0].x * 2 + what->geometries[1].x) >> 1;
  temp->y = (what->geometries[0].y * 2 + what->geometries[1].y) >> 1;
  strcpy(temp->name, what->name);
  temp->next = result;
  result = temp;
}
//---------------------------------------------------------------------------
void TPetriConvertor::ConvertArcs(UMLElement *what)
{
  UMLElement *a = what;
  while (a != NULL)
  {
    if (a->geomcount > 0)
    {
      if (a->type == FLOW)
        ConvertArc(a);
    }
    a = a->next;
  }
}
//---------------------------------------------------------------------------
void TPetriConvertor::ConvertArc(UMLElement *what)
{
```

```
  PetriElement *from, *to;
  from = FindById(what->fromid);
  to = FindById(what->toid);
  if (from->type != to->type) //Place to place, transition to transition
    ConvertSimple(what);
  else
    ConvertBreak(what, from->type);
}
//---------------------------------------------------------------------------
void TPetriConvertor::ConvertSimple(UMLElement *what)
{
  int i;
  PetriElement *temp = new PetriElement;
  temp->type = ARC;
  temp->id = maxid;
  maxid++;
  temp->fromid = what->fromid;
  temp->toid = what->toid;
  strcpy(temp->name, what->name);
  temp->geomcount = what->geomcount;
  temp->geometries = new Geometry[temp->geomcount];
  for (i = 0; i < temp->geomcount; i++)
  {
     temp->geometries[i].x = what->geometries[i].x;
     temp->geometries[i].y = what->geometries[i].y;
  }
  temp->next = result;
  result = temp;
}
//---------------------------------------------------------------------------
void TPetriConvertor::ConvertBreak(UMLElement *what, int type)
{
  int i, lineid1 = maxid, lineid2 = maxid + 1, insertid = maxid + 2, breakpos, midx, midy;
  PetriElement *line1 = new PetriElement, *line2 = new PetriElement, *insert = new
PetriElement;
  maxid += 3;
  breakpos = FindLongest(what->geomcount, what->geometries);
  line1->id = lineid1;
  line2->id = lineid2;
  insert->id = insertid;
  line1->type = ARC;
  line2->type = ARC;
  if (type == PLACE)
  {
    insert->type = TRANSITION;
    insert->position = VERTICAL; /* TODO : review */
  }
  else
    insert->type = PLACE;
  line1->fromid = what->fromid;
  line2->fromid = insertid;
  line1->toid = insertid;
  line2->toid = what->toid;
  strcpy(line1->name, what->name);
  strcpy(line2->name, what->name);
  strcpy(insert->name, what->name);

  midx = (what->geometries[breakpos].x + what->geometries[breakpos + 1].x) >> 1;
  midy = (what->geometries[breakpos].y + what->geometries[breakpos + 1].y) >> 1;
  insert->x = midx;
  insert->y = midy;
  insert->tokens = 0;
  insert->inputs = 1;
  insert->outputs = 1;
  insert->uni = 0;
  insert->uno = 0;

  line1->geomcount = breakpos + 2;
  line1->geometries = new Geometry[breakpos + 2];
  for (i = 0; i < line1->geomcount - 1; i++)
  {
```

```
    line1->geometries[i].x = what->geometries[i].x;
    line1->geometries[i].y = what->geometries[i].y;
  }
  line1->geometries[breakpos + 1].x = midx;
  line1->geometries[breakpos + 1].y = midy;

  line2->geomcount = what->geomcount - breakpos;
  line2->geometries = new Geometry[what->geomcount - breakpos];
  line2->geometries[0].x = midx;
  line2->geometries[0].y = midy;
  for (i = 1; i < what->geomcount - breakpos; i++)
  {
    line2->geometries[i].x = what->geometries[breakpos + i].x;
    line2->geometries[i].y = what->geometries[breakpos + i].y;
  }

  line1->next = result;
  result = line1;
  line2->next = result;
  result = line2;
  insert->next = result;
  result = insert;
}
//---------------------------------------------------------------------------
PetriElement *TPetriConvertor::FindById(int id)
{
  PetriElement *a = result;
  while (a != NULL)
  {
    if (a->id == id)
      return a;
    a = a->next;
  }
  return NULL;
}
//---------------------------------------------------------------------------
int TPetriConvertor::FindLongest(int gc, Geometry *gcs)
{
  int retval = 0, maxval = Dist(gcs[0].x, gcs[0].y, gcs[1].x, gcs[1].y), i;
  for (i = 1; i < gc - 1; i++)
    if (Dist(gcs[i].x, gcs[i].y, gcs[i + 1].x, gcs[i + 1].y) > maxval)
    {
      maxval = Dist(gcs[i].x, gcs[i].y, gcs[i + 1].x, gcs[i + 1].y);
      retval = i;
    }
  return retval;
}
//---------------------------------------------------------------------------
void TPetriConvertor::FixArrows()
{
  PetriElement *a = result, *temp;
  while (a != NULL)
  {
    if (a->type == ARC)
    {
      if ((temp = FindById(a->fromid))->type == TRANSITION)
        FixTransition(temp, &(a->geometries[1]), &(a->geometries[0]));
      if ((temp = FindById(a->toid))->type == TRANSITION)
        FixTransition(temp, &(a->geometries[a->geomcount - 2]), &(a->geometries[a-
>geomcount - 1]));
    }
    a = a->next;
  }
  a = result;
  while (a != NULL)
  {
    if (a->type == ARC)
    {
      if ((temp = FindById(a->fromid))->type == PLACE)
        FixPlace(temp, &(a->geometries[1]), &(a->geometries[0]));
      if ((temp = FindById(a->toid))->type == PLACE)
```

```
        FixPlace(temp, &(a->geometries[a->geomcount - 2]), &(a->geometries[a->geomcount -
1]));
    }
    a = a->next;
  }
}
//---------------------------------------------------------------------------
int TPetriConvertor::Dist(int x1, int y1, int x2, int y2)
{
  return (x2-x1)*(x2-x1)+(y2-y1)*(y2-y1);
}
//---------------------------------------------------------------------------
void TPetriConvertor::FixTransition(PetriElement *where, Geometry *p1, Geometry *p2)
{
  if (where->position == VERTICAL) /* TODO : review, add horizontal */
  {
    if (p1->x > where->x)
      p2->x = where->x + THALFWIDTH;
    else
      p2->x = where->x - THALFWIDTH;
    p2->y = where->y;
  }
}
//---------------------------------------------------------------------------
void TPetriConvertor::FixPlace(PetriElement *where, Geometry *p1, Geometry *p2)
{
  double totaldist = sqrt(Dist(p1->x, p1->y, where->x, where->y));
  double delta = (totaldist - (double)PRADIUS) / totaldist;
  p2->x = where->x;
  p2->y = where->y;
  p2->x = (p2->x - p1->x) * delta + p1->x;
  p2->y = (p2->y - p1->y) * delta + p1->y;
}
#pragma package(smart_init)
```

TesterUnit.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "TesterUnit.h"
PetriElement *TTester::FindToken(int x, int y, PetriElement *where)
{
  PetriElement *retval = where;
  while (retval != NULL)
  {
    if (retval->type == PLACE)
      if ( ((x - retval->x)*(x - retval->x) + (y - retval->y)*(y-retval->y)) <
PRADIUS*PRADIUS)
        return retval;
    retval = retval->next;
  }
  return retval;
}
//---------------------------------------------------------------------------
void TTester::AddToken(int x, int y, PetriElement *where)
{
  PetriElement *target;
  if ((target = FindToken(x, y, where)) != NULL)
  {
    target->tokens++;
  }
}
//---------------------------------------------------------------------------
void TTester::RemoveToken(int x, int y, PetriElement *where)
{
  PetriElement *target;
  if ((target = FindToken(x, y, where)) != NULL)
```

```
  {
    if (target->tokens > 0)
      target->tokens--;
  }
}
//---------------------------------------------------------------------------
TTester::TTester()
{
  startunits = endunits = currentunits = inclist = NULL;
  pos = NULL;
}
//---------------------------------------------------------------------------
TTester::~TTester()
{
  ClearStartUnits();
  ClearEndUnits();
  ClearCurrentUnits();
  ClearPositions();
}
//---------------------------------------------------------------------------
void TTester::ClearStartUnits()
{
  PlaceList *b;
  while (startunits != NULL)
  {
    b = startunits->next;
    delete startunits;
    startunits = b;
  }
}
//---------------------------------------------------------------------------
void TTester::ClearEndUnits()
{
  /* TODO : module w/ clearstartunits */
  PlaceList *b;
  while (endunits != NULL)
  {
    b = endunits->next;
    delete endunits;
    endunits = b;
  }
}
//---------------------------------------------------------------------------
void TTester::ClearCurrentUnits()
{
  /* TODO : module w/ clearstartunits */
  PlaceList *b;
  while (currentunits != NULL)
  {
    b = currentunits->next;
    delete currentunits;
    currentunits = b;
  }
}
//---------------------------------------------------------------------------
void TTester::ClearIncList()
{
  /* TODO : module w/ clearstartunits */
  PlaceList *b;
  while (inclist != NULL)
  {
    b = inclist->next;
    delete inclist;
    inclist = b;
  }
}
//---------------------------------------------------------------------------
void TTester::FindStartUnits(PetriElement *where)
{
  PetriElement *search = where;
  PlaceList *temp;
```

```
  ClearStartUnits();
  while (search != NULL)
  {
    if (search->type == PLACE && search->tokens > 0 && search->outputs > 0)
    {
      temp = new PlaceList;
      temp->Place = search;
      temp->next = startunits;
      startunits = temp;
    }
    search = search->next;
  }
}
//---------------------------------------------------------------------------
int TTester::RunStep(PetriElement *where)
{
  PlaceList *finder1;
  PetriElement *finder2, *finder3;
  ClearIncList();
  FindStartUnits(where);
  if (startunits == NULL)
    return STEPNOTOKENS;
  ClearCurrentUnits();
  finder1 = startunits;
  while (finder1 != NULL)
  {
    finder2 = where;
    while (finder2 != NULL)
    {
      if (finder2->type == ARC && finder2->fromid == finder1->Place->id)
      {
        finder3 = where;
        while (finder3 != NULL)
        {
          if (finder3->id == finder2->toid)
            AddToCurrent(finder3);
          finder3 = finder3->next;
        }
      }
      finder2 = finder2->next;
    }
    finder1 = finder1->next;
  }
  ProcessCurrents(where);
  ProcessIncList();
  return STEPOK;
}
//---------------------------------------------------------------------------
void TTester::AddToCurrent(PetriElement *what)
{
  PlaceList *s = currentunits, *temp;
  while (s != NULL)
  {
    if (s->Place == what)
      return;
    s = s->next;
  }
  temp = new PlaceList;
  temp->Place = what;
  temp->next = currentunits;
  currentunits = temp;
}
//---------------------------------------------------------------------------
void TTester::ProcessCurrents(PetriElement *where)
{
  PlaceList *s;
  int k, delta;
  bool doloop = true;
  while (doloop)
  {
    doloop = false;
```

```
    s = currentunits;
    while (s != NULL)
    {
      delta = s->Place->inputs;
      k = CountIns(s->Place->id, where);
      if (k >= delta)
      {
        DecrIns(s->Place->id, delta, where);
        PassNext(s->Place->id, where);
        doloop = true;
      }
      s = s->next;
    }
  }
}
//---------------------------------------------------------------------------
int TTester::CountIns(int id, PetriElement *where)
{
  PetriElement *finder1 = where, *finder2;
  int sum = 0;
  while (finder1 != NULL)
  {
    if (finder1->type == ARC && finder1->toid == id)
    {
      finder2 = where;
      while (finder2 != NULL)
      {
        if (finder2->id == finder1->fromid)
          sum += finder2->tokens;
        finder2 = finder2->next;
      }
    }
    finder1 = finder1->next;
  }
  return sum;
}
//---------------------------------------------------------------------------
void TTester::DecrIns(int id, int value, PetriElement *where)
{
  PetriElement *finder1 = where, *finder2;
  int sum = value;
  while (finder1 != NULL)
  {
    if (finder1->type == ARC && finder1->toid == id)
    {
      finder2 = where;
      while (finder2 != NULL)
      {
        if (finder2->id == finder1->fromid && finder2->tokens > 0 && sum > 0)
        {
          finder2->tokens--;
          sum--;
        }
        finder2 = finder2->next;
      }
    }
    finder1 = finder1->next;
  }
}
//---------------------------------------------------------------------------
void TTester::PassNext(int id, PetriElement *where)
{
  PetriElement *finder1 = where, *finder2;
  PlaceList *temp;
  while (finder1 != NULL)
  {
    if (finder1->type == ARC && finder1->fromid == id)
    {
      finder2 = where;
      while (finder2 != NULL)
      {
```

```
        if (finder2->id == finder1->toid)
        {
          temp = new PlaceList;
          temp->Place = finder2;
          temp->next = inclist;
          inclist = temp;
        }
        finder2 = finder2->next;
      }
    }
    finder1 = finder1->next;
  }
}
//---------------------------------------------------------------------------
void TTester::ProcessIncList()
{
  PlaceList *t = inclist;
  while(t != NULL)
  {
    t->Place->tokens++;
    t = t->next;
  }
  ClearIncList();
}
//---------------------------------------------------------------------------
bool TTester::NotAtEnd(PetriElement *start)
{
  PetriElement *list = start;
  while (list != NULL)
  {
    if (list->type == PLACE && list->tokens > 0 && list->outputs > 0)
      return true;
    list = list->next;
  }
  return false;
}
//---------------------------------------------------------------------------
void TTester::ClearPositions()
{
  PositionList *a = pos, *b;
  while (a != NULL)
  {
    b = a->next;
    delete[] a->positions;
    delete a;
    a = b;
  }
}
//---------------------------------------------------------------------------
bool TTester::SamePosition(PetriElement *where)
{
  int i;
  PositionList *n = new PositionList, *k = pos;
  PetriElement *t = where;
  bool flag;
  if (pos == NULL)
  {
    placecount = 0;
    while (t != NULL)
    {
      if (t->type == PLACE)
        placecount++;
      t = t->next;
    }
    n->positions = new int[placecount];
    FillPos(where, n);
    n->next = pos;
    pos = n;
    return false;
  }
  else
```

```
  {
    n->positions = new int[placecount];
    FillPos(where, n);
    while (k != NULL)
    {
      flag = true;
      for (i = 0; i < placecount; i++)
        if (n->positions[i] != k->positions[i])
          flag = false;
      if (flag)
      {
        delete n;
        return true;
      }
      k = k->next;
    }
    n->next = pos;
    pos = n;
    return false;
  }
}
//---------------------------------------------------------------------------
void TTester::FillPos(PetriElement *where, PositionList *n)
{
  int i = 0;
  PetriElement *t = where;
  while (t != NULL)
  {
    if (t->type == PLACE)
    {
      n->positions[i] = t->tokens;
      i++;
    }
    t = t->next;
  }
}
#pragma package(smart_init)
```

Drawer.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <math.h>
#pragma hdrstop

#include "Drawer.h"

void DiaDrawer::DrawArrow(TCanvas *drawing, int x1, int y1, int x2, int y2)
{
  double translatedX, translatedY, originalX = 8, originalY = 3;
  double theta = atan2(y1-y2, x2-x1) + 3.1415;
  int lx, ly;

  translatedX = (originalX * cos(theta)) + (originalY * sin(theta));
  translatedY = (-originalX * sin(theta)) + (originalY * cos(theta));
  lx = x2 + (int)translatedX;
  ly = y2 + (int)translatedY;
  drawing->MoveTo(x2, y2);
  drawing->LineTo(lx, ly);
  originalX = 8;
  originalY = -3;
  translatedX = (originalX * cos(theta)) + (originalY * sin(theta));
  translatedY = (-originalX * sin(theta)) + (originalY * cos(theta));
  lx = x2 + (int)translatedX;
  ly = y2 + (int)translatedY;
  drawing->MoveTo(x2, y2);
  drawing->LineTo(lx, ly);

}
```

```
//---------------------------------------------------------------------------
void DiaDrawer::Draw(TImage *where, PetriElement *what)
{
  PetriElement *a = what;
  AnsiString TokenString, Caption;
  int i;
  TCanvas *drawing = where->Canvas;
  drawing->Brush->Color = clWhite;
  drawing->FillRect(Rect(0, 0, where->Width, where->Height));
  drawing->Pen->Color = clBlack;
  drawing->Brush->Style = bsClear;
  while (a != NULL)
  {
    switch (a->type)
    {
      case PLACE:
        Caption = a->name;
        TextOut(drawing, a->x, a->y + PRADIUS * 2, 0, 0, Caption);
        drawing->Ellipse(a->x - PRADIUS, a->y - PRADIUS, a->x + PRADIUS, a->y + PRADIUS);
        if (a->tokens > 0)
        {
          TokenString = IntToStr(a->tokens);
          drawing->TextOutA(a->x - (drawing->TextWidth(TokenString) >> 1) , a->y -
(drawing->TextHeight(TokenString) >> 1), TokenString);
        }
      break;
      case TRANSITION:
        Caption = a->name;
        TextOut(drawing, a->x, a->y + THALFHEIGHT * 2, 0, 0, Caption);
        if (a->position == VERTICAL)
          drawing->Rectangle(a->x - THALFWIDTH, a->y - THALFHEIGHT,a->x + THALFWIDTH, a->y
+ THALFHEIGHT);
        else
          drawing->Rectangle(a->x - THALFHEIGHT, a->y - THALFWIDTH,a->x + THALFHEIGHT, a->y
+ THALFWIDTH);
      break;
      case ARC:
        for (i = 0; i < a->geomcount - 1; i++)
        {
          drawing->MoveTo(a->geometries[i].x,a->geometries[i].y);
          drawing->LineTo(a->geometries[i + 1].x,a->geometries[i + 1].y);
        }
        if (a->geomcount > 0)
          DrawArrow(drawing, a->geometries[a->geomcount - 2].x, a->geometries[a->geomcount
- 2].y, a->geometries[a->geomcount - 1].x, a->geometries[a->geomcount - 1].y);
      break;

      default:;
    }
    a = a->next;
  }
}
//---------------------------------------------------------------------------
void DiaDrawer::Draw(TImage *where, UMLElement *what)
{
  UMLElement *a = what;
  int i;
  AnsiString Caption;
  TCanvas *drawing = where->Canvas;
  drawing->Brush->Color = clWhite;
  drawing->FillRect(Rect(0, 0, where->Width, where->Height));
  drawing->Pen->Color = clBlack;
  drawing->Brush->Style = bsClear;
  while (a != NULL)
  {

    switch (a->type)
    {
      //Flow
      case FLOW:
        for (i = 0; i < a->geomcount - 1; i++)
```

```
        {
          drawing->MoveTo(a->geometries[i].x,a->geometries[i].y);
          drawing->LineTo(a->geometries[i + 1].x,a->geometries[i + 1].y);
        }
        if (a->geomcount > 0)
          DrawArrow(drawing, a->geometries[a->geomcount - 2].x, a->geometries[a->geomcount
- 2].y, a->geometries[a->geomcount - 1].x, a->geometries[a->geomcount - 1].y);
      break;
      //Object
      case BUFFER:
        if (a->geomcount > 0)
        {
          drawing->Rectangle(a->geometries[0].x, a->geometries[0].y, a->geometries[0].x +
a->geometries[1].x, a->geometries[0].y + a->geometries[1].y);
          Caption = a->name;
          TextOut(drawing, a->geometries[0].x, a->geometries[0].y, a->geometries[1].x, a-
>geometries[1].y, Caption);
        }
      break;
      //Action
      case ACTION:
        if (a->geomcount > 0)
        {
          drawing->RoundRect(a->geometries[0].x, a->geometries[0].y, a->geometries[0].x +
a->geometries[1].x, a->geometries[0].y + a->geometries[1].y, 15, 15);
          Caption = a->name;
          TextOut(drawing, a->geometries[0].x, a->geometries[0].y, a->geometries[1].x, a-
>geometries[1].y, Caption);
        }
      break;
      //Fork &  Join
      case FORK:
      case JOIN:
        if (a->geomcount > 0)
        {
          drawing->Brush->Color = clBlack;
          drawing->Brush->Style = bsSolid;
          drawing->Rectangle(a->geometries[0].x, a->geometries[0].y, a->geometries[0].x +
a->geometries[1].x, a->geometries[0].y + a->geometries[1].y);
          drawing->Brush->Color = clWhite;
          drawing->Brush->Style = bsClear;
        }
      break;
      //Decision & Merge
      case DECISION:
      case MERGE:
        if (a->geomcount > 0)
        {
          drawing->MoveTo(a->geometries[0].x, a->geometries[0].y + (a->geometries[1].y >>
1));
          drawing->LineTo(a->geometries[0].x + (a->geometries[1].x >> 1), a-
>geometries[0].y);
          drawing->LineTo(a->geometries[0].x + a->geometries[1].x, a->geometries[0].y + (a-
>geometries[1].y >> 1));
          drawing->LineTo(a->geometries[0].x + (a->geometries[1].x >> 1), a-
>geometries[0].y + a->geometries[1].y);
          drawing->LineTo(a->geometries[0].x, a->geometries[0].y + (a->geometries[1].y >>
1));
        }
      break;
      case STARTNODE:
        if (a->geomcount > 0)
        {
          drawing->Brush->Color = clBlack;
          drawing->Brush->Style = bsSolid;
          drawing->Ellipse(a->geometries[0].x, a->geometries[0].y, a->geometries[0].x + a-
>geometries[1].x, a->geometries[0].y + a->geometries[1].y);
          drawing->Brush->Color = clWhite;
          drawing->Brush->Style = bsClear;
        }
      break;
```

```
      case FINALNODE:
        if (a->geomcount > 0)
        {
          drawing->Brush->Color = clBlack;
          drawing->Brush->Style = bsSolid;
          drawing->Ellipse((a->geometries[0].x * 4+ a->geometries[1].x) >> 2, (a-
>geometries[0].y * 4 + a->geometries[1].y) >> 2, (a->geometries[0].x * 4 + a-
>geometries[1].x  * 3) >> 2, (a->geometries[0].y * 4 + a->geometries[1].y * 3) >> 2);
          drawing->Brush->Color = clWhite;
          drawing->Brush->Style = bsClear;
          drawing->Ellipse(a->geometries[0].x, a->geometries[0].y, a->geometries[0].x + a-
>geometries[1].x, a->geometries[0].y + a->geometries[1].y);
        }
      break;

      default:;
    }
    a = a->next;
  }
}
//---------------------------------------------------------------------------
void DiaDrawer::TextOut(TCanvas *where, int x1, int y1, int x2, int y2, AnsiString what)
{
  int i, w, h, delta = 0;
  AnsiString temp;
  if (what.Pos("\n") > 0)
  {
    temp = what;
    while ((i = temp.Pos("\n")) > 0)
    {
      delta += where->TextHeight(temp.SubString(1, i - 1));
      temp = temp.SubString(i+1, temp.Length() - 2);
    }
    delta += where->TextHeight(temp);
    delta  = delta >> 1;
    temp = what;
    while ((i = temp.Pos("\n")) > 0)
    {
      TextOut(where, x1, y1 - delta + (where->TextHeight(temp.SubString(1, i - 1)) >> 1),
x2, y2, temp.SubString(1, i - 1));
      delta -= where->TextHeight(temp.SubString(1, i - 1));
      temp = temp.SubString(i+1, temp.Length() - 2);
    }
    TextOut(where, x1, y1 - delta + (where->TextHeight(temp) >> 1), x2, y2, temp);
  }
  else
  {
    h = where->TextHeight(what);
    w = where->TextWidth(what);
    where->TextOutA(x1 + (x2 >> 1) - (w >> 1), y1 + (y2 >> 1) - (h >> 1), what);
  }
}
//---------------------------------------------------------------------------
#pragma package(smart_init)
```

XMLParse.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "XMLParse.h"

//---------------------------------------------------------------------------
TXMLTreeParser::TXMLTreeParser()
{
  TXMLTreeParser(NULL);
}
//---------------------------------------------------------------------------
```

```
TXMLUMLTree *TXMLTreeParser::FindByXMIType(char *search, TXMLUMLTree *where)
{
/* TODO : MODULE */
  TXMLUMLTree *a = where;
  while (a) //Visiting all the tree elements from starting from where+1
  {
    if (a->Children != NULL)
      a = a->Children;
    else
      if (a->Next != NULL)
        a = a->Next;
      else
      {
        while(a->Parent->Next == NULL)
        {
          if (a->Parent != NULL)
            a = a->Parent;
          else
            return NULL;
          if (a->Next == NULL && a->Parent == NULL)
            return NULL;
        };
        a = a->Parent->Next;
      }
    if (!strcmp(a->XMIType, search))
      return a;
  }
  return NULL;
}
//---------------------------------------------------------------------------
TXMLUMLTree *TXMLTreeParser::FindByCharName(char *search, TXMLUMLTree *where)
{
/* TODO : MODULE */
  TXMLUMLTree *a = where;
  while (a) //Visiting all the tree elements from starting from where+1
  {
    if (a->Children != NULL)
      a = a->Children;
    else
      if (a->Next != NULL)
        a = a->Next;
      else
      {
        while(a->Parent->Next == NULL)
        {
          if (a->Parent != NULL)
            a = a->Parent;
          else
            return NULL;
          if (a->Next == NULL && a->Parent == NULL)
            return NULL;
        };
        a = a->Parent->Next;
      }
    if (!strcmp(a->CharName, search))
      return a;
  }
  return NULL;
}
//---------------------------------------------------------------------------
TXMLTreeParser::TXMLTreeParser(TMemo *TM)
{
  XMLTree = NULL;
  Dump = TM;
}
//---------------------------------------------------------------------------
void TXMLTreeParser::BuildTree(TIcXMLElement *what)
{
  if (XMLTree != NULL)
    delete XMLTree;
  XMLTree = new TXMLUMLTree(NULL);
```

```
  VisitAll(what, XMLTree);
}
//---------------------------------------------------------------------------
void TXMLTreeParser::GetParams(TIcXMLElement *what, TXMLUMLTree *where)
{
  AnsiString temp;
  TIcXMLText *temp2;
  strcpy(where->Name, what->GetName().c_str());
  strcpy(where->Value, what->GetValue().c_str());
  if (!strcmp(where->Name, "geometry"))
    if ((temp2 = what->GetFirstCharData()) != NULL)
      strcpy(where->Value, temp2->GetValue().c_str());
  strcpy(where->XMIId, what->GetAttribute("xmi:id").c_str());
  strcpy(where->XMIType, what->GetAttribute("xmi:type").c_str());
  strcpy(where->Source, what->GetAttribute("source").c_str());
  strcpy(where->Target, what->GetAttribute("target").c_str());
  strcpy(where->XMIValue, what->GetAttribute("xmi:value").c_str());
  strcpy(where->XMIIdRef, what->GetAttribute("xmi:idref").c_str());
  strcpy(where->ElementClass, what->GetAttribute("elementClass").c_str());
  strcpy(where->CharName, what->GetAttribute("name").c_str());
}
//---------------------------------------------------------------------------
void TXMLTreeParser::VisitAll(TIcXMLElement *what, TXMLUMLTree *where)
{
  TXMLUMLTree *temp;
  TIcXMLElement *next;

  GetParams(what, where);
  Application->ProcessMessages();
  //Visiting children
  next = what->GetFirstChild();
  if (next != NULL)
  {
    temp = new TXMLUMLTree(where);
    VisitAll(next, temp);
  }
  //Visiting siblings
  next = what->NextSibling();
  if (next != NULL)
  {
    temp = new TXMLUMLTree(where->Parent);
    VisitAll(next, temp);
  }
}
//---------------------------------------------------------------------------
UMLElement *TXMLTreeParser::BuildTree(char *schema)
{
  TXMLUMLTree *head = FindByCharName(schema, XMLTree), *searchpos;
  result = NULL;
  idnum = 0;
  //finding XMI elements
  searchpos = head->Children;

  BuildXMITree(searchpos);
  BuildGraphics();
  BuildIndeces();
  return result;
}
//---------------------------------------------------------------------------
void TXMLTreeParser::BuildXMITree(TXMLUMLTree *where)
{

  if (strcmp(where->XMIType, "uml:ActivityFinalNode") == 0)
  {
    AddXmi(where);
    result->type = FINALNODE;
  }

  if (strcmp(where->XMIType, "uml:InitialNode") == 0)
  {
    AddXmi(where);
```

```
    result->type = STARTNODE;
  }

  if (strcmp(where->XMIType, "uml:CallBehaviorAction") == 0)
  {
    AddXmi(where);
    result->type = ACTION;
  }

  if (strcmp(where->XMIType, "uml:DecisionNode") == 0)
  {
    AddXmi(where);
    result->type = DECISION;
  }

  if (strcmp(where->XMIType, "uml:ForkNode") == 0)
  {
    AddXmi(where);
    result->type = FORK;
  }

  if (strcmp(where->XMIType, "uml:CentralBufferNode") == 0)
  {
    AddXmi(where);
    result->type = BUFFER;
  }

  if (strcmp(where->XMIType, "uml:ObjectFlow") == 0 || strcmp(where->XMIType,
"uml:ControlFlow") == 0 )
  {
    AddXmi(where);
    result->type = FLOW;
    strcpy(result->xmifrom, where->Source);
    strcpy(result->xmito, where->Target);
  }

  if (where->Children != NULL)
    BuildXMITree(where->Children);
  if (where->Next != NULL)
    BuildXMITree(where->Next);

}
//---------------------------------------------------------------------------
void TXMLTreeParser::AddXmi(TXMLUMLTree *where)
{
  UMLElement *temp;
  temp = new UMLElement;
  temp->next = result;
  temp->id = idnum;
  idnum++;
  result = temp;
  strcpy(result->name, where->CharName);
  strcpy(result->xmiid, where->XMIId);
}
//---------------------------------------------------------------------------
int TXMLTreeParser::FindIdByXmi(char *xmiid)
{
  UMLElement *temp = result;
  while (temp != NULL)
  {
    if (strcmp(temp->xmiid, xmiid) == 0)
      return temp->id;
    temp = temp->next;
  }
  return -1;
}
//---------------------------------------------------------------------------
void TXMLTreeParser::BuildIndeces()
{
  UMLElement *temp = result, *temp2;
  while (temp != NULL)
```

```
  {
    if (temp->type == FLOW)
    {
      temp->fromid = FindIdByXmi(temp->xmifrom);
      temp->toid = FindIdByXmi(temp->xmito);
    }
    temp = temp->next;
  }
  temp = result;
  while (temp != NULL)
  {
    if (temp->type != FLOW)
    {
      temp->goesinto = 0;
      temp->goesfrom  = 0;
      temp2 = result;
      while (temp2 != NULL)
      {
        if (temp2->type == FLOW && temp2->geomcount > 0)
        {
          if (temp2->fromid == temp->id)
            temp->goesfrom++;
          if (temp2->toid == temp->id)
            temp->goesinto++;
        }
        temp2 = temp2->next;
      }
    }
    temp = temp->next;
  }
}
//---------------------------------------------------------------------------
void TXMLTreeParser::DumpUML()
{
  UMLElement *temp = result;
  while (temp != NULL)
  {
    Dump->Lines->Add("id: " + IntToStr(temp->id));
    Dump->Lines->Add("Name: " + AnsiString(temp->name));
    if (temp->type == FLOW)
    {
      Dump->Lines->Add("From: " + IntToStr(temp->fromid));
      Dump->Lines->Add("TO: " + IntToStr(temp->toid));
    }
    else
    {
      Dump->Lines->Add("Goes from: " + IntToStr(temp->goesfrom));
      Dump->Lines->Add("Goes to: " + IntToStr(temp->goesinto));
    }

    temp = temp->next;
  }
}
//---------------------------------------------------------------------------
void TXMLTreeParser::BuildGraphics()
{
  UMLElement *temp = result;
  TXMLUMLTree *finder;
  while (temp != NULL)
  {
    if ((finder = FindByXmiIdref(temp->xmiid)) == NULL)
      temp->geomcount = 0;
    else
      if ((finder = FindGeometry(finder->Parent)) == NULL)
        temp->geomcount = 0;
      else
        FormGeometry(temp, finder->Value);
    temp = temp->next;
  }
}
//---------------------------------------------------------------------------
```

```
TXMLUMLTree *TXMLTreeParser::FindByXmiIdref(char *xmiidref)
{
  /* TODO : Search from model fraphics start, not the root!!! or by hash!!! */
  return FindByXmiIdref(xmiidref, XMLTree);
}
//---------------------------------------------------------------------------
TXMLUMLTree *TXMLTreeParser::FindByXmiIdref(char *xmiidref, TXMLUMLTree *where)
{
/* TODO : MODULE */
  TXMLUMLTree *a = where;
  while (a) //Visiting all the tree elements from starting from where+1
  {
    if (a->Children != NULL)
      a = a->Children;
    else
      if (a->Next != NULL)
        a = a->Next;
      else
      {
        while(a->Parent->Next == NULL)
        {
          if (a->Parent != NULL)
            a = a->Parent;
          else
            return NULL;
          if (a->Next == NULL && a->Parent == NULL)
            return NULL;
        }
        a = a->Parent->Next;
      }
    if (strcmp(a->XMIIdRef, xmiidref) == 0 && strcmp(a->Name, "elementID") == 0 )
      return a;
  }
  return NULL;
}
//---------------------------------------------------------------------------
TXMLUMLTree *TXMLTreeParser::FindGeometry(TXMLUMLTree *where)
{
  TXMLUMLTree *a = where->Children;
  while (a != NULL)
  {
    if (strcmp(a->Name, "geometry") == 0)
      return a;
    a = a->Next;
  }
  return NULL;
}
//---------------------------------------------------------------------------
void TXMLTreeParser::FormGeometry(UMLElement *where, char *geom)
{
  AnsiString temp, temp2, temp3, temp4;
  unsigned int gc = 0, i; /* TODO : variable names! */
  if (where->type == FLOW)
  {
    for (i = 0; i < strlen(geom); i++)
      if (geom[i] == ';')
        gc++;
    where->geomcount = gc;
    if (gc > 0)
      where->geometries = new Geometry[gc];
    else
      where->geometries = NULL;
    temp = geom;
    gc--;
    while ((i = temp.Pos(";")) > 1)
    {
      temp2 = temp.SubString(1, i);
      temp.Delete(1, i);
      temp.Trim();
      temp2.Trim();
      temp2.SetLength(temp2.Length()-1);
```

```
      temp2.Trim();
      temp3 = temp2.SubString(1, temp2.Pos(",") - 1);
      temp3.Trim(); //x
      temp2.Delete(1, temp2.Pos(","));
      temp2.Trim(); //y
      where->geometries[gc].x = StrToInt(temp3);
      where->geometries[gc].y = StrToInt(temp2);
      gc--;
    }
  }
  else
    if (strlen(geom) > 0)
    {
      where->geomcount = 2;  /* TODO : REDO!!!! */
      where->geometries = new Geometry[2];
      temp = geom;
      temp2 = temp.SubString(1, temp.Pos(",") - 1);
      temp.Delete(1, temp.Pos(","));
      temp3 = temp.SubString(1, temp.Pos(",") - 1);
      temp.Delete(1, temp.Pos(","));
      temp4 = temp.SubString(1, temp.Pos(",") - 1);
      temp.Delete(1, temp.Pos(","));
      temp.Trim();
      temp2.Trim();
      temp3.Trim();
      temp4.Trim();
      where->geometries[0].x = StrToInt(temp2);
      where->geometries[0].y = StrToInt(temp3);
      where->geometries[1].x = StrToInt(temp4);
      where->geometries[1].y = StrToInt(temp);
    }
    else
      where->geomcount = 0;
}
//---------------------------------------------------------------------------
#pragma package(smart_init)
```