RĪGAS TEHNISKĀ UNIVERSITĀTE
Datorzinātnes un informācijas tehnoloģijas fakultāte
Lietišķo datorsistēmu institūts

Konstantīns Gusarovs
Doktora studiju programmas "Datorsistēmas" doktorants

METODES IZSTRĀDE KODA ĢENERĒŠANAI NO DIVPUSLOŽU MODEĻA

Promocijas darbs

Zinātniskā vadītāja
profesore *Dr. sc. ing.*
OKSANA ŅIKIFOROVA

RTU Izdevniecība
Rīga 2021

Anotācija

Attīstoties programmatūras inženierijai, tās uzdevumi paplašinās, un mūsdienās tie ietver sevī ne tikai programmatūras koda izstrādi, bet arī biznesa procesu analīzi. Informācija, kas tiek iegūta šīs analīzes rezultāta var tikt izmantota modeļu, kas apraksta automatizējamos procesus, izstrādei. Modeļvadāma programmatūras izstrāde paredz arī šo modeļu izmantošanu programmatūras koda vai citu artefaktu ģenerēšanai. Modeļvadāma programmatūras izstrāde, atšķirībā no tā saucamas "uz modeļiem bāzētās" izstrādes, paredz stingri formalizētu modeļu un to apstrādes algoritmu izmantošanu visa programmatūras izstrādes dzīves cikla gaitā.

Lai gan idejas, kas atrodas modeļvadāmas izstrādes pamatā, sniedz vairākas priekšrocības – piemēram, sistēmas sākuma reprezentāciju, kas ir saprotama ne tikai tās izstrādātājiem, bet arī problēmsfēras ekspertiem un, iespējams, pasūtītājiem – tās ieviešana joprojām atrodas sākuma posmos. Tas var būt skaidrojams gan ar zemo automatizācijas līmeni, gan arī ar nepietiekami piemēroto avota modeļu izmantošanu.

Dotajā darbā tā autors piedāvā iespējamo risinājumu šai problēmai, definējot transformācijas likumus, kas no divpusložu modeļa (biznesa procesu un konceptu diagrammu apvienojums) ļauj iegūt programmatūras kodu. Papildus tiek piedāvāti arī dota modeļa uzlabojumi, kas ļauj šos transformācijas likumus realizēt, kā arī klašu attiecību definēšanas algoritms, ko ir iespējams izmantot arī ārpus modeļvadāmas programmatūras izstrādes jomas. Darba ietvaros ir apskatīti gan transformācijas likumi, kas tiek definēti ar pseidokoda palīdzību, gan arī to pielietošanas piemērs un rezultējošs programmatūras kods Java programmēšanas valodā.

Promocijas darbs satur ievadu, 7 nodaļas, secinājumus, 5 pielikumus, literatūras sarakstu (122 avoti), 125 attēlus un 11 tabulas, kopā 208 lappuses.

Abstract

Software engineering is continuously evolving, and its tasks are expanding. Nowadays, it includes not only software code development, but business process analysis as well. Information that is gained as a result of such analysis can be used for model that describe processes under automation development. Model driven software engineering also includes application of such models for automated software code or other artefact generation. In comparison to so called “model-based” engineering, model driven engineering uses strictly defined models and its processing algorithms during whole software lifecycle.

While ideas, that model driven software engineering is based upon, seem to provide several benefits – for example, initial system representation understood not only by developers, but also by business area experts and, possibly, customers – its adoption is still in the initial phase. This can be explained by both the low level of automation, as well as, inappropriate source model usage.

In this thesis, author offers a possible solution to this problem by defining transformation rules for so called Two-Hemisphere Model. This model is a combination of business process and concept diagrams and offered transformation rules allow for a software code generation from it. Additionally, improvements for the Two-Hemisphere model are offered. These improvements are required for enabling of code generation. Also, an algorithm for defining class relationships that can be used also outside of model driven software engineering is described. In this thesis author provides an insight to defined transformation rules by using pseudocode, as well as, an example of its’ application and resulting Java programming language code that was generated from the example model.

The thesis includes introduction, 7 chapters, and conclusion. It contains 208 pages, 125 figures and 11 tables, 122 titles large bibliography and 5 appendixes.

Saturs

Anotācija 2
Abstract 3
Saturs 4
Attēlu saraksts 6
Tabulu saraksts 9
Saīsinājumu saraksts 10
Ievads 11
1. Divpusložu modelis un aktuālas transformācijas metodes 20
1.1. Divpusložu modelis 20
1.2. Transformācijas metodes 21
2. Esošie divpusložu modeļa lietošanas ierobežojumi koda ģenerēšanas uzdevumā 24
2.1. Ierobežojumi divpusložu modeļa notācijā 24
2.2. Ierobežojumi divpusložu modeļa transformācijas metodēs 27
2.3. Divpusložu modeļa lietošanas ierobežojumi koda ģenerēšanas uzdevumā 31
3. Domēna-specifiskā valoda divpusložu modeļa definēšanai 32
3.1. Domēna-specifiskās valodas 33
3.2. Divpusložu modeli aprakstoša domēna-specifiskā valoda 34
3.2.1. Divpusložu modeli aprakstošas domēna-specifiskās valodas kopējie elementi. 35
3.2.2. Atribūtu definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā .. 36
3.2.3. Koncepta definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā . 37
3.2.4. Procesa definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā 39
3.2.5. Datu plūsmas definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā 40
3.2.6. Procesu diagrammas definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā 41
3.2.7. DSL parsētāja realizācijas detaļas 42
3.2.8. DSL izmantošanas piemērs 44
3.3. Secinājumi par divpusložu modeli aprakstošu domēna-specifisko valodu 46
4. Uzlabotais klašu attiecību noteikšanas algoritms 48
4.1. Realizāciju un mantošanu noteikšana 50
4.2. Agregāciju noteikšana 53
4.3. Atkarību noteikšana 54
4.4. Asociāciju noteikšana 55
4.5. Algoritma darbības piemērs 55
4.6. Klašu attiecību noteikšanas algoritma analīze un secinājumi par to 59

5. Koda ģenerēšana no divpusložu modeļa 60
5.1. Mērķa programmēšanas valodas izvēle 60
5.2. Koda ģenerēšanas stratēģijas definēšana 62
5.3. Datu struktūru definēšana koda ģenerēšanai 65
5.4. Biznesa procesu diagrammas minimizēšana 78
5.4.1. Metožu signatūru definēšana 79
5.4.2. Biznesa procesu diagrammas pārveidošana procesu izsaukumu grafā 80
5.4.3. Procesu izsaukumu grafs 83
5.4.4. Procesu izsaukumu grafa minimizēšana 85
5.5. Minimizēta procesu izsaukumu grafa apstrāde 114
5.6. Koda ģenerēšanas algoritms – kopsavilkums 126
6. Koda ģenerēšanas algoritma novērtēšana 129
6.1. Algoritma darbības piemērs 129
6.2. Algoritma darbības rezultātu validācija 142
6.3. Java koda iegūšana no instrukciju secības 146
6.4. Saistīto pētījumu apskats 161
6.5. Koda ģenerēšanas algoritma novērtēšanas rezultāts 166
7. Darba rezultātu praktiskais pielietojums 167
7.1. Elektroģitāras efektu pārslēgšanas sistēma 167
7.2. Sistēmas divpusložu modelis 169
7.3. Koda ģenerēšanas procesa pielāgošana 171
7.4. Iegūtie rezultāti 172
Nobeigums 177
Bibliogrāfija 182
Pielikumi 189
Programmatūras izstrādātāju un arhitektu aptaujas rezultāti 190
Biznesa analītiķu aptaujas rezultāti 191
Divpusložu modeli aprakstošas domēna-specifiskās valodas ANTLR likumi 197
Piemēra divpusložu modelis 202
Elektroģitāras efektu pārslēgšanas sistēmas kods 205

Attēlu saraksts

1.1. att. Divpusložu modeļa piemērs 21
2.1. att. Biznesa procesu modeļa sastāvdaļa, kas satur iespējamas alternatīvas 26
2.2. att. Dzīvnieku klašu hierarhija 29
2.3. att. Anēmiskā dzīvnieku klašu hierarhija 30
3.1. att. Divpusložu modeli aprakstošas valodas kopējie elementi 36
3.2. att. Divpusložu modeli aprakstošas valodas atribūtu definīcijas 36
3.3. att. Koncepta definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā 38
3.4. att. Procesa definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā 39
3.5. att. Datu plūsmas definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā 40
3.6. att. Procesu diagrammas definīcija divpusložu modeli aprakstošā DSL 42
3.7. att. ANTLR likumi identifikatora parsēšanai 43
3.8. att. ANTLR uzģenerētas novērotāja metodes 43
3.9. att. Vienkārša divpusložu modeļa grafiskā notācija 44
3.10. att. Vienkārša divpusložu modeļa DSL notācija 45
4.1. att. Mantošanas attiecības definēšanas funkcija 50
4.2. att. Mantošanas definēšanas algoritms 51
4.3. att. Realizāciju definēšanas algoritms 53
4.4. att. Agregāciju definēšanas algoritms 54
4.5. att. Iespējamās atkarības definēšanas funkcija 54
4.6. att. Atkarību noteikšanas algoritms 55
4.7. att. Asociāciju definēšanas algoritms 55
4.8. att. Karšu sistēmas klases 56
4.9. att. Karšu sistēmas klases pēc pirmās mantošanu noteikšanas algoritma iterācijas. 56
4.10. att. Karšu sistēmas klases pēc otrās mantošanu noteikšanas algoritma iterācijas. 57
4.11. att. Karšu sistēmas klases pēc realizāciju noteikšanas algoritma izpildes. 58
4.12. att. Rezultējošās klases un to attiecības. 58
5.1. att. Java valodas koda piemērs 66
5.2. att. Java valodas koda piemērs abstrakta sintaktiska koka formā 67
5.3. att. Java valodas koda piemērs vienkāršota abstrakta sintaktiska koka formā 68
5.4. att. Java valodas koda piemērs JVM baitu koda formā 68
5.5. att. Baitu kods bez nevajadzīgām iezīmēm 69
5.6. att. Modificētais baitu kods 70
5.7. att. Programmatūras koda reprezentācija 77
5.8. att. Metodes signatūras definīcija 79
5.9. att. Metodes signatūras definīcija ar izpildes nosacījumu 79
5.10. att. Metožu signatūru definēšanas algoritms 80
5.11. att. Procesu priekšteču un pēcteču noteikšana 81
5.12. att. Starpprocesu pāreju tabulas izveidošana 81
5.13. att. Procesu izsaukumu grafa izveidošana 82
5.14. att. Sākuma un beigu virsotņu pievienošana procesu izsaukumu grafam 82
5.15. att. Procesu izsaukumu grafa fragments 83
5.16. att. Virsotņu apvienošanas rezultāts 84
5.17. att. Procesu izsaukumu grafa fragments pirms cilpas aizvietošanas 86
5.18. att. Procesu izsaukumu grafa fragments pēc cilpas aizvietošanas 86
5.19. att. Cilpu aizvietošanas algoritms 87
5.20. att. Procesu izsaukumu grafa fragments pirms loku dublikātu likvidēšanas 87

5.21. att. Procesu izsaukumu grafa fragments pēc loku dublikātu likvidēšanas 87
5.22. att. Loku dublikātu likvidēšanas algoritms 88
5.23. att. Procesu izsaukumu grafa fragments pirms virsotņu apvienošanas 88
5.24. att. Procesu izsaukumu grafa fragments pēc virsotņu apvienošanas 88
5.25. att. Virsotņu apvienošanas algoritms 89
5.26. att. Stipri saistīta grafa komponente, kas satur vairākus ciklus 90
5.27. att. Tarjana algoritms 91
5.28. att. Procesu izsaukumu grafa attēlojuma algoritms 92
5.29. att. Atgriezeniskais attēlojums 92
5.30. att. Galvenās Džonsona algoritma funkcijas 93
5.31. att. Papildus Džonsona algoritma funkcijas 94
5.32. att. Procesu izsaukumu grafa fragments ar ciklu 95
5.33. att. Procesu izsaukumu grafa fragments ar aizvietoto ciklu 95
5.34. att. Procesu izsaukumu grafa fragments ar netriviālu ciklu 96
5.35. att. Procesu izsaukumu grafa fragments ar aizvietotu netriviālo ciklu 97
5.36. att. Procesu izsaukumu grafa fragments ar aizvietotu netriviālo lineāro ciklu 98
5.37. att. Ciklu aizvietošanas algoritms 99
5.38. att. Cikla apakšgrafs ar ieejas alternatīvām 100
5.39. att. Cikla apakšgrafa ar ieejas alternatīvām apstrādes rezultāts 100
5.40. att. Ciklu ieejas un izejas punktu apstrādes algoritms 101
5.41. att. Cikla ieejas punkti pirms pārkārtošanas 102
5.42. att. Cikla ieejas punkti pēc pārkārtošanas 102
5.43. att. Likvidējamais cikla izejas punkts 103
5.44. att. Cikls pēc izejas punkta likvidēšanas 104
5.45. att. Cikls ar izejas punktu 104
5.46. att. Cikls pēc izejas punkta apstrādes 105
5.47. att. Cikls ar izejas punktu dublikātiem 106
5.48. att. Cikls bez izejas punktu dublikātiem 106
5.49. att. Cikls, kur pirmā virsotne satur izpildes nosacījumu 107
5.50. att. Cikls ar priekšnosacījumu 107
5.51. att. Secība ar nosacījumu pirms aizvietošanas 108
5.52. att. Secība ar nosacījumu pēc aizvietošanas 108
5.53. att. Disjunkcijas situācija grafā 109
5.54. att. Disjunkcijas apstrādes rezultāts 109
5.55. att. Disjunkcijas definēšanas algoritms 110
5.56. att. Algoritmu pielietošanas funkcijas 111
5.57. att. Informācijas par mainīgajiem iegūšana no datu plūsmām 115
5.58. att. Informācijas par mainīgajiem iegūšana no izpildes nosacījumiem 115
5.59. att. Informācijas par mainīgajiem iegūšana no ciklu ieejas un izejas punktiem 116
5.60. att. Informācijas par mainīgajiem iegūšana no ciklu izpildes nosacījumiem 116
5.61. att. Iezīmju definēšanas algoritms 119
5.62. att. Zarošananās definēšanas algoritms 121
5.63. att. Rezultātu klašu definēšanas algoritms 123
5.64. att. Konceptu pārveidošana par klasēm 123
5.65. att. Mainīgo definēšanas instrukciju ģenerēšana 124
5.66. att. Secības elementa apstrāde 126
5.67. att. Transformācijas likumu pielietošana 128
6.1. att. Piemēra divpusložu modelis – konceptu diagramma 129
6.2. att. Piemēra divpusložu modelis – biznesa procesu diagramma 130
6.3. att. Iegūtais procesu izsaukumu grafs 133

6.4. att. Procesu izsaukumu grafs pēc cikla aizvietošanas 134
6.5. att. Cikla procesu izsaukumu grafs 135
6.6. att. Procesa izsaukumu rezultātu klases 137
6.7. att. Mainīgo definēšanas instrukcijas 139
6.8. att. Uzģenerētā instrukciju secība 140
6.9. att. Transformācijas validācijas grafa virsotņu definēšana 142
6.10. att. Transformācijas validācijas grafa loku definēšana 142
6.11. att. Atgriezmeklēšana, apmeklējot tikai iezīmes 144
6.12. att. Transformācijas validācijas algoritms 144
6.13. att. Piemēra transformācijas validācijas grafs 145
6.14. att. Mainīgo definēšana JVM baitu kodā 149
6.15. att. Metodes prologs 151
6.16. att. Metodes baitu kods 152
6.17. att. Metodes baitu kods (turpinājums) 153
6.18. att. Nosacījumu aizvietošana pēc dekompilācijas 154
6.19. att. Domēna klases, kas tika iegūtas no konceptiem 155
6.20. att. Procesu izsaukumu rezultātu klases Java valodā 156
6.21. att. Metodes, kas tika iegūtas procesu transformācijas rezultātā 157
6.22. att. Java mainīgo definīcijas 158
6.23. att. Java biznesa loģikas kods 159
6.24. att. FernFlower dekompilācijas rezultāts 160
7.1. att. Elektroģitāras efektu pārslēgšanas sistēmas shēma 167
7.2. att. Signāla pārraides ceļš ar izslēgtiem efektiem 168
7.3. att. Signāla pārraides ceļš ar vairākiem ieslēgtiem efektiem 168
7.4. att. Definētas konceptu modelis 169
7.5. att. Definētas procesu modelis 171
7.6. att. Uzģenerētas datu struktūras 173
7.7. att. Uzģenerētas funkcijas 174
7.8. att. Uzģenerētais `main()` funkcijas kods 175

Tabulu saraksts

2.1. tabula: Esošie divpusložu modeļa notācijas un transformācijas metožu ierobežojumi 31
3.1. tabula: Divpusložu modeļa notācijas ierobežojumu risināšana ar DSL valodas palīdzību47
5.1. tabula: 10 populārākās programmēšanas valodas pēc TIOBE datiem 61
5.2. tabula: Procesu izsaukumu grafa virsotņu veidi..... 84
5.3. tabula: Algoritmu grupas..... 112
6.1. tabula: Mainīgie, kas tika iegūti no datu plūsmām..... 136
6.2. tabula: Mainīgie, kas tika iegūti procesu izsaukumu grafa analīzes rezultātā 137
6.3. tabula: Definētās iezīmes 138
6.4. tabula: Definētās zarošanās 139
6.5. tabula: Instrukciju kartēšana uz JVM baitu kodu..... 147

Saīsinājumu saraksts

Nr.	Saīsinājums	Izvērsums/Paskaidrojums
1	AST	Abstract syntax tree
2	BPMN	Business Process Model and Notation
3	CASE	Computer-aided software engineering
4	CIL	Common Intermediate Language
5	DFD	Data-flow diagram
6	DSL	Domain-specific language
7	EBNF	Extended Backus–Naur form
8	ER	Entity-Relationship
9	GPL	General Purpose Language
10	HSC	Hierarchical Syntax Chart
11	IDDFS	Iterative deepening depth-first search
12	JVM	Java Virtual Machine
13	MDA	Model Driven Architecture
14	MDSD	Model Driven Software Development
15	MSVC	Model-Service-View-Controller
16	MVC	Model-View-Controller
17	OMG	Object Management Group
18	UML	Unified Modeling Language
19	WORA	Write once, run anywhere
20	XML	Extensible Markup Language

IEVADS

Programmatūras inženierija turpina strauji attīstīties. 20. gadsimta 50. gados programmēšana nozīmēja zema līmeņa mašīnas instrukcijas noteiktu uzdevumu risināšanai [1]. Mūsdienās programmatūras inženierijas uzdevumi ir paplašinājušies, un tā tagad ietver sevī gan programmatūras koda izstrādi, gan biznesa procesu, kas tiek automatizēti, analīzi, gan arī projektu pārvaldību. Darba [1] autori izsaka domu, ka mūsdienu programmatūras inženierija ir spējās (angl. *Agile*) metodoloģijas un modeļvadāmas arhitektūras (angl. *Model-Driven Architecture* – MDA) attīstības rezultāts.

Viena no jaunākajām programmatūras inženierijas paradigmām ir modeļvadāmā programmatūras izstrāde (angl. *Model-Driven Software Development* – MDSD) [20],[92], kas joprojām attīstās un to pielāgo gan pētnieki, gan arī industrijas pārstāvji. Modeļvadāmas programmatūras izstrādes galvenā ideja ir modeļu izmantošana programmatūras koda, vai arī citu programmatūras artefaktu ģenerēšanai [20]. Atšķirībā no tā sauktās "uz modeļiem bāzētās" izstrādes, kas paredz modeļu izmantošanu dažādos izstrādes cikla posmos (modeļi var būt jebkāda veida, piemēram, pat ar roku zīmēti), MDSD paredz stingri formalizētu modeļu un to apstrādes algoritmu izmantošanu.

Darba [92] autors ir definējis divas galvenās modeļvadāmās programmatūras lietotāju grupas – pirmā grupa uzskata modeļus par analītisku instrumentu, kas ir paredzēts problēmsfēras analīzei un biznesa specifikas attēlošanai. Otrā lietotāju grupa definē modeļus kā augsta līmeņa izpildāmo artefaktu, kas var tikt izmantots zemāka līmeņa artefaktu (piemēram, programmatūras kods, dokumentācija un citi) automātiskai ģenerēšanai. Analizējot abu lietotāju grupu modeļa lomas definīcijas programmatūras izstrādē, var redzēt, ka tās nav pretrunā viena ar otro. Augsta līmeņa artefakts var tikt izmantots gan biznesa analīzes, gan arī koda ģenerēšanas nolūkiem. Tas nozīme to, ka problēmsfēras modelis var būt saprotams gan biznesa pārstāvjiem, gan programmatūras izstrādātājiem, gan arī piemērots automātiskai programmatūras koda ģenerēšanai no tā.

Modeļu definēšanai MDSD kontekstā tiek izmantotas tā saucamās modelēšanas valodas, kas var tikt uzskatītas par nākamo soli programmēšanas valodu attīstībā [20],[92]. Analizējot programmatūras inženierijas vēsturi [1], var redzēt, ka programmēšanas valodas nepārtraukti attīstās, katrai jaunai valodu paaudzei sasniedzot lielāku abstrakcijas līmeni.

Tēmas aktualitāte

Pēc autora uzskatiem un pieredzes programmatūras izstrādē, viena no galvenajām modeļvadāmās programmatūras izstrādes ieviešanas problēmām ir industrijā izmantotās [35] vienotās modelēšanas valodas (angl. *Unified Modelling Language* – UML) [113] sarežģītība. Lai gan UML izmantošana piedāvā plašas koda ģenerēšanas iespējas no tās standarta modeļiem (piemēram, [6],[7],[20],[99],[103],[114] un citi), promocijas darba autors to skaidro ar UML ciešo saistību ar programmatūras kodu – lielākā daļa no UML diagrammām apraksta jau gatavus risinājumus, piemēram, klases, to sadarbību, metožu izsaukumu secību, vai arī citus artefaktus, nevis sākotnējo problēmsfēru. OMG (angl. *Object Management Group*) mājaslapā [113] UML tiek definēta kā "grafiskā valoda uz objektiem bāzēto sistēmu vizualizācijai, specificēšanai, konstruēšanai un dokumentēšanai". Tas nozīme to, ka UML nav domāta sākotnējo prasību modelēšanai, bet ir domāta gatavo arhitektūras risinājumu aprakstīšanai. Līdz ar to promocijas darba autors uzskata UML par nepiemērotu biznesa procesu modelēšanai un MDSD atbalstam. Tas nozīme nepieciešamību pēc citas modelēšanas notācijas, kas ļautu veidot sistēmas, nevis to arhitektūras aprakstu.

Līdzīgas domas izvirza arī pētījuma [91] autori, uzsverot, ka pašreizējās modelēšanas valodas (tai skaitā UML) ir saprotamas modeļvadāmās inženierijas ekspertiem, bet nav saprotamas problēmsfēras ekspertiem un biznesa analītiķiem. Šī darba autori īpaši uzsver nepieciešamību pēc modelēšanas valodas, ko var saprast, definēt un izmantot ne tikai programmatūras inženieri. Pētījuma [8] autors arī atzīmē faktu, ka UML ir pārāk sarežģīta modelēšanas valoda, lai to varētu pilnībā ievest industrijā. Šis autors ir arī definējis, ka darbā ar klientiem un problēmsfēras ekspertiem būtu nepieciešams definēt trīs galvenos artefaktus, kas turpmāk var tikt izmantoti sistēmas modelēšanai un izstrādei:

- Lietošanas gadījumu saraksts.
- Biznesa procesu modeļi, kas šos lietošanas gadījumus apraksta (piemēram, BPMN – angl. *Business Process Model and Notation* [10] notācijā).
- Konceptu modeļi, kas satur informāciju par problēmsfēras domēna objektiem.

Šo artefaktu izvēli autors pamato ar to, ka šos artefaktus var izmantot gan programmatūras inženieri, gan arī problēmsfēras eksperti. Autors izsaka domu par to, ka "ir nepieciešams definēt pietiekami detalizētus modeļus, kas ne tikai ļautu veikt koda ģenerēšanu, bet arī ir būtu saprotami gala lietotājiem un klientiem".

Idejas par UML vājo piemērotību biznesa procesu modelēšanai izsaka arī [105] autori, definējot, ka UML vietā ir vēlams izmantot vienkāršākas modelēšanas valodas, kas ir

saprotamas problēmsfēras ekspertiem un citām iesaistītām pusēm (angl. *stakeholders*). Autori izsaka domu, ka UML modeļi ir "pārāk līdzīgi programmatūras kodam, lai tos varētu izmantot".

Tātad var redzēt, ka ir iespējams veikt programmatūras koda ģenerēšanu no modeļa, tomēr uz doto brīdi izmantojamie modeļi ir vāji piemēroti ieviešanai to sarežģītības dēļ. Šī darba autors uzskata, ka modeļvadāmas programmatūras izstrādi varētu pārveidot par industrijai "draudzīgāku", izmantojot cita veida modeļus problēmsfēras analīzei. Tiek arī izvirzīta hipotēze par to, ka tāda valoda varētu sevī ietvert divus artefaktus (līdzīgi ka [8]):

- Biznesa procesa modeli, kuram var izmantot dažāda veida notācijas notiekošo procesu aprakstam – piemēram, BPMN [10], datu plūsmu diagrammas (angl. *Data Flow Diagram* – DFD) [29],[102] vai arī citas notācijas.
- Domēna jeb konceptuālo modeli, kas, piemēram, var būt veidots ER-diagrammas (angl. *Entity-Relationship*) [17] veidā.

Pēc autora domām ir iespējams lietošanas gadījumu modeļa vietā izmantot biznesa procesu modeli, tas ir, katram lietošanas gadījumam definēt attiecīgos biznesa procesus. Promocijas darba autors uzskata, ka šāda veida notācija ļaus definēt augstāka līmeņa modeļus, kurus varēs izmantot arī problēmsfēras eksperti un cita iesaistītās puses.

Lai pārbaudītu šo hipotēzi šī darba autors ir savā darbavietā veicis divas aptaujas. Pirmās aptaujas mērķa auditorija bija programmatūras izstrādātāji un arhitekti, kas katru dienu strādā ar programmatūras artefaktiem un to izstrādi, specificēšanu un aprakstiem. Šeit respondentiem tika uzdots viens jautājums par to, kuras no UML diagrammām tie ir spējīgi definēt. Aptaujas rezultāti ir sniegti 1. pielikumā (tā kā oficiālā komunikācijas valoda kompānijā ir angļu – līdz ar ārzemju darbinieku iesaisti, arī jautājumi tika uzdoti angļu valodā). Kopā tika saņemtas 227 atbildes, pēc kurām var secināt, kā lielākā respondentu daļa ir spējīga definēt klašu, secību, stāvokļu, aktivitāšu un lietošanas gadījumu diagrammas. Pēc respondentu domām tas ir skaidrojams ar faktu, ka tieši šīs UML diagrammas (izņemot lietošanas gadījumu) ir domātas tieši programmatūras koda un tā uzvedības aprakstam, kas nozīme to, ka ar to palīdzību ir iespējams aprakstīt lietojuma arhitektūru un procesus, kas notiek sistēmā.

Otrās aptaujas mērķa auditorija bija biznesa analītiķi, kuru galvenais darba uzdevums ir biznesa procesu analīze un to formalizācija, izmantojot modeļus, kas ir saprotami iesaistītām pusēm, kā arī programmatūras izstrādātājiem. Respondentiem tika uzdoti vairāki jautājumi – līdzīgi kā programmatūras izstrādātājiem tika jautāts par spēju izmantot dažāda veida UML diagrammas, kā arī par biznesa procesu un ER-diagrammu izmantošanu.. Kopā tika saņemtas 46 atbildes, kas kopā ar jautājumiem ir apkopotas 2. pielikumā. Ir jāņem vērā, ka ne visi respondenti iesniedza teksta atbildes. Var redzēt, ka lielākā biznesa analītiķu daļa saprot, kas ir

ER-diagrammas un biznesa procesu diagrammas, spēj tās aprakstīt un izmantot. Tas apstiprina pieņēmumu par to, ka modeļvadāmai programmatūras izstrādei tieši šie divi modeļi ir piemēroti, un to izmantošana visticamāk atvieglotu MDSD ieviešanu industrijā. Līdz ar to promocijas darba autors uzskata, ka ir nepieciešams veikt programmatūras koda ģenerēšanu, izmantojot modelēšanas valodu, kas šos divus modeļus atbalsta vienā vai citā veidā.

Kā tāds modelis, kas atbalsta ER-diagrammas un biznesa procesu diagrammas, tika izvēlēts divpusložu modelis [68],[80]. Vairāki pētījumi, kas tika veikti līdz šim brīdim un ir veltīti divpusložu modeļa izmantošanai UML (lietošanas gadījumu, secību, stāvokļu un klašu) diagrammu ģenerēšanai, parāda, ka šis modelis satur pietiekami daudz informācijas gan statisko, gan arī dinamisko sistēmas analīzes artefaktu definēšanai [68],[69]. Pēc transformācijām iegūtais rezultāts ir UML diagrammu komplekss, kas, savukārt ļauj veikt koda ģenerēšanu [87].

Uz doto brīdi ir atzīmējams CASE-rīku, kas atbalsta automatizēto koda ģenerēšanu, trūkums [98]. Šis trūkums galvenokārt izriet no nepietiekamas sistēmu analīzes dinamisko elementu izmantošanas [67]. Neskatoties uz to, eksistē virkne pētījumu, kas parāda to, ka ir iespējama arī šāda veida elementu iegūšana no divpusložu modeļa [79],[81], kā arī programmatūras koda iegūšana no dinamiskiem sistēmas analīzes artefaktiem [7],[103]. Līdz ar to, tiek izvirzīta hipotēze par to, kā ir iespējams no divpusložu modeļa pāriet uz programmatūras kodu (ja ir iespējamas transformācijas divpusložu modelis → UML un UML → kods, tad ir jābūt iespējamai arī divpusložu modelis → kods transformācijai). Šo apgalvojumu atbalsta arī pētījumi [39],[40].

Šādas izmaiņas ļaus koda ģenerēšanai izmantot tikai vienu modeli – divpusložu modeli, kas, savukārt, var tikt iegūts biznesa prasību analīzes rezultātā. Tas nozīmē to, ka nākotnē rīka izstrādei būs nepieciešams atbalstīt tikai vienu modeli, nav jāatbalsta UML modeļi, kas var izraisīt vairākas problēmas [8],[70],[91],[122] gan no atbalsta, gan arī no izmantošanas viedokļa, un ir iespējams piedāvāt pilnu risinājumu iesaistītām pusēm.

Promocijas darba mērķis

Promocijas darbam tiek izvirzīts mērķis izstrādāt programmatūras koda ģenerēšanas algoritmu no divpusložu modeļa, tā validācijas metodi, kā arī pārbaudīt izstrādāta algoritma darbību praktiski.

Promocijas darba uzdevumi

Promocijas darba mērķa sasniegšanai ir izvirzīti šādi uzdevumi:

1. Novērtēt divpusložu modeļa piemērotību programmatūras koda ģenerēšanai un, ja ir nepieciešams, papildināt to.
2. Izvēlēties mērķa programmēšanas valodu programmatūras koda ģenerēšanai un pamatot tās izvēli.
3. Izstrādāt algoritmu vai algoritmu kopu divpusložu modeļa transformēšanai programmatūras kodā.
4. Definēt validācijas metodoloģiju transformāciju pareizību pārbaudei.
5. Pārbaudīt izstrādāto algoritmu ar praktiskā uzdevuma palīdzību, pielietojot to programmatūras sistēmas izstrādei.

Pētījuma objekts

Promocijas darba pētījuma objekts ir divpusložu modelis.

Pētījuma priekšmets

Darba pētījuma priekšmets ir divpusložu modeļa transformācijas metodes programmatūras koda ģenerēšanai.

Pētījuma metodes

Promocijas darba teorētiskajā daļā ir izmantotas šādas teorētiskās metodes:

1. Zinātniskās literatūras teorētiskā izpēte un saistīto pētījumu analīze.
2. Datu ieguves metodes – aptaujas.

Praktiskās daļas izstrādē ir izmantotas šādas metodes:

1. Problēmas un iespējamo risinājumu analīze.
2. Statistiskās metodes – eksperimentu veikšana un to rezultātu analīze.
3. Prototipēšana koda transformācijas metodes izstrādei.

Darba zinātniskais jaunieguvums ir šāds:

1. Tika veikta divpusložu modeļa analīze. Identificētas tā priekšrocības un ierobežojumi programmatūras koda ģenerēšanai. Pēc analīzes tika piedāvāti vairāki modeļa uzlabojumi.
2. Tika izstrādāti jauni transformācijas likumi divpusložu modelim, kas ļauj no tā iegūt programmatūras kodu. Darba ietvaros tika apskatīta Java [50] koda ģenerēšana, tomēr piedāvātie likumi var tikt pielietoti jebkurai koda ģenerēšanai jebkurā programmēšanas valodā.

3. Lai panāktu transformācijas likumu neatkarību no mērķa programmēšanas valodas, tika izstrādāts starpmodelis koda reprezentācijai, ko ir iespējams izmantot programmatūras koda ģenerēšanai dažādu paradigmu valodās.
4. Tika izstrādāts klašu attiecību noteikšanas algoritms, ko ir iespējams izmantot arī ārpus modeļvadāmas programmatūras izstrādes darbības sfēras. Iespējamās pielietošanas jomas var būt koda pārrakstīšana (angl. *refactoring*), esošo sistēmu un to komponenšu analīze un citas.

Pētījuma praktiskā nozīmība

Promocijas darba izstrādes laikā iegūtie rezultāti tika izmantoti arī praktiska uzdevuma risināšanai – mikrokontroliera programmatūras izstrādei. Tās izstrādes laikā autors ir konstatējis, ka divpusložu modelis ir saprotams ne tikai programmatūras izstrādātājiem, bet arī citām iesaistītām pusēm, kas atviegloja savstarpējo komunikāciju. Izstrādātie transformācijas likumi, ir no mērķa programmēšanas valodas neatkarīgi, un var tikt izmantoti ne tikai ar objektorientētām programmēšanas valodām, kas arī tika praktiski pieradīts.

Pētījuma rezultātu aprobācija

Promocijas darba rezultāti ir atspoguļoti 9 publikācijās starptautiskos un Latvijas Zinātnes padomes atzītos zinātniskos izdevumos:

1. Nikiforova, O., Gusarovs, K. Anemic Domain Model vs Rich Domain Model to Improve the Two-Hemisphere Model-Driven Approach. *Applied Computer Systems*, 2020, Volume 25, Issue 1. (Accepted for publication) (Web of Science). (Ieguldījums publikācijā ~50 %).
2. Gusarovs, K., Nikiforova, O. An Intermediate Model for the Code Generation from the Two-Hemisphere Model. In: *Proceedings of the 14th International Conference on Software Engineering Advances (ICSEA 2019)*, accepted for publishing. ISBN: 978-1-61208-752-8. (Ieguldījums publikācijā ~70%).
3. Gusarovs, K. An Analysis on Java Programming Language Decompiler Capabilities. *Applied Computer Systems*, 2018, Volume 23, No. 2. pp. 109-117. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.2478/acss-2018-0014.
4. Gusarovs, K., Nikiforova, O., Giurca, A. Simplified Lisp Code Generation from the Two-hemisphere Model. *Procedia Computer Science*, 2017, Volume 104, Issue C. pp. 329-337. Available from: doi:10.1016/j.procs.2017.01.142. (SCOPUS, Web of Science). (Ieguldījums publikācijā ~70 %).
5. Gusarovs, K., Nikiforova, O. Workflow Generation from the Two-Hemisphere Model. *Applied Computer Systems*, 2017, Volume 22, Issue 1. pp. 36-46. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.1515/acss-2017-0016. (Web of Science). (Ieguldījums publikācijā ~60 %).
6. Nikiforova, O., Gusarovs, K., Ressin, A. An Approach to Generation of the UML Sequence Diagram from the Two-Hemisphere Model. In: *Proceedings of the 11th International Conference on Software Engineering Advances (ICSEA 2016)*. Wilmington: IARIA, 2016, pp. 142-149. ISBN 978-1-61208-498-5. (Ieguldījums publikācijā ~40 %).

7. Gusarovs, K., Nikiforova, O., Jukss, M. A Prototype of Description Language for the Two-Hemisphere Model. *Applied Computer Systems*, 2015, Volume 18, Issue 1. pp. 15-20. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0014. (Ieguldījums publikācijā ~60%).
8. Nikiforova, O., Gusarovs, K. Kozacenko, L., Ahilcenoka, D., Ungurs, D. An Approach to Compare UML Class Diagrams Based on Semantical Features of Their Elements. In: *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA 2015)*. Wilmington: IARIA, 2015, pp. 147-153. ISBN 978-1-61208-438-1. (Ieguldījums publikācijā ~40%).
9. Zusane, U.I., Nikiforova, O., Gusarovs, K. Several Issues on the Model Interchange Between Model-Driven Software Development Tools. In: *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA 2015)*. Wilmington: IARIA, 2015, pp. 451-457. ISBN 978-1-61208-438-1. (Ieguldījums publikācijā ~20%).

Promocijas darba galvenie rezultāti prezentēti 7 starptautiskās zinātniskās konferencēs:

1. ICSEA 2015 - The Tenth International Conference on Software Engineering Advances, November 15-20, 2015 - Barcelona, Spain, "Several Issues on the Model Interchange Between Model-Driven Software Development Tools" and "An Approach to Compare UML Class Diagrams Based on Semantical Features of Their Elements".
2. Riga Technical University 56th International Scientific Conference, October 14-17, 2015 - Riga, Latvia, "A Prototype of Description Language for the Two-Hemisphere Model".
3. ICSEA 2016 - The Eleventh International Conference on Software Engineering Advances. August 21-25, 2016 - Rome, Italy. "An Approach to Generation of the UML Sequence Diagram from the Two-Hemisphere Model".
4. Riga Technical University 58th International Scientific Conference, October 12-15, 2017 - Riga, Latvia, "Workflow Generation from the Two-Hemisphere Model".
5. 15th International Conference of Numerical Analysis and Applied Mathematics ICNAAM 2017, 7th Symposium on Computer Languages, Implementation and Tools - SCLIT 2017, September, 26-30, 2017, Thessaloniki, Greece, "Several Issues of Two-Hemisphere Model-Driven Approach to Improve with Anemic Domain Model".
6. Riga Technical University 59th International Scientific Conference, October 10-12, 2018 - Riga, Latvia, "An Analysis on Java Programming Language Decompiler Capabilities".
7. ICSEA 2019 - The Fourteenth International Conference on Software Engineering Advances, November 24-28, 2019 - Valencia, Spain, "An Intermediate Model for the Code Generation from the Two-Hemisphere Model".

11 citas publikācijas dotajā jomā:

1. Nikiforova, O., Ahilcenoka, D., Ungurs, D., Gusarovs, K., Kozacenko, L. Several Issues on the Layout of the UML Sequence and Class Diagram. In: *Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA 2014),* 2014, pp. 40-47. (Ieguldījums publikācijā ~20%).
2. Nikiforova, O., Bohomaz, Y., Gusarovs, K. A Comparison of the Implementation Means for Development of Modelling Tool. In: *Proceedings of the 2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS 2017)*, 2017, pp. 1-6. Available from: doi:10.1109/WITS.2017.7934623. (SCOPUS, Web of Science). (Ieguldījums publikācijā ~20 %).

3. Nikiforova, O., El Marzouki, N., Gusarovs, K., Vangheluwe, H., Bures, T., Al-Ali, R., Iacono, M., Esquivel, P.O., Leon, F. The Two-Hemisphere Modelling Approach to the Composition of Cyber-Physical Systems. In: *In Proceedings of the 12th International Conference on Software Technologies*. Portugal: SciTePress, 2017, pp. 286-293. ISBN 978-989-758-262-2. Available from: doi:10.5220/0006424902860293. (SCOPUS, Web of Science). (Ieguldījums publikācijā ~10%).
4. Nikiforova, O., Gorbiks, O., Gusarovs, K., Ahilcenoka, D., Bajovs, A., Kozacenko, L., Skindere, N., Ungurs, D. Development of BrainTool for Generation of UML Diagrams from the Two-hemisphere Model Based on the Two-Hemisphere Model Transformation Itself. In: *Proceedings of the International Scientific Conference "Applied Information and Communication Technologies"*. Latvia: LLU, 2013, pp. 267-274. ISSN 2255-8586. (Ieguldījums publikācijā ~20%).
5. Nikiforova, O., Gusarovs, K. Comparison of BrainTool to Other UML Modeling and Model Transformation Tools. *AIP Conference Proceedings*, 2017, Volume 1863, No. 1, id. 330005. Available from: doi:10.1063/1.4992503. (SCOPUS, Web of Science). (Ieguldījums publikācijā ~40%).
6. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. BrainTool. A Tool for Generation of the UML Class Diagrams. In: *Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012. Lisbon: IARIA, 2012, pp. 60-69. ISBN 9781612082301. (Ieguldījums publikācijā ~35%).
7. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. Improvement of the Two-Hemisphere Model-Driven Approach for Generation of the UML Class Diagram. *Applied Computer Systems*, 2013, Volume 14, Issue 1, pp. 19-30. ISSN 2255-8691. Available from: doi:10.2478/acss-2013-0003. (Ieguldījums publikācijā ~30%).
8. Nikiforova, O., Kozacenko, L., Ungurs, D., Ahilcenoka, D., Bajovs, A., Skindere, N., Gusarovs, K., Jukss, M. BrainTool v2.0 for Software Modeling in UML. *Applied Computer Systems*, 2015, Volume 16, Issue 1, pp 33-42. ISSN 2255-8691. Available from: doi:10.1515/acss-2014-0011. (Ieguldījums publikācijā ~10%).
9. Nikiforova, O., Kozacenko, L., Ahilcenoka, D., Gusarovs, K., Ungurs, D., Jukss, M. Comparison of the Two-Hemisphere Model-Driven Approach to Other Methods for Model-Driven Software Development. *Applied Computer Systems*, 2016, Volume 18, Issue 1, pp. 5-14. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0013. (Ieguldījums publikācijā ~20%).
10. Nikiforova, O., Pavlova, N., Gusarovs, K., Gorbiks, O., Vorotilovs, J., Zaharovs, A., Umanovskis, D., Sejans, J. Development of the Tool for Transformation of the Two-Hemisphere Model to the UML Class Diagram: Technical Solutions and Lessons Learned. In: *Proceedings of the 5th Internation Scientific Conference "Applied Information and Communication Technology 2012"*. Latvia: LLU, 2012, pp. 11-19. ISBN 978-9984-48-065-7. (Ieguldījums publikācijā ~30%).
11. Nikiforova, O., Sukovskis, U., Gusarovs, K. Application of the Two-Hemisphere Model Supported by BrainTool: Football Game Simulation. *AIP Conference Proceedings*, 2015, Volume 1648, No. 1, id. 310004. Available from: doi:10.1063/1.4912557. (Web of Science). (Ieguldījums publikācijā ~25 %).

Aizstāvēšanai izvirzītās tēzes

1. Divpusložu modelis ir piemērots sistēmas modelēšanai, jo tas apvieno sevī ER-diagrammai līdzīgo domēna modeļa atspoguļošanas veidu un biznesa procesu modeli.

Šie modeļi tiek uzskatīti par piemēroto avota modeli, kas ir saprotams visām iesaistītām pusēm programmatūras izstrādes procesā.

2. Tā kā, ir iespējams no divpusložu modeļa iegūt UML diagrammas, kas var tikt transformētas programmatūras kodā, ir iespējams veikt arī tiešo koda ģenerēšanu no avota modeļa.

Darba struktūra

Promocijas darbs ir strukturēts šādi. 1. sadaļā tiek apskatīts divpusložu modelis un tā aktuālas transformāciju metodes. 2. sadaļā tiek veikta divpusložu modeļa ierobežojumu analīze un piedāvātas to novēršanas iespējas. 3. sadaļā tiek aprakstīta domēna-specifiskā valoda divpusložu modeļa definēšanai. 4. sadaļa ir veltītā uzlabotajam klašu attiecību noteikšanas algoritmam, bet 5. sadaļa definē pieeju programmatūras koda ģenerēšanai, sakot no mērķa programmēšanas valodas izvēles un beidzot ar koda starpmodeļa iegūšanu. 6. sadaļā ir apskatīts transformācijas likumu pielietošanas piemērs, iekļaujot arī transformāciju rezultātu validāciju un iegūto rezultātu analīzi. 7. sadaļā ir sniegts praktiskā pielietojuma piemērs. Darba nobeigumā tiek izteikti secinājumi par iegūtiem rezultātiem un nākotnes pētījumu virzieni.

1. DIVPUSLOŽU MODELIS UN AKTUĀLAS TRANSFORMĀCIJAS METODES

Kā tika definēts promocijas darba ievadā, tā autors par modeli, kas ir piemērots koda ģenerēšanai, uzskata modeli, kas ļauj definēt domēna modeli ER-diagrammām līdzīgā formātā un aprakstīt sistēmā notiekošos biznesa procesus ar modeli, kas ir līdzīgs biznesa procesu diagrammām. Šāda sākotnēja modeļa izvēle ir pamatota gan ar autora pieredzi programmatūras izstrādē, gan ar citu autoru pētījumu rezultātiem, gan arī ar biznesa analītiķu un programmatūras izstrādātāju aptaujas rezultātiem. Iegūtie rezultāti ļauj secināt par vienotas modelēšanas valodas (UML) vājo piemērotību sākotnējai sistēmas modelēšanai un nepieciešamību pēc modeļa, kas ir saprotams ne tikai programmatūras izstrādātājiem, bet arī biznesa analītiķiem un, potenciāli, sistēmas pasūtītājiem un lietotājiem.

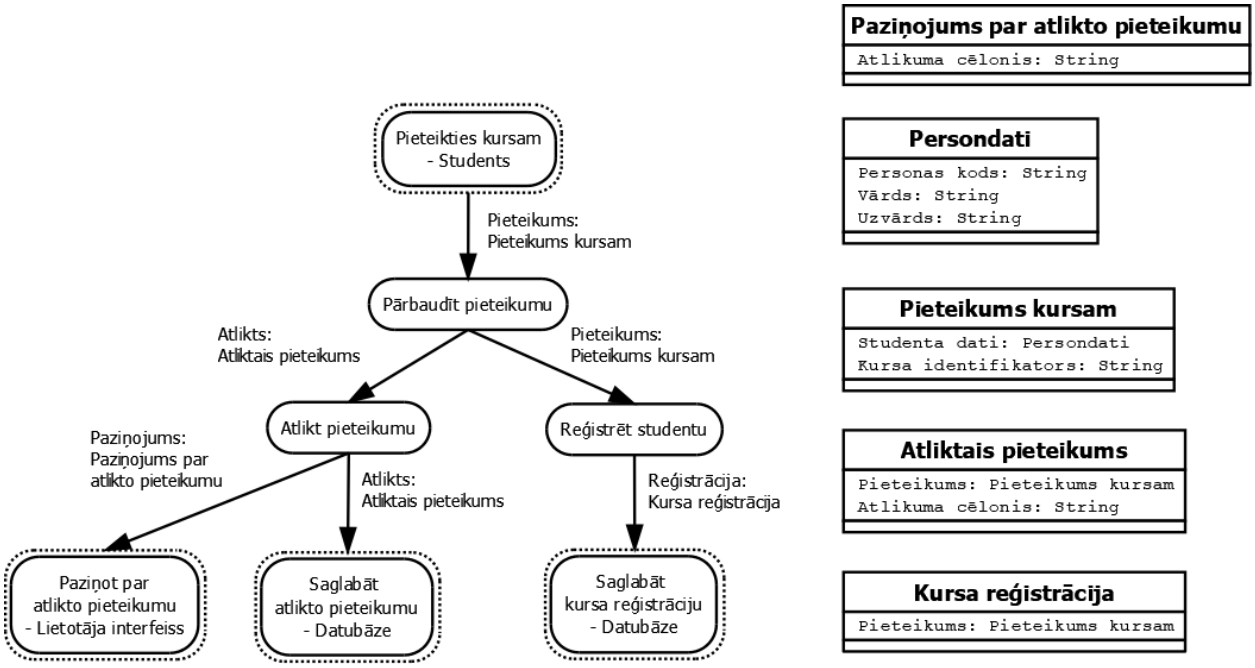
Līdz ar to promocijas darba ietvaros kā tāds modelis tika izvēlēts divpusložu modelis, kas arī tiek aprakstīts šajā sadaļā.

1.1. Divpusložu modelis

Divpusložu modeļa nosaukums tika izvēlēts balstoties uz kognitīvās psiholoģijas uzskatiem [2], kas cilvēka smadzenes dala divās puslodēs, kur viena atbild par loģiku, otra – par konceptiem. Ir nepieciešama abu pusložu koordinēta darbība, lai nodrošinātu cilvēka normālu funkcionēšanu. Līdzīgs princips tiek izmantots arī divpusložu modelī. Tas sastāv no divām diagrammām, kas ir biznesa procesu diagramma, kas tiek attēlota modeļa kreisajā daļā (līdzīgi smadzeņu kreisajai puslodei, kas atbild par loģiku), un konceptu diagramma, kas apraksta datus, kas tiek izmantoti modelī. Biznesa procesu diagramma pēc savas būtības ir līdzīga datu plūsmu diagrammai – tā satur ārējus un iekšējus procesus, kur ārēji procesi nozīmē datu saņemšanu no ārpuses, vai arī nodošanu uz ārpusi [68],[80]. Vēl viens biznesa procesu diagrammas elements ir datu plūsmas, kas savieno procesus, definējot izpildes secību un procesu ieejas un izejas datus. Konceptu diagramma divpusložu modelī ir līdzīga ER-diagrammai, tomēr tās notācija atšķiras. Konceptu diagramma satur konceptus, kas reprezentē domēna modeļu elementus. Koncepti satur atribūtus, kas ir vai nu primitīvie datu tipi, vai nu citi koncepti, vai arī abu datu tipu masīvi (jeb kolekcijas) [73].

Abas diagrammas ir savstarpēji saistītas. Tas tiek nodrošināts ar konceptu no konceptu diagrammas piesaisti datu plūsmām biznesa procesu diagrammā. Šī piesaiste ļauj ne tikai savienot abas diagrammas modelī, bet arī nodrošina to, ka visām datu plūsmām ir stingri definēts pārnesamo datu tips [73]. Tas nozīme, ka katram procesam ir skaidri zināms to ieejas un izejas datu formāts. Papildus ir iespējams definēt procesu izpildītājus, kas var būt sistēmas lietotāji, citas sistēmas, vai arī abstrakti jēdzieni – piemēram, datubāze.

1.1. att. ir parādīts divpusložu modeļa piemērs. Tiek aprakstīts process, kur students veic pieteikšanos kursam. Sākumā tiek pārbaudīts, vai saņemtais pieteikums ir pareizs – gadījumā, kad tas tā nav, tiek atgriezts ziņojums par nepareizu pieteikumu. Pēc tam tiek pārbaudīts, vai ir iespējams pieņemt šo studentu (piemēram, konkursa dēļ). Ja tas tā nav, pieteikums tie uzglabāts atlikto pieteikumu datubāzē, gadījumam, ja dotajā kursā būs pieejamas papildus vietas. Par to arī tiek paziņots studentam. Citādi, students tiek reģistrēts kursam, un informācija par studentu tiek saglabāta reģistrēto studentu datubāzē.



1.1. att. Divpusložu modeļa piemērs

1.2. Transformācijas metodes

Uz promocijas darba izstrādes brīdi eksistē vairākas divpusložu modeļa transformācijas metodes, kas var tikt izmantotas dažāda veida artefaktu iegūšanai no tā:

- Darbs [84] piedāvā metodi UML [113] klašu diagrammas iegūšanai no divpusložu modeļa. Transformācijas nolūkiem avota modelis tiek pārveidots par starpmodeli, kas savukārt tiek izmantots komunikācijas diagrammas iegūšanai. Komunikācijas diagramma, savukārt kalpo par avota modeli rezultējošai klašu diagrammai. Klašu savstarpēju attiecību noteikšanai tiek izmantota informācija no biznesa procesu diagrammas – ieejošo un izejošo datu plūsmu skaits attiecīgiem procesiem.
- Darbā [78] tiek piedāvāti vairāki uzlabojumi iepriekšējai transformāciju metodei ar mērķi iegūt precīzāko klašu diagrammu, kā arī precīzākas ģenerējamo klašu specifikācijas. Šī mērķa sasniegšanai darba autori piedāvā papildus transformācijas soli, kas ir domāts papildus informācijas iegūšanai – pusautomātisko iespējamo parēju matricu veidošanu. Šīs papildus solis ir domāts informācijas iegūšanai par konceptu savstarpējo saistību un ietver sevī vairāku jautājumu uzdošanu. Katrs no jautājumiem tiek formulēts kā "Vai ir iespējams no koncepta `A` iegūt konceptu `B`".
- Darbs [81] ir veltīts UML secību diagrammu ģenerēšanai no divpusložu modeļa. Piedāvātā metode ir balstīta uz biznesu procesa modeļa struktūras analīzi. Transformāciju laikā tiek definēti šādi secību diagrammu elementi: aktieri, objekti, ziņojumi, kā arī paralēlas mijiedarbības fragmenti.
- Darbā [79] tā autori pirmoreiz piedāvā alternatīvo pieeju divpusložu modeļa transformēšanai. Biznesa procesu modelis tiek salīdzināts ar galīgo automātu [57],[100],[121], un transformāciju pamatā atrodas tas fakts, ka jebkuru galīgo automātu ir iespējams pārveidot par regulāro izteiksmi. Šo pārveidošanu rezultātā ir iespējams iegūt starpmodeli, kas tiek izmantots kā pamats secību diagrammas veidošanai. Transformāciju rezultātā tiek iegūta UML secību diagramma, kas satur aktierus, objektus, ziņojumus, kā arī divu veidu mijiedarbības fragmentus – cikla un alternatīvo izpildi.
- Darbs [39] ir iepriekšejas transformācijas uzlabojums, kas ir veltīts programmatūras koda iegūšanai no divpusložu modeļa "pa taisno", tas ir, neizmantojot UML diagrammas. Rezultātā darba autori piedāvā algoritmu, ka ļauj iegūt programmatūras kodu ar vairākiem ierobežojumiem.

- Darbs [40] ir pamats šīm promocijas darbam. Tajā tiek turpmāk attīstītas idejas, kas tika aprakstītas darbos [39] un [79] – biznesa procesu modeļa salīdzināšana ar galīgo automātu un tā apstrāde, izmantojot dažādus algoritmus.

Var redzēt, ka divpusložu modelim ir definētas arī vairākas transformāciju metodes, kas ļauj iegūt gan dažāda veida UML diagrammas, gan arī citus artefaktus, piemēram, daļu no programmatūras koda. Pēdējie pētījumi dotajā jomā ir veltīti grafu apstrādes algoritmu pielietošanai biznesa procesu modeļa transformēšanai un programmatūras koda iegūšanai no avota modeļa. Promocijas darbā tiek attīstītā šī pieeja.

2. ESOŠIE DIVPUSLOŽU MODEĻA LIETOŠANAS IEROBEŽOJUMI KODA ĢENERĒŠANAS UZDEVUMĀ

Uz promocijas darba izstrādes sākuma brīdi transformācijas likumi, kas ir paredzēti divpusložu modeļa apstrādei ļauj iegūt galvenokārt statisku informāciju – klašu sarakstu, to atribūtus, un izpildāmās metodes, kas definē projektējamās sistēmas uzvedību kopā ar informāciju par attiecībām starp šīm klasēm [82]. Lai gan ir piedāvāta arī metode secību diagrammu ģenerēšanai no procesu diagrammām, kas tiek ietvertas divpusložu modelī [81] – modeli, kas definē projektējamās sistēmas dinamiku – šī metode nav pilnīga, jo autori atzīmē faktu, ka nav līdz galam atrisināts jautājums par mijiedarbības fragmentu ģenerēšanu no procesu diagrammas. Lai gan ir iespējams noteikt mijiedarbības fragmenta ietvarus, nav skaidrs, kāda veida fragments būtu jāģenerē – tā var būt paralēla izpilde, vai alternatīva. Līdz ar to ir iespējams secināt, ka esošie divpusložu modeļa transformācijas likumi ir ierobežoti ar statiskas informācijas ģenerēšanu, kas nozīmē to, ka daļa no informācijas, kas atrodas avota modelī, netiek izmantotā pilnā apjomā.

Šāds fakts nozīme to, ka esošās transformācijas ir nepieciešams uzlabot un papildināt, lai būtu iespējams pārnest vairāk informācijas no avota (divpusložu) modeļa uz mērķa modeli jeb programmatūras kodu. Tomēr, pirms metožu uzlabojuma ir nepieciešams veikt arī paša divpusložu modeļa analīzi ar mērķi noteikt, vai pašreizēja notācija ļauj iegūt visu nepieciešamo informāciju programmatūras koda ģenerēšanai. Šī analīze tiek aprakstīta šajā nodaļā.

2.1. Ierobežojumi divpusložu modeļa notācijā

Tā kā promocijas darba uzdevums ir koda ģenerēšana no divpusložu modeļa, ir nepieciešams detalizēti apskatīt mērķa modeli, lai saprastu, kāda papildus informācija, salīdzinot ar avota modeli, tiek ietverta tajā, un vai ir nepieciešams vismaz daļu no šīs informācijas pievienot divpusložu modeļa notācijai. Ja tas tā ir, tad var secināt to, ka pašreizējā divpusložu modeļa notācijā ir identificēts ierobežojums, ko ir nepieciešams atrisināt.

Mērķa modeli jeb programmatūras kodu ISO/IEC 2382:2015 standarts definē kā "sintaktisko vienību, kas atbilst noteiktas programmēšanas valodas likumiem un sastāv no deklarācijām, instrukcijām vai operatoriem, kas ir nepieciešami noteikta uzdevuma vai

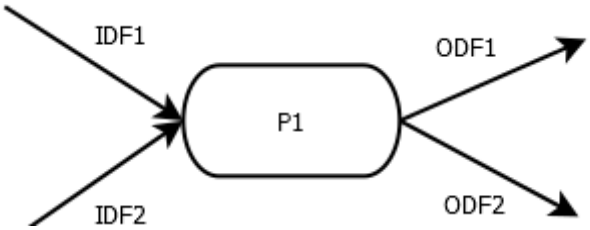
problēmas risināšanai" [47]. Analizējot šo definīciju, var redzēt, ka programmatūras kods faktiski sastāv no divām pamatdaļām:

1. Deklarācijas ir programmēšanas valodas konstrukcijas, kas definē vienu vai vairākus identifikatorus un to interpretācijas veidus.
2. Instrukcijas vai operatori savukārt definē izpildāmas darbības, to operandus un iegūstamos rezultātus. Instrukciju vai operatoru secība definē vienu vai vairākus algoritmus, kas ļauj programmatūras kodam sasniegt nepieciešamos rezultātus.

Analizējot abas mērķa modeļa sastāvdaļas, var redzēt, ka tās pēc būtības atbilst divpusložu modelī esošajām diagrammām. Ir iespējams sasaistīt konceptuālo datu modeli ar programmatūras koda deklarācijām, jo to uzdevums ir līdzīgs. Tapāt var arī runāt par biznesa procesu diagrammas pārveidošanu instrukciju vai operatoru secībā. Esošie divpusložu modeļa transformācijas likumi ļauj pārveidot sistēmas konceptuālo modeli par klašu kopu, kas pašreiz ir pietiekami deklarāciju definēšanai. Tomēr, ir jāņem vērā fakts, ka pašreiz aktuālās klašu kopas ģenerēšanas metodes, kas definētas [82], izmanto informāciju no biznesa procesu modeļos iekļautās informācijas, lai noteiktu attiecības starp klasēm. Lai gan tas ir pareizi no tāda skatu punkta, ka informācija par to, kuras klases komunicē savā starpā, ļauj noteikt tādas attiecības kā asociācija un atkarība, tas ne vienmēr ļauj secināt par mantošanu, realizāciju un agregāciju. Šai informācijai būtu jārodas arī no uzģenerētās klašu kopas – pēc klašu kopas ģenerēšanas ir nepieciešams veikt papildus analīzi papildus klašu attiecību noteikšanai. Tas nenozīmē, ka ir problēmas ar divpusložu modeļa notāciju, lai ģenerētu deklarācijas, tomēr tam tiks pievērsta uzmanība, veicot turpmāko analīzi. Kopumā ir iespējams secināt, ka pašreizējā divpusložu modeļa notācija pilnīgi atbalsta koda deklarāciju ģenerēšanu, un papildus izmaiņas šeit nav vajadzīgas.

Savukārt, instrukciju jeb operatoru secības ģenerēšana uz doto brīdi tiek veikta tikai UML secību diagrammas formā [81]. Šīs informācijas ģenerēšanai tiek izmantota sākotnējā informācija no biznesa procesu modeļa. Vienkāršu instrukciju ģenerēšana var tikt pilnīgi atbalstīta ar esošo divpusložu modeļa notāciju. Par vienkāršām instrukcijām tiek uzskatīti metožu izsaukumi, kas notiek secīgi viens pēc otra – bez zarošanās. Tomēr bez vienkāršām instrukcijām eksistē arī citas – alternatīvas un cikli. Veicot procesu modeļa analīzi, ir iespējams noteikt ciklus tajā, tomēr nav skaidrs, pie kura noteikuma cikls turpina izpildīties, pie kura tiek apstādināts. Tas pats attiecas uz izpildes alternatīvām. Ja vienā procesā ieiet vairākas datu plūsmas – tas var nozīmēt, vai nu to, ka process var izpildīties tikai saņemot visu nepieciešamo informāciju, vai arī to, ka process var turpināt strādāt, saņemot tikai daļu no tās. Tas pats attiecas uz no procesa izejošām datu plūsmām – vai process veido vairākas plūsmas vienlaicīgi, vai arī

veido tikai kādu no datu plūsmām, kas nozīmētu alternatīvas darba secības izpildes iespējas. Šādas situācijas piemērs ir sniegts 2.1. att.



2.1. att. Biznesa procesu modeļa sastāvdaļa, kas satur iespējamas alternatīvas

Šajā attēla ir redzams process `P1`, kas saņem divas ienākošas datu plūsmas – `IDF1` un `IDF2`. Savukārt, šis process arī veido divas datu plūsmas – `ODF1` un `ODF2`. Izmantojot tikai šo informāciju, nav iespējams noteikt, vai procesam ir nepieciešamas abas ieejas, vai tikai viena no tām. Tapāt, nav skaidrs, vai procesa izpildes rezultātā parādīsies abas datu plūsmas, vai tikai viena.

Analizējot šo informāciju, ir iespējams secināt, ka pašreizējā divpusložu modeļa notācija nesatur informāciju, kas ļautu definēt procesu izpildes nosacījumus. Tas arī tiek atzīmēts kā pirmais divpusložu modeļa notācijas ierobežojums koda ģenerēšanas uzdevuma izpildē. Uzreiz ir iespējams definēt arī otro notācijas ierobežojumu – vairāku izejošo datu plūsmu gadījuma nav skaidrs, kādus datus biznesa process producē.

Pašreizējie divpusložu modeļa transformācijas likumi ir balstīti uz faktu, ka jebkura procesu modeļa datu plūsma tiek asociēta ar vienu un tikai vienu konceptu no konceptuālā modeļa. Analizējot mūsdienu programmēšanas valodu iespējas ir iespējams definēt, ka šāda pieeja var ierobežot koda ģenerēšanu, kā arī problēmsfēras aprakstīšanas iespējas. Procesi no biznesa procesu diagrammas tiek pārveidot par klašu metodēm, kamēr datu plūsmas tiek pārveidotas par to parametriem un atgriežamajām vērtībām. Metodes mūsdienu programmēšanas valodās var saņemt vairākus argumentus, kas var būt gan primitīvi datu tipi, gan klases, gan arī primitīvo datu tipu un objektu kolekcijas. Tas pats ir attiecināms uz metožu izpildes rezultātiem. Papildus ir jāatzīmē, ka eksistē metodes, kurām nav nepieciešami gan ieejas, gan izejas parametri.. Zemāk ir doti vairāki piemēri šādām klašu metodēm no programmēšanas valodas Java klases `java.awt.Graphics` [50]:

1. Metode `Graphics Graphics.create()` nesaņem nevienu parametru un atgriež objektu kā izpildes rezultātu.

2. Metode `void Graphics.dispose()` nesaņem nevienu parametru, kā arī neko neatgriež.
3. Metode `Graphics Graphics.create(int x, int y, int width, int heigth)` kā parametrus saņem vairākus primitīvus datu tipus, atgriežot objektu kā izpildes rezultātu.
4. Metode `void Graphics.fillPolygon(int[] xPoints, int[] yPoints, int nPoints)` kā parametrus saņem gan primitīvu datu masīvus, gan arī primitīvu datu tipu.
5. Metode `boolean Graphics.drawImage(Image img, int x, int y, ImageObserver observer)` kā parametrus saņem gan primitīvus datu tipus, gan objektus, atgriežot primitīvu datu tipu kā izpildes rezultātu.

Protams, šie piemēri nav vienīgie – ir iespējams definēt arī citus, bet jau vienas klases ietvaros ir iespējams redzēt to, ka klašu metodes var saņemt un atgriezt dažāda veida informāciju, kas ir definēta, izmantojot dažādus datu tipus.

Visu šo ir iespējams attiecināt arī uz biznesa procesiem, kas var saņemt un atgriezt dažādus rezultātus. Lai gan, uz doto brīdi ir iespējams definēt vairākas ieejošās datu plūsmas, citas iespējas netiek atbalstītas. Līdz ar to ir iespējams definēt trešo divpusložu modeļa notācijas ierobežojumu – dati, ko spēj pārnest datu plūsmas, ir ierobežoti ar vienu konceptu uz vienu datu plūsmu.

2.2. Ierobežojumi divpusložu modeļa transformācijas metodēs

Pašreizējie divpusložu modeļa transformācijas likumi definē klašu kopu un attiecības starp tām, balstoties uz informāciju no divpusložu modeļa. Tomēr pēc transformācijas netiek veikta klašu kopas apstrāde, kas nozīmē to, ka papildu attiecības starp uzģenerētajām klasēm netiek definētas. Piemēram, definējot vairākus dažādus konceptus ar līdzīgu struktūru – vienāda tipa atribūtiem ar vienādiem nosaukumiem, metodes nepiedāvā definēt mantošanu starp šādām klasēm, kā arī izdalīt atsevišķu bāzes klasi. Rezultātā tas var novest pie vairākiem atkārtojumiem programmatūras kodā, kas var ietekmēt tā uzturēšanas iespējas – jo izmaiņas būs nepieciešams veikt vairākās vietās. Tātad, pirmais esošo transformāciju ierobežojums ir uzģenerētās klašu kopas analīzes posma neesamība.

Turpinot esošo transformācijas metožu analīzi, var redzēt, ka tās ģenerē tā saucamo bagāto datu modeli. Jebkuras programmatūras sistēmas galvenais uzdevums vienmēr var tikt

reducēts uz datu apstrādes uzdevumu. Programmatūra atbild par datu ievadi, to apstrādi un apstrādes rezultātu izvadi, kas var tikt darīts dažādos veidos.

Objektorientētā paradigma paredz programmatūras komponenšu definēšanu objektu veidā. Arī dati ir daļa no programmatūras, līdz ar to objektorientētā programmēšanas pieejā dati arī tiek definēti kā objekti. Objektorientētai paradigmai attīstoties, parādījās divas galvenās pieejas datu definēšanai [13],[106] – anēmiskais [24],[26] un bagātie datu modeļi.

Bagātā datu modeļa pamatā atrodas četri galvenie objektorientētās paradigmas pamatprincipi: mantošana, iekapsulēšana, abstrakcija un polimorfisms. Veidojot šādu datu modeli, domēna klases satur ne tikai datus, bet arī to apstrādes loģiku. Lai gan šī pieeja vairāk atbilst objektorientētai paradigmai, tai arī piemīt vairāki ierobežojumi:

1. Izmantojot bagāto datu modeli, ne vienmēr ir iespējams realizēt MVC arhitektūru [111], jo modelis satur daļu no kontroliera loģikas, šādā veidā, apvienojot divus neatkarīgus arhitektūras slāņus. Rezultātā tiek zaudēta viena no MVC arhitektūras galvenajām priekšrocībām – ne vienmēr ir iespējams aizvietot objektu tikai vienā no slāņiem. Piemēram, jauna kontroliera ieviešana var prasīt arī izmaiņu veikšanu modelī.
2. Izstrādājot dalītās sistēmas (angl. *distributed systems*), kuru sastāvdaļas ir realizētas dažādās programmēšanas valodās, var rasties nepieciešamība pēc datu apstrādes loģikas atkārtošanas dažādās dalītās sistēmas komponentēs. Piemēram, ja šādas sistēmas komponentes apmainās ar domēna līmeņa objektiem, kas satur papildus loģiku, tad bez vienkāršas datu struktūras ir nepieciešams arī nodot informāciju par izpildāmo kodu, vai arī dublēt to. Rezultātā izmaiņu veikšana prasa papildus darbu, kas savukārt palielina kļūdu rašanās iespēju.

Vairāki autori [24],[26],[59] uzskata, ka loģikas apvienošana ar domēna modeli, atbilst objektorientētās paradigmas principiem, taču pārkāpj citus principus – tādus kā koncepciju atdalīšana. Šie autori piedāvā programmatūras izstrādē izmantot citu pieeju datu modelēšanai – anēmisko datu modeli, kura pamatprincips ir "dati ir dati". Šīs pieejas ietvaros tiek uzskatīts, ka objektiem, kas apraksta datus, nav nepieciešami papildus atribūti un metodes, kas neatbild par informācijas uzglabāšanu. Anēmiskais datu modelis ļauj atdalīt datus no to apstrādes loģikas.

Kā papildus anēmiskā datu modeļa priekšrocību var definēt tā savienojamību ar esošajiem objektu-relāciju kartēšanas ietvariem, piemēram, Hibernate [41], kā arī to, ka šāda pieeja vairāk atbilst MVC arhitektūras [111] principiem. MVC arhitektūra tiek plaši izmantota

mūsdienu ietvaros, piemērām, AngularJS [3]. MVC pamatprincips ir programmatūras sistēmas sadalīšana trīs komponentēs:

1. Modelis (angl. *Model*) atbild par informācijas uzglabāšanu un informācijas pieprasījumu apstrādi.
2. Skats (angl. *View*) atbild par modeļa attēlošanu lietotājam uzskatāmajā formā.
3. Kontrolieris (angl. *Controller*) apvieno modeli un skatu, definējot biznesa loģiku.

Mūsdienās MVC arhitektūra tiek attīstīta un vairāki autori, piemēram, [59] runā par tā saucamo MSVC, jeb *Model-Service-View-Controller* arhitektūru, kas pievieno papildus servisu slāni starp modeli un kontrolieri. Servisu slānis ir atbildīgs par biznesa loģikas realizāciju, kamēr kontrolieris MSVC gadījumā tikai un vienīgi savieno servisus un skatus, kas dod papildus elastīgumu.

Veicot MVC un MSVC pieejas analīzi, ir iespējams redzēt, ka anēmiskais datu modelis ir vairāk piemērots to realizācijai, lai gan var izskatīties, ka tas daļēji neatbalsta objektorientētās programmēšanas pamatprincipus. Tomēr, mūsdienu ietvari, piemēram, AngluarJS [3] bieži to izmanto, jo tas ļauj atdalīt datus no biznesa loģikas.

Zemāk tiek apskatīts vienkāršs piemērs, kas parāda atšķirību starp bagāto un anēmisko datu modeli. Attēlā 2.2. ir dots vienkāršs klašu hierarhijas piemērs, kas ir paredzēts dzīvnieku reprezentācijai.

```
abstract class Animal:
      name: String
      sayHello: abstract function() → String

class Dog extends Animal
      sayHello: function() → "Woof!"

class Cat extends Animal
      sayHello: function() → "Meow!"
```

2.2. att. Dzīvnieku klašu hierarhija

Dotajā pseidokoda fragmentā ir sniegts viens no klasiskiem polimorfisma piemēriem: eksistē divas klases `Cat` un `Dog`, kas manto no klases `Animal`, pārdefinējot vienu no tās metodēm. Rezultātā tiek iegūtas 3 klases ar vienu un to pašu interfeisu jeb kontraktu, kas ļauj abstrahēties no konkrētās realizācijas. Savukārt, anēmiskais datu modelis definē šādu klašu hierarhiju kā nepareizu, pamatojoties uz faktu, ka klases tajā satur ne tikai datus, bet arī daļu no programmatūras loģikas – šajā gadījumā metodi `sayHello()`. Vadoties pēc anēmiskā datu

modeļa principiem šādai klašu hierarhijai ir jābūt pārdefinētai (pie tam pārkāpjot arī objektorientētos abstrakcijas un iekapsulēšanas pamatprincipus) – skat. 2.3. att.

```
enum AnimalType { CAT, DOG }

class Animal:
      name: String
      type: AnimalType

class AnimalHelloService:
      sayHello: function(animal: Animal): String {
            if (animal.type is CAT) → "Meow!"
            if (animal.type is DOG) → "Woof!"
            …
      }
```

2.3. att. Anēmiskā dzīvnieku klašu hierarhija

Tomēr, pēc šī darba autora, kā arī citu autoru (piemēram, [106]) pieredzes, dati jeb domēna objekti reālas dzīves gadījumos gandrīz nekad nesatur programmatūras loģiku – tie var būt domāti datubāžu tabulu attēlošanai programatūras kodā, vai arī datu saņemšanai, uzglabāšanai un izvadei, kas nozīme to, ka:

1. Mantošana domēna klasēs ir nepieciešama tikai atribūtu pievienošanai.
2. Datiem parasti nav raksturīgs polimorfisms – atšķirība star dažādām datu klasēm var tikt definēta ar atribūtu pievienošanas palīdzību mantošanas ietvaros.
3. Parasti servisi, kas nodarbojas ar datu apstrādi, strādā ar konkrētiem tipiem, minimizējot nepieciešamību pēc abstrakcijas, iekapsulēšanas un polimorfisma domēna objektos.
4. Vairāki eksistējoši datu apstrādes ietvari un bibliotēkas, piemēram, plaši izmantojamais objektu-relāciju kartēšanas ietvars Hibernate [41], izmanto tieši anēmisko datu modeli.

Ņemot vērā iepriekš minētos faktorus, promocijas darba autors uzskata, ka anēmiskais datu modelis nav anti-paraugs, un tā izmantošana ir pilnīgi pieļaujama arī modeļvadāmā programmatūras izstrādē.

Tātad, ir redzams, ka anēmiskais datu modelis tiek plaši izmantots industrijā, vairāki plaši izmantotie ietvari ir balstīti tieši uz šī datu modeļa pielietošanas, un fakts, ka esošie divpusložu modeļa transformācijas likumi neatbalsta anēmisko datu modeli [76], var tikt uzskatīts par metožu ierobežojumu, jo perspektīvā tas var ietekmēt metodes ieviešanas iespējas – uzģenerētais kods prasīs papildus pielāgošanu eksistējošajiem ietvariem un risinājumiem. Šajā darbā tas tiek definēts kā otrais esošo transformācijas metožu ierobežojums.

Nākamais esošo transformāciju ierobežojums ir neiespējamība noteikt nosacījumus metožu izpildei un zarošanai uzģenerētās secību diagrammas gadījumā. Šis ierobežojums gan izriet no divpusložu modeļa notācijas ierobežojuma, kas ir neiespējamība definēt palīginformāciju šādu konstrukciju noteikšanai transformācijas brīdī. Tomēr, šī darba autors papildus atzīmē arī šo kā transformācijas metožu ierobežojumu.

2.3. Divpusložu modeļa lietošanas ierobežojumi koda ģenerēšanas uzdevumā

Kā ir parādīts iepriekšējās sadaļās, gan divpusložu modeļa esošai notācijai, gan arī transformācijas likumiem piemīt vairāki ierobežojumi, kas var apgrūtināt vai padarīt par neiespējamu koda ģenerēšanas uzdevumu. Zemāk, 2.1. tabulā ir dots kopsavilkums visiem definētajiem ierobežojumiem un to risināšanas piedāvājumi.

2.1. tabula

Esošie divpusložu modeļa notācijas un transformācijas metožu ierobežojumi

Ierobežojums	Atrašanās vieta	Risināšanas iespējas
Modelī nav iespējams definēt procesu izpildes nosacījumus	Notācija	Divpusložu modeļa notācija ir jāpapildina ar elementiem, kas ļauj noteikt izpildes kārtību
Vairāku izejas datu plūsmu gadījumā nav iespējams definēt, vai process var producēt tikai vienu, vai arī vairākas datu plūsmas	Notācija	Pastāv vairākas risināšanas iespējas. Var katrai izejošajai datu plūsmai pievienot informāciju par to, vai tā vienmēr tiks radīta procesa rezultātā. Otra iespēja ir apvienot visas procesa producētās datu plūsmas vienā objektā, kas saturēs atbilstošos atribūtus, kuru vērtības kā arī pašu producēšanas faktu var noteikt procesam atbilstošajā programmatūras kodā.
Dati, ko spēj pārnest datu plūsmas ir ierobežoti ar vienu konceptu, kas ir piesaistīts datu plūsmai. Primitīvu datu tipu un objektu masīvu/kolekciju definīcija datu plūsmas ietvaros nav iespējama. Tāpat nav iespējams definēt datu plūsmu, kas datus nepārnes.	Notācija	Ir nepieciešams mainīt veidu, kā dati (jeb datu tipi) tiek piesaistīti datu plūsmām divpusložu modelī. Notācijai ir jāpiedāvā iespējas piesaistīt datu plūsmai primitīvos datu tipus, masīvus/kolekcijas, kā arī nepiesaistīt vispār – šajā gadījuma datu plūsma var tikt dēvēta par kontroles plūsmu, jo tā vairs neatbild par informācijas apmaiņu, bet gan definē izpildes secību.

2.1. tabula (turpinājums)

Pēc klašu kopas ģenerēšanas, tā netiek analizēta, rezultātā ir iespējama uzģenerētā programmatūras koda atkārtojumi, kā var nebūt noteiktas visu klašu attiecības.	Transformācijas	Ir nepieciešams definēt algoritmu vai algoritmus, kas var tikt izmantoti klašu kopas apstrādei papildus attiecību noteikšanai.
Divpusložu modeļa transformācijas likumi neatbalsta anēmiskā datu modeļa ģenerēšanu [76].	Transformācijas	Tā kā anēmiskais datu modelis ir vairāku mūsdienu ietvaru un risinājumu pamatā, ir nepieciešams atbalstīt šāda datu modeļa ģenerēšanas iespējas. Turpmāk analizējot mūsdienu ietvarus un pieeju programmatūras izstrādei, var secināt, ka bagātā datu modeļa ģenerēšana sākotnēji vispār var būt neatbalstīta – tieši anēmiskais datu modelis ir pieprasītāks industrijā.
Lai gan secību diagrammu ģenerēšanas algoritms atbalsta mijiedarbības fragmentu veidošanu, tas nespēj definēt šādu fragmentu tipus un nosacījumus to izpildei.	Transformācijas	Šīs ierobežojums izriet no fakta, ka avota modelis nesatur informāciju, kas varētu palīdzēt šādu konstrukciju definēšanai. Līdz ar to ir nepieciešams papildināt gan modeļa notāciju, kas jau ir minēts iepriekš, gan arī transformācijas likumus notācijas papildinājuma atbalstam.

3. DOMĒNA-SPECIFISKĀ VALODA DIVPUSLOŽU MODEĻA DEFINĒŠANAI

Iepriekšējā sadaļā tika apskatīti esošie divpusložu modeļa un tā transformācijas metožu ierobežojumi, kuru novēršana prasa gan izmaiņas modeļa notācijā, gan arī pašās transformācijās. Transformācijas metožu izmaiņas ir attiecīgo algoritmu modificēšana un jaunu algoritmu definēšana, kas var tikt veikta jebkurā programmēšanas valodā un nedefinē nekādus ierobežojumus. Notācijas modifikācijas, savukārt, var izmainīt pašu modeļu būtību, un veicot tās, būtu nepieciešams to darīt tādā veidā, kas prasītu pēc iespējas mazāk izmaiņu atbilstošajā programmatūras kodā, kurā ir definēts pats divpusložu modelis un tā elementi.

Iepriekšējie pētījumi, kas bija veltīti divpusložu modeļa un transformāciju izmantošanai, tika veidoti ar attiecīgiem programmatūras rīkiem, kas atbalsta divpusložu modeļa grafisko notāciju – BrainTool [77] un BrainTool 2.0 [82]. Lai gan šie rīki piedāvā uzskatāmu divpusložu modeļa vizualizāciju, promocijas darba autors neuzskata, ka tie ir piemēroti eksperimentiem,

kas tiek veikti ar divpusložu modeli, mainot tā notāciju – grafiskais rīks var prasīt salīdzinoši daudz papildus darba jauna elementa pievienošanai [73],[85] – jo ir nepieciešams atbalstīt tā vizuālo attēlojumu un definēt nepieciešamos interfeisa elementus darbam ar jauno notācijas elementu. Notācijas uzlabošana pati par sevi paredz eksperimentēšanu – jauni elementi var tikt pievienoti, esošie elementi izņemti vai modificēti iteratīvā veidā līdz modelis satur pietiekamu informācijas daudzumu. Izmantojot esošos rīkus, jebkurai notācijas izmaiņai būtu nepieciešams veikt nopietnas izmaiņas grafiskajā redaktorā, pat ja eksperiments pats par sevi būs neveiksmīgs.

Līdz ar to ir nepieciešams cits divpusložu modeļa reprezentācijas veids, kas ir viegli modificējams un labāk atbalsta eksperimentus. Šādam modeļa definēšanas veidam tiek definētas vairākas prasības:

1. Ir jāpastāv iespējai izmainīt metamodeli ar salīdzinoši nelielām pūlēm. Modeļvadāmas izstrādes gadījumā modeļi parasti tiek definēti ar grafisko diagrammu palīdzību, kas ir cilvēkam saprotams un viegli uztverams definēšanas veids, bet, kas prasa papildus darbu, ieviešot izmaiņas metamodelī, jo ir nepieciešams izmainīt ne tikai uzglabāšanas, bet arī attēlošanas loģiku.
2. Jaunajam modeļa definēšanas veidam ir jābūt pēc iespējas vienkāršākam no cilvēka uztveres skata punkta, tādā veidā, saglabājot grafiskās reprezentācijas priekšrocības.
3. Jaunajam modeļa definēšanas veidam ir jābūt pēc iespējas vienkāršākam no apstrādes skata punkta, kas nozīmē nepieciešamību samazināt papildu koda, kas ir nepieciešams modeļa reprezentācijas pārveidošanai transformācijas algoritma datu struktūrās, apjomu.

Analizējot šīs prasības, promocijas darba autors ir nonācis pie secinājuma par domēna-specifiskās valodas (angl. *Domain Specific Language* – DSL) izmantošanas iespēju, kas tiek apskatīta šajā nodaļā.

3.1. Domēna-specifiskās valodas

Salīdzinājumā ar universālajām programmēšanas valodām (angl. *General Purpose Language* – GPL), domēna-specifiskās valodas ir paredzētas konkrētas problēmas risināšanai, vai arī konkrēta biznesa domēna informācijas precīzākai definēšanai [9],[63]. Šīs valodas universalitāte tiek samazināta, palielinot ekspresivitāti konkrētajā jomā. DSL valodu

izmantošana ļauj ieviest papildus verifikācijas, analīzes un optimizācijas iespējas, kas ne vienmēr ir sasniedzamas, izmantojot universālās programmēšanas valodas [63].

Piemēram, ir iespējams runāt par XML valodas (angl. *eXtensible Markup Language*) dialektiem kā DSL piemēru – kamēr universālais XML var tikt izmantots jebkuras informācijas definēšanai, tā dialekti ir paredzēti konkrēta veida informācijas definēšanai, un parasti ļauj to panākt efektīvāk un ar augstāku ekspresivitāti.

DSL izstrāde ļauj iegūt papildu priekšrocības, bet tas var būt sarežģīts uzdevums, un rezultējošās valodas izmantošanas priekšrocībām ir jākompensē tās izstrādes izmaksas [63]. DSL ir vērts izstrādāt gadījumos, kad viens vai vairāki tās potenciālie lietotāji ir biznesa domēna, nevis programmatūras izstrādes eksperti – papildus ekspresivitāte atvieglos izstrādātās valodas uztveršanu.

Domēna-specifiskās valodas veidošanai ir nepieciešams definēt problēmas domēna galvenos elementus un izveidot to reprezentācijas, izmantojot izvelēto sintaksi. Kad visas bāzes konstrukcijas ir definētas, ir nepieciešams definēt validācijas un semantikas likumus, šādā veidā noslēdzot DSL izstrādi. Atkarībā no izvēlētās DSL sintakses, ir iespējams to aprakstīt ar dažāda veida notāciju palīdzību. Piemēram, ja valoda ir balstīta uz XML, ir iespējams definēt konkrēta XML dialekta shēmu, izmantojot XML shēmas definēšanas valodu [117]. Tekstuālas valodas gadījumā ir iespējams aprakstīt tās elementus ar paplašināto Bekusa-Naura formu (angl. *Extended Backus-Naur Form* – EBNF) palīdzību [34]. Promocijas darba autors izmanto tieši šo pieeju.

Modeļvadāmās izstrādes gadījumā par DSL bieži uzskata grafisko modeli [9],[11],[54],[63],[115]. Pēc šī darba autora domām tas ir saistīts ar lielu teorētisko darbu skaitu šajā jomā. Tikai neliela daļa no vairāku autoru piedāvātajām metodēm ir atbalstīta ar rīkiem, kas ļauj definēt un transformēt attiecīgos modeļus – šāda rīka izstrāde prasa daudz pūles [77],[82] un parasti ierobežo eksperimentēšanas iespējas – jo grafiskās modeļa reprezentācijas izmaiņas parasti prasa lielas izmaiņas izstrādātajā rīkā [71],[73],[77],[85].

3.2. Divpusložu modeli aprakstoša domēna-specifiskā valoda

Kā tika minēts iepriekšējā apakšnodaļā, domēna-specifiskās valodas izstrādei ir nepieciešams definēt problēmsfēras elementus un izveidot to attiecīgās reprezentācijas izvelētajā sintaksē. Šeit par problēmsfēru kalpo divpusložu modelis. Definējot attiecīgās DSL konstrukcijas, promocijas darba autors arī veic vairākus esošās notācijas uzlabojumus, ņemot

vērā prasības, kas tika definētas iepriekšējā sadaļā. Rezultātā tika izveidota divpusložu modeli aprakstoša DSL valoda, kuras elementi un konstrukcijas tiek apskatīti tālāk [37].

Analizējot informāciju par divpusložu modeļa notāciju, ir iespējams identificēt tā galvenos elementus, kas savukārt tiks pārveidoti DSL elementos. Divpusložu modelis sastāv no viena konceptu modeļa un viena vai vairākiem procesu modeļiem. Tā kā konceptu modelis vienmēr eksistē tikai vienā eksemplārā, promocijas darba autors definē četrus galvenos divpusložu modeļa elementus:

1. Process.
2. Koncepts.
3. Datu plūsma.
4. Procesu modelis.

Kā ir parādīts tālāk, šie elementi ir nepieciešami un pietiekami divpusložu modeļa definēšanai un tiek pārveidoti par definētās domēna-specifiskās valodas sintaktiskajiem elementiem.

3.2.1. Divpusložu modeli aprakstošas domēna-specifiskās valodas kopējie elementi

Kā tika definēts iepriekšējā apakšnodaļā, divpusložu modeli aprakstoša valoda ietver četrus galvenos elementus: konceptu, procesu, datu plūsmu un procesu diagrammu. Pirms šie elementi tiek aprakstīti, tiek definēta sintakse elementiem, kas tiek izmantoti visās DSL konstrukcijās. Šo elementu definīcijas izmantojot EBNF-notāciju ir parādītas 3.1. att.

Elements `string` (simbolu virkne) ir paredzēts simbolu virkņu definēšanai un tā sintakse atbilst simbolu virknes literāļa sintaksei programmēšanas valodā Java [50]. Simbolu virkne ir unikoda [112] simbolu kopa, kas tiek ievietota dubultās pēdiņās un var ietvert sevī tā saucamās atsoļa sekvences, kas ļauj definēt speciālos simbolus – tabulācijas, pēdiņas un citus.

Elements `identifier` (identifikators) tiek izmantots divpusložu modeļa elementu identifikatoru definēšanai, tā sintakse atbilst identifikatora sintaksei vairākās programmēšanas valodās (piemēram, Java, C, Pascal) ar vienu izņēmumu – DSL identifikatoros pasvītrošanas simboli nav atļauti. Identifikators sastāv no latīņu burtiem un simboliem, tam ir jāsākas ar burtu.

Elementam `glue` nav speciālas semantiskas nozīmes – šis elements ir domāts atsevišķu valodas elementu sasaistei un var tikt reprezentēts ar divu marķieru – `"WITH"` un `"AND"`

palīdzību. Līdzīgi kā scenāriju valodas Gherkin [19] elementi `And`, `When` un `Then`, šis elements ir paredzēts modeļa definīcijas ekspresivitātes palielināšanai.

```
character          = jebkurš Unicode simbols, izņemot '\' un '"';
escape_sequence    = '\b' | '\t' | '\n' | '\f' | '\r' | '\"' | '\'';
string_character   = character | escape_sequence;
string             = '"' string_character? '"';

identifier_letter = 'A' - 'Z' | 'a' - 'z';
digit              = '0' - '9';
identifier_letter_or_digit
                   = identifier_letter | digit;
identifier         = identifier_letter identifier_letter_or_digit*;

glue               = 'WITH' | 'AND';
```

3.1. att. Divpusložu modeli aprakstošas valodas kopējie elementi

3.2.2. Atribūtu definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

```
cardinality        = 'SINGLE' | 'ARRAY' | 'COLLECTION';
cardinality_def    = 'HAVING' cardinality 'CARDINALITY';

attribute_def      = string '(' identifier ')' cardinality_def?;
```

3.2. att. Divpusložu modeli aprakstošas valodas atribūtu definīcijas

Attēlā 3.2. ir parādītas DSL konstrukcijas, kas ir paredzētas atribūtu definēšanai. Jebkuram atribūtam ir iespējams definēt vārdu, attiecīgo datu tipu, kas var būt primitīvs datu tips vai koncepts (lai gan pašas konstrukcijas nekādā veidā neierobežo to, kas ir datu tips, tas tiek darīts DSL faila apstrādes laikā) un kardinalitāti, kas nosaka vai dotais atribūts ir vienkārša vērtība, masīvs vai kolekcija. Ir jāatzīmē, ka domēna-specifiskā valoda neatbalsta atribūtu definēšanu heštabulu veidā.

Elements `cardinality` ļauj definēt atribūta kardinalitāti un tam ir trīs atļautās vērtības, kas nosaka, vai attiecīgais atribūts ir vienkārša vērtība, masīvs vai kolekcija.

Elements `attribute_def` tiek izmantots atribūta definēšanai. Tas sastāv no atribūta vārda, kas ir simbolu virkne, kam seko attiecīgais datu tips un kardinalitāte. Tips tiek definēts kā identifikators un teorētiski tas var būt gan primitīvais datu tips no jebkuras programmēšanas valodas, gan arī identifikators, kas definē citu modeļa elementu. Šī valodas konstrukcija neierobežo to, kas ir datu tips. Noteikt, vai attiecīga datu tipa identifikators ir pareizs, ir DSL parsētāja uzdevums. Tas ļauj dinamiski pievienot dažādus likumus datu tipu definēšanai, piemēram:

1. `"x" (int) HAVING SINGLE CARDINALITY` definē atribūtu ar vārdu `x`, kas ir `int` datu tipa atribūts.
2. `"z" (double) HAVING ARRAY CARDINALITY` definē atribūtu ar vārdu `y`, kas ir `double` datu tipa elementu masīvs.
3. `"documents" (Document) HAVING COLLECTION CARDINALITY` definē atribūtu ar vārdu `documents`, kas ir `Document` datu tipa elementu kolekcija. `Document` var būt gan koncepts, kas ir definēts konceptu modelī, gan arī programmēšanas valodā iebūvēta klase ar šādu vārdu.

3.2.3. Koncepta definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

Koncepta definīcijas paplašinātās Bekusa-Naura formas ir parādītas 3.3. att. Koncepta definīcija sākas ar `DEFINE CONCEPT` atslēgvārdiem un tā identifikatoru un ietver sevī vienu vai vairākus koncepta parametrus. Tiek definēti šādi koncepta parametri:

1. Koncepta vārds – simbolu virkne, kas atbilst dotā koncepta nosaukumam, piemēram `"Rēķins"`.
2. Koncepta apraksts – simbolu virkne, kas satur informāciju par konceptu. Koncepta apraksts ir brīvas formas teksts, kas ir domāts modeļa uztveramības uzlabošanai. Apraksta piemērs – `"Pārdevēja rēķins par pircējam nosūtītām precēm"`.
3. Komentārs – simbolu virkne, kas satur papildus komentārus par doto konceptu. Piemērs – `"!!! Rēķinām nav jāsatur PVN"`.
4. Atribūts – `attribute_def` tipa konstrukcija ar papildus parametriem, kas nosaka attiecīgā atribūta redzamību Java valodai līdzīgā veidā – tas var būt

`public`, `private` vai `protected`. Redzamības definīcija nav obligāts koncepta konstrukcijas elements – ja tā nav noradīta, tiek uzskatīts, ka atribūts ir privāts.

Koncepta definīcija tiek aizvērta ar atslēgvārdiem `END CONCEPT`. Līdzīgi kā ar atribūta definīciju, valodas konstrukcijas nenosaka nekādus validācijas noteikumus. Līdz ar to koncepta definīcijas validācija ir parsētāja uzdevums. Pēc noklusējuma tā definīcijas validācijai ir definēti vairāki noteikumi:

1. Ja vairākiem konceptiem ir piešķirti vienādi identifikatori, tiek izraisīta parsēšanas kļūda.
2. Ja koncepta vārds vai apraksts ir definēts vairāk nekā vienu reizi, tiek izraisīta parsēšanas kļūda.
3. Ja koncepta nosaukums nav definēts, tiek izraisīts parsēšanas brīdinājums un konceptam tiek piešķirts automātiski uzģenerēts nosaukums formā `"Concept X"`, kur `X` ir automātiski inkrementējams skaitlis, sākot ar 1.

Ja ir nepieciešams, tad ir iespējams pievienot arī citus validācijas noteikumus, kas, līdzīgi kā atribūtu datu tipu gadījumā, var notikt dinamiski. Tas ļauj paplašināt DSL iespējas – piemēram, aizliegt dot konceptiem nosaukumus, kas sakrīt ar programmēšanas valodā iebūvēto klašu nosaukumiem vai atslēgvārdiem.

Konceptam un citiem divpusložu modeļa elementiem tiek definēts nosaukums un identifikators, kam ir dažādas nozīmes – nosaukums tiek izmantots transformāciju laikā un atbilst attiecīgā elementa nosaukumam grafiskajā notācijā, kamēr identifikators tiek izmantots tikai un vienīgi modeļa apraksta nolūkiem un ir DSL sastāvdaļa.

```
visibility_def    = 'PRIVATE' | 'PROTECTED' | 'PUBLIC';

name_def          = glue 'NAME' string;
description_def   = glue 'DESCRIPTION' string;
comment_def       = glue 'COMMENT' string;
concept_attr_def  = glue visibility_def? attribute_def;

concept_param_def = name_def | description_def | comment_def
                  | concept_attr_def;
concept           = 'DEFINE CONCEPT' identifier
                    concept_param_def*
                    'END CONCEPT';
```

3.3. att. Koncepta definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

3.2.4. Procesa definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

```
process_type      = 'INTERNAL' | 'EXTERNAL';

process_type_def  = glue 'TYPE' process_type;
performer_def     = glue 'PERFORMER' string;
process_param     = name_def | description_def | comment_def
                  | process_type_def | performer_def;

guard_def         = 'WITH GUARD' string;

process           = 'DEFINE PROCESS' identifier guard_def?
                    process_param*
                    'END PROCESS';
```

3.4. att. Procesa definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

Procesa definīcijas paplašinātās Bekusa-Naura formas ir parādītas 3.4. att. Definīcija sākas ar `DEFINE PROCESS` atslēgvārdiem un procesa identifikatoru. Šai konstrukcijai var sekot nosacījums procesa izpildei, kas tiek definēts kā simbolu virkne. Tas ļauj procesa izpildes nosacījumu aprakstīt brīvā formā, kas varētu būt noderīgs, ģenerējot kodu dažādās programmēšanās valodās – jo attiecīgo valodu konstrukcijas var atšķirties. Procesa izpildes nosacījums nav obligāts parametrs un, ja tas nav noradīts, tiek uzskatīts, ka process izpildās vienmēr. Šīm definīcijām seko procesa parametri. Procesa parametri ietver procesa nosaukumu, aprakstu un komentārus, kas ir līdzīgi attiecīgajiem koncepta parametriem. Papildus procesam var tikt definēts tā tips. Procesa tipa definīcija sākas ar atslēgvārdu `TYPE`, aiz kura seko viens no definētajiem procesa tipiem: `INTERNAL` (iekšējais) vai `EXTERNAL` (ārējais). Vēl viens papildus procesa parametrs ir tā izpildītājs, kas ir definēts ar atslēgvārdu `PERFORMER`, kuram seko izpildītāja nosaukumu nosakoša simbolu virkne. Procesa definīcija tiek aizvērta ar `END PROCESS` atslēgvārdiem. Procesa definīcijai ir definēti šādi validācijas noteikumi:

1. Ja vairākiem procesiem ir piešķirti vienādi identifikatori, tiek izraisīta parsēšanas kļūda.
2. Ja procesa nosaukums, tips vai apraksts ir definēts vairāk nekā vienu reizi, tiek izraisīta parsēšanas kļūda.

3. Ja procesa nosaukums nav definēts, tiek izraisīts parsēšanas brīdinājums, un procesam tiek piešķirts automātiski uzģenerēts nosaukums formā `"Process X"`, kur `X` ir automātiski inkrementējams skaitlis, sākot ar 1.
4. Ja procesa tips nav definēts, tas tiek uzskatīts par iekšējo procesu.

Līdzīgi kā koncepta gadījumā, ir iespējams dinamiski pievienot arī citus validācijas noteikumus.

3.2.5. Datu plūsmas definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

Datu plūsmas definīcijas paplašinātās Bekusa-Naura formas ir parādītas 3.5. att. Tās definīcija sākas ar `DEFINE DATA FLOW` atslēgvārdiem un datu plūsmas identifikatoru, aiz kuriem seko avota un mērķa procesu identifikatori. Datu plūsmas parametri ietver nosaukumu, aprakstu un komentārus, kas ir līdzīgi attiecīgajiem koncepta un procesa parametriem.

```
data_def          = glue 'PARAMETER' attribute_def;
dataflow_param    = name_def | description_def | comment_def | data_def;

data_flow         = 'DEFINE DATA FLOW' identifier 'FROM' identifier
                    'TO' identifier
                    dataflow_param*
                    'END DATA FLOW';
```

3.5. att. Datu plūsmas definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

Papildus datu plūsmas parametrs ir pārnesamie dati, kas ir `attribute_def` tipa konstrukcija. Ir iespējams definēt nevienu vai vairākus datu plūsmas atribūtus, kas raksturo pārnesamos datus. Datu plūsmas definīcija tiek aizvērta ar atslēgvārdiem `END DATA FLOW`. Datu plūsmas definīcijai ir definēti šādi validācijas noteikumi:

1. Ja vairākām datu plūsmām ir piešķirti vienādi identifikatori, tiek izraisīta parsēšanas kļūda.
2. Ja datu plūsmas nosaukums vai apraksts ir definēts vairāk nekā vienu reizi, tiek izraisīta parsēšanas kļūda.

3. Jā vairākiem datu plūsmas atribūtiem (pārnesamiem datiem) ir definēti vienādi nosaukumi, tiek izraisīta parsēšanas kļūda.
4. Ja datu plūsmas nosaukums nav definēts, tiek izraisīts parsēšanas brīdinājums, un datu plūsmai tiek piešķirts automātiski uzģenerēts nosaukums formā `"Data Flow X"`, kur `X` ir automātiski inkrementējams skaitlis, sākot ar 1.
5. Ja viens no procesu identifikatoriem vai piesaistīta koncepta identifikators norāda uz elementu, kas nav definēts divpusložu modeļa aprakstā, tiek izraisīta parsēšanas kļūda.

Ir svarīgi norādīt, ka pārnesamais datu tips var būt jebkāds – noteikt, vai tas ir pareizs ir parsētāja uzdevums, kas var tikt realizēts vairākos veidos. Piemēram, līdzīgi kā atribūtu definīcijas gadījumā var izveidot atļauto datu tipu sarakstu, vai arī atļaut pārnest tikai konceptus, kas ir definēti konceptu modelī. DSL konstrukcijas šeit nenosaka nekādus ierobežojumus.

3.2.6. Procesu diagrammas definīcija divpusložu modeli aprakstošā domēna-specifiskā valodā

Pēdējais DSL elements ir procesu diagrammas definīcija, kuras paplašinātās Bekusa-Naura formas ir parādītas 3.6. att. Tā tiek izveidota pēc līdzīgiem pamatprincipiem, kā iepriekš aprakstītie elementi. Procesu diagrammas parametri ietver nosaukumu, aprakstu un komentārus, kā arī atsauces uz procesiem, kuri ir iekļauti šajā procesu diagrammā. Norādes uz datu plūsmām netiek iekļautas, jo tās ir iespējams identificēt ar procesu palīdzību.

Procesu diagrammas definīcijai ir definēti šādi validācijas noteikumi:

1. Ja vairākām procesu diagrammām ir piešķirti vienādi identifikatori, tiek izraisīta parsēšanas kļūda.
2. Ja procesu diagrammas nosaukums vai apraksts ir definēts vairāk nekā vienu reizi, tiek izraisīta parsēšanas kļūda.
3. Ja procesu diagrammas nosaukums nav definēts, tiek izraisīts parsēšanas brīdinājums, un procesu diagrammai tiek piešķirts automātiski uzģenerēts nosaukums `"Process Diagram X"`, kur `X` ir automātiski inkrementējams skaitlis, sākot ar 1.
4. Ja viens no procesu identifikatoriem norāda uz elementu, kas nav definēts divpusložu modeļa aprakstā, tiek izraisīta parsēšanas kļūda.

5. Ja viena no datu plūsmām, kas tiek identificētas procesu diagrammas apstrādes laikā, satur norādi uz procesu, kas netiek iekļauts dotajā procesu diagrammā, tiek izraisīta parsēšanas kļūda.
6. Ja eksistē vairākas norādes uz vienu un to pašu procesu, tiek izraisīta parsēšanas kļūda.

```
process_ref_def   = glue 'PROCESS' identifier;
process_m_param   = name_def | description_def | comment_def |
                    process_ref_def;

process_model     = 'DEFINE PROCESS MODEL'
                    process_model_id
                    process_m_param*
                    'END PROCESS MODEL';
```

3.6. att. Procesu diagrammas definīcija divpusložu modeli aprakstošā DSL

3.2.7. DSL parsētāja realizācijas detaļas

Tā kā valodas sintakse ir aprakstīta ar paplašināto Bekusa-Naura formu palīdzību, ir iespējams izveidot rekursīvu lejupejošu parsētāju [34], kas realizēs attiecīgās gramatikas likumu pārbaudi un konstruēs parsējamā koda abstrakto sintaktisko koku (angl. *Abstract Syntax Tree* – AST). Šādu parsētāju ir iespējams realizēt manuāli, tomēr mūsdienās eksistē vairāki rīki, kas ļauj definēt gramatikas likumus ar speciālas sintakses (kas ir līdzīga EBNF notācijai) palīdzību un automātiski ģenerē attiecīgā parsētāja kodu – piemēram, [4],[108].

Promocijas darba autors valodas parsētāja realizācijai ir izvēlējies ANTLR [4] rīku, kas ģenerē parsētāja kodu Java valodā [50]. Šāda izvēle tika izdarīta pamatojoties uz to, ka pēdējā rīka, kas atbalsta uz divpusložu modeļa izmantošanu balstīto modeļvadāmo pieeju, versija tika realizēta arī programmēšanas valodā Java [82]. Tiek plānots veikt šī rīka uzlabošanu, un līdz ar to DSL parsētājam arī ir jābūt realizētam Java valodā.

ANTLR izmanto gramatikas likumu notāciju, kas ir līdzīga paplašinātajām Bekusa-Naura formām. 3.7. att. ir parādīti ANTLR likumi identifikatora parsēšanai, visi likumi aprakstītai domēna-specifiskajai valodai ir uzskaitīti.3. pielikumā. Viena no ANTLR priekšrocībām ir iespēja izmantot burtzīmju klases, kas tiek definētas ar regulāro izteiksmju burtzīmju klašu [94] notācijas palīdzību.

ANTLR uzģenerētais Java kods satur klasi, kas realizē novērotāja šablonu [28], un kura var tikt paplašināta gramatisko konstrukciju analīzes realizācijas nolūkam. Katram gramatikas likumam tiek uzģenerētas divas metodes ar vārdiem `enterGrammarConstruction` un `exitGrammarConstrution`, kas tiek izsauktas parsētājam uzsākot un beidzot attiecīgā likuma apstrādi. Attiecīgo metožu piemērs ir parādīts 3.8. att. Metodes šajā attēlā tika uzģenerētas gramatikas likuma `processType` (skat. 3. pielikumu) apstrādes rezultātā.

```
fragment
IdentifierLetter
   :  [a-zA-Z]
   ;

fragment
IdentifierLetterOrDigit
   :  [a-zA-Z0-9]
   ;

Identifier
   :  IdentifierLetter IdentifierLetterOrDigit*
   ;
```

3.7. att. ANTLR likumi identifikatora parsēšanai

```
/**
 * Enter a parse tree produced by {@link ThmlParser#processType}.
 * @param ctx the parse tree
 */
void enterProcessType(ThmlParser.ProcessTypeContext ctx);
/**
 * Exit a parse tree produced by {@link ThmlParser#processType}.
 * @param ctx the parse tree
 */
void exitProcessType(ThmlParser.ProcessTypeContext ctx);
```

3.8. att. ANTLR uzģenerētas novērotāja metodes

Izmantojot ANTLR, izmaiņu ieviešana modeļa metadatos prasa divus soļus:

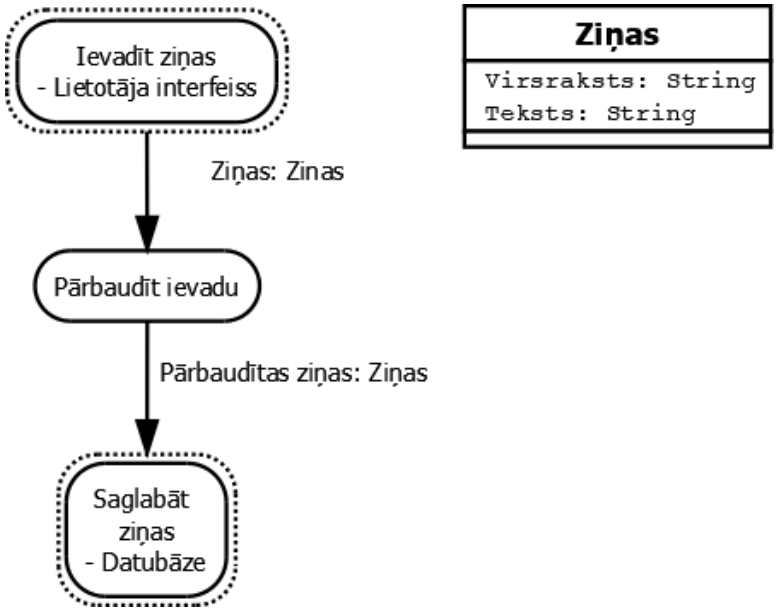
1. Jauna ANTLR likuma definēšana vai esoša izmaiņa – ir nepieciešams aprakstīt jaunas sintaktiskās konstrukcijas, vai izmainīt esošās, izmantojot ANTLR likumu notāciju.
2. Loģikas pievienošana attiecīgajai novērotāja klasei – ir nepieciešams vai nu realizēt jauno sintaktisko konstrukciju apstrādes metodes, vai modificēt esošās.

Rezultātā bibliotēka ANTLR ļauj ātri ieviest izmaiņas divpusložu modeļa notācijā, kas ļauj veikt eksperimentus modeļa transformāciju uzlabošanai. Šāda pieeja pēc promocijas darba autora uzskatiem piedāvā vairāk iespēju nekā grafiskā notācija, pie tam saglabājot pietiekami augstu modeļa uztveramību. Par DSL izmantošanas ierobežojumiem var atzīmēt to, ka

divpusložu modeļa attēlošana ar tās palīdzību (kā parādīts nākamajā sadaļā) vizuāli aizņem vairāk vietas nekā grafiskā notācija. Kā arī to, ka atšķirībā no divpusložu modeļa grafiskās notācijas, domēna-specifiskajai valodai neeksistē redaktors, kas palīdzētu veidot modeļus ar tās palīdzību. Tomēr šāda redaktora izveidošanas uzdevums var būt atsevišķa pētījuma tēma un šajā darbā tas netiek apskatīts.

DSL definēšanai izmantotie ANTLR likumi ir sniegti 3. pielikumā.

3.2.8. DSL izmantošanas piemērs



3.9. att. Vienkārša divpusložu modeļa grafiskā notācija

Izstrādātās domēna-specifiskās valodas iespēju demonstrēšanai tiek apskatīts vienkāršs divpusložu modelis, kas apraksta lietošanas gadījumu datu pievienošanai datubāzei. Tas sastāv no viena koncepta, kas apraksta pievienojamo datu struktūru, un procesa modeļa ar trim procesiem, divi no kuriem ir ārējie – informācijas saņemšana no lietotāja un tās saglabāšana datubāzē, bet viens – iekšējais – informācijas validācija. 3.9. att. ir parādīts attiecīgais divpusložu modelis.

Var redzēt, ka abiem ārējiem procesiem ir pievienota informācija par to izpildītajiem – datu ievades gadījumā par to atbild programmas grafiskais interfeiss, kamēr par datu saglabāšanu atbild datubāze.

```
DEFINE CONCEPT News
    WITH NAME "Ziņas"
    AND ATTRIBUTE "Virsraksts"(String)
    AND ATTRIBUTE "Teksts"(String)
END CONCEPT

DEFINE PROCESS EnterNews
    WITH NAME "Ievadīt ziņas"
    AND PERFORMER "Lietotāja interfeiss"
    AND TYPE EXTERNAL
END PROCESS

DEFINE PROCESS ValidateNews
    WITH NAME "Pārbaudīt ievadu"
END PROCESS

DEFINE PROCESS SaveNews
    WITH GUARD "Pareizs ievads"
    AND NAME "Saglabāt ziņas"
    AND PERFORMER "Datubāze"
    AND TYPE EXTERNAL
END PROCESS

DEFINE DATA FLOW EnteredNews FROM EnterNews TO ValidateNews
    WITH NAME "Ziņas"
    AND PARAMETER "n"(News)
END DATA FLOW

DEFINE DATA FLOW ValidatedNews FROM ValidateNews TO SaveNews
    WITH NAME "Pārbaudītas ziņas"
    AND PARAMETER "n"(News)
    AND PARAMETER "Pārbaudes rezultāts"(Boolean)
END DATA FLOW

DEFINE PROCESS MODEL ValidateAndSaveNews
    WITH NAME "Pārbaudīt un saglabāt ziņas"
    AND PROCESS EnterNews
    AND PROCESS ValidateNews
    AND PROCESS SaveNews
END PROCESS MODEL
```

3.10. att. Vienkārša divpusložu modeļa DSL notācija

3.10. att. ir parādīts tas pats divpusložu modelis, kas tiek attēlots ar DSL palīdzību. Atšķirībā no rīkā izveidotā modeļa, tas satur papildus informāciju par to, kad ir jāizpildās informācijas saglabāšanas procesam. Bez tā ar DSL definētajā modelī viena no datu plūsmām pārnes ne tikai vienu konceptu, bet arī informāciju par iepriekšējā procesa izpildes rezultātu.

Var redzēt, ka divpusložu modeļa definēšana ar DSL palīdzību aizņem vairāk vietas salīdzinājumā ar grafisko notāciju, kas var ietekmēt tā uztveramību. Tomēr, DSL izmantošana ļauj salīdzinoši vienkāršā veidā pievienot jaunus notācijas elementus, kā arī mainīt esošos. Pat dotajā vienkāršajā piemēra ir redzami jauni notācijas elementi, kas nav definējami modeļu grafiskajā redaktorā BrainTool [82] rīkā.

3.3. Secinājumi par divpusložu modeli aprakstošu domēna-specifisko valodu

Šajā sadaļā tika apskatīta promocijas darba autora izstrādāta domēna-specifiskā valoda, kas ir paredzēta divpusložu modeļa definēšanai. Var redzēt, ka šī valoda ļauj definēt modeli tapāt kā grafiskās notācijas gadījumā. Modeļa tekstuāla reprezentācija vizuāli aizņem vairāk vietas salīdzinājumā ar grafisko notāciju. Tomēr DSL izmantošana sniedz papildu iespējas – vienkāršotais notācijas modificēšanas process, kas sastāv no diviem soļiem (gramatiskas konstrukcijas definēšana izmantojot paplašinātas Bekusa-Naura formas un loģikas implementācijas attiecīgās konstrukcijas apstrādei).

Iepriekšējā sadaļā tika definēti trīs divpusložu modeļa notācijas ierobežojumi, divus no tiem izveidotā domēna-specifiskā valoda risina (skat. tabulu 3.1).

Var redzēt, ka bez citām priekšrocībām, ko sniedz DSL izmantošana, tās izstrādes laikā tika ņemti vērā arī esošie divpusložu modeļa notācijas ierobežojumi, kas tika veiksmīgi atrisināti. Līdz ar to šī darba autors uzskata, ka šādas aprakstošas valodas izstrāde ir viens no svarīgākajiem soļiem koda ģenerēšanas virzienā – tagad divpusložu modeļa notācija sniedz vairāk informācijas un ir piemērotāka divpusložu modeļa pārveidošanai programmatūras kodā.

Bez tā divpusložu modeli aprakstošai domēna-specifiskajai valodai var būt arī citi pielietojumi. Piemēram, to var izmantot modeļu glabāšanai. Rīki BrainTool [77] un BrainTool 2.0 [82] šim nolūkam izmanto XML valodu, kas izstrādes laikā izraisīja vairākas problēmas – piemērām, mainoties attiecīgai XML dialekta shēmai, vecie faili ne vienmēr tika pareizi apstrādāti to ielādes laikā. Tas prasīja manuālu konkrēto failu labošanu, kas ne vienmēr bija vienkāršs uzdevums. Lai gan XML valoda ir izveidota tā, lai to varētu lasīt un saprast gan cilvēks, gan arī datorprogrammas, attiecīgo failu struktūra var būt sarežģīta. Savukārt, DSL gadījumā šādu operāciju veikšana prasītu mazākas pūles, jo kā var redzēt piemērā iepriekšējā sadaļā, valodas konstrukcijas ir salīdzinoši vienkāršas un tās var labot neko nezinot par rīka iekšējām datu struktūrām, kas ir domātas modeļa reprezentācijai. XML gadījumā, savukārt, faila formāts atbilst šādām struktūrām un labošana prasa, lai cilvēks, kas to veic, zinātu par tām.

Papildus ir iespējams atbalstīt vairākas DSL versijas, kas salīdzinājumā ar XML izmantošanu atkal prasa mazākas pūles. XML izmantošanas gadījumā veco versiju atbalsts parasti prasa vairāku datu struktūru vai arī to apstrādes mehānismu uzturēšanu. Savukārt DSL gadījumā, pateicoties ANTLR [4] bibliotēkai, šādu izmaiņu ieviešana ir vairāk automatizēta, kas arī samazina cilvēciskā faktora ietekmi.

3.1. tabula

Divpusložu modeļa notācijas ierobežojumu risināšana ar DSL palīdzību

Notācijas ierobežojums	Risinājums ar DSL palīdzību
Modelī nav iespējams definēt procesu izpildes nosacījumus	Divpusložu modeļa notācijai tiek pievienotas konstrukcijas, kas ļauj definēt procesa izpildes nosacījumus
Vairāku izejas datu plūsmu gadījumā nav iespējams definēt, vai process var producēt tikai vienu, vai arī vairākas datu plūsmas	DSL nerisina šo ierobežojumu – promocijas darba autors uzskata, ka šo uzdevumu var atrisināt vēlāk, tas ir, transformāciju laikā. Šādu lēmumu arī ietekmē fakts, ka transformāciju rezultātā var eksistēt iespēja iegūt programmatūras kodu dažādās programmēšanas valodās. Savukārt vairākas programmēšanas valodas, piemēram Python [119], atbalsta vairāku objektu atgriešanu no funkcijām – līdz ar to vairāku datu plūsmu producēšana ir absolūti korekts gadījums.
Dati, ko spēj pārnest datu plūsmas ir ierobežoti ar vienu konceptu, kas ir piesaistīts datu plūsmai. Primitīvu datu tipu un objektu masīvu/kolekciju definīcija datu plūsmas ietvaros nav iespējama. Tapāt nav iespējams definēt datu plūsmu, kas datus nepārnes.	DSL maina divpusložu modeļa notāciju, ļaujot datu plūsmai pārnest primitīvus, objektus, to masīvus vai kolekcijas, kā arī vispār nepārnest datus.

4. UZLABOTAIS KLAŠU ATTIECĪBU NOTEIKŠANAS ALGORITMS

Kā ir parādīts [69],[72],[74],[83],[85],[86] un citos pētījumos, esošie divpusložu modeļa transformācijas likumi ļauj definēt ne tikai klases, bet arī attiecības starp tām. Tomēr, kā viens no esošo transformācijas likumu ierobežojumiem tika atzīmēta klašu attiecību noteikšana, kas ir balstīta vienīgi uz informāciju no procesa modeļa. Lai gan šī pieeja ir pareiza, jo analizējot, kādas klases komunicē savā starpā, ir iespējams noteikt tādas klašu attiecības kā atkarība vai asociācija [113]. Citu attiecību – agregāciju, mantošanu un realizāciju – noteikšana var prasīt arī ģenerētās klašu kopas analīzi. Piemēram, transformāciju rezultātā var tikt uzģenerētas vairākas klases ar līdzīgiem atribūtiem, kas būtu jāapvieno vienā klašu hierarhijā [62]. Lai šādas klašu kopas uzturēšana būtu vienkāršāka, izmaiņas būtu nepieciešams veikt pēc iespējas mazāk. Līdzīgi, agregāciju noteikšana neizmanto informāciju, ko sniedz konceptuālais modelis – piemēram, ja viens no konceptiem tiek ietverts citā kā atribūts, starp attiecīgajām uzģenerētajām klasēm agregācijas attiecība netiek definēta. Tas nozīme to, ka esošās transformācijas neizmanto visu pieejamo informāciju, ko sniedz divpusložu modelis, un līdz ar to ir nepieciešams novērst šo ierobežojumu, kas ļautu ne tikai ģenerēt programmatūras kodu, kas strādā, bet atbalstāmo kodu. Kā norāda [62] autors: "Pat slikts kods var funkcionēt, bet, ja kods nav tīrs, tas var nolikt programmatūras izstrādes organizāciju uz ceļiem. Katru gadu neskaitāmi laika un resursu apmēri tiek pazaudēti slikti uzrakstīta koda dēļ". Tātad, lai programmatūras kods, kas ir iegūts divpusložu modeļa transformāciju rezultātā, būtu labāk uzturams, ir nepieciešams uzlabot arī klašu attiecību noteikšanas algoritmu.

Analizējot šāda algoritma ieejas un izejas datus – sākotnējo klašu kopu un uzlaboto klašu kopu, kas satur informāciju par klašu attiecībām, var redzēt, ka šāda algoritma izstrāde nav atkarīga no modeļa transformācijas algoritma. Klašu attiecību noteikšana tiek veikta pēc transformācijas algoritma izpildes. Tā kā šādam algoritma ir jāstrādā ar klašu kopu, šī klašu kopa var tikt reprezentēta vairākos veidos – tā var būt gan UML klašu kopa, gan arī klašu kopa no jau uzģenerēta programmatūras koda. Līdz ar to šāds algoritms var strādāt ar dažādu transformācijas algoritmu izejas modeļiem un var tikt izpildīts, kā atsevišķs transformāciju solis.

Izstrādājot šādu algoritmu, būtu nepieciešams noteikt, kādā veidā tiks izveidoti jauno definēto klašu nosaukumi. Piemērām, veicot klašu kopas analīzi, ir iespējams noteikt, ka klases `A` un `B` satur vienādus atribūtus, līdz ar to ir nepieciešamas izveidot bāzes klasi un definēt

mantošanas attiecības starp bāzes klasi, `A` un `B`. Šāds uzdevums prasa ne tikai jaunās bāzes klases struktūras (tas ir atribūtu un metožu kopas) noteikšanu, bet arī jaunās klases nosaukuma izveidošanu. To var darīt, piemēram, automātiski, apvienojot klašu A un B nosaukumus un pievienojot rezultātam prefiksu vai postfiksu – `ABParent`, `BaseAB`. Taču automātiskai klašu nosaukumu ģenerēšanai piemīt arī vairāki ierobežojumi – sarežģītas klašu hierarhijas gadījumā var izveidoties situācija, kad arī bāzes klasēm tiks izveidotas bāzes klases, kas var novest, piemēram, pie tādiem nosaukumiem kā `ABParentCDParentParent`, kas ir gan pārāk gari, gan arī slikti uztverami. Cita problēma, kas ir saistīta ar automātisku bāzes klašu nosaukumu ģenerēšanu, ir tas, ka uzģenerētie nosaukumi var iet pretrunā ar "tīra koda" [62] principiem – bāzes klases nosaukums saturēs informāciju par visām klasēm, kas manto no tās. Līdz ar to promocijas darba autors piedāvā šādus klašu nosaukumus ģenerēt ar cilvēka, kas veic transformācijas palīdzību. Nosakot jauno bāzes klasi, algoritmam būtu jāpārtrauc sava darbība un jāuzdod jautājumu par to, kā šādu klasi ir jānosauc.

Ņemot vērā visu iepriekš rakstīto, ir iespējams definēt šādas prasības klašu attiecību noteikšanas algoritmam:

1. Algoritmam ir jāapstrādā klašu kopa, kas var būt prezentēta vairākos veidos – gan programmatūras koda klašu kopas veidā, gan arī UML klašu kopas veidā. Iespējams arī citos veidos. To var panākt, izmantojot *Template* projektēšanas šablonu [28], kas programmēšanas valodā Java [50] tiek realizēts ar vispārējo tipu un metožu palīdzību. Papildus Java valodas priekšrocība šajā gadījumā ir tāda, ka tajā ir iespējams definēt papildu prasības vispārējiem tipiem – piemēram, konstrukcija `<T extends ClassDesc>` norāda uz to, ka tipam `T` ir jāmanto no vai ir jārealizē tipu `ClassDesc`.
2. Algoritmam ir jāņem vēra ne tikai informāciju par to, kuras klases komunicē savā starpā, bet arī klašu struktūru – to atribūtus un metodes.
3. Lai būtu iespējams noteikt, kuras klases komunicē savā starpā, klašu kopai ir jāsatur arī papildu informācija. Viens no veidiem, kā var noteikt šādu komunikāciju, ir metožu parametru un to atgriežamo tipu analīze – piemēram, ja klases `A` metode `m` kā parametru saņem klasi `B` un klasi `C` atgriež kā rezultātu, tad ir iespējams definēt attiecīgās atkarības starp šīm klasēm. Tomēr, metode `m` var izmantot arī citas klases – piemēram, `D`, `E`, `F`, savas izpildes laikā. Tikai un vienīgi analizējot metožu definīcijas, šādus gadījumus nav iespējams noteikt. Līdz ar to klašu atkarību noteikšanas algoritma ieejas datiem ir jāsatur arī papildus informācija par to, kuras

klases komunicē savā starpā. Bez šādas informācijas attiecību noteikšana var būt nepilnīga.

4. Jauno uzģenerēto klašu nosaukumiem ir jābūt nevis automātiski uzģenerētiem, bet gan cilvēka noteiktiem.

Tālāk šajā sadaļā tiek apskatīts algoritms, kas ir izveidots pēc šīm prasībām. Piedāvātais algoritms sastāv no četriem galvenajiem soļiem, pirmajā no kuriem tiek identificētas realizācijas un mantošanas, otrajā – agregācijas, trešajā – atkarības, bet ceturtajā – asociācijas. Algoritma ieejas dati ir klašu kopa ar metodēm un atribūtiem, kā arī informācija par klašu savstarpējo komunikāciju, izejas dati – klašu kopa ar savstarpējām attiecībām.

4.1. Realizāciju un mantošanu noteikšana

Realizāciju un mantošanu noteikšanas gaitā ir nepieciešama cilvēka līdzdalība, kas nozīme to, ka šis algoritma solis tiek veikts pusautomātiskā veidā.

Sākumā piedāvātais algoritms veic ieejas klašu kopas analīzi mantošanas attiecību noteikšanai. Mantošanas attiecību noteikšanai tiek izmantota datu palīgstruktūra, kas tiek saukta par kopīgo elementu tabulu. Kopīgo elementu tabula ir heštabula, kur par atslēgu kalpo kortežs `<A, B>`, kur `A` un `B` ir klases ar kopīgiem elementiem. Šim kortežam ir noteikta prasība: `<A, B> == <B, A>`, kas nozīmē to, ka klašu secība šajā korteža nav svarīga. Par kopīgo elementu tabulas vērtību kalpo klašu `A` un `B` kopīgo elementu saraksts `L`. Rezultātā kopīgo elementu tabula var tikt definēta kā attiecību `<A, B> → L` kopa.

Kā viena no piedāvātā algoritma sastāvdaļām tiek definēta funkcija, kas definē mantošanas attiecību divām klasēm. Šī funkcija ir parādīta 4.1. att. Tā saņem četrus parametrus – bāzes klasi `parent`, atvasināto klasi `child`, klašu kopīgo elementu sarakstu `commonElements` un eksistējošo klašu attiecību kopu `relations`. Funkcijas izpildes laikā visi kopīgie elementi tiek pievienoti bāzes klasei un izņemti no atvasinātās klases. Kopīgo elementu izņemšana nodrošina to, ka algoritms nepiedāvās analītiķim definēt vienu un to pašu attiecību vairākas reizes.

```
function defineGeneralization(parent, child, commonElements, relations):
      parent += commonElements
      child -= commonElements
      relations += generalization(child, parent)
```

4.1. att. Mantošanas attiecības definēšanas funkcija

```
changesWereMade = true
relations = []

while changesWereMade:
      changesWereMade = false
      commonElementTable = tukšā kopīgo elementu tabula <A, B> → L

      for a in inputClasses:
            for b in inputClasses:
                  commonElementTable[a, b] = klašu a un b kopīgie
                                             elementi

      while not changesWereMade:
            generalizationCandidate = commonElementTable tabulas
                                      ieraksts ar vislielāko kopīgo
                                      klašu skaitu

            action = askForAction(generalizationCandidate)
            if action == MAKE_A_PARENT:
                  defineGeneralization(
                        generalizationCandidate.a,
                        generalizationCandidate.b,
                        generalizationCandidate.commonElements,
                        relations)
                  changesWereMade = true
            else if action == MAKE_B_PARENT:
                  defineGeneralization(
                        generalizationCandidate.b,
                        generalizationCandidate.a,
                        generalizationCandidate.commonElements,
                        relations)
                  changesWereMade = true
            else if action == INTRODUCE_SUPERCLASS:
                  s = createNewSuperClass()
                  inputClasses += s

                  defineGeneralization(s,
                        generalizationCandidate.a,
                        generalizationCandidate.commonElements,
                        relations)
                  defineGeneralization(s,
                        generalizationCandidate.b,
                        generalizationCandidate.commonElements,
                        relations)
                  changesWereMade = true
```

4.2. att. Mantošanas definēšanas algoritms

Mantošanas definēšanas algoritms ir parādīts 4.2. att. Sākumā tiek analizētas ieejas klases un tiek konstruēta kopīgo elementu tabula. Pēc kopīgās elementu tabulas izveidošanas tiek veikts mantošanas attiecību definēšanas cikls, kura izpildes laikā no kopīgo elementu tabulas tiek izvēlēts ieraksts `<A, B> → L` ar vislielāko kopīgo elementu skaitu. Tad analītiķim tiek piedāvāta izvēle no 4 darbībām:

1. Pārveidot klasi `A` par bāzes klasi, klasi `B` par atvasināto,

2. Pārveidot klasi `B` par bāzes klasi, klasi `A` par atvasināto,
3. Izveidot jaunu bāzes klasi `S`, klases `A` un `B` pārveidot par klases `S` atvasinātajām klasēm,
4. Neko nedarīt.

Atkarībā no darbības izvēles tiek modificēta definēto klašu attiecību kopa un ieejas klašu kopa. Gadījumā, ja tika izvēlēta viena no darbībām 1, 2 vai 3, kopīgo elementu tabula tiek izveidota no jauna. Ja analītiķis nav veicis nevienu izmaiņu, mantošanas definēšanas algoritms tiek pārtraukts un tiek uzskatīts, ka visas iespējamās mantošanas attiecības ir definētas. Šādā gadījumā tiek izpildīts nākamais algoritma apakšsolis – realizāciju attiecību noteikšana.

Realizāciju noteikšanas gaitā tiek analizētas klases, kuras ietver sevī vienādas metodes. Tā kā visas iespējamās mantošanas jau tika noteiktas iepriekšējā solī, tiek uzskatīts, ka klases ar vienādām metodēm var realizēt kopīgu interfeisu. Realizāciju noteikšanas algoritms kā ieejas datus izmanto visu klašu metožu kopu. Tas ir parādīts 4.3. att. Sākumā realizāciju definēšanas algoritms aizpilda potenciālo interfeisu implementējošo klašu heštabulu, kur par atslēgu kalpo attiecīgā metode, bet par vērtību – klašu, kas satur šo metodi, kopa. Pēc potenciālo interfeisu realizētāju tabulas aizpildes sākas tās analīze, kas ir līdzīga analīzei mantošanas noteikšanas gaitā.

Analizējot potenciālo interfeisu realizētāju tabulu, algoritms piedāvā analītiķim definēt interfeisu klašu kopai. Gadījumā, ja interfeiss tiek definēts, algoritms pārbauda, vai jau eksistē interfeiss, kas satur visas attiecīgas klases. Gadījumā, ja šāds interfeiss netiek atrasts, tiek izveidots jauns interfeiss, kas savukārt tiek pievienots ieejas klašu kopai. Pēc attiecīgā interfeisa noteikšanas tam tiek pievienota klašu kopīgā metode, kas arī tiek izņemta no klasēm.

Var redzēt, ka šī algoritma soļa izpildes laikā ir nepieciešama intensīva cilvēka piedalīšanās. Pie tam šim cilvēkam ir nepieciešams saprast objektorientētās pieejas un "tīra koda" pamatprincipus. Tas nozīme to, ka ideālā gadījumā šo soli būtu jāveic vai nu projekta arhitektam, vai programmatūras izstrādātājam. Protams, ir iespējams izņemt šo soli no klašu atkarību noteikšanas algoritma vai arī izmantot iespēju automātiski definēt jauno klašu nosaukumus, vienmēr definējot kopējo bāzes klasi (tā ir 3. alternatīva iepriekš definētajā sarakstā), bet tad pēc transformāciju beigām uzģenerētais programmatūras kods prasīs papildus pārveidošanu. Pēc promocijas darba autora domām, šādu pārveidošanu labāk ir veikt jau atkarību noteikšanas laikā, jo prasība pēc vairākām atbildēm ir ne tik strikta, salīdzinājumā ar nepieciešamību restrukturizēt uzģenerēto programmatūras kodu.

```
changesWereMade = true
relations = []

while changesWereMade:
      changesWereMade = false
      possibleImplementers =  heštabula <M, C>, kur M – metode, C –
                              klašu, kas satur šo metodi, saraksts

      for m in methods:
            if m ir definēta vairākas klasēs:
                  possibleImplementers[m] = klases, kas satur metodi m

      for m in possibleImplementers:
            action = askForAction(possibleImplementers)

            if action == DEFINE_INTERFACE:
                  i = findOrCreateInterface(possibleImplementers)

                  i += m
                  inputClasses += i

                  for class in possibleImplementers.classes:
                        relations += implementation(class, i)
                        class -= m

                  changesWereMade = true
                  break
```

4.3. att. Realizāciju definēšanas algoritms

4.2. Agregāciju noteikšana

Nākamais klašu attiecību veids, kas tiek noteikts ar promocijas darba autora piedāvātā algoritma palīdzību, ir agregācija – tās noteikšana prasa salīdzinoši vienkāršu klašu kopas analīzi. Šāda analīze ietver pārbaudi, vai noteiktā klase satur citas klases no ieejas kopas atribūtu veidā. Jā viena klase satur citu klasi, tad tiek definēta agregācijas attiecība. Pie tam, ir iespējama situācija, kad klase `A`, kas ir klases `B` bāzes klase, satur sevī `B` kā atribūtu. Lai gan tas nav pareizi no "tīra koda" principu viedokļa [62], jo tas nozīmē, ka bāzes klase "zina" par klasēm, kuras no tās manto, tāda situācija tomēr ir iespējama. Ja ir konstatēts tāds gadījums, tad agregācijas attiecība definēta netiek, jo jau eksistē mantošanas attiecība. Ir iespējams, ka klase `B`, kas manto no klases `A`, satur `A` kā atribūtu. Atkal, šajā gadījumā agregācija netiek definēta. Tas pats attiecas uz realizācijas attiecību. Algoritma pseidokods ir parādīts 4.4. att.

```
for a in inputClasses:
      for b in inputClasses:
            if a satur b:
                  if relations satur (a, b) or (b, a) pāri:
                        relation = relations[(a, b) or (b, a)]

                        if (relation is generalization or
                              relation is implementation):

                              // attiecība netiek noteikta
                  else:
                        relations += aggregation(a, b)
```

4.4. att. Agregāciju definēšanas algoritms

4.3. Atkarību noteikšana

Atkarību noteikšanas nolūkiem tiek izmantota gan informācija, ko satur klašu metožu definīcijas, gan arī papildu informācija par klašu savstarpējo komunikāciju. Tātad šis algoritma solis ir sadalīts divos apakšsoļos – sākumā tiek veikta klašu metožu parametru un atgriežamo vērtību analīze, pēc tam tiek analizēta papildu informācija par klašu savstarpējo komunikāciju. Pie tam tiek ņemts vērā komunikācijas virziens – ja klase `A` savā darbā izmanto klasi `B`, tad attiecīgās atkarības virziens būs no `A` uz `B` nevis otrādi.

Pie tam, ir jāņem vērā jau esošās attiecības starp klasēm. Gadījumā, kad starp klasēm `A` un `B` eksistē iepriekš definēta attiecība jebkurā virzienā, atkarība netiek definēta – algoritms uzskata, ka mantošanas, realizācijas un agregācijas attiecības ir "svarīgākas" par atkarību.

Tā kā atkarības var tikt definētas vairākos gadījumos, un visur ir nepieciešams pārbaudīt, vai attiecība starp divām klasēm jau ir definēta, šajā solī tiek izmantota palīgfunkcija `definePossibleDependency`, kas saņem trīs parametrus – klases `a` un `b`, starp kurām ir nepieciešams definēt iespējamo atkarības attiecību un klašu attiecību sarakstu `relations`. Šīs funkcijas pseidokods ir parādīts 4.5.att.

```
function definePossibleDependency(a, b, relations):
      if relations satur (a, b) or (b, a):
            return

      relations += dependency(a, b)
```

4.5. att. Iespējamās atkarības definēšanas funkcija

Atkarību noteikšanas algoritma pseidokods ir parādīts 4.6. att.

```
for c in inputClasses:
      for m in c.methods:
            if m.returnType in inputClasses:
                  definePossibleDependency(c, m.returnType)

            for p in m.parameters:
                  if p in inputClasses:
                        definePossibleDependency(c, p)

      usedClasses = communicationInformation[c]
      for c1 in usedClasses:
            definePossibleDependency(c, c1)
```

4.6. att. Atkarību noteikšanas algoritms

Var redzēt, ka atkarību noteikšana ir salīdzinoši vienkāršs process, kas tiek balstīts uz klašu metožu analīzi, tomēr tas pieprasa papildu informāciju no iepriekšējā transformācijas soļa –informāciju par to, kuras klases komunicē savā starpā, ņemot vērā komunikācijas virzienu. Tātad, realizējot citas transformācijas, šis fakts ir jāņem vērā.

4.4. Asociāciju noteikšana

Pēdējais promocijas darba autora piedāvātā algoritma solis ir asociāciju noteikšana. Asociāciju noteikšanas laikā tiek analizētas jau definētās atkarības starp klasēm. Gadījumā, ja starp divām klasēm eksistē vairākas atkarības, tās tiek aizvietotas ar asociāciju. Algoritma pseidokods ir parādīts 4.7. att.

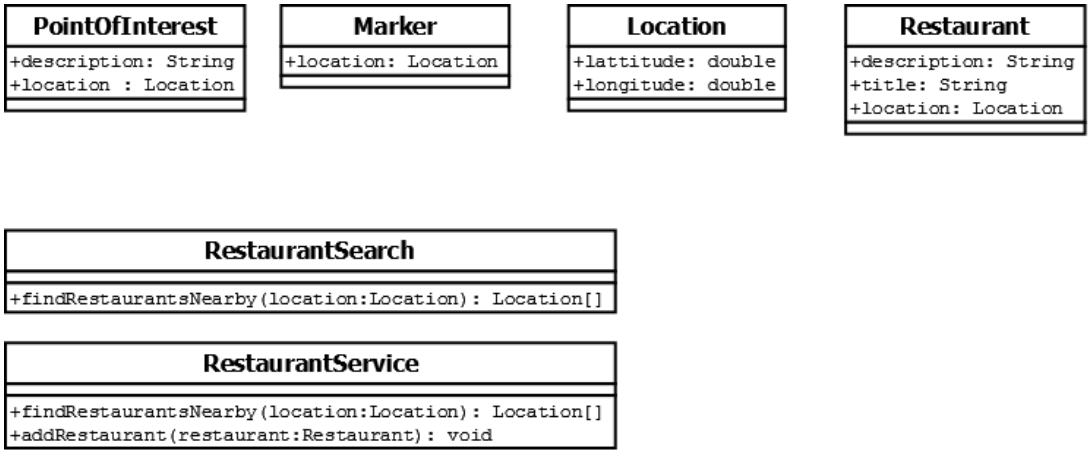
```
for a in inputClasses:
      for b in inputClasses:
            if eksistē vairākas atkarības starp a un b:
                  relations -= visas atkarības starp a un b
                  relations += association(a, b)
```

4.7. att. Asociāciju definēšanas algoritms

4.5. Algoritma darbības piemērs

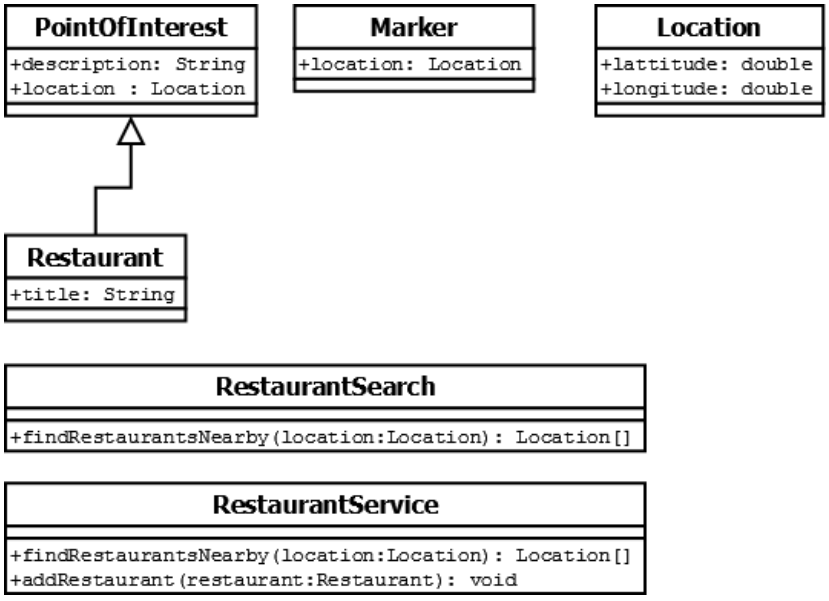
Iepriekšējās sadaļās tiek apskatīts šī darba autora piedāvātais algoritms, kas ir paredzēts klašu attiecību noteikšanai. Šis algoritms izmanto gan informāciju par analizējamo klašu

struktūru, gan arī papildu informāciju par klašu savstarpējo komunikāciju. Šajā sadaļā tiek apskatīts piedāvātā algoritma darbības piemērs.



4.8. att. Karšu sistēmas klases

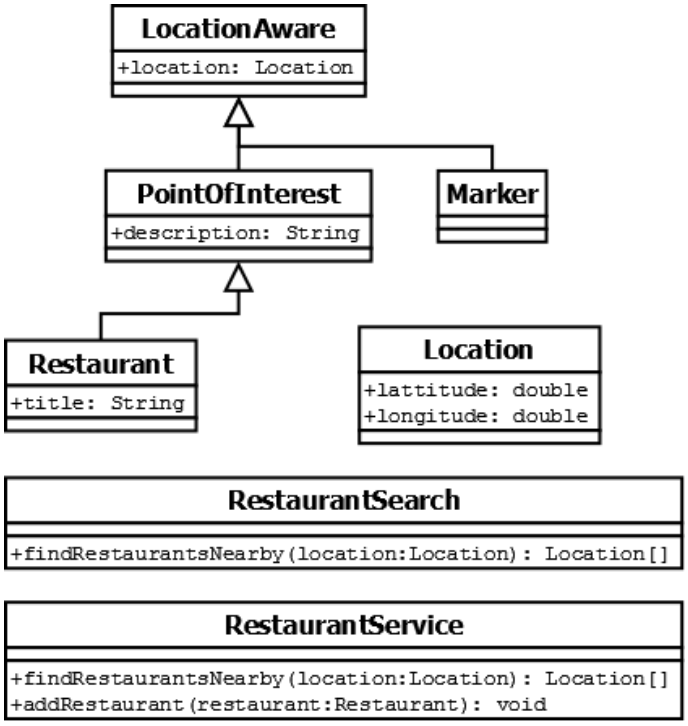
Par ieejas klašu kopu kalpo klases, kas apraksta abstraktas karšu sistēmas, kas ir līdzīga, piemēram, Google Maps [31]. Sākotnējā klašu kopa ir parādīta 4.8. att. Sākumā neviena attiecība starp klasēm nav definēta, un darbību uzsāk mantošanu definēšanas algoritms. Klašu analīzes rezultātā analītiķim tiek uzdots jautājums par superklases izveidošanu klasēm `Restaurant` un `PointOfInterest` ar kopīgiem laukiem `location` un `description`. Analītiķis izvēlas iespēju pārveidot klasi `PointOfInterest` par bāzes klasi, rezultātā tiek iegūta jauna klašu kopa, kas ir parādīta 4.9. att.



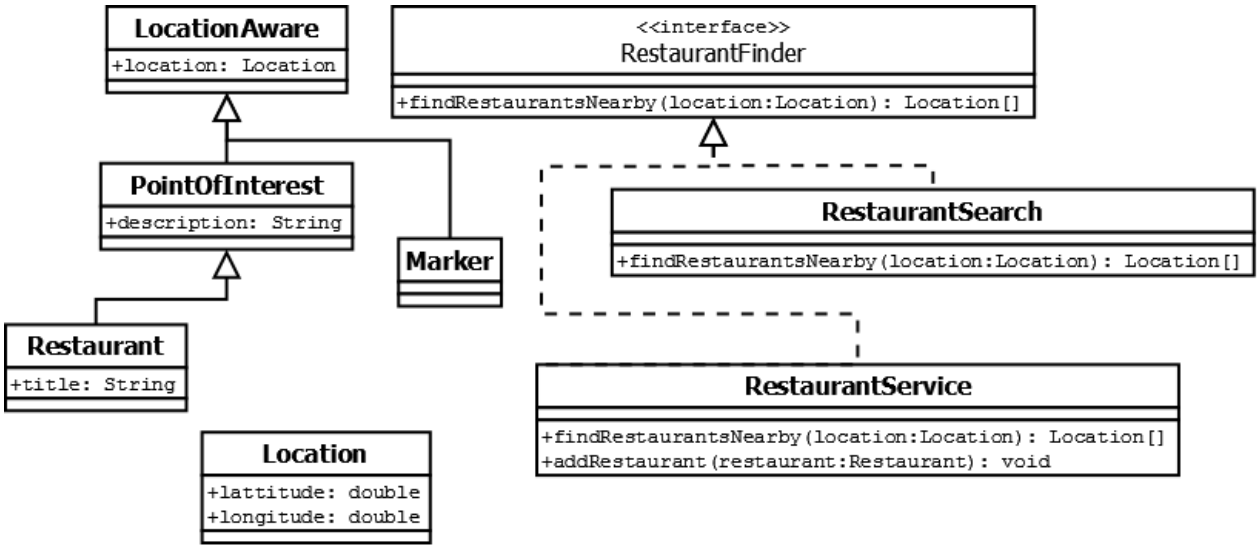
4.9. att. Karšu sistēmas klases pēc pirmās mantošanu noteikšanas algoritma iterācijas.

Nākamas mantošanu definēšana algoritmu iterācijas laikā tiek piedāvāts izveidot bāzes klasi klasēm `Location` un `PointOfInterest`, analītiķis piekrīt un izveido jaunu superklasi – `LocationAware`. Rezultējošā klašu kopa ir parādīta 4.10. att.

Trešās iterācijas laikā analītiķim tiek piedāvāts izveidot bāzes klasi klasēm `RestaurantSearch` un `RestaurantService`. Analītiķis izvēlas neveidot šādu attiecību, un tā kā nekādas klašu kopas izmaiņas netika veiktas, darbību uzsāk realizāciju definēšanas algoritms, kas piedāvā izveidot kopīgu interfeisu klasēm `RestaurantSearch` un `RestaurantService`, kam analītiķis piekrīt. Citas izmaiņas realizāciju definēšanas algoritms neveic, jo neeksistē realizāciju kandidāti. Klašu kopa pēc realizāciju definēšanas algoritma izpildes ir parādīta 4.11. att.

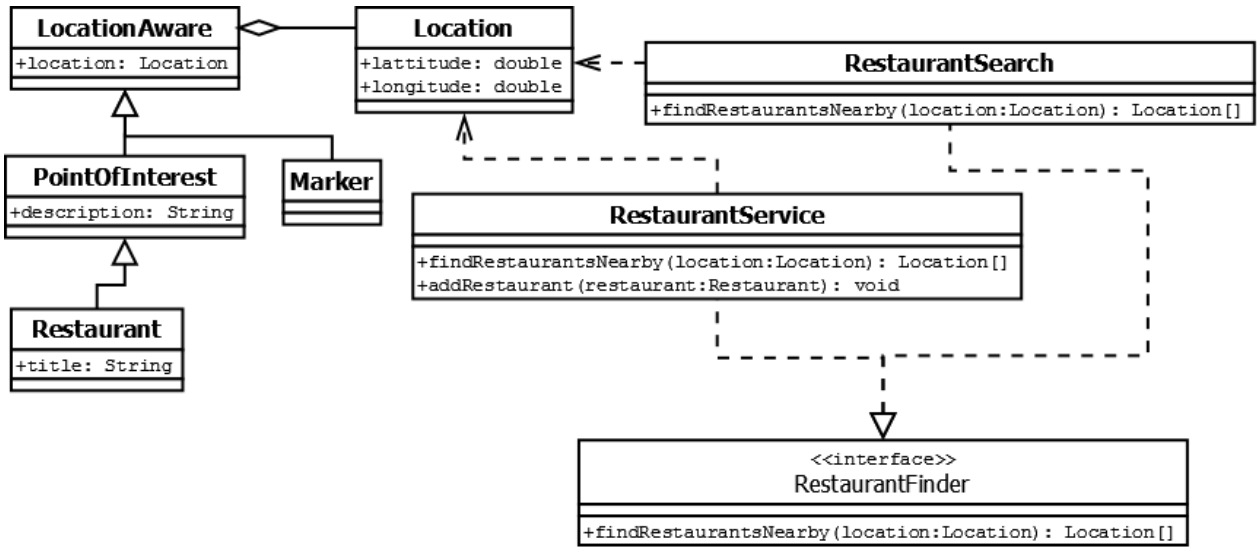


4.10. att. Karšu sistēmas klases pēc otrās mantošanu noteikšanas algoritma iterācijas.



4.11. att. Karšu sistēmas klases pēc realizāciju noteikšanas algoritma izpildes.

Agregāciju noteikšanas laikā tiek definēta agregācija `LocationAware` → `Location`. Atkarību noteikšanas laikā tiek definētas divas atkarības attiecības: `RestaurantSearch` → `Location`, `RestaurantService` → `Location`. Asociācijas netiek definētas, jo neizpildās to definēšanas likums. Rezultātā tiek iegūta klašu diagramma, kas ir parādīta 4.12. att.



4.12. att. Rezultējošās klases un to attiecības.

4.6. Klašu attiecību noteikšanas algoritma analīze un secinājumi par to

Šajā sadaļā tiek aprakstīts algoritms, kas ir paredzēts attiecību noteikšanai starp klasēm. Tiek aprakstīti visi piedāvātā algoritma soļi, kā arī tiek dots algoritma darbības piemērs. Kā ieejas datus algoritms saņem klašu kopu un informāciju par klašu savstarpējo komunikāciju. Izpildes rezultātā ieejas klašu kopa tiek papildināta ar informāciju par klašu attiecībām.

Algoritma izpildes laikā ir iespējama jaunu klašu vai interfeisu izveidošana, kā arī esošo klašu restrukturizēšana, tas ir atribūtu un metožu pārvietošana citās klasēs. Šādas koda transformācijas, kas tiek veiktas ar cilvēka piedalīšanos, var palīdzēt ģenerēt uzturamu kodu, kas atbilst "tīra koda" [62] principiem. Salīdzinājumā ar esošajām transformācijām, tas dod papildus priekšrocības, lai gan arī prasa cilvēka piedalīšanos. Protams, ir iespējams veikt šādas transformācijas automātiski, bet rezultātā var būt nepieciešams mainīt uzģenerēto kodu, kas tik un tā prasīs cilvēka piedalīšanos – tikai citā programmatūras izstrādes cikla solī.

Kā var redzēt iepriekš definētajā piemērā, algoritms ir spējīgs no klašu kopas, kas nesatur nevienu attiecību, izveidot pilnīgi saistītu klašu kopu. Protams, ne vienmēr tas ir iespējams, bet pēc promocijas darba autora pieredzes, tādu gadījumu skaits nebūs liels – parasti objektorientētas programmatūras kods nesatur pilnīgi izolētas klases, kas nekomunicē ar citām klasēm, vai arī nesatur citas klases kā atribūtus. Pat ja šādas klases ir atrodamas, visticamāk, ir iespējams tās pilnīgi izdzēst.

Kā citu piedāvātā algoritma priekšrocību promocijas darba autors vēlas atzīmēt tā iespēju strādāt ar klašu kopu, kas ir definēta vairākos veidos – gan ar UML [113] klašu diagrammām, gan arī ar programmatūras kodu – galvenais, lai pastāvētu iespēja iegūt informāciju par klašu signatūrām – to atribūtiem un metodēm. Līdz ar to piedāvāto klašu attiecību noteikšanas algoritmu var izmantot ne tikai divpusložu modeļa transformācijai, bet arī citiem nolūkiem, piemēram, koda pārveidošanas (angl. *refactoring*) [27] nolūkiem. Ir iespējams izmantot šo algoritmu arī ar citām modeļvadāmām metodēm.

Ņemot vērā iepriekš minētās algoritma priekšrocības, promocijas darba autors uzskata, ka izstrādātais klašu attiecību definēšanas algoritms var būt noderīgs un izmantojams ne tikai šī darba ietvaros, bet arī citās problēmsfērās. Papildus ir vērts atzīmēt, ka algoritma darbības analīzei tika izstrādāta arī UML klašu diagrammu salīdzināšanas metode [75], kas netiek detalizēti apskatīta promocijas darba ietvaros.

5. KODA ĢENERĒŠANA NO DIVPUSLOŽU MODEĻA

Šī darba galvenais mērķis ir algoritma, kas spēj pārveidot divpusložu modeli par programmatūras kodu, izstrāde. Lai būtu iespējams veiksmīgi sasniegt mērķi, ir nepieciešams izpildīt šādus uzdevumus:

1. Izvēlēties mērķa programmēšanas valodu, kurā tiek ģenerēts programmatūras kods. Šādai valodai ir jābūt objektorientētai.
2. Definēt transformāciju algoritmu, kas ļauj no divpusložu modeļa izveidot datu struktūru, kas būtu piemērota koda ģenerēšanai.
3. Apstrādāt iepriekš definēto datu struktūru programmatūras koda ģenerēšanai.
4. Aprobēt izstrādāto algoritmu, izmantojot reālas dzīves problēmsfēru un veikt iegūto rezultātu analīzi.

Šajā un nākamajās sadaļās tiek apskatīti visi šie uzdevumi un to risinājumi.

5.1. Mērķa programmēšanas valodas izvēle

Programmatūras koda ģenerēšanai sākumā ir nepieciešams izvēlēties programmēšanas valodu, kurā tiks ģenerēts programmatūras kods. Šādai valoda ir jābūt objektorientētai un pietiekami populārai programmatūras izstrādātāju vidē. Ja izvēlētā programmēšanas valoda nav populāra un netiek aktīvi izmantota, šī darba rezultāti nevarētu tikt izmantoti arī praksē. Pēc TIOBE programmēšanas valodu popularitātes indeksa [110] datiem 2018. gada maijā 10 populārākas programmēšanas valodas ir šādas – skat. 5.1. tabulu. Var redzēt, ka gandrīz visas populārākas programmēšanas valodas ir universālas un objektorientētas, kas nozīmē to, ka teorētiski var ģenerēt programmatūras kodu jebkurā no tām. Tātad, ir nepieciešami papildus kritēriji mērķa programmēšanas valodas izvēlei.

Par pirmo tādu kritēriju var kalpot valodas tipizācijas veids. Stingri tipizētas programmēšanas valodas izmantošana ļauj vienkāršot programmatūras koda ģenerēšanu – ir iespējams sekot līdzi mainīgo tipiem un to struktūrai, kas paliek nemainīga. Jau izveidotai klasei ir salīdzinoši grūti dinamiski izmainīt iekšējo struktūru (protams, tādas iespējas pastāv un gandrīz visas uzskaitītās valodas to atļauj darīt, bet tas prasa zema līmeņa valodas konstrukciju izmantošanu un ne vienmēr ir iespējams – piemēram, programmēšanas valodas Java [50] gadījumā pastāv vairāki ierobežojumi, kas attiecas uz jau ielādētām klasēm).

5.1. tabula

10 populārākās programmēšanas valodas pēc TIOBE datiem

Vieta	Valoda	Reitings	Valodas raksturojums
1	Java [50]	16.38%	Stingri tipizēta objektorientēta universālā programmēšanas valoda ar drazu savācēju
2	C [49]	14%	Stingri tipizēta universālā programmēšanas valoda
3	C++ [101]	7.688%	Stingri tipizēta objektorientēta universālā programmēšanas valoda
4	Python [119]	5.192%	Daudzu paradigmu (tai skaitā objektorientēta) universālā dinamiskā programmēšanas valoda ar drazu savācēju
5	C# [12]	4.402%	Stingri tipizēta objektorientēta universālā programmēšanas valoda ar drazu savācēju
6	The Visual Basic .NET [116]	4.124%	Objektorientēta programmēšanas valoda
7	PHP [93]	3.321%	Skriptu universālā dinamiskā programmēšanas valoda ar drazu savācēju, kas atbalsta objektorientēto paradigmu
8	JavaScript [52]	2.923%	Daudzu paradigmu (tai skaitā objektorientēta) universālā dinamiskā programmēšanas valoda ar drazu savācēju
9	SQL [48]	1.987%	Deklaratīva vaicājumu valoda, kas ir paredzēta datubāzēm
10	Ruby [96]	1.182%	Daudzu paradigmu (tai skaitā objektorientēta) universālā dinamiskā programmēšanas valoda ar drazu savācēju

Tātad, mērķa programmēšanas valodai ir jābūt universālai, objektorientētai un stingri tipizētai. Analizējot piedāvāto programmēšanas valodu sarakstu, var redzēt, ka šiem kritērijiem atbilst programmēšanas valodas Java, C++ un C#. Java un C# izmanto drazu savācēju atmiņas pārvaldībai, kas nozīme to, ka definējot mainīgo jeb objektu nav nepieciešams sekot līdzi tā dzīves ciklam un atbrīvot izdalīto atmiņu. Tas atkal ļauj atvieglot koda ģenerēšanas procesu – ir iespējams tikai definēt mainīgos bez nepieciešamības sekot līdzi brīdim, kad šie mainīgie

kļūst nesasniedzami un neizmantojami un būtu nepieciešams veikt aizņemtās atmiņas atbrīvošanu.

Rezultātā, kā mērķa programmēšanas valodu ir iespējams izmantot vai un Java, vai C#. No šīm programmēšanas valodām Java ir populārāka, tomēr tas arī nevar kalpot par noteicošo kritēriju valodas izvēlei – pēc TIOBE indeksa [110] datiem var redzēt, ka valodu popularitāte visu laiku mainās, un nav noteicošais faktors. Tātad, ir nepieciešams veikt turpmāko kandidātu analīzi.

Viens no programmēšanas valodas Java pamatprincipiem ir WORA (angl. *write once, run anywhere*) [50], kas nozīmē to, ka kodu, kas ir uzrakstīts šajā valodā, ir iespējams palaist vairākās platformās. C# valodas gadījumā izpildes platforma ir .NET Framework [65], kas atbalsta visu funkcionalitāti tikai Microsoft Windows [120] operētājsistēmas vidē. Eksistē arī platforma .NET Core [18], kas ir oficiāli atbalstīta Linux [109] un macOS [60] vidēs, tomēr pastāv vairāki ierobežojumi, kas neļauj uzskatīt C# par pilnīgu multiplatformu programmēšanas valodu. Var pieminēt arī Mono platformu [43], kas ir atvērta koda .NET platformas realizācija un teorētiski atbalsta vairāk platformas nekā .NET Core, tomēr Mono projekta mājaslapā var redzēt, ka tā nerealizē visas .NET Framework iespējas. Java valodas gadījumā, savukārt, atbalstīto platformu skaits ir lielāks, kas pēc promocijas darba autora domām arī ir viens no tās popularitātes iemesliem.

Tātad, pēc populārāko programmēšanas valodu analizēs, tika pieņemts lēmums kā mērķa valodu izmantot Java. Līdz ar to piedāvātā koda ģenerēšanas algoritma uzdevums ir atbalstīt šo programmēšanas valodu.

5.2. Koda ģenerēšanas stratēģijas definēšana

Pēc mērķa programmēšanas valodas izvēles ir nepieciešams noteikt stratēģiju, kas ļaus informāciju no divpusložu modeļa pārveidot par attiecīgās programmēšanas valodas kodu. Neskatoties uz to, ka šī darba ietvaros tiek izmantota noteikta programmēšanas valoda, šādai stratēģijai ir jābūt paplašināmai – ir jābūt iespējai pievienot arī citas mērķa valodas, minimāli modificējot transformāciju metodi.

Šim nolūkam tiek piedāvāts divpusložu modeļa transformācijas procesu sadalīt vairākos soļos. Sākumā ir nepieciešams definēt klases, kuras tiks izveidotas transformāciju rezultātā. Pēc tam ir nepieciešams apstrādāt informāciju, ko sniedz biznesa procesu diagrammas, metodes un to satura definēšana. Šajā solī netiek noteikti attiecīgo metožu īpašnieki jeb klases, kas saturēs

attiecīgo metodi. Kā jau iepriekš tika minēts, maksimālai divpusložu modeļa transformācijas likumu kopas elastībai būtu nepieciešams atbalstīt gan anēmisko, gan arī bagāto datu modeli [24]. Līdz ar to attiecīgo metožu piešķiršana noteiktām klasēm var notikt vēlāk. Šī soļa uzdevums, savukārt, ir ģenerējamo metožu definīciju (jeb signatūru) izveide un (ja tas ir iespējams) metožu satura jeb to iekšējās biznesa loģikas definēšana. Lai gan runa ir par vienu transformācijas soli, transformāciju realizācijas laikā tas var tikt sadalīts vairākos, kur katrs no soļiem ģenerē starpmodeli, kas tiek izmantots nākamās šī soļa iterācijas veikšanai.

Pēdējais transformāciju solis ir attiecīgās programmēšanas valodas koda ģenerēšana no starpmodeļa. Šajā solī ir iespējams attiecīgo starpmodeli pārveidot par programmatūras kodu dažādās programmēšanas valodās. Pie tam, izmantojot šādu pieeju, jaunas mērķa valodas pievienošana prasīs tikai šī soļa modifikāciju, kas atbilst prasībai par minimālu transformācijas likumu modifikāciju.

Tātad, programmatūras koda ģenerēšanas stratēģiju ir iespējams definēt šādi (skat. formulas (5.1)-(5.8)):

$$\begin{pmatrix} Concept\ Model \\ (BP\ Model_1 \dots BP\ Model_n) \end{pmatrix} \underset{Step\ 1}{\Longrightarrow} (Class\ Def_1 \dots Class\ Def_m) \tag{5.1}$$

$$Class\ Def = \begin{pmatrix} Name \\ (\langle Attr_1 | Type_1 \rangle \dots \langle Attr_k | Type_k \rangle) \end{pmatrix} \tag{5.2}$$

$$\begin{pmatrix} (Class\ Def_1 \dots Class\ Def_m) \\ (BP\ Model_1 \dots BP\ Model_n) \end{pmatrix} \Longrightarrow$$
$$\Longrightarrow \overbrace{\left\| \begin{matrix} Intermediate \\ Model \end{matrix} \right\|}^{for\ step\ in\ Step\ 2 \dots StepX} \Longrightarrow \begin{pmatrix} (Class\ Def_1 \dots Class\ Def_m) \\ (Method\ Def_1 \dots Method\ Def_o) \\ (Method\ Body\ Def_1 \dots Method\ Body\ Def_l) \end{pmatrix} \tag{5.3}$$

$$Method\ Def = \begin{pmatrix} Name \\ (\langle Input_1 | Type_1 \rangle \dots \langle Input_k | Type_k \rangle) \\ (\langle Output_1 | Type_1 \rangle \dots \langle Output_k | Type_k \rangle) \end{pmatrix} \tag{5.4}$$

$$Method\ Body\ Def = (Op_1 \dots Op_k) \tag{5.5}$$

$$\begin{pmatrix} (Class\ Def_1 \dots Class\ Def_m) \\ (Method\ Def_1 \dots Method\ Def_o) \end{pmatrix} \xRightarrow[Method\ Assignment]{} (EClass\ Def_1 \dots EClass\ Def_m) \tag{5.6}$$

$$EClass\ Def = \begin{pmatrix} Name \\ (\langle Attr_1 | Type_1 \rangle \dots \langle Attr_k | Type_k \rangle) \\ (Method\ Def_1 \dots Method\ Def_o) \end{pmatrix} \tag{5.7}$$

$$\begin{pmatrix} (EClass\ Def_1 \dots EClass\ Def_m) \\ (Method\ Def_1 \dots Method\ Def_o) \\ (Method\ Body\ Def_1 \dots Method\ Body\ Def_l) \end{pmatrix} \xRightarrow[Code\ Generation]{} Code \tag{5.8}$$

Sākumā konceptu modelis `Concept Model` un viens līdz `n` biznesa procesu modeļi `BP Model`$_1$...`BP Model`$_n$ tiek pārveidoti par ģenerējamo klašu definīcijām `Class Def`$_1$...`Class Def`$_m$, kas satur informāciju par attiecīgo klašu nosaukumiem un to atribūtiem. Pēc tam tiek izpildītas vairākas starptransformācijas, kas iteratīvi ģenerē attiecīgos starpmodeļus. Šo iterāciju rezultātā tiek iegūta informācija par ģenerējamo metožu, kas atbilst biznesa procesiem, signatūrām (`Method Def`$_1$...`Method Def`$_k$) un to saturu jeb biznesa loģiku. Metožu signatūras satur informāciju par metožu nosaukumiem, to ieejas un izejas parametriem, kamēr metožu saturs tiek reprezentēts kā izpildāmo operāciju un programmatūras koda konstrukciju secība, kas var tikt pārveidota par attiecīgās programmēšanas valodas kodu. Šīs definīcijas var attēlot dažādos veidos, un nākamā sadaļa ir veltīta piemērota attēlojuma (attiecīgas datu struktūras) analīzei. Kad visa nepieciešamā informācija ir uzģenerēta, klasēm var piesaistīt metodes, pārejot no `Class Def`$_1$...`Class Def`$_m$ uz `EClass Def`$_1$...`EClass Def`$_m$, kur attiecīgās modificētās klašu definīcijas satur informāciju ne tikai par to nosaukumiem un atribūtiem, bet arī par metodēm, kas pieder attiecīgajai klasei. Beigās šī informācija ar koda ģeneratora palīdzību tiek pārveidota par attiecīgās programmēšanas valodas programmatūras kodu.

Rezultātā, šāda stratēģija ļauj gan mainīt izmantoto datu modeli (bagātais vai anēmiskais [24]), modificējot transformācijas (5.6) algoritmu, gan arī izmantot dažādas mērķa programmēšanas valodas, modificējot transformācijas (5.8) algoritmu.

5.3. Datu struktūru definēšana koda ģenerēšanai

Lai būtu iespējams veikt divpusložu modeļa transformāciju programmatūras kodā, to ir nepieciešams pārveidot attiecīgās datu struktūrās, kas atbalsta koda ģenerēšanu. Divpusložu modelis sastāv no divām diagrammām – biznesa procesu diagrammas un konceptu diagrammas. Pie tam, viens divpusložu modelis vienmēr ietver vienu konceptu diagrammu un vienu vai vairākas biznesa procesu diagrammas.

Konceptu diagramma var tikt apskatīta kā konceptu kopa, kas satur informāciju par problēmsfērā izmantotiem konceptiem. Šo konceptu kopu var arī saukt par domēna modeli. Katru konceptu raksturo tā nosaukums un signatūra, kas ir atribūtu un to datu tipu saraksts. Konceptu diagramma transformāciju laikā kļūst par pamatu klašu kopas ģenerēšanai – katrs tā koncepts var tikt pārveidots par attiecīgo klasi, saglabājot visu informāciju, kas ir definēta konceptu modelī.

Biznesa procesu diagrammas savukārt var tikt raksturotas kā orientētais multigrafs ar cilpām [23]. Tās reprezentē atsevišķo lietošanas gadījumu biznesa loģiku, definējot procesu, kas veic datu apstrādi, izpildes secību, kā arī datus, kurus katrs no procesiem saņem un producē. Šīs diagrammas tiek izmantotas kā pamats biznesa loģikas ģenerēšanai – transformāciju uzdevums ir šo diagrammu pārveidošana par metodēm, kas turpmāk var tikt piesaistītas dažādām klasēm. Šīs transformācijas laikā ir nepieciešams ne tikai definēt metožu signatūras, bet arī pārveidot biznesa loģikas definīcijas, ko satur biznesa procesu diagrammas, par programmatūras kodu.

Lai būtu iespējams iegūt programmatūras kodu, kas reprezentē biznesa procesu diagrammās notiekošos procesus, ir nepieciešamas pārveidot tās par attiecīgām datu struktūrām, kas atbilst elementiem $\texttt{Method Body Def}_1 \ldots \texttt{Method Body Def}_1$ formulā (5.3). Šādas datu struktūras reprezentē programmatūras kodu, kas ir atbildīgs par programmatūras loģikas realizāciju un pēc savas būtības ir mērķa modeļa reprezentācijas veids.

Mērķa modelis jeb programmatūras kods var tikt reprezentēts vairākos veidos – to var definēt gan teksta formā, gan arī ka abstraktu sintakses koku [34]. Lai gan abas šīs programmatūras koda reprezentācijas var tikt izmantotas koda ģenerēšanas nolūkiem, promocijas darba autors vēlas pievērst uzmanību citām reprezentācijas iespējām. Jebkura programmēšanas valoda kompilācijas vai arī izpildes laikā tiek pārveidota tā saucamajā mašīnvalodā [97]. Mašīnvaloda ir centrālā procesora instrukciju kopa, kas tiek uzdota binārā formā. Lai arī cilvēks būtu spējīgs uztvert mašīnvalodu, pastāv programmēšanas valodu kopa,

ko sauc par asambleriem [97]. Asamblera valodas ļauj definēt programmatūras kodu salīdzinoši zemā, bet tomēr cilvēkam saprotamā līmenī. Java [50] valodas gadījumā par asamblera valodu var saukt Java Virtuālas Mašīnas (angl. *Java Virtual Machine* – JVM) baitu koda valodu [107], kas salīdzinājumā ar citām asamblera valodām ir pietiekami vienkārša un satur ~200 instrukcijas, kamēr, piemēram, x86-64 asamblera valoda [46] satur ~1000 instrukcijas. Līdzīgi arī .NET [65] saimes valodas (piemēram, C#, C++ .NET, Visual Basic .NET) sākumā tiek pārveidotas baitu koda instrukcijās, kas tiek sauktas par CIL (angl. *Common Intermediate Language*) – tas ļauj vienu un to pašu programmatūras kodu izpildīt vairākās platformās, pārveidojot baitu koda instrukcijas par attiecīgās platformas mašīnvalodas instrukcijām, kas var notikt gan izpildes laikā, gan arī pie programmas palaišanas.

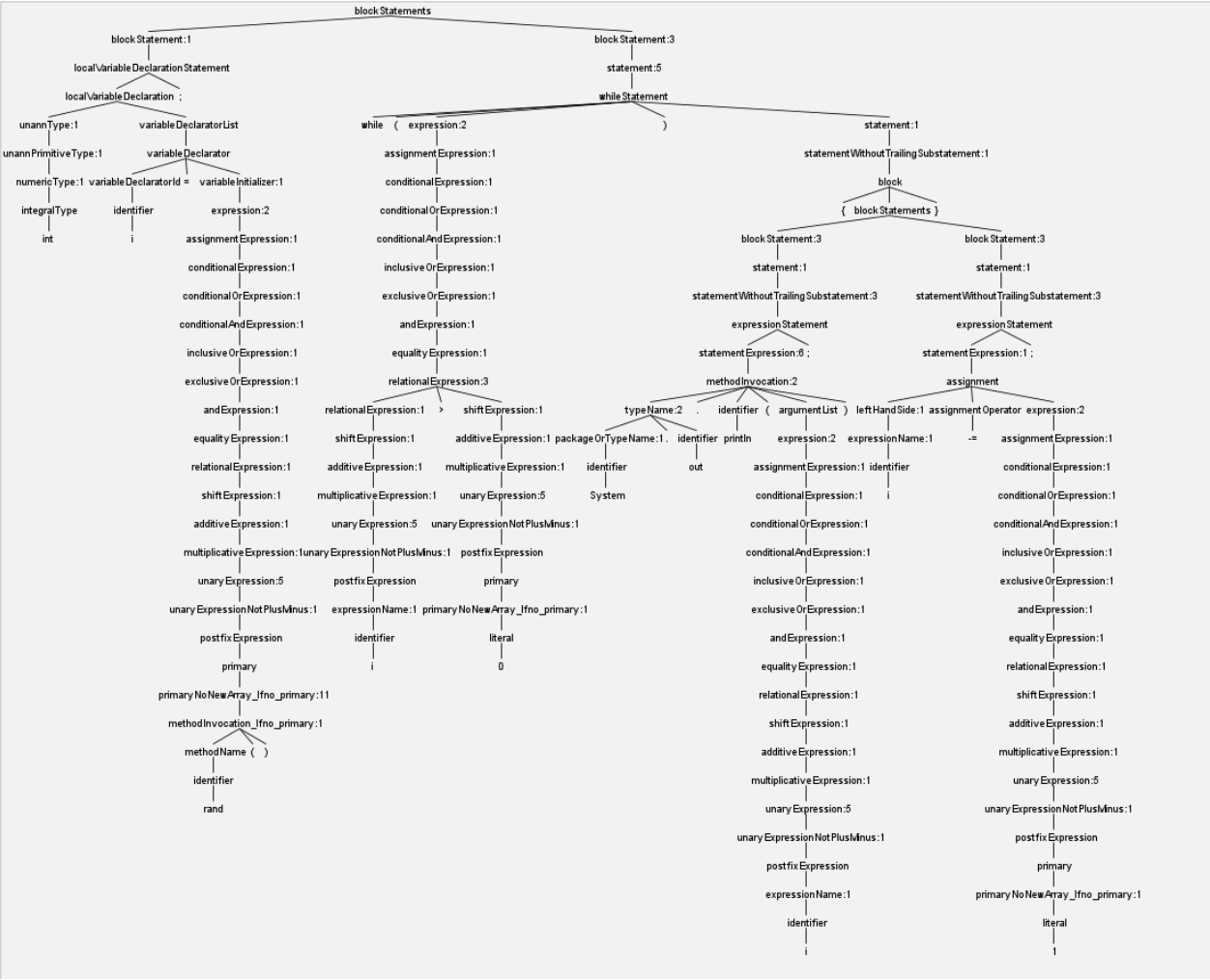
Tātad programmatūras kodu var reprezentēt vairākos veidos – teksta formā, kā abstrakto sintaktisko koku, vai asamblerim līdzīgā veidā. Lai demonstrētu visas trīs reprezentācijas iespējas un veiktu to turpmāko analīzi piemērotākas alternatīvas izvēlei, autors piedāvā apskatīt vienkāršu Java valodas koda fragmentu, kas ir parādīts 5.1. att. Piedāvātais koda fragments satur mainīgā `i` definīciju, kas ir metodes `rand()`, kas ģenerē gadījuma skaitli, izpildes rezultāts un ciklu `while`, kas izpildās, kamēr `i` ir lielāks par 0, katrā iterācijā izvadot mainīgā `i` vērtību un samazinot to par 2. Kopā šis koda fragments satur 5 rindiņas.

```
int i = rand();
while (i > 0) {
    System.out.println(i);
    i -= 2;
}
```

5.1. att. Java valodas koda piemērs

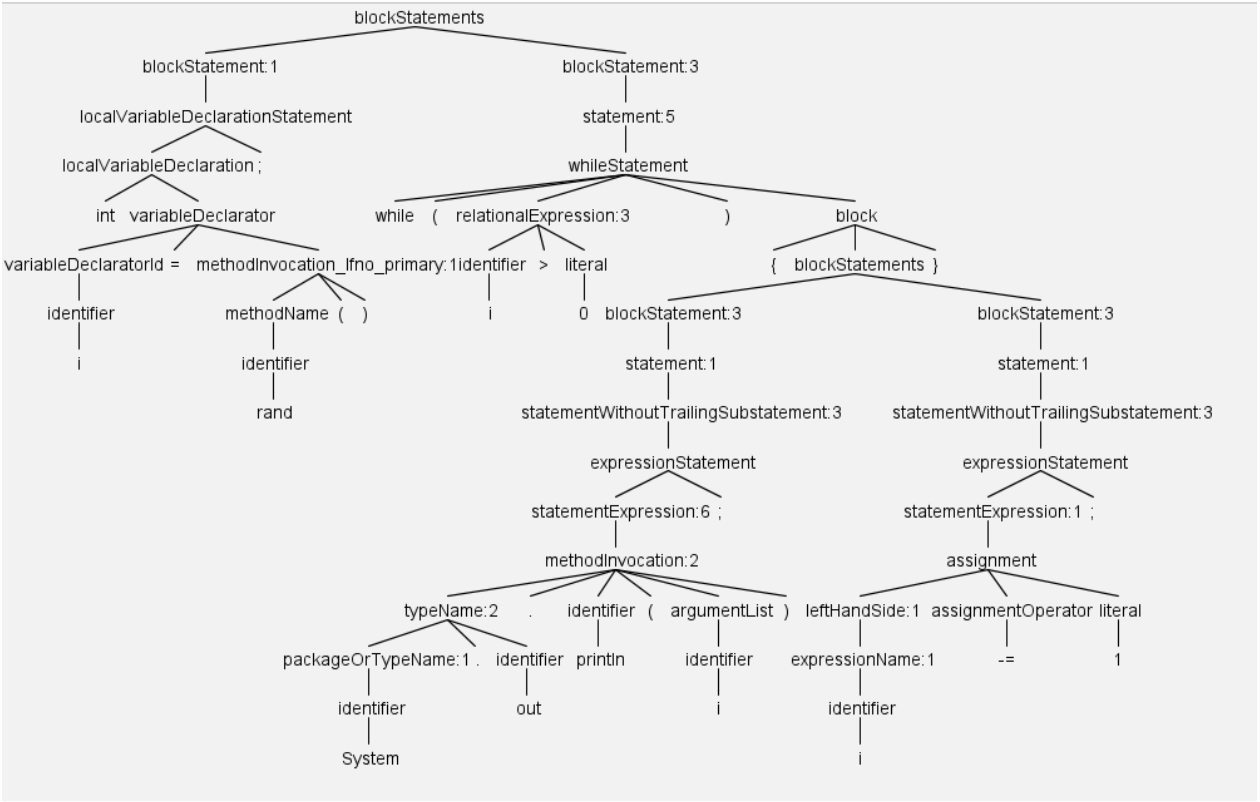
5.2. att. ir parādīta attiecīgā koda fragmenta reprezentācija ar abstrakta sintaktiskā koka palīdzību. Vizualizācijai tika izmantota ANTLR [4] bibliotēka ar Java valodas gramatikas definīciju.

Var redzēt, ka tā paša koda reprezentācija ar abstrakta sintaktiskā koka palīdzību aizņem vairāk vietas un nav tik vienkārši uztverama. Tas ir skaidrojams ar to, ka piedāvātā ANTLR [4] gramatika Java valodai ir pietiekami sarežģīta – tā satur informāciju par visām iespējamām Java valodas konstrukcijām. var būt vienkāršota. Izņemot no ANTLR gramatikas elementus, kuri netiek pielietoti, tos var ar neattēlot arī abstrakti sintaktiskajā kokā, tādā veidā to vienkāršojot. Piemēram, ir iespējams no attiecīgās ANTLR gramatikas izņemt nevajadzīgās konstrukcijas, atstājot tikai tās, kas var tikt uzģenerētas transformāciju laikā. Šāda vienkāršota abstrakta sintaktiska koka piemērs ir parādīts 5.3. att.



5.2. att. Java valodas koda piemērs abstrakta sintaktiska koka formā

Pēdējais programmatūras koda konstrukciju reprezentācijas variants, ko šī darba autors vēlas apskatīt šajā darbā ir asamblerim līdzīgas valodas izmantošana [38]. Tas pats Java koda fragments kompilācijas laikā tiek pārveidots par JVM baitu kodu. Attiecīgais baitu kods ir parādīts 5.4. att. Var redzēt, ka valodas konstrukcijas tika pārveidotas lineārā struktūrā jeb operāciju secībā, kurai tika pievienotas iezīmes zarošanās realizācijai. Pie tam, baitu kods satur divu veidu zarošanās instrukcijas – nosacījuma zarojumu, kas atbilst cikla `while (i > 0)` definīcijai un vienkāršas pārejas operatoru, kas atbild par šī cikla beigām.



5.3. att. Java valodas koda piemērs vienkāršota abstrakta sintaktiska koka formā

```
L0
    INVOKESTATIC Example.rand ()I
    ISTORE 0
L1
    ILOAD 0
    IFLE L2
L3
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    ILOAD 0
    INVOKEVIRTUAL java/io/PrintStream.println (I)V
L4
    IINC 0 -2
    GOTO L1
L2
    RETURN
```

5.4. att. Java valodas koda piemērs JVM baitu koda formā

Veicot piedāvātā koda reprezentācijas veidu analīzi, var redzēt, ka tekstuāla reprezentācija vizuāli aizņem vismazāk vietas un ir vienkāršāk uztverama, abstraktais sintaktiskais koks var aizņemt salīdzinoši daudz vietas un tā konstruēšana nav triviāls uzdevums – jo ir nepieciešams ņemt vērā vai nu visas iespējamās valodas gramatikas konstrukcijas, vai nu reducēt attiecīgās valodas gramatiku uz vienkāršāku. Baitu koda jeb asamblerim līdzīga reprezentācija, savukārt ir instrukciju secība jeb lineāra datu struktūra, ko, piemēram, var

definēt ar masīva vai saraksta palīdzību. Pie tam šādu reprezentāciju ir iespējams izmantot arī turpmākai koda ģenerēšanai – ir iespējams veikt iezīmju un attiecīgo zarošanās punktu (jeb kontroles nodošanu uz iezīmēto punktu) analīzi augsta līmeņa zarošanās konstrukciju ģenerēšanai. Zemāk ir apskatīts šādas transformācijas realizācijas piemērs.

Algoritmam, kas apstrādā secīgu instrukciju un iezīmju kopu būtu jāņem vērā vairāki faktori. Sākumā ir nepieciešams izņemt nevajadzīgās iezīmes, kas nav zarošanās mērķi. Šim nolūkam nepieciešams atrast visas zarošanās instrukcijas un sastādīt attiecīgo iezīmju kopu. 5.4. att. sniegtajā baitu koda fragmenta tās ir `L1` un `L2`. Pēc šādas iezīmju kopas definēšanas ir iespējams izvairīties no citām iezīmēm. Tātad, baitu kods no 5.4. var tikt pārveidots par baitu kodu, kas ir parādīts 5.5. att. Nākamais šāda algoritma solis varētu būt iezīmju mērķu analīze – var redzēt, ka kontroles nodošana uz iezīmi `L1` notiek pēc tās definīcijas. Tas nozīmē to, ka šajā gadījumā ir nepieciešams ģenerēt ciklu, kura sākums ir iezīme `L1`, bet beigas – pēdējā kontroles nodošanas vieta uz šo iezīmi. Ir nepieciešams izmantot tieši pēdējo kontroles nodošanas punktu, jo vairākas programmēšanas valodas, piemēram, C [49], Java [50], C++ [101], kā arī citas, atbalsta `continue` operatoru, kas ļauj uzsākt jaunu cikla iterāciju, nepabeidzot pašreizējo.

```
    INVOKESTATIC Example.rand ()I
    ISTORE 0
L1
    ILOAD 0
    IFLE L2
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    ILOAD 0
    INVOKEVIRTUAL java/io/PrintStream.println (I)V
    IINC 0 -2
    GOTO L1
L2
    RETURN
```

5.5. att. Baitu kods bez nevajadzīgām iezīmēm

Pēc aprakstītās transformācijas var tikt iegūts modificētais baitu kods, kas ir parādīts 5.6.att. Var redzēt, ka tajā ir palikusi tikai viena iezīme, ko ir nepieciešams apstrādāt zarošanās konstrukciju ģenerēšanai.

```
    INVOKESTATIC Example.rand ()I
    ISTORE 0
LOOP 1
    ILOAD 0
    IFLE L2
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    ILOAD 0
    INVOKEVIRTUAL java/io/PrintStream.println (I)V
    IINC 0 -2
END LOOP 1
L2
    RETURN
```

5.6. att. Modificētais baitu kods

Var definēt arī citus likumus iezīmju apstrādei, tomēr tas nav šī piemēra uzdevums – tā mērķis bija parādīt iespēju izmantot instrukciju un iezīmju secību koda ģenerēšanas nolūkiem. Pēc promocijas darba autora uzskatiem šāda datu struktūra ir pilnīgi piemērota un to var izmantot kā starpmodeli divpusložu modeļa transformāciju laikā [38].

Tātad kā sākotnējo biznesa procesu diagrammas transformācijas rezultātu, tika pieņemts lēmums izmantot asamblerim vai baitu kodam līdzīgu instrukciju un iezīmju secību, kas var tikt reprezentēta masīva vai arī saraksta veidā. Lai būtu iespējams veikt šādas datu struktūras ģenerēšanu, ir nepieciešams definēt visus iespējamos instrukciju veidus, kas var tikt iekļauti šādā secībā. Analizējot JVM baitu koda instrukciju sarakstu [107], autors instrukcijas ir sadalījis šādās apakšgrupās:

1. Matemātiskas operācijas, kas ir instrukcijas, kas veic darbības ar skaitļiem, piemēram, `DADD` (divu `double` tipa vērtību saskaitīšana).
2. Instrukcijas darbam ar JVM steku – atšķirībā no, piemēram, x86-64 arhitektūras [46], kur tiek izmantoti reģistri un steks mainīgo uzglabāšanai un apstrādei, Java Virtuāla Mašīna izmanto tikai steku. Šīs apakšgrupas instrukcijas ļauj ievietot datus stekā un izņemt no tā (piemēram, `ALOAD` ievieto steka virsotnē norādi uz lokālo mainīgo, bet `ASTORE` izņem norādi no steka virsotnes un ieraksta to mainīgajā), kā arī izveidot steka virsotnē jaunus objektus un masīvus (piemēram, `NEW` izdala atmiņu objektam un ievieto to steka virsotnē, bet `NEWARRAY` izdala atmiņu objektu vai primitīvu masīvam). Pie šīs apakšgrupas pieder arī instrukcijas, kas ļauj kopēt steka elementus (piemēram, `DUP`, kas izveido steka virsotnes kopiju un ievieto to stekā) un izņemt elementus no tā (piemēram, `POP`, kas izņem no steka tā virsotni neievietojot nekādā lokālajā mainīgajā).

3. Tipu pārveidošanas instrukcijas, kas ir domātas skaitliskiem tipiem, piemēram, D2I, kas pārveido `double` tipa mainīgo par `int` tipa mainīgo.
4. Tipu pārbaudes instrukcijas, kas ļauj pārbaudīt, vai steka virsotne var tikt pārveidota par noteikta tipa mainīgo (`CHECKCAST`, `INSTANCEOF`).
5. Skaitlisko tipu salīdzināšanas instrukcijas, kas salīdzina divus augšējos steka elementus, piemēram, `LCMP`, kas salīdzina divas `long` tipa vērtības.
6. Bloķēšanas instrukcijas `MONITORENTER` un `MONITOREXIT`, kas ļauj bloķēt pieeju noteiktam resursam.
7. Metožu izsaukuma instrukcijas, kas ļauj izsaukt noteikta objekta metodes, piemēram, `INVOKEVIRTUAL`.
8. Instrukcijas, kas ļauj iegūt un mainīt objektu atribūtu vērtības. Piemēram, `PUTFIELD` maina noteikta objekta atribūta vērtību, kamēr `GETSTATIC` ļauj iegūt statiska jeb klases atribūta vērtību.
9. Instrukcijas vērtību atgriešanai no metodēm. |Piemēram, `ARETURN`, kas ļauj kā metodes izsaukuma rezultātu atgriezt objektu.
10. Zarošanās instrukcijas, kas ir paredzētas kontroles nodošanai uz noteiktu iezīmi. JVM baitu kods iekļauj gan nosacījuma zarošanās instrukcijas (piemēram, `IFACMP_EQ`, kas izpilda kontroles nodošanu, ja divi salīdzināmie objekti ir vienādi pēc norādēm), gan vienkāršas parejas instrukcijas (piemēram, `GOTO`, kas izpilda kontroles nodošanu vienmēr), gan arī tā saucamās tabulas zarošanās instrukcijas, kas ir domātas `switch` valodas elementa apstrādei, piemēram, `TABLESWITCH`.
11. Atkļūdošanas instrukcija `BREAKPOINT`, kas netiek iekļauta kompilētā kodā, bet tiek pievienota dinamiski, kad attiecīgā programma tiek palaista atkļūdošanas režīmā.
12. Izņēmumu apstrādes instrukcija `ATHROW`.
13. Instrukcija `ARRAYLENGTH`, kas ļauj iegūt masīva garumu.
14. Instrukcija `NOP`, kas nozīme tukšu operāciju.

Arī .NET CIL baitu koda valoda [65] un x86-64 asambleris [46] satur līdzīgas instrukciju grupas. Tātad, arī definējot koda ģenerēšanai nepieciešamo instrukciju kopu ir nepieciešams definēt līdzīgas instrukciju grupas. Tomēr, salīdzinājumā ar baitu kodu un asamblera valodu, koda ģenerēšanai nepieciešamo instrukciju grupu skaits var būt mazāks . Zemāk promocijas darba autors veic katras instrukciju grupas analīzi ar mērķi noteikt, vai to ir nepieciešams iekļaut ģenerējamajā instrukciju secībā [38].

Matemātiskas operācijas netiek definētas biznesa procesu diagrammā – tās ir realizācijas detaļas, kas var tikt izmantotas noteiktu algoritmu realizācijai. Protams, var paredzēt gadījumus, kad biznesa procesa izpildes nosacījums var tikt definēts ar šādu operāciju palīdzību, tomēr tas nav nepieciešams. Biznesa procesu diagrammas saturs ir procesi, kas tiek izpildīti, dati kas tiek nodoti starp biznesa procesiem, kā arī procesu izpildes secība. Tātad, šādu instrukciju ģenerēšana nav obligāts nosacījums, un šī darba autors uzskata, ka tās (vismaz pagaidām) nav nepieciešamas.

Instrukcijas, kas ļauj strādāt ar steku (vai x86-64 [46] gadījumā ar steku un procesora reģistriem), ir nepieciešams atbalstīt vismaz minimālā apjomā – tās ir nepieciešamas gan datu nodošanai procesiem, gan arī procesu izpildes rezultātu saglabāšanai mainīgajos un atmiņā.

Tipu pārveidošanas instrukciju atbalsts nav obligāts – šādas instrukcijas pēc būtības ir realizācijas detaļas. Transformāciju sākotnējais uzdevums ir definēt abstrakto instrukciju secību. Tālāk ģenerējot programmatūras kodu no šīs instrukciju secības var tikt pievienot visas nepieciešamās konkrētās programmēšanas valodas konstrukcijas.

Tas pats attiecas uz tipu pārbaudes instrukcijām – tās var ievietot konkrētās programmēšanas valodas kompilators, apstrādājot jau uzģenerēto programmatūras kodu. Biznesa procesu diagramma satur informāciju par konkrētiem datu tipiem, tātad transformāciju laikā var uzskatīt, ka visi izmantotie mainīgie ir ar pareizām datu tipa vērtībām.

Līdzīgi kā matemātiskas operācijas, skaitlisko tipu salīdzināšanas instrukcijas nav nepieciešams obligāti ģenerēt – tās var būt gan realizācijas detaļas, gan arī ievietotas ar kompilatora palīdzību, pārveidojot uzģenerēto programmatūras kodu.

Bloķēšanas instrukcijas tiek izmantotas, realizējot laiksakritīgos algoritmus. Teorētiski, biznesa procesu modelis var saturēt informāciju par procesiem, kas tiek izpildīti paralēli, tomēr šādus gadījumus vienmēr ir iespējams pārveidot par secīgu izpildi. Līdz ar to, šīs instrukciju grupas atbalsts nav obligāts.

Tā kā biznesa procesu diagramma satur informāciju par procesu izpildes secību, un biznesa procesi transformāciju laikā tiek pārveidoti par klašu metodēm, ir nepieciešams atbalstīt metožu izsaukumu instrukcijas.

Metožu atgriežamie rezultāti var būt gan objekti, gan primitīvie datu dati, gan arī vairāku objektu apvienojums (piemēram, ja ir nepieciešams atbalstīt iespēju no vienas metodes vienlaicīgi atgriezt vairākus rezultāta objektus vai primitīvus datu tipus). Lai vēlāk būtu iespējams strādāt ar atsevišķām šādu apvienojumu sastāvdaļām, ir nepieciešams atbalstīt arī instrukcijas darbam ar objektu atribūtiem.

Ir nepieciešams atbalstīt instrukcijas vērtību atgriešanai no metodēm gan kompilējamā koda ģenerēšanai (ja metodes definīcijā ir noradīts atgriežamais datu tips, tad ir nepieciešams, lai visi metodes zari atgrieztu to – šo pārbaudi veic gan Java [50], gan C++ [101], gan arī citu valodu kompilatori), gan arī sarežģītāku transformāciju veiksmīgai realizācijai.

Tā kā biznesa procesu diagrammas satur informāciju par iespējamām izpildes alternatīvām (bagātinātā divpusložu modeļa notācija ļauj definēt arī procesu izpildes nosacījumus), ir nepieciešams atbalstīt zarošanas instrukcijas gan ar nosacījumiem, gan arī bez tiem. Eksistē divi ciklu veidi – ar nosacījuma pārbaudi cikla iterācijas sākumā vai iterācijas beigās. Ja cikla nosacījums tiek pārbaudīts cikla iterācijas sākumā, tad cikla iterācijas beigās ir nepieciešama beznosacījumu kontroles nodošana.

Atkļūdošanas un izņēmuma apstrādes instrukcijas nav nepieciešamas koda ģenerēšanai – pirmās parasti vispār netiek ievietotas kompilētajā kodā, kamēr otrā instrukciju veida apstrāde var prasīt arī papildus informācijas ievietošanu – speciālo iezīmju, kas atdala izņēmuma apstrādes blokus. Vispār, biznesa procesu diagramma nesniedz pietiekami daudz informācijas, lai būtu iespējams noteikt šādu bloku robežas.

Instrukcijas, kas ir domātas masīvu apstrādei, piemēram, masīva garuma noteikšanas instrukcija, arī nav obligātas transformāciju laikā – biznesa procesu diagramma nesatur informāciju par masīvu apstrādes realizācijas detaļām.

Tas pats attiecas arī uz `NOP` instrukciju, kas pēc savas būtības ir tukša instrukcija, kas var tikt ievietota jebkurā vietā, bet koda ģenerēšanas laikā nav vajadzīga – jo tā nesniedz nekādu informāciju par biznesa procesu izpildi.

Bez šīm instrukcijām ir nepieciešams ievest arī papildus instrukcijas – mainīgo definēšanai, kas ļaus definēt konteinerus attiecīgajām datu plūsmām koda ģenerēšanas laikā.

Tātad, ir iespējams definēt minimālo koda ģenerēšanas instrukciju kopu, kas atbalstīs visu informāciju, ko satur biznesa procesu diagrammas. Zemāk šādas instrukcijas ir uzskaitītas un īsi aprakstītas.

```
Label<Name>
```
(5.9)

Formulā (5.9) ir parādīts iezīmes ievietošanas instrukcijas formāts, kas satur informāciju par iezīmes simbolisko vārdu, kas var būt gan skaitlisks identifikators (piemēram, `Label<1>`), gan arī simbolu virkne, kas raksturo iezīmes nosaukumu (piemēram, `Label<FirstLabel>`). Svarīgākais, lai būtu iespējams šo vārdu izmantot arī citās koda ģenerēšanas instrukcijās.

```
Var<Id, Name, Type>
```
(5.10)

Formulā (5.10) ir parādīta mainīgā definēšanas instrukcija, kas satur informāciju gan par attiecīgā mainīgā unikālo identifikatoru, kas var būt gan skaitliska vērtība, gan arī simbolu virkne, mainīgā nosaukumu, gan arī tā datu tipu. Nosaukums šajā gadījumā ir simbolu virkne, kas ir datu plūsmas nosaukuma transformācijas rezultāts. Atkarībā no mērķa programmēšanas valodas šī nosaukuma formāts var būt dažāds – piemēram, programmēšanas valodas Java [50] gadījumā parasti tiek izmantots tā saucamais camel case stils (`firstVariable`), kamēr valodas C [49] gadījumā vairāki vārdi parasti tiek atdalīti ar apakšsvītras simbolu (`first_variable`). Līdz ar to var izmantot arī netransformētu datu plūsmas nosaukumu, lai turpmākās transformācijas (programmatūras koda ģenerēšana) varētu pārveidot to par attiecīgās valodas mainīgo formatētu pareizā stilā. Ir jāņem vērā arī tas, ka dažādās programmēšanas valodās var atšķirties rezervētie vārdi – piemēram, valoda C [49] ļauj izmantot vārdu `new` kā identifikatoru, kamēr C++ [101] un Java [50] tas ir rezervētais vārds. Atkal, vienkārši izmantojot datu plūsmas nosaukumu, šī problēma šajā solī netiek apskatīta. Tomēr, ir palikusi vēl viena problēma, ko ir nepieciešams atrisināt – datu plūsma modificētajā divpusložu modeļa notācijā var saturēt vairākus objektus vai primitīvus. Tas nozīme to, ka vienkārši izmantojot tās nosaukumu, var nonākt pie situācijas, kad tiks izveidoti vairāki mainīgie ar vienādiem nosaukumiem, bet atšķirīgiem datu tipiem, kas parasti netiek atļauts (dinamiski tipizētas programmēšanas valodas, piemēram, Python [119], Ruby [96] un citas to ļauj darīt, tomēr tas tiek uzskaitīts par slikto praksi). Ir iespējams atlikt šīs problēmas risināšanu – bez mainīgā nosaukuma šī instrukcija satur arī informāciju par datu tipu, ko var izmantot, pārveidojot uzģenerēto instrukciju secību par programmatūras kodu un papildinot mainīgā nosaukumu ar datu tipa informāciju, kas ļauj atrisināt vienādu nosaukumu problēmu.

Trešais instrukcijas parametrs ir datu tips. Nav striktu prasību tā nosaukumam– galvenā prasība ir atsekojamība. Izmantojot datu tipa parametru, programmatūras koda ģeneratoram ir jābūt spējīgam noteikt pareizu konkrētās valodas datu tipu, kas var būt gan primitīvais datu tips, gan valodas bibliotēkā iebūvēta klase, gan arī transformāciju laikā definētā klase.

Kā mainīgā definēšanas instrukcijas piemēru var definēt šādu konstrukciju (formula (5.11)):

```
Var<1, Name, String>
```
(5.11)

Kur 1 ir unikālais identifikators, `Name` ir mainīgā nosaukums, bet `String` – tā datu tips.

```
Invoke<Method, Inputs, Output?>
```
(5.12)

Formulā (5.12) ir parādīta konstrukcija, kas definē metodes izsaukumu. Tajā ir 3 parametri – `Method` ir norāde uz izsaucamo metodi, kas var būt gan metodes nosaukums, gan arī tās identifikators vai cita veida informācija, kas palīdz noteikt izsaucamo metodi. `Inputs` ir masīvs, kas satur lokālo mainīgo identifikatorus, kas ir metodes izsaukuma parametri. `Output` ir lokālā mainīgā identifikators, kas definē lokālo mainīgo, kurā tiks ierakstīts metodes izsaukuma rezultāts, kas nav obligāts, jo ir metodes, kas neatgriež vērtības. Lai gan vairākas programmēšanas valodas ļauj definēt metodes, kas atgriež vairākus rezultātus, promocijas darba autors koncentrējas uz viena mainīgā atgriešanas gadījumu. Tas ļauj palielināt transformācijas pielietojamību, jo vairākus mainīgos vienmēr var apvienot vienā objektā vai otrādi – no viena objekta izveidot vairākus mainīgos. Izmantojot lokālos mainīgos, nepieciešams definēt tikai norādes uz tiem, jo informācija par parametriem un to datu tipiem jau ir definēta metodes signatūrā. Šādas konstrukcijas piemērs ir sniegts formulā (5.13). Šeit `transformNumbers` ir metodes identifikators; `a`, `b` un `c` – tās ieejas parametri, bet `result` – mainīgais, kurā tiek ierakstīts metodes izsaukuma rezultāts.

```
Invoke<transformNumbers, [a, b, c], result>
```
(5.13)

Tā kā metodes izsaukuma rezultāts vienmēr ir viens mainīgais, ir nepieciešams definēt konstrukcijas darbam ar objektu atribūtiem, kas ļauj iegūt atribūta vērtību un ierakstīt to lokālajā mainīgajā, vai otrādi – no lokālā mainīgā vērtību ievietot attiecīgā objekta atribūtā. Šādas konstrukcijas ir definētas formulās (5.14) un (5.15).

```
GetField<Object, Field, Target>
```
(5.14)

```
PutField<Object, Field, Source>
```
(5.15)

Šeit abas operācijas strādā ar objekta `Object`, kas ir lokālā mainīgā identifikators, atribūtu `Field`, kas ir attiecīgā atribūta identifikators, kas var būt uzdots gan teksta formā, gan arī kā unikālais skaitlis. `Source` un `Target` ir lokālo mainīgo identifikatori, kuros tiek ierakstīta vai no kuriem tiek nolasīta attiecīgā atribūta vērtība. Šo konstrukciju piemēri ir doti formulās (5.16), kur no lokālā mainīgā ar identifikatoru `1` tiek nolasīts atribūts `x`, kas tiek ierakstīts

mainīgajā ar identifikatoru `3`, un (5.17), kur lokālā mainīgā `2` atribūtā `y` tiek ierakstīta mainīgā `3` vērtība.

```
GetField<1, x, 3>
```
(5.16)

```
PutField<2, y, 3>
```
(5.17)

Iepriekš definētās konstrukcijas ļauj definēt iezīmes un lokālos mainīgos, strādāt ar to atribūtiem un izsaukt metodes. Bez tām ir nepieciešamas arī konstrukcijas, kas atbalsta zarošanās iespējas. Vienkāršākā no šādām konstrukcijām ir beznosacījuma kontroles nodošanas konstrukcija, kas ir parādīta formulā (5.18). Tā satur vienu parametru `Label`, kas ir mērķa iezīmes identifikators.

```
Jump<Label>
```
(5.18)

Nākamā konstrukcija, kuru autors definē programmatūras koda reprezentācijai, ir saistīta ar nosacījuma pārbaudi. Tā kā modificētā divpusložu modeļa notācijā procesa izpildes nosacījums tiek definēts ar teksta palīdzību, kas pagaidām netiek apstrādāts (tas ir, nenotiek nosacījuma analīze), šī konstrukcija var tikt pārveidota dažādos koda fragmentos. Galvenais ir saglabāt informāciju par nosacījumu, kas tika definēts biznesa procesu modelī. Vērā jāņem tas, ka jebkuru izpildes nosacījumu var reducēt uz Bula loģiku – nosacījuma pārbaudes rezultāts ir vai nu patiess, vai nē. Tātad, nosacījuma pārbaudes rezultātu var ievietot lokālajā mainīgajā ar attiecīgo datu tipu, kas var būt, piemēram `boolean` vai `int`. Konstrukcijas definīcija ir parādīta formulā (5.19). Tā satur divus parametrus – `Var`, kas ir attiecīga lokālā mainīgā identifikators un `Guard`, kas ir nosacījuma teksts no biznesa procesu diagrammas.

```
Check<Var, Guard>
```
(5.19)

Kad nosacījums ir pārbaudīts, un tā pārbaudes rezultāts tika ierakstīts lokālajā mainīgajā, ir iespējams izmantot šo mainīgo zarošanās nolūkiem. Attiecīgās konstrukcijas ir parādītas formulās (5.20) un (5.21). Abas šīs konstrukcijas satur divus parametrus – `Var`, kas ir lokālais mainīgais, kas satur nosacījuma pārbaudes rezultātu, identifikators un `Label`, kas ir attiecīgās iezīmes identifikators. Atšķirība starp šīm divām konstrukcijām ir tāda, ka pirmā izpildās, ja attiecīgais mainīgais ir patiess, otrā – ja tas nav patiess.

```
JumpIf<Var, Label>
```
(5.20)

```
JumpIfNot<Var, Label>
```
(5.21)

Nākamā konstrukcija ir vērtības atgriešanas konstrukcija, kas ir parādīta formulā (5.22). Tā satur vienu neobligātu parametru, kas ir lokālā mainīgā identifikators, kas tiek atgriezts metodes izpildes rezultātā.

```
Return<Var?>
```
(5.22)

Lai parādītu šādas koda reprezentācijas iespējas, autors vēlas izmantot programmatūras kodu no 5.1. att. Pielietojot iepriekš definētās konstrukcijas, var izveidot šī koda reprezentāciju, kas ir parādīta 5.7. att. Var redzēt, ka attiecīgā reprezentācija satur visu to pašu informāciju, kas ir iekļauta programmatūras kodā ar vienu izņēmumu – tajā trūkst matemātiska operācija, kas izmaina lokālā mainīgā vērtību. Tomēr, kā tika iepriekš definēts, vismaz pagaidām biznesa procesu diagrammas nesatur šādu informāciju. Šādas informācijas atbalsts varētu būt viens no nākotnes pētījumu virzieniem.

```
Var<1, i, int>
Var<2, b, boolean>
Invoke<rand, [], 1>
Label<L1>
Check<2, "i > 0">
JumpIf<2, L2>
Invoke<System.out.println, [i], Ø>
Label<L2>
Return<Ø>
```

5.7. att. Programmatūras koda reprezentācija

Izmantojot piedāvātās konstrukcijas programmatūras koda reprezentācijai, ir iespējams pārrakstīt formulu (5.5) šādi (skat. formulas (5.23) un (5.24)):

$$Instr = \begin{cases} Label\langle Name\rangle \\ Var\langle Id, Name, Type\rangle \\ Invoke\langle Method, Inputs, Output\rangle \\ GetField\langle Object, Field, Target\rangle \\ PutField\langle Object, Field, Source\rangle \\ Jump\langle Label\rangle \\ Check\langle Var, Guard\rangle \\ JumpIf\langle Var, Label\rangle \\ JumpIfNot\langle Var, Label\rangle \\ Return\langle Var\rangle \end{cases} \quad (5.23)$$

$$Method\ Body\ Def = (Instr_1 \dots Instr_k) \quad (5.24)$$

5.4. Biznesa procesu diagrammas minimizēšana

Lai būtu iespējams no biznesa procesu diagrammas, kas pēc savas būtības ir orientēts multigrafs ar cilpām [23], pāriet uz lineāru struktūru, kas ir aprakstīta iepriekšēja sadaļā, ir nepieciešams veikt tās apstrādi, pārveidojot grafa struktūru. 2016. gada pētījuma, kurā piedalījās arī promocijas darba autors [79], rezultātā tika piedāvāts biznesa procesu modeļa diagrammas grafu apskatīt kā galīgo automātu [57],[100],[121]. Tas ļauj izmantot esošos algoritmus, kas ir paredzēti galīgo automātu apstrādei, arī biznesa procesu diagrammas apstrādei. Darbā [79] tika piedāvāts pārveidot biznesa procesu diagrammu par galīgo automātu un to, savukārt, par regulāro izteiksmi, kas pēc savas būtības ir lineāra leksisko elementu secība. Tomēr, kā viens no pētījuma rezultātiem, tika atzīmēta šādas regulārās izteiksmes pārmērība – vienas un tās pašas leksisko elementu secības var tikt atkārtotas, kas rezultātā noved pie atkārtojumiem arī programmatūras kodā. Neskatoties uz to, šī darba autors uzskata, ka biznesa procesu diagrammas reprezentācija galīgā automāta (vai tamlīdzīgas datu struktūras – jo galīgie automāti parasti ir domāti gramatiku apstrādei) formā var tikt izmantota diagrammas vienkāršošanai un transformācijai lineārā struktūrā. Šajā sadaļā tiek apskatītas tehnikas, kuras autors piedāvā izmantot biznesa procesu diagrammas pārveidošanai.

5.4.1. Metožu signatūru definēšana

Katrs biznesa process no biznesa procesu diagrammas tiek pārveidots par metodi, kura, savukārt, pēc tam tiek piesaistīta vienai no uzģenerētajām klasēm. Programmēšanas valodas Java [50] gadījumā eksistē metodes signatūras jēdziens, kas ietver sevī metodes vārdu un tās parametrus. Tomēr, var redzēt, ka šī definīcija neiekļauj sevī metodes atgriežamo tipu, kas nosaka to, kāda veida datus metode atgriež. Veicot biznesa procesu diagrammas transformāciju, katrs no procesiem tiek pārveidots par datu struktūru, kas ir parādīta formulā (5.4) un satur informāciju par ģenerējamās metodes nosaukumu, parametriem un atgriežamajiem datiem. Šajā solī informācija no sistēmas konceptuālā modeļa netiek izmantota, un faktiski parametri un atgriežamie dati ir nekas cits, kā datu plūsmas, savukārt metodes nosaukums var tikt ņemts no informācijas par biznesa procesu [78]. Tātad, formulu (5.4) pseidokodā var definēt kā klasi, kas ir parādīta 5.8. att.

```
class MethodSignature {
	const BusinessProcess process;
	const Set<DataFlow> parameters;
	const Set<DataFlow> outputs;

	constructor(Set<DataFlow> parameters, Set<DataFlow> outputs,
		BusinessProcess process) {

		this.process = process;
		this.parameters = parameters;
		this.outputs = outputs;
	}
}
```

5.8. att. Metodes signatūras definīcija

```
class MethodSignature {
	const BusinessProcess process;
	const Set<DataFlow> parameters;
	const Set<DataFlow> outputs;

	String guard;

	constructor(Set<DataFlow> parameters, Set<DataFlow> outputs,
		BusinessProcess process) {

		this.process = process;
		this.parameters = parameters;
		this.outputs = outputs;
	}
}
```

5.9. att. Metodes signatūras definīcija ar izpildes nosacījumu

Lai gan šī informācija ir pietiekama metodes ģenerēšanai, transformāciju gaitā ir nepieciešams arī noteikt, pie kāda nosacījuma metode tiks izsaukta, līdz ar to ir nepieciešams metodes signatūras definīcijai pievienot arī šo informāciju. Viens no promocijas darba autora piedāvātiem uzlabojumiem ir iespēja pievienot biznesa procesa definīcijai tā izpildes nosacījumu (skat 3.4. att.), un pievienojot šo informāciju datu struktūrai no 5.8. att., tiek iegūta datu struktūra, kas ir parādīta 5.9. att. Šī datu struktūra arī tiks izmantota turpmāk, transformāciju laikā.

Lai būtu iespējams no biznesa procesu diagrammas iegūt metožu signatūras definīcijas, ir nepieciešams pārbaudīt visus tajā definētos biznesa procesus, analizējot informāciju par procesā ienākošām un no tā izejošām datu plūsmām, kā arī informāciju par metodes izpildes nosacījumu. Šo uzdevumu risina algoritms, kas ir parādīts 5.10. att.

```
processMapping = {}
for p in processDiagram.processes:
       Set<DataFlow> inputs = p.getIncomingDataFlows()
       Set<DataFlow> outputs = p.getOutgoingDataFlows()

       processMapping[p] = new MethodSignature(inputs, outputs, p)
       if (p.hasGuard()):
              processMapping[p].guard = p.guard
```

5.10. att. Metožu signatūru definēšanas algoritms

Var redzēt, ka metožu signatūru definēšanas algoritms izveido asociatīvo masīvu jeb heštabulu, kur katram biznesa procesu diagrammas procesam atbilst metodes signatūra. Turpmāk šī informācija tiks izmantota biznesa procesu modeļa pārveidošanai. Pats par sevi algoritms ir salīdzinoši vienkāršs – tas veic visu procesu apstrādi, pārveidojot tos par attiecīgajām datu struktūrām.

5.4.2. Biznesa procesu diagrammas pārveidošana procesu izsaukumu grafā

Pēc metožu signatūru definēšanas soļa ir iespējams veikt pirmo biznesa procesu diagrammas pārveidošanu, rezultātā iegūstot grafu, ko šī darba autors ir nosaucis par procesu izsaukumu grafu. Šis grafs gandrīz vai pilnīgi sakrīt ar sākotnējo procesu diagrammu ar diviem izņēmumiem:

1. Grafa loki vairs neatbilst datu plūsmām un tiek pārveidi par savienojošiem elementiem, kuru vienīgais uzdevums ir definēt visas iespējamās procesu izsaukumu secības. Biznesa procesi savukārt tiek aizvietoti ar attiecīgām metožu signatūrām.
2. Grafam tiek pievienotas sākuma un beigu virsotnes, kas neatbilst nevienam procesam, bet nosaka sākotnējo un terminālo sistēmas stāvokli. Visi ārējie procesi, kas tikai producē datu plūsmas, nesaņemot nekādus datus, tiek savienoti ar sākotnējo virsotni. Ārējie procesi, kas tikai saņem datus, neproducējot nekādu rezultātu, kļūst par beigu virsotnes priekštečiem.

Transformāciju ir iespējams sadalīt vairākos soļos. Pirmais solis veic datu plūsmu analīzi, nosakot katram procesam atbilstošos priekštečus un pēctečus. Tā izpildes rezultātā tiek iegūta informācija par iespējamo procesu izpildes secību. Algoritma pseidokods ir parādīts 5.11.att.

```
parents = {}
children = {}
for d in processDiagram.dataFlows:
	source = d.getSource()
	target = d.getTarget()
	parents[target] += source
	children[source] += target
```

5.11. att. Procesu priekšteču un pēcteču noteikšana

Kad ir noteikti biznesa procesu priekšteči un pēcteči, ir iespējams, izmantojot informāciju no *processMapping* heštabulas (skat. 5.10. att.), izveidot iespējamo starpprocesu pāreju tabulu, kur diviem procesiem – `a` un `b` atbilst to savienojošais grafa loks `t`. Šī tabula tiek izmantota, lai novērstu situāciju, kad diviem biznesa procesiem atbilstošās virsotnes tiek savienotas ar vairāk nekā vienu loku. Algoritma pseidokods ir parādīts 5.12. att.

```
transitions = {}
for p in processDiagram.processes:
	List<Process> next = children[p]

	for np in next:
		i = processMapping[p]
		o = processMapping[np]

		if not (p, np) in transitions:
			transitions[(p, np)] = new Edge()
```

5.12. att. Starpprocesu pāreju tabulas izveidošana

Pēc starpprocesu pāreju tabulas izveidošanas ir iespējams to pārveidot par biznesa procesu modelim atbilstošu grafu – procesu izsaukumu grafu. Tas tiek izveidots, apstaigājot šo tabulu un pārveidojot informāciju no tās par grafa virsotnēm un lokiem. Algoritma pseidokods ir parādīts 5.13. att.

```
graph = new ProcessInvocationGraph()
for (from, to) => edge in transitions:
       if not graph.hasVertex(from):
              graph.addVertex(from)

       if not graph.hasVertex(to):
              graph.addVertex(to)

       graph.addEdge(from, to, edge)
```

5.13. att. Procesu izsaukumu grafa izveidošana

Var redzēt, ka izmantojot iepriekš sagatavoto starpprocesu pāreju tabulu, procesu izsaukumu grafa izveidošana kļūst par vienkāršu uzdevumu, kas tikai un vienīgi prasa informācijas pārnešanu no tabulas veidojamā grafā. Pēc šī soļa izpildes ir nepieciešams tikai pievienot sākuma un beigu stāvokļus, nosakot to priekštečus un pēctečus. Šo uzdevumu ir iespējams paveikt, izmantojot iepriekš apkopoto informāciju par procesu priekštečiem un pēctečiem. Sākumā procesu izsaukumu grafam tiek pievienotas attiecīgās sākuma un beigu virsotnes, kas turpmāk tiek savienotas ar atbilstošajām virsotnēm. Algoritma pseidokods ir parādīts 5.14. att.

```
initial = new Vertex()
terminal = new Vertex()

graph.addVertex(initial)
graph.addVertex(terminal)

for p in processDiagram.processes:
       if parents[p] is empty:
              sig = processMapping[p]
              graph.addEdge(initial, sig, new Edge())

       if children[p] is empty:
              sig = processMapping[p]
              graph.addEdge(sig, terminal, new Edge())
```

5.14. att. Sākuma un beigu virsotņu pievienošana procesu izsaukumu grafam

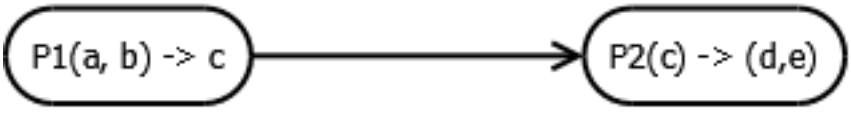
5.4.3. Procesu izsaukumu grafs

Iepriekšēja sadaļā tika apskatīts tā saucamā procesu izsaukumu grafa iegūšanas process no biznesa procesu diagrammas. Var redzēt, ka šāds grafs satur virsotnes, kas var attēlot informāciju par attiecīgo procesu izpildi, vai arī tukšos sistēmas stāvokļus – piemēram, sākuma un beigu stāvoklis. Šāds grafs ir līdzīgs funkcionālās struktūras modelim, kas ir aprakstīts darbā [33]. Šajā sadaļā tiek apskatīti citi iespējamie virsotņu satura varianti, kas tiks izmantoti grafa apstrādes procesā.

Apskatot iespējamo procesa izsaukumu grafa saturu, promocijas darba autors ir nonācis pie secinājuma, ka tas var tikt izveidots līdzīgi regulārajām izteiksmēm [57],[100],[121]. Faktiski, jebkurai regulārai izteiksmei atbilst noteikts galīgais automāts, kas apraksta sistēmas stāvokļus un iespējamās parejas starp tiem. Var redzēt, ka procesu izsaukumu grafs ir līdzīgs galīgajam automātam – tas apraksta iespējamos sistēmas stāvokļus, kas tiek reprezentēti ar procesu izsaukumiem. Tas ir, sistēmas stāvoklim atbilst noteikta procesa izsaukums. Šis grafs arī satur informāciju par iespējamām pārejām no viena stāvokļa citā jeb procesu izsaukumu iespējamo secību. Turpinot atbalstīt šo ideju, autors piedāvā līdzīgā veida apskatīt arī atsevišķas šī grafa virsotnes. Autors ir nonācis pie secinājuma, ka gadījumos, ja virsotnē viena procesa izsaukuma vietā tiks ievietots galīgais automāts, kas ir pierakstīts regulāras izteiksmes veidā, aizvietošana ir pilnīgi korekta un ļauj vienkāršot grafa apstrādi – piemēram, apvienojot virsotnes.

Zemāk ir apskatīts šīs pieejas izmantošanas piemērs. 5.15.att. ir parādīts procesu izsaukumu grafa fragments, kas satur informāciju par diviem procesu izsaukumiem, kas seko viens otram:

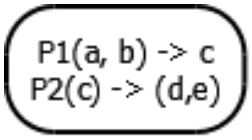
1. `P1`, kas saņem `a` un `b` kā parametrus un atgriež `c` kā rezultātu.
2. `P2`, kas saņem `c` kā parametru, atgriežot `d` un `e` kā rezultātu.



5.15. att. Procesu izsaukumu grafa fragments

Šie divi procesi ir savienoti ar vienu loku, kas nozīmē, ka starp tiem var eksistēt tikai iespējama tikai viena parēja – `P1 → P2`. Šādā gadījumā būtu iespējams apvienot šīs virsotnes,

izveidojot jaunu virsotni, kas satur informāciju par abiem procesu izsaukumiem. Šāda virsotne ir parādīta 5.16. att.



5.16. att. Virsotņu apvienošanas rezultāts

Var redzēt, ka virsotņu apvienošanas rezultātā tika izveidota jauna virsotne, kas satur informāciju par diviem procesiem un to izpildes secību. Šāda apvienošanas operācija ļauj gan minimizēt virsotņu skaitu grafā, kas tiek aprakstīts nākamajā sadaļā, kā arī iekļaut vienā virsotnē informāciju par vairāk nekā vienu procesu.

Iepriekšējā piemērā tika apskatīta viena no iespējām, kā viena virsotne var iekļaut informāciju par vairākiem procesiem. Promocijas darba izstrādes laikā autors ir identificējis arī citas iespējas apvienot informāciju par vairākiem procesiem, kā arī definējis citus iespējamos procesu izsaukumu grafa virsotņu veidus. Šie virsotņu veidi ir aprakstīti 5.2. tabulā, bet konkrēta nepieciešamība pēc tiem, kā arī to izmantošana ir sīkāk aprakstīta nākamajā sadaļā.

5.2. tabula

Procesu izsaukumu grafa virsotņu veidi

Tips	Pieraksts	Īss apraksts
Tukšais stāvoklis	∅	Virsotne, kas nesatur informāciju par procesu izsaukumiem, bet tikai un vienīgi apzīmē iespējamo sistēmas stāvokli. Piemēram, sākuma un beigu stāvoklis
Procesa izpilde	(If guard) $P(a_1, a_2, \ldots, a_n) = (b_1, b_2, \ldots, b_n)$	Definē noteiktu procesu `P`, kas saņem parametrus $a_1, a_2, \ldots, a_n$ un atgriež $b_1, b_2, \ldots, b_n$ kā rezultātu, izpildi. Var saturēt arī neobligātu procesa izpildes nosacījumu `guard`.
Cikla sākums	`LS: N`	Apzīmē punktu grafā, pēc kura sākas cikls `N`.
Cikla beigas	`LT: N`	Apzīmē punktu grafā, kur cikls `N` beidzas.
Cikla ieejas punkts	`LE: from`	Apzīmē punktu, no kura var sākties noteikta cikla izpilde. Satur speciālo parametru `from`, kas apzīmē iepriekšējo procesu izsaukumu grafa virsotni (vai arī daļu no tās), no kuras tiek novirzīta kontrole.

5.1. tabula (turpinājums)

Cikla izejas punkts	`LB: to`	Apzīmē punktu, kur noteiktā cikla izpilde var tikt pārtraukta. Satur speciālo parametru `to`, kas apzīmē iepriekšējo procesu izsaukumu grafa virsotni (vai arī tās daļu), kur tiek novirzīta kontrole pēc cikla pārtraukšanas.
Cikls	`(do/while)` `guard?: w`	Nozīmē to, ka dotās virsotnes saturs `w` var tikt atkārtots vairākas reizes, pie nosacījuma `guard` (kas arī var būt tukšs). Papildus tiek pievienota informācija par cikla veidu – ar pēcnosacījumu (`do`), vai ar pirmsnosacījumu (`while`).
Secība	`(w1, w2, …, wn)`	Apvieno vairākas cita veida virsotnes `w1, w2, …, wn`, kas var būt jebkura cita veida virsotnes, tai skaitā arī secības. Saturs tiek izpildīts secīgi.

5.4.4. Procesu izsaukumu grafa minimizēšana

Pēc procesu izsaukuma grafa izveidošanas tas satur tikai divu veidu virsotnes – tukšas un procesa izpildes virsotnes, kas attēlo informāciju no procesu modeļa. Promocijas darba autors ir izvirzījis hipotēzi, ka, izmantojot iepriekš aprakstīto virsotņu apvienošanu, ir iespējams šādu grafu novest līdz stāvoklim, kad tas saturēs tikai un vienīgi vienu secības tipa virsotni, kas, savukārt, satur visu informāciju par grafa sākuma stāvokli. Šāda minimizēšana ir nepieciešama tāpēc, ka iegūtā virsotne pēc savas būtības ir ne kas cits, kā lineārā struktūra, kas atbilst mērķa modelim – programmatūras kodam.

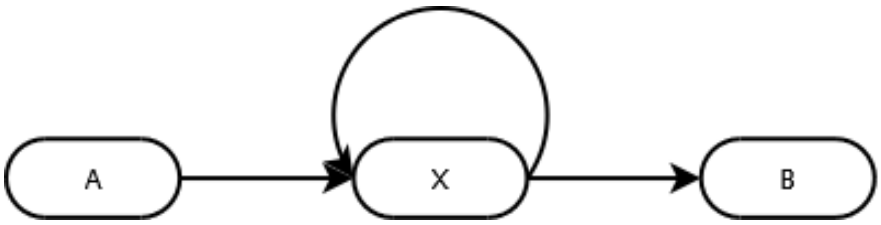
Lai būtu iespējams veikt šo uzdevumu, ir nepieciešams definēt vairākus grafa apstrādes algoritmus, kas, mainot tā struktūru, spēj saglabāt visu informāciju, kas šajā grafā ir attēlota. Šajā sadaļā tiek aprakstīti šie algoritmi, pēc tam tiek noteikti to pielietošanas noteikumi, jo atkarībā no transformāciju algoritmu pielietošanas secības var mainīties arī rezultāts. Līdz ar to ir nepieciešams noteikt, kādā secībā šos algoritmus būtu jāpielieto.

5.4.4.1. Cilpu aizvietošana

Pirmais algoritms, kas spēj mainīt procesu izsaukuma grafa struktūru, saglabājot visu informāciju, ir cilpu aizvietošanas algoritms. Šim algoritmam atbilst hipotēze: ja grafā eksistē virsotne ar cilpu, tad ir iespējams šo cilpu likvidēt, aizvietojot attiecīgo virsotni ar cikla virsotni. Lai demonstrētu, ka šī hipotēze ir pareiza, promocijas darba autors piedāvā apskatīt grafa fragmentu, kas ir parādīts 5.17. att. Analizējot šo fragmentu, var redzēt, ka, apstaigājot visas virsotnes pēc orientētājiem lokiem, pastāv divi iespējamie ceļi:

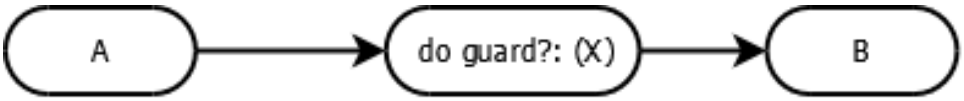
1. `A → X → B`, kad netiek apstaigāts cilpas loks.
2. `A → X → X* → B`, kas nozīme, ka pēc virsotnes A apstaigāšanas virsotne X var tikt apmeklēta vismaz vienu reizi – ieejot no virsotnes A. Pēc tam, 0 vai vairākas reizes var tikt apmeklēts cilpas loks.

Turpinot šī gadījuma analīzi, var redzēt, ka pēc savas būtības pirmais gadījums ir ne kas cits, kā otra gadījuma variants ar 0 cilpas apmeklēšanām. Tātad, šo grafa fragmentu var reprezentēt kā `A → X+ → B`, kur + nozīme vienu vai vairākas apmeklēšanas. Tas atbilst ciklam ar pēcnosacījumu no tabulas 5.2.



5.17. att. Procesu izsaukumu grafa fragments pirms cilpas aizvietošanas

Tātad, cilpu procesu izsaukumu grafā var aizvietot ar virsotni, kas satur ciklu ar pēcnosacījumu, kas arī ir parādīts 5.18. att.



5.18. att. Procesu izsaukumu grafa fragments pēc cilpas aizvietošanas

Šādu aizvietošanu realizē 5.19. att. parādītais pseidokods.

```
for v in graph.vertices:
       Set<Edge> incidents = v.getIncidents()
       Set<Edge> circles = incidents
              .find(e such that e.source = v and e.dest = v)

       if circles is not empty:
              List<Edge> inc = v.getIncomingEdges() - circles
              List<Edge> out = v.getOutgoingEdges() - circles

              for c in circles:
                     graph.removeEdge(c)

              Vertex loop = new DoLoop(v)
              for e in inc:
                     graph.removeEdge(e)
                     graph.addEdge(e.source, loop, e)

              for e in out:
                     graph.removeEdge(e)
                     graph.addEdge(loop, e.dest, e)

              graph.removeVertex(v)
```

5.19. att. Cilpu aizvietošanas algoritms

5.4.4.2. Loku dublikātu likvidēšana

Nākamo procesu izsaukumu grafa struktūru modificējošo algoritmu autors ir nosaucis par loku dublikātu likvidēšanas algoritmu. Tam atbilst šāda hipotēze: ja starp divām grafa virsotnēm eksistē vairāk nekā viens loks, pie tam vairāku loku orientācija sakrīt, tad šos lokus var apvienot, aizstājot ar vienu jaunu loku. Hipotēzes pareizības demonstrācija seko tālāk: 5.20.att. ir parādīts grafa fragments, kurā divas virsotnes `A` un `B` ir savienotas ar vairākiem lokiem. Veicot šo loku apstaigāšanu var pamanīt, ka, neatkarīgi no pirmā izvēlētā loka, virsotņu apstaigāšanas secība vienmēr ir viena un tā pati – `A → B → A`. Apvienojot vienādi orientētus lokus, tiek iegūta situācija, kas ir parādīta 5.21. att. Vienīgais iespējamas ceļš starp grafa virsotnēm tiek saglabāts. Tātad, šāda aizvietošana ir pieļaujama. Attiecīgā algoritma pseidokods ir parādīts 5.22. att.

5.20. att. Procesu izsaukumu grafa fragments pirms loku dublikātu likvidēšanas

5.21. att. Procesu izsaukumu grafa fragments pēc loku dublikātu likvidēšanas

```
for s in graph.vertices:
       for f in graph.vertices:
              Set<Edge> edges = graph.getAllEdgesBetween(s, f)
              if edges.size > 1:
                     for e in edges:
                            graph.removeEdge(e)

                     graph.addEdge(s, f, new Edge())
```

5.22. att. Loku dublikātu likvidēšanas algoritms

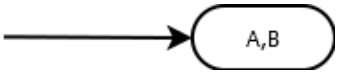
5.4.4.3. Secīgu virsotņu apvienošana

Hipotēze: ja procesu izsaukumu grafā eksistē tādas divas virsotnes `A` un `B`, ka:

1. Virsotne `A` ir savienota ar virsotni `B`, pie tam `B` seko `A`.
2. Vienīgais loks, kas ienāk virsotnē `B`, iziet no virsotnes `A`.
3. Virsotnē `A` ienāk 0 vai 1 loki.
4. No virsotnes `B` iziet 0 vai 1 loki.

Tad ir iespējams veikt secīgu virsotņu `A` un `B` apvienošanu `(A, B)`. Lai pierādītu šo hipotēzi, autors piedāvā apskatīt dotās situācijas piemēru, kas ir parādīts 5.23. att.

5.23. att. Procesu izsaukumu grafa fragments pirms virsotņu apvienošanas



5.24. att. Procesu izsaukumu grafa fragments pēc virsotņu apvienošanas

```
for s in graph.vertices:
       for f in graph.vertices:
              Set<Edge> edges = graph.getAllEdgesBetween(s, f)
              if edges.size == 1:
                     Set<Edge> inEdges = s.getIncomingEdges()
                     int oS = s.getOutgoingEdges().size
                     int iF = f.getIncomingEdges().size
                     Set<Edge> outEdges = f.getOutgoingEdges()
                     int iS = inEdges.size
                     int oF = outEdges.size

                     if iS <= 1 and oS <= 1 and iF <= 1 and oF <= 1:
                            Vertex seq = new Sequence(s, f)
                            graph.addVertex(seq)

                            for e in inEdges:
                                   graph.addEdge(e.source, seq, new Edge())
                                   graph.removeEdge(e)

                            for e in outEdges:
                                   graph.addEdge(seq, e.dest, new Edge())
                                   graph.removeEdge(e)

                            graph.removeVertex(s)
                            graph.removeVertex(f)
```

5.25. att. Virsotņu apvienošanas algoritms

Var redzēt, ka šajā grafa fragmentā starp divām dotajām virsotnēm eksistē viens iespējamais ceļš: `A → B`. Apvienojot virsotnes `A` un `B` vienā virsotnē, kas satur informāciju par šo virsotņu iepriekšējo saturu – `A,B` – tiek iegūta situācija, kas ir parādīta 5.24. att.

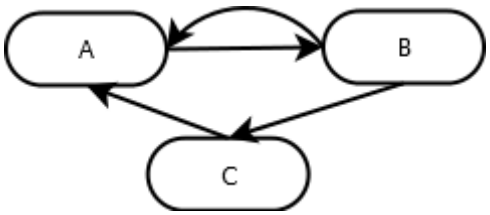
Vienīgais iespējamais ceļš starp abu virsotņu saturu tiek saglabāts, līdz ar to šāda aizvietošana ir atļauta. Attiecīgā algoritma pseidokods ir parādīts 5.25**Error! Reference source not found.**. att.

5.4.4.4. Ciklu aizvietošana

Procesu izsaukumu grafā var tikt atrasti cikli, kas prasa atsevišķu apstrādi. Lai būtu iespējams veikt ciklu apstrādi, sākumā ir nepieciešams šādus ciklus atrast. Pie tam, ir nepieciešams atrast tikai tā saucamos elementāros ciklus [53],[57],[104] jeb tādus ciklus, kuros visi iespējamie ceļi satur dažādas virsotnes, izņemot sākuma un beigu virsotni, kas savukārt sakrīt. Šim nolūkam promocijas darba autors piedāvā izmantot Džonsona algoritmu [53], kas ir balstīts uz Tarjana algoritma [104] izmantošanu. Zemāk tiek dots abu algoritmu un to darbības principu apraksts.

Tarjana algoritms [104] ir algoritms, kas ir paredzēts tā saucamo "stipri saistīto komponenšu" identificēšanai orientētā grafā. Stipri saistīta grafa komponente ir maksimālā

orientēta grafa virsotņu kopa, kurā visas virsotnes ir stipri saistītas [57]. Par stipri saistītām virsotnēm sauc tādas divas virsotnes `A` un `B`, ja grafā eksistē gan orientētais ceļš no virsotnes `A` uz virsotni `B`, gan orientētais ceļš no virsotnes `B` uz virsotni `A`. Var redzēt, ka stipri saistītas komponentes pēc savas būtības ir cikli, jo izvēloties jebkuru grafa virsotni `A`, un identificējot visus ceļus, kas sākas ar šo virsotni, būtu nepieciešams šajā virsotnē arī atgriezties, lai izpildītos stiprās saistības prasība. Tomēr, viena stipri saistīta komponente var saturēt vairākus ciklus. Šādas komponentes piemērs ir parādīts 5.26. att.



5.26. att. Stipri saistīta grafa komponente, kas satur vairākus ciklus

Kā var redzēt no dotā piemēra, stipri saistīta komponente ne vienmēr ir elementārais cikls, tomēr šādu komponenšu identifikācija ļauj šādus ciklus noteikt. Tarjana algoritma pseidokods ir parādīts 5.27. att.

```
stack = []
indexMap = {}
lowLinkMap = {}
index = 0

function tarjan(graph):
	result = []
	for v in graph.vertices:
		if not v in indexMap:
			sccs = strongConnect(graph, v)
			result += sccs

	return result

function strongConnect(graph, v):
	indexMap[v] = index
	lowLinkMap[v] = index
	index += 1
	stack.push(v);

	result = []
	for w in graph.getSuccessors(v):
		if not w in indexMap:
			c = strongConnect(w)
			result += c

			updateLowLinks(v, w)
		else if w in stack:
			updateLowLinks(v, w)

	vl = lowLinkMap[v] or 0
	vi = indexMap[v] or 0

	if vl == vi:
		scc = []

		while true:
			w = stack.pop()
			scc += w
			if w == v:
				break

		if scc is not empty:
			result += scc

	return result

function updateLowLinks(v, w):
	vl = lowLinkMap[v] or 0
	wl = lowLinkMap[w] or 0
	lowLinkMap[v] = min(vl, wl)
```

5.27. att. Tarjana algoritms

Pēc stipri saistīto komponenšu noteikšanas, šādas komponentes var tikt izmantotas elementāro ciklu meklēšanai dotajā grafā. Šo uzdevumu pilda Džonsona algoritms [53]. Algoritma variants, kas ir aprakstīts [53], ir paredzēts tādu grafu apstrādei, kuru virsotnes ir veseli skaitļi, tomēr procesu izsaukumu grafs nav šāda veida grafs. Līdz ar to promocijas darba autors piedāvā sākumā izveidot Džonsona algoritmam piemērotu grafa attēlojumu un tad veikt tā apstrādi. Attēlojums šajā gadījumā ir grafs, kur virsotnes ir veseli skaitļi, bet loki nesatur

nekādu papildu informāciju. Veidojot procesu izsaukuma grafa attēlojumu, katrai oriģinālai virsotnei tiek piešķirts numurs, sākot ar 1. Šī attēlojuma izveidošanas pseidokods ir parādīts 5.28. att.

```
dg = new Graph(int for vertex, any for edge)
dictionary = {}

function buildDg(graph):
	reverseDict = {}
	i = 1

	for v in graph.vertices:
		dictionary[i] = v
		reverseDict[v] = i
		dg.addVertex(i)
		i += 1

	for e in graph.edges:
		source = reverseDict[e.source]
		dest = reverseDict[e.dest]

		dg.addEdge(source, dest, new Edge())
```

5.28. att. Procesu izsaukumu grafa attēlojuma algoritms

Iegūto procesu izsaukumu grafa attēlojumu ir iespējams apstrādāt ar Džonsona algoritma palīdzību. Tomēr, pēc šādas apstrādes, informācija par elementāriem cikliem būs jāpārveido pretējā virzienā – iegūtos attēlojuma elementāros ciklus būs jāpārveido par procesu izsaukumu grafa cikliem.

Šādu operāciju palīdz paveikt funkcija, kuras pseidokods ir parādīts 5.29. att. Funkcija saņem attiecīgo virsotņu, kas ir elementāra cikla sastāvdaļas, sarakstu un pārveido to par procesu izsaukumu grafa virsotņu sarakstu. Tagad, definējot funkcijas procesu izsaukumu grafa attēlojumam, ir iespējams izpildīt Džonsona algoritmu, iegūstot elementāro ciklu sarakstu.

```
function restoreVertices(Set<int> circuit):
	result = []
	for i in circuit:
		v = dictionary[i]
		result += v

	return result
```

5.29. att. Atgriezeniskais attēlojums

5.30. att. ir parādītas divas Džonsona algoritma galvenās funkcijas. Pirmā no tām – `johnson()` saņem apstrādājamo grafu, izveido tā attēlojumu, veic elementāro ciklu meklēšanu

un tad atgriež iegūtos rezultātus, izmantojot atgriezeniska attēlojuma funkciju. Otrā – `findCircuits()` – veic ciklu meklēšanu, izmantojot citas palīgfunkcijas.

```
blocked = {}
circuits = []
blockedNodes = {}

function johnson(graph):
	buildDg(graph)
	findCircuits()

	result = []
	for c in circuits:
		orderedCircuit = sort(c)
		remappedCircuit = restoreVertices(orderedCircuit)
		result += remappedCircuit

	return result

function findCircuits():
	stack = []
	s = 1

	while s < gd.vertices.size:
		subGraph = subGraphFrom(s, dg)
		leastScc = findLeastScc(subGraph)

		if leastScc.vertices.size == 0:
			s = dg.vertices.size
		else:
			s = min(leastScc.vertices)
			for i in leastScc.vertices:
				blocked[i] = false
				blockedNodes[i] = []

			circuit(leastScc, s, s, stack)
			s += 1
```

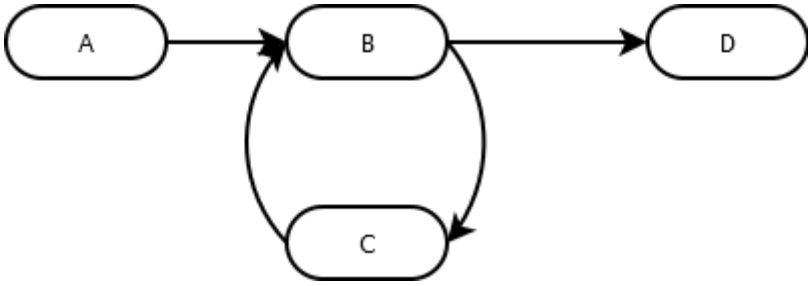
5.30. att. Galvenās Džonsona algoritma funkcijas

5.31. att. ir parādītas papildus funkcijas, kuras izmanto Džonsona algoritms. Algoritma izpildes rezultātā tiek atgriezti vairāki virsotņu saraksti, kur katrs no sarakstiem atbilst vienam elementāram ciklam.

```
function subGraphFrom(i, ing):
	result = new Graph(int for vertex, any for edge)
	for f in ing.vertices:
		if f >= i:
			for t in ing.getSuccessors(fromf
				if t >= i:
					result.addEdge(f, t, ing.findEdge(f, t))
	return result

function findLeastScc(g):
	sccs = tarjan(g), min = ∞, minScc = []
	for scc in sccs:
		for i in scc:
			if i < min:
				minScc = scc, min = i
	return addEdges(minScc, g)

function addEdges(list, g):
	result = new Graph(int for vertex, any for edge)
	for i in list:
		for edge in g.getOutgoingEdges(i):
			to = dg.getOpposite(i, edge)
			if to in list:
				result.addEdge(i, to, edge)
	return result

function circuit(g, v, s, stack):
	stack.push(v)
	blocked[v] = true, f = false
	for w in g.getSuccessors(v):
		if w == s:
			stack.push(s)
			c = copy of stack, f = true
			circuits.add(c), stack.pop()
		else if not (blocked[w] or false):
			f |= circuit(g, w, s, stack)
	if f:
		unblock(v)
	else:
		for w in g.getSuccessors(v):
			bnodes = blockedNodes[w] or []
			if not v in bnodes:
				bnodes += v
			blockedNodes[w] = bnodes
	stack.pop()
	return f

function unblock(u):
	blocked[u] = false
	while (blockedNodes[u] or []) is not empty:
		w = blockedNodes[u].removeFirstElement()
		if blocked[w]:
			unblock(w)
```

5.31. att. Papildus Džonsona algoritma funkcijas

Hipotēze: veicot procesu izsaukumu grafa apstrādi, ir iespējams tajā atrastos ciklus aizvietot ar citiem procesu izsaukumu grafiem, nozīmējot sākuma un beigu virsotnes, kur dotais cikls ir ietverts. Pēc [47] dotās definīcijas var redzēt, ka programmēšanas valodu gadījumā ciklu var definēt kā "instrukciju secību, kas tiek atkārtota". Tā kā procesu izsaukumu grafs pēc savas būtības atbilst instrukciju secībai, šādai aizvietošanai ir jābūt atļautai. 5.32. att. ir parādīts procesu izsaukumu grafa fragments, kas satur elementāro ciklu `(B, C)`.

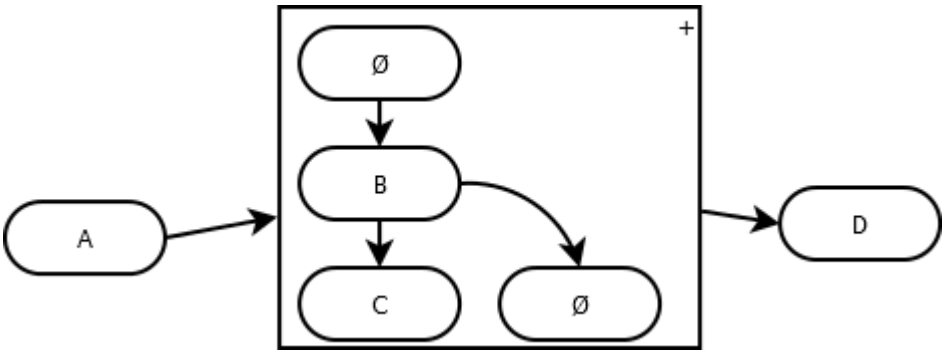


5.32. att. Procesu izsaukumu grafa fragments ar ciklu

Analizējot visus iespējamos ceļus, kas var eksistēt starp virsotnēm `A` un `B`, var izdalīt divus iespējamos scenārijus:

1. `A → B → D`, ciklam neizpildoties.
2. `A → (B → C)+ → D`, ciklam izpildoties.

Abus šos scenārijus var apvienot, iegūstot `A → (B → C?)+ → D`, kur "?" nozīme neobligātu attiecīgā procesa izpildi. Pārveidojot secību `B → C?` par atsevišķu procesu izsaukumu grafu, un ievietojot to oriģinālajā grafā, tiek iegūta situācija, kas ir parādīta 5.33. att.



5.33. att. Procesu izsaukumu grafa fragments ar aizvietoto ciklu

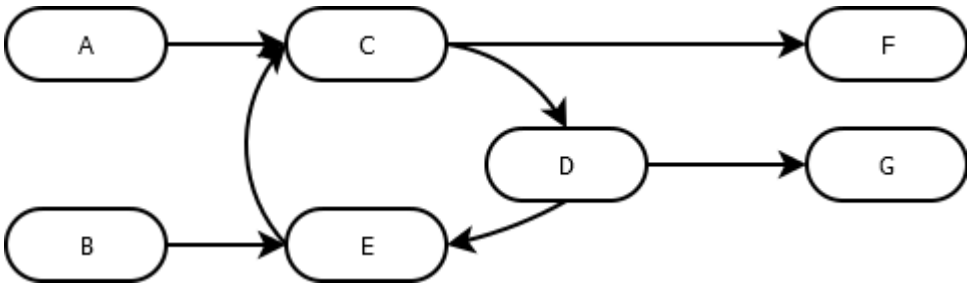
Šeit jaunais procesu izsaukumu grafs, kas aizvieto atrasto ciklu, satur divus iespējamos scenārijus:

1. `B`, kad virsotne `C` netiek apstaigāta.
2. `B → C`, kad virsotne `C` tiek apstaigāta.

Var redzēt, ka abi scenāriji sākas ar virsotnes B apstaigāšanu, kamēr virsotne C var netikt apstaigāta, kas ļauj šos divus scenārijus apvienot vienā: `B → C?`, savukārt, oriģinālais procesu izsaukumu grafa fragments šo virsotņu secību iekļauj vismaz vienu reizi. To var pierakstīt kā `A → (B → C?)+ → D`, kas pilnīgi atbilst iespējamiem ceļiem oriģinālajā grafā. Tātad, šāda aizvietošana ir atļauta.

Ir svarīgi arī atzīmēt faktu, ka jaunizveidotajā procesu izsaukumu grafā eksistē arī papildu virsotnes, kas šajā gadījumā ir sākuma un beigu virsotnes. Šīs virsotnes nesatur informāciju par procesu izsaukumiem un tāpēc arī netiek ņemtas vērā, rēķinot iespējamos ceļus.

Tomēr, situācija, kas tika apskatīta 5.32. att. un 5.33. att. ir salīdzinoši vienkārša: šeit eksistē viens loks, kas kalpo kā cikla ieeja, un arī viens loks, kas ir cikla izeja. Reālajos grafos ir iespējamas arī sarežģītākas situācijas – kad ciklu ir iespējams uzsākt no vairākām vietām grafā, kā arī izejas no cikla var vest uz vairākām iespējamām virsotnēm. Šāda grafa piemērs ir dots 5.34. att.



5.34. att. Procesu izsaukumu grafa fragments ar netriviālu ciklu

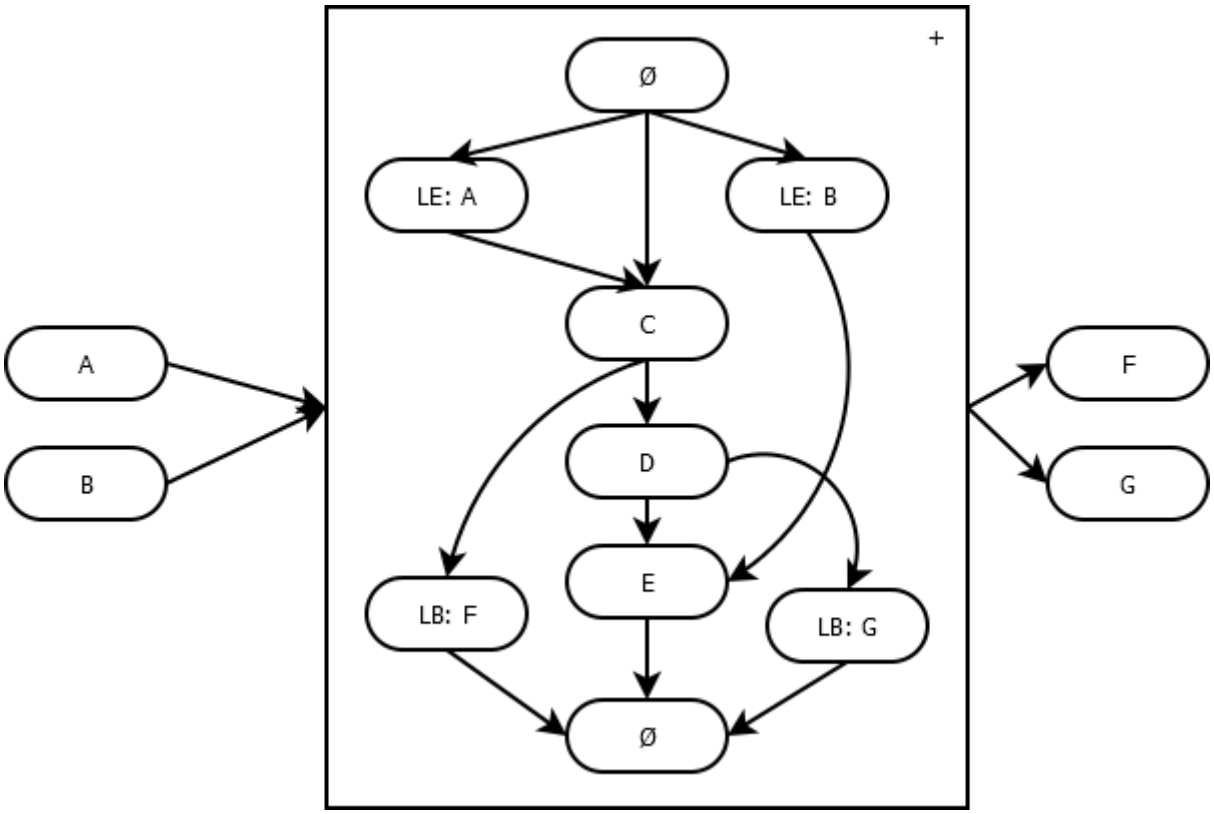
Piemērā eksistē divas virsotnes `A` un `B`, no kurām ir iespējams ieiet ciklā. Pie tam, no virsotnes `A` ciklā var ieiet sākot ar virsotni `C`, bet no virsotnes `B` – sākot ar virsotni `E`. Tapāt, no cikla var iziet gan no virsotnes `C`, gan arī no virsotnes `D`. Iespējamo ceļu analīzi šajā grafā promocijas darba autors piedāvā veikt pakāpeniski, sākot ar visiem ceļiem, kas sākas virsotnē `A`. Tādi ceļi ir:

1. `A → C → F`
2. `A → C → D → G`
3. `A → (C → D → E)+ → C → F`
4. `A → (C → D → E)+ → C → D → G`

Šeit parādās arī papildus nosacījumi, piemēram, virsotnē `G` ir iespējams nokļūt tikai no virsotnes `D`. Līdz ar to autors nemēģina apvienot visus šos ceļus vienā izteiksmē, bet gan izmantos tos turpmākai analīzei. Līdzīgi var definēt arī visus iespējamos ceļus, kas sākas ar virsotni `B`:

1. `B → E → C → F`
2. `B → E → C → D → G`
3. `B → E → (C → D → E)+ → C → F`
4. `B → E → (C → D → E)+ → C → D → G`

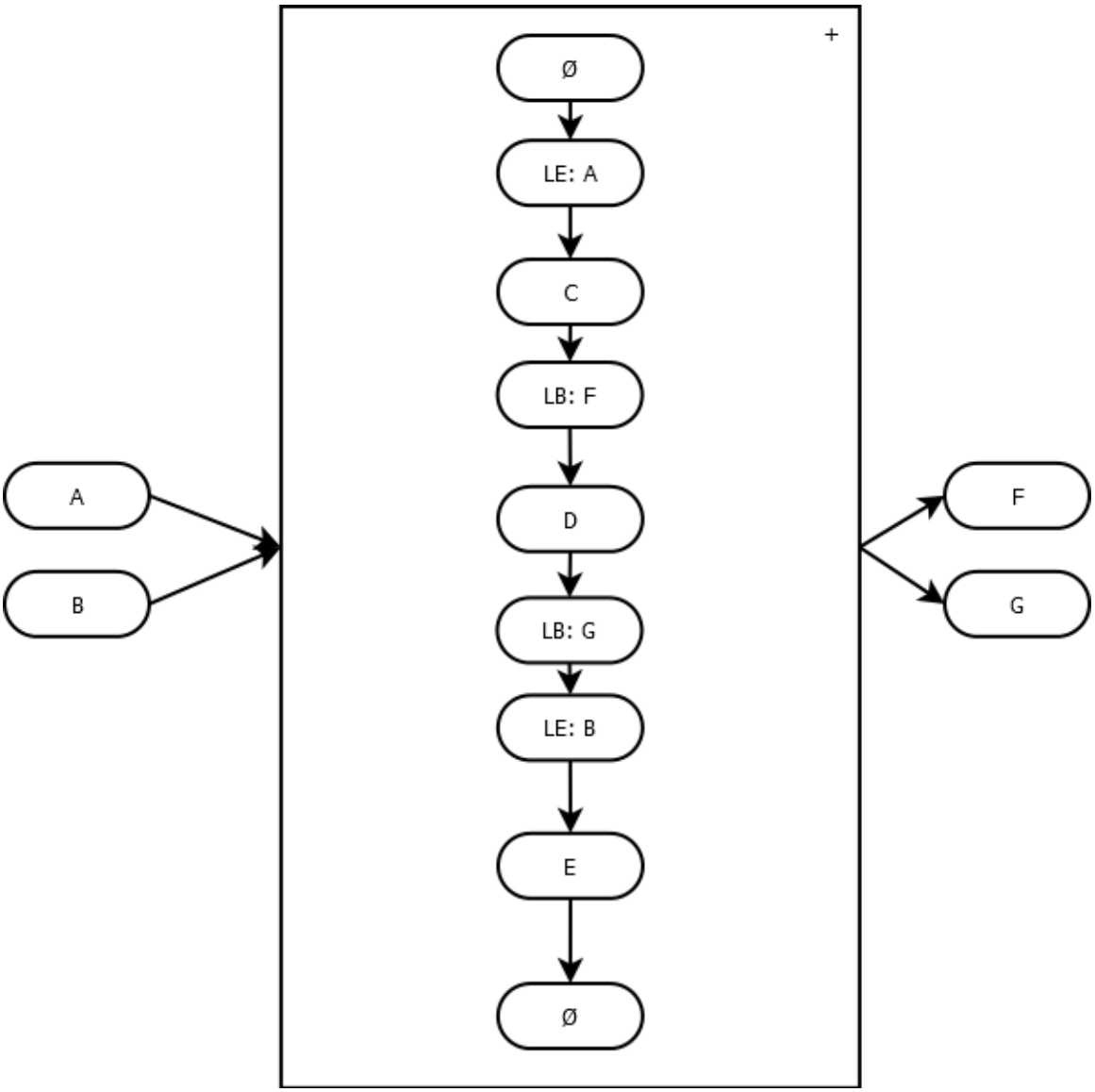
Atkal, iegūtie ceļi satur papildus nosacījumus, kas izriet no virsotņu secības oriģinālajā grafā. Autors piedāvā šo nosacījumu saglabāšanai izmantot speciālos elementus, kas tika definēti tabulā 5.2. un ir paredzēti cikla ieeju un cikla izeju apzīmēšanai. Izmantojot šādus elementus, var pārveidot doto grafa fragmentu, iegūstot aizvietojumu, kas ir parādīts 5.35. att.



5.35. att. Procesu izsaukumu grafa fragments ar aizvietotu netriviālo ciklu

Līdzīgi kā iepriekšējos gadījumos, var veikt aizvietojuma analīzi un atrast visus iespējamos ceļus pēc aizvietošanas. Var redzēt, ka tie sakrīt ar iepriekš aprakstītajiem ceļiem, kā arī var redzēt, ka papildu nosacījumi, kurus promocijas darba autors ir minējis iepriekš, arī tiek saglabāti – virsotnē `G` ir iespējams nokļūt no virsotnes `D`, bet virsotnē `F` – no virsotnes `C`. Tapāt, pēc virsotnes `A` nākamā virsotne ir `C`, bet pēc virsotnes `B` – `E`. Līdz ar to, arī šāda aizvietošana ir atļauta.

Tā kā ir zināms, no kuras virsotnes pārejas uz kurām virsotnēm ir atļautas, grafu no 5.35.att. var definēt arī citā veidā, kas ir parādīts 5.36. att.



5.36. att. Procesu izsaukumu grafa fragments ar aizvietotu netriviālo lineāro ciklu

Šeit apakšgrafs, kas atbilst ciklam, ir lineārs, bet ņemot vērā informāciju par cikla ieejas un izejas punktiem, visi iespējamie ceļi tiek saglabāti. Tas vienkāršo cikla atbilstoša procesu izsaukumu grafa struktūru un ļauj to apstrādāt, izmantojot vienkāršākus paņēmienus.

Procesu izsaukumu grafā ciklu aizvietošana ļauj visu turpmāko darbu veikt ar grafiem, kas ciklus nesatur un tas atvieglo to turpmāko apstrādi. Ciklu aizvietošanas algoritma pseidokods ir parādīts 5.37. att.

```
while true:
	loops = johnson(graph)
	if loops is not empty:
		loop = loops.pop()
		subGraph = buildLoopGraph(loop)
		with graph and loop do:
			replace loop with subGraph in graph
```

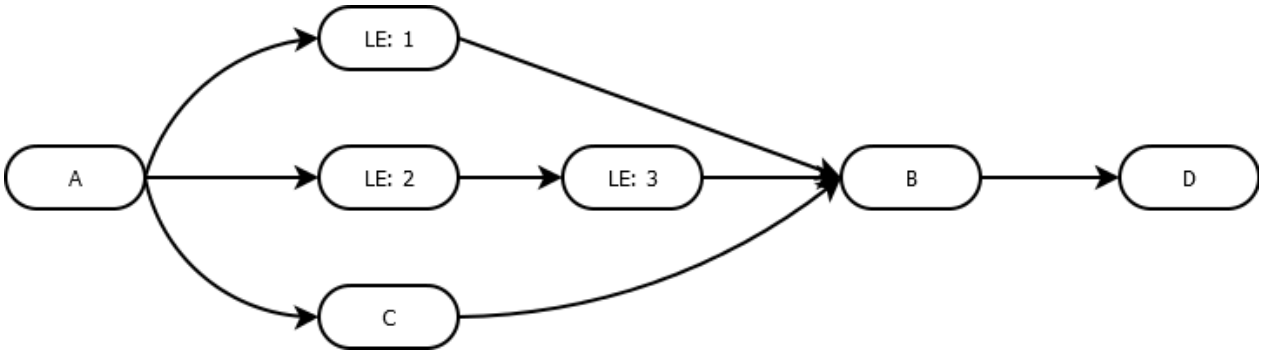
5.37. att. Ciklu aizvietošanas algoritms

5.4.4.5. Aizvietoto ciklu apstrāde

Veicot ciklu aizvietošanu procesu izsaukumu grafā, rodas iespēja apstrādāt aciklisku (ciklus nesaturošu) grafu. Tomēr pēc ciklu aizvietošanas, ir iespējams veikt papildus transformācijas iegūto ciklu apakšgrafu vienkāršošanai. Tas ļauj vienkāršot to turpmāko apstrādi.

Viena no tādām operācijām ir ceļu, kas satur tikai un vienīgi cikla ieejas punktus vai cikla izejas punktus, apstrāde. Hipotēze: ja starp divām grafa virsotnēm `A` un `B` eksistē vairāki ceļi, no kuriem daži satur tikai cikla ieejas punktus vai cikla izejas punktus, tad ir iespējams šos ceļus apvienot vienā, paņemot visus šos punktus, izveidojot jaunu lineāru apakšgrafu (grafu, kas satur dotās virsotnes, kas ir secīgi savienotas viena ar otru) un ievietojot šo apakšgrafu pirms attiecīgās virsotnes `B`. Dotā grafa transformācija ir paredzēta alternatīvo ceļu skaita samazināšanai apstrādājamajā grafā, un ir balstīta uz faktu, ka virsotnes, kas apzīmē cikla ieejas un izejas punktus, nesatur informāciju par procesu izsaukumiem un ir nepieciešamas tikai papildus informācijas uzglabāšanai. Līdz ar to, ja pēc virsotnes A atrodas vairākas šādas virsotnes, to apstrādes secība nav svarīga. 5.38. att. ir parādīts grafa fragments, kas atbilst dotajam nosacījumam. Var redzēt, ka virsotnes A un B ir savienotas ar trīs ceļu palīdzību:

1. `A → LE: 1 → B → D`
2. `A → LE: 2 → LE: 3 → B → D`
3. `A → C → B → D`



5.38. att. Cikla apakšgrafs ar ieejas alternatīvām

Ir nepieciešams vēl minēt to, ka cikla ieejas un izejas punkti var tikt izpildīti tikai vienu reizi – tie ir ciklam atbilstošā apakšgrafa saistības punkti, kas apraksta, no kurienes šajā apakšgrafā var iekļūt, vai arī – kur iziet. Grafa fragments, kas ir parādīts 5.38. att. ir cikla apakšgrafs. Līdz ar to var secināt, ka doto cikla apakšgrafu ir iespējams aprakstīt arī ar šādu ceļu palīdzību:

1. `B → D`, kad tiek izpildīta ieeja jebkurā no dotajiem cikla ieejas punktiem, bet notiek tikai viena iterācija.
2. `B → D → (A → C → B → D)+` , kad tiek izpildīta ieeja jebkurā no dotajiem cikla ieejas punktiem, un notiek vairākas iterācijas.

5.39. att. Cikla apakšgrafa ar ieejas alternatīvām apstrādes rezultāts

5.39. att. ir parādīts piedāvātās aizvietošanas izpildes rezultāts. Analizējot visus iespējamos ceļus šajā grafā, var redzēt, ka divi iepriekš aprakstītie ceļi ir saglabāti. Līdzīgi ir iespējams veikt arī ciklu izejas punktu apstrādi.

5.40. att. ir dots attiecīgā algoritma, kas veic iepriekš aprakstīto aizvietošanu, pseidokods. Dotais algoritms sastāv no vairākām funkcijām, kas tiek izsauktas no divām apstrādes funkcijām – `processLoopEntries()` un `processLoopBreaks()`.

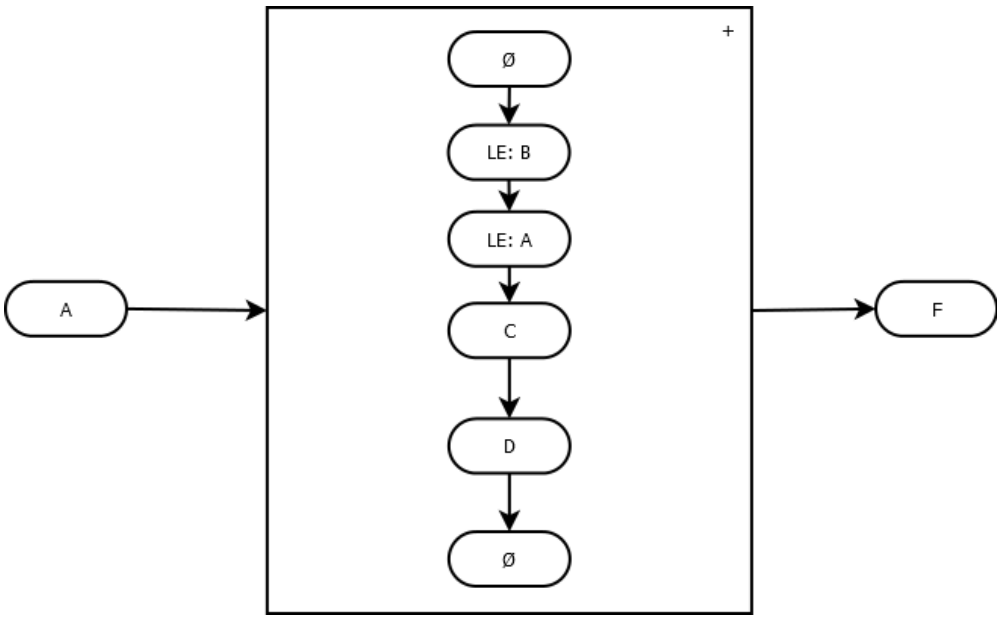
```
function processLoop(graph, vertexKind):
	for s in graph.vertices:
		for f in graph.vertices:
			routes = graph.getAllRoutesBetween(s, f)
			if routes.size > 1:
				if canProcess(routes, vertexKind):
					disjoint(s, f, routes, vertexKind)

function disjoint(s, f, routes, vertexKind):
	kindRoutes = routes find(r such that ∀ v ∈ r: v.kind is vertexKind)
	nonKindRoutes = routes find(
			r such that ∃ v ∈ r: v.kind is not vertexKind)

	proxyState = ∅
	allVs = all vertices from kindRoutes

	source = s
	for wp in vs:
		graph.removeVertex(wp)
		graph.addEdge(source, wp, new Edge())
		source = wp

	graph.addEdge(source, proxyState, new Edge())
	if nonKindRoutes is empty:
		for e in graph.findEdgeSet(s, f):
			graph.removeEdge(e)
	else:
		for route in nonKindRoutes:
			if route is empty:
				for e in graph.findEdgeSet(s, f):
					graph.removeEdge(e)
			else:
				source = proxyState
				for wp in route:
					graph.removeVertex(wp)
					graph.addEdge(source, wp, new Edge())
					source = wp

				graph.addEdge(source, f, new Edge())

function canProcess(routes, vertexKind):
	return routes exists(r such that ∀ v ∈ r: v.kind is vertexKind)

function processLoopEntries(graph):
	processLoop(graph, LoopEntry)

function processLoopBreaks(graph):
	processLoop(graph, LoopBreak)
```

5.40. att. Ciklu ieejas un izejas punktu apstrādes algoritms

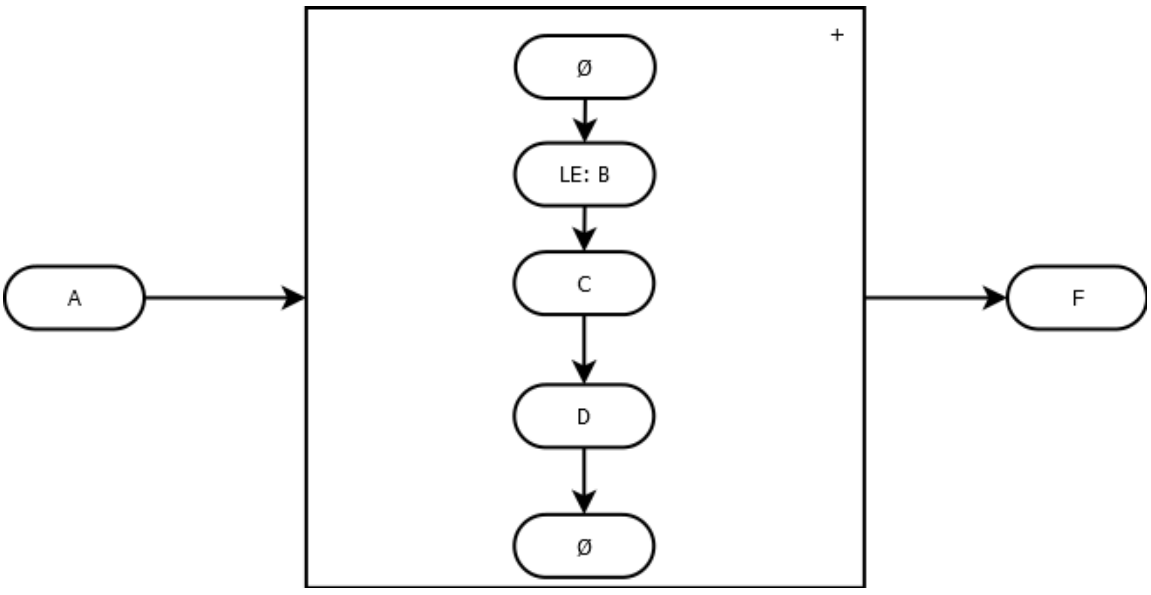
Veicot ciklu ieejas un izejas punktu analīzi, var arī redzēt, ka tos ir iespējams pārkārtot apstrādei nepieciešamā secībā gadījumos, kad vairāki šāda veida punkti seko viens otram. Piemēram, 5.39. att. redzamie cikla ieejas punkti `LE: 1`, `LE: 2` un `LE: 3` var tikt izpildīti jebkurā secībā – tas neietekmē procesu izsaukumu secību, bet varētu būt nepieciešams turpmākai apstrādes vienkāršošanai.

5.41. att. ir dots grafa fragments, kas satur cikla apakšgrafu. Var redzēt, ka šo ciklu ir iespējams uzsākt no divām virsotnēm – `A` un `B`, ko definē cikla ieejas punkti `LE: A` un `LE: B`.

Pie tam, `LE: B` atrodas pirms `LE: A`. Pārkārtojot šos ieejas punktus, rodas iespēja cikla ieejas punktu `LE: A` vispār likvidēt – jo tik un tā pēc virsotnes `A` apmeklēšanas nākamā darbība būs ieeja ciklā, kas sakrīt ar likvidēto virsotni.



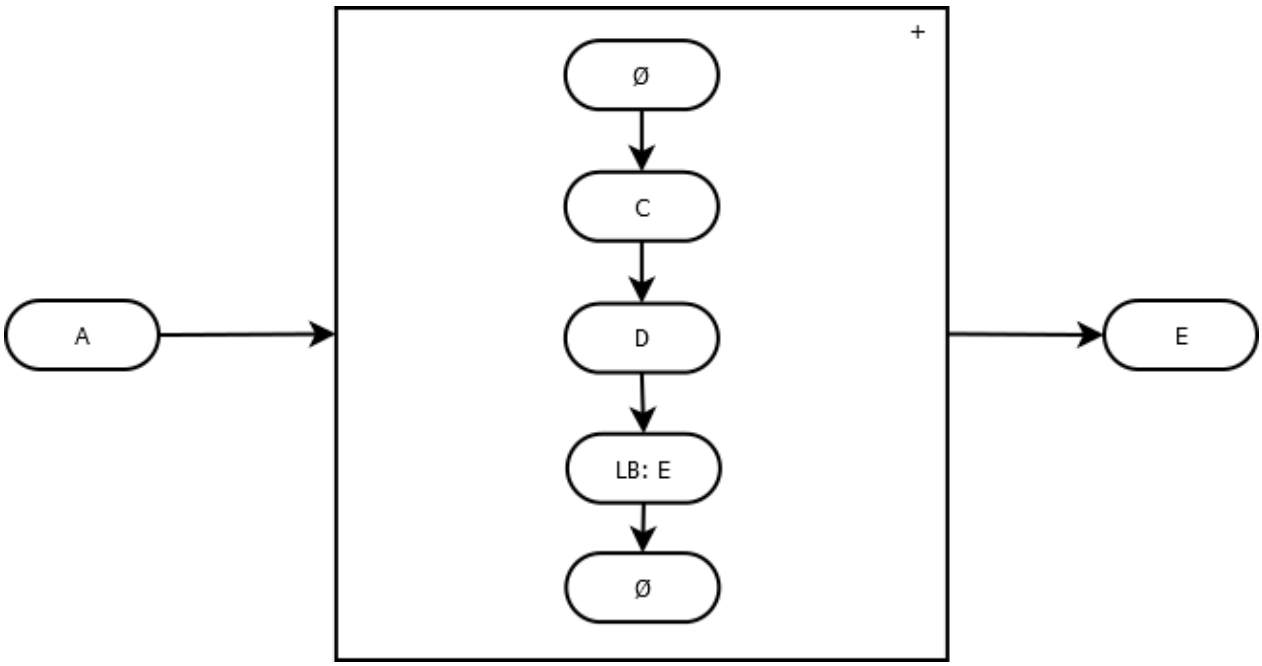
5.41. att. Cikla ieejas punkti pirms pārkārtošanas



5.42. att. Cikla ieejas punkti pēc pārkārtošanas

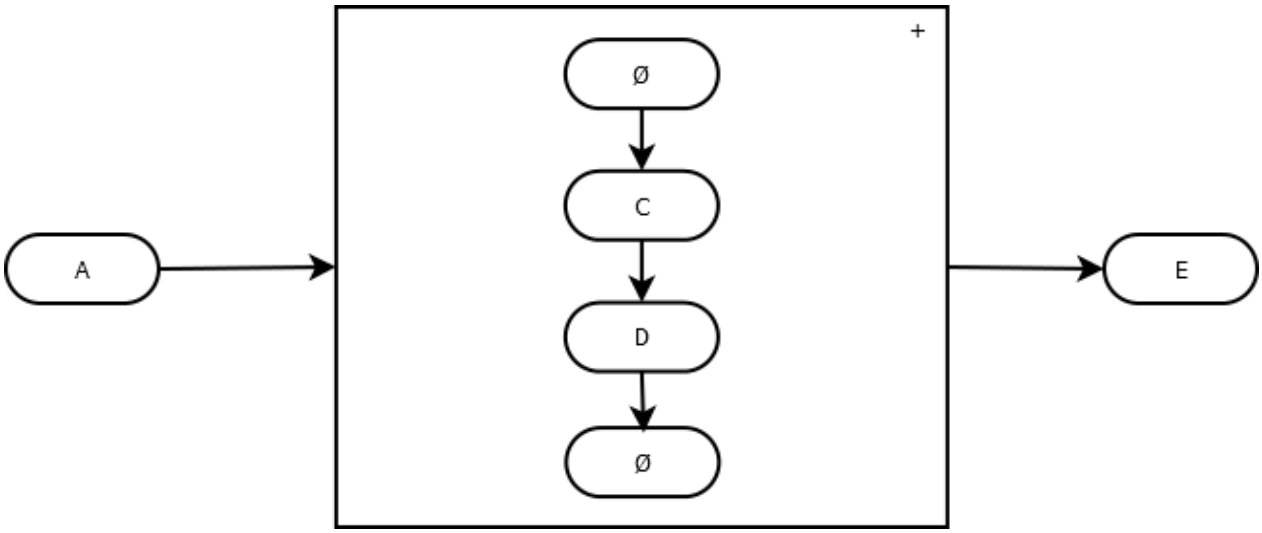
5.42. att. ir parādīta situācija pēc cikla ieejas punktu pārkārtošanas un ieejas punkta `LE: A` likvidēšanas. Var redzēt, ka iespējamā grafa virsotņu apmeklēšanas secība nav mainījusies. Līdz ar to ir iespējams veikt šādas operācijas.

Ir iespējams arī veikt vairākas operācijas ar cikla izejas punktiem, vienkāršojot un likvidējot tos, ja tas ir iespējams. Pirmā šāda operācija ir cikla noslēdzošo izejas punktu apstrāde. Ja cikla beigās atrodas izejas punkts, kura mērķa virsotne atrodas uzreiz aiz cikla, tad to ir iespējams likvidēt. 5.43. att. var redzēt grafa fragmentu pirms šādu izejas punktu likvidēšanas.



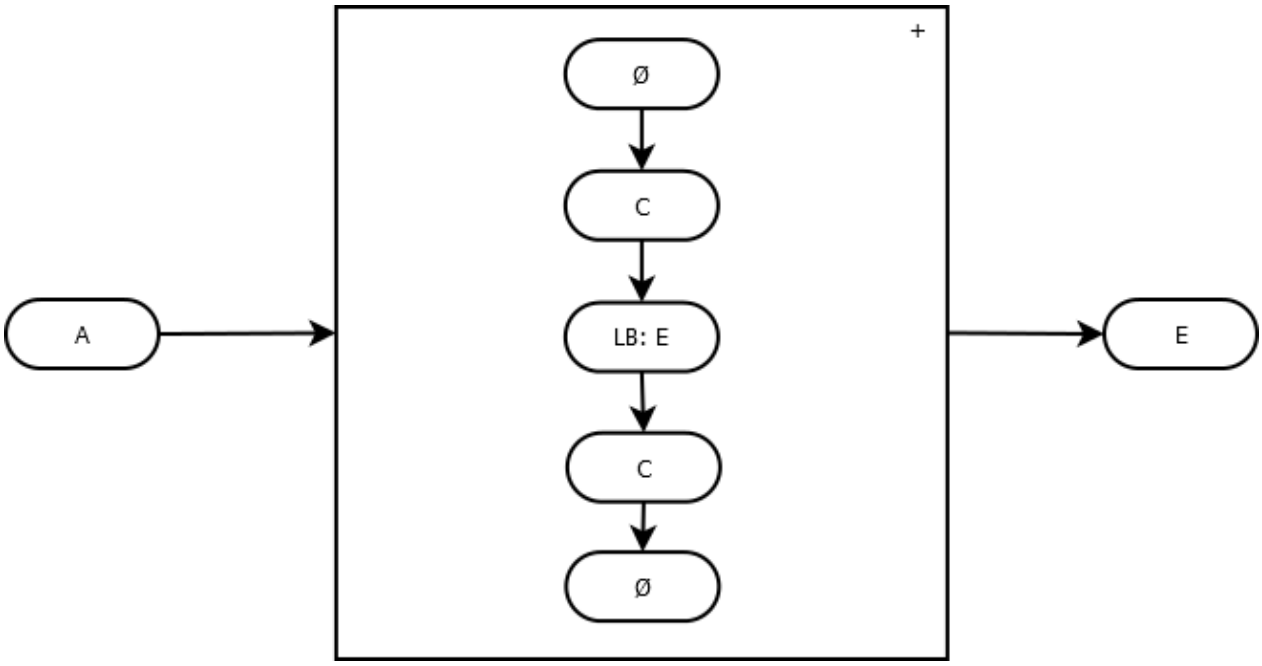
5.43. att. Likvidējamais cikla izejas punkts

Šeit pirms cikla apakšgrafa beigām eksistē izejas virsotne, kas norāda uz galvenā grafa virsotni, kas atrodas uzreiz pēc cikla apakšgrafa. Var redzēt, ka pastāv divas iespējas iziet no dotā cikla – vai nu iterācijām beidzoties, vai nu apmeklējot virsotni `LB: E`, kas norāda uz cikla izeju. Ir iespējams definēt doto situāciju kā `A → (C → D)+ → E`. Tātad, var redzēt, ka dotā izejas punkta virsotne nemaina darbību secību, un to ir iespējams likvidēt. Situācija pēc dotā punkta likvidēšanas ir parādīta 5.44. att., un iespējamā virsotņu apmeklēšanas secība joprojām ir `A → (C → D)+ → E`. Tātad, šāda izejas punktu virsotņu likvidēšanas operācija ir atļauta.



5.44. att. Cikls pēc izejas punkta likvidēšanas

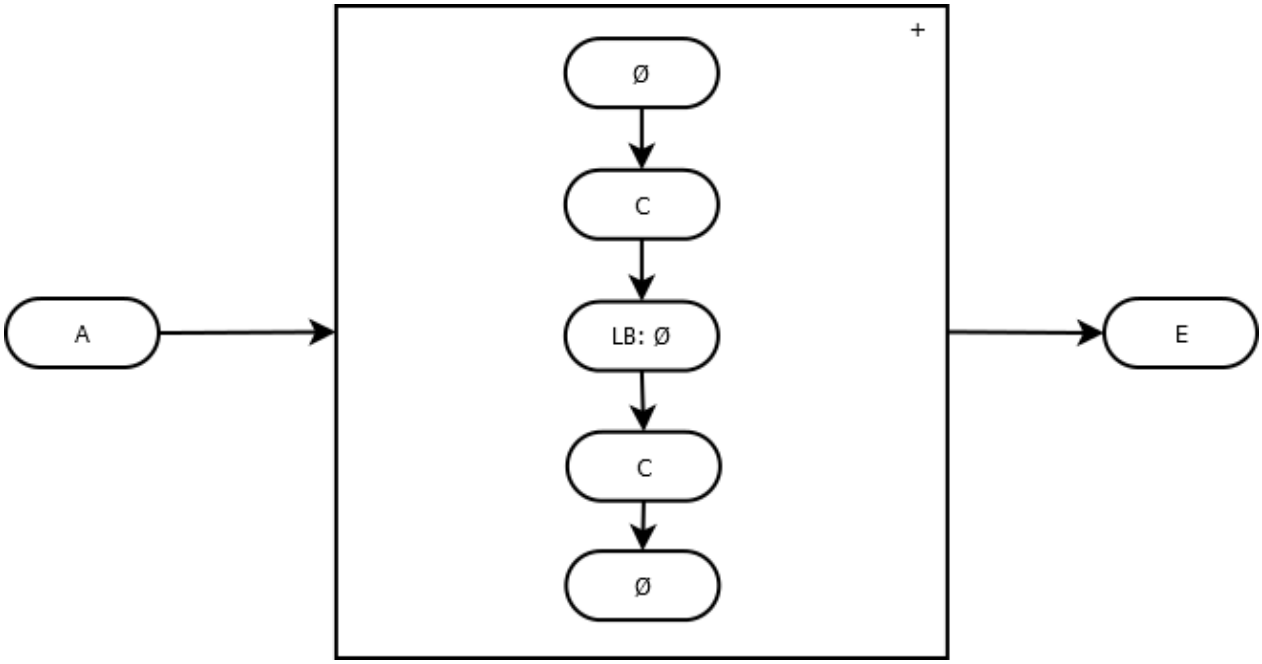
Vēl viena operācija, ko ir iespējams veikt ar cikla izejas punktiem, ir to mērķa virsotņu apstrāde. 5.45. att. ir parādīts cikla apakšgrafs ar izejas punktu, kas ir vērsts uz ciklam sekojošo virsotni. Hipotēze: šādu izejas punktu var pārrakstīt, izņemot no tā definīcijas mērķa virsotni.



5.45. att. Cikls ar izejas punktu

Analizējot iespējamos ceļus šajā grafa fragmentā, var redzēt, ka tie ir pierakstāmi šādā formā: `A → (C → D?)+ → E`. Izņemot no izejas punkta `LB: E` informāciju par izejas mērķa virsotni, tiek iegūts grafs, kas ir parādīts 5.46. att. Šeit, izejas punkts vairs nesatur informāciju par mērķa virsotni, bet vienkārši norāda, ka ir iespējams pārtraukt ciklu. Tātad, ir iespējams

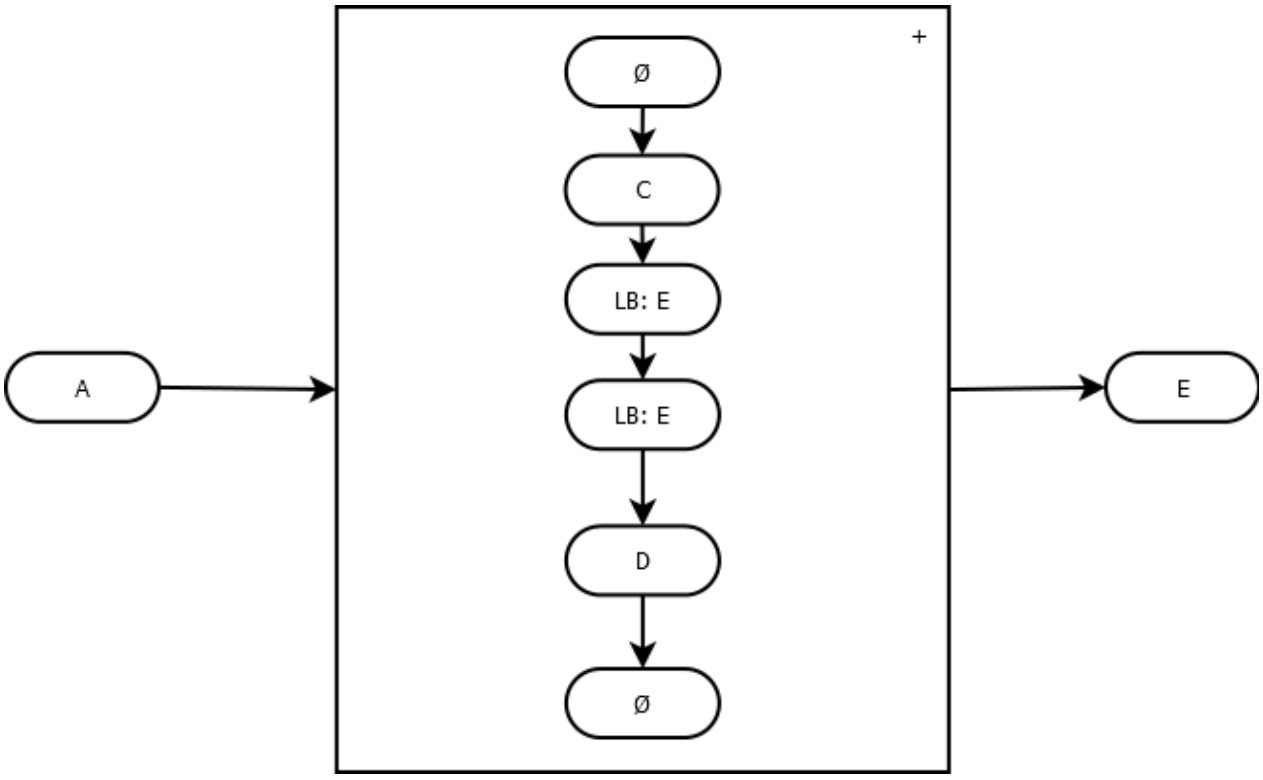
veikt šādu aizvietošanu. Var redzēt, ka pati par sevi aizvietošana nemaina grafa struktūru, tomēr tiek vienkāršots cikla izejas punkts, kas ir noderīgi turpmākai apstrādei.



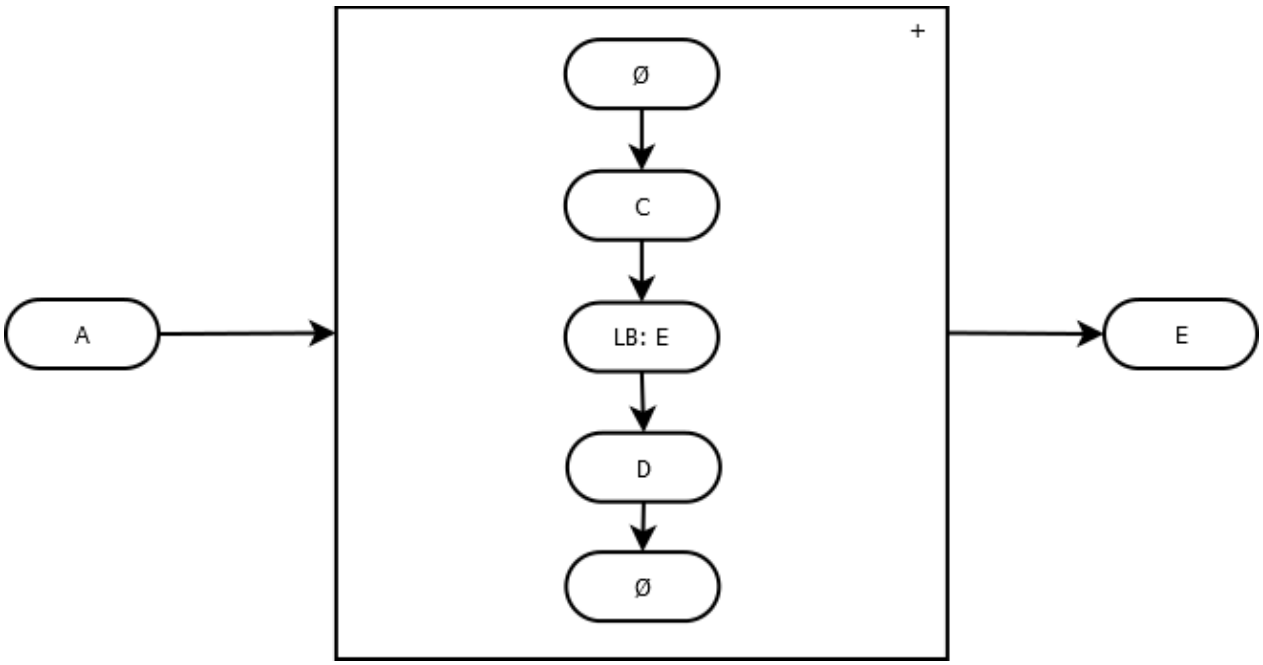
5.46. att. Cikls pēc izejas punkta apstrādes

Veicot vairākas operācijas ar cikla apakšgrafā esošām virsotnēm, ir iespējama arī situācija, kad vairāki cikla izejas punkti ar vienādu mērķi seko viens otrām. Šādā gadījumā ir iespējams likvidēt visus šī izejas punkta dublikātus, atstājot tikai vienu izejas punktu. Lai pārliecinātos, ka šāda aizvietošana ir iespējama, promocijas darba autors piedāvā apskatīt grafa fragmentu, kas ir parādīts 5.47. att. Šeit divi vienādi cikla izejas punkti – `LE: E` seko viens otrām. Analizējot iespējamos ceļus šajā grafā, var redzēt, ka tos ir iespējams definēt kā `A → (C → D?)+ → E`. Apvienojot abas šīs virsotnes, tiek iegūta situācija, kas ir parādīta 5.48. att. Analizējot jaunā grafā esošos iespējamos ceļus, var redzēt, ka tie nav izmainījušies salīdzinājumā ar situāciju pirms apvienošanas. Tātad, šādu apstrādi ir iespējams veikt.

Iepriekšējā rindkopā apskatītā operācija ir pēdējā no cikla ieejas un izejas punktu apstrādes operācijām, ko definē šī darba autors. Tomēr, veicot ciklu apstrādi, ir iespējams pielietot vēl vienu operāciju, ko autors ir nosaucis par cikla nosacījuma definēšanu.



5.47. att. Cikls ar izejas punktu dublikātiem

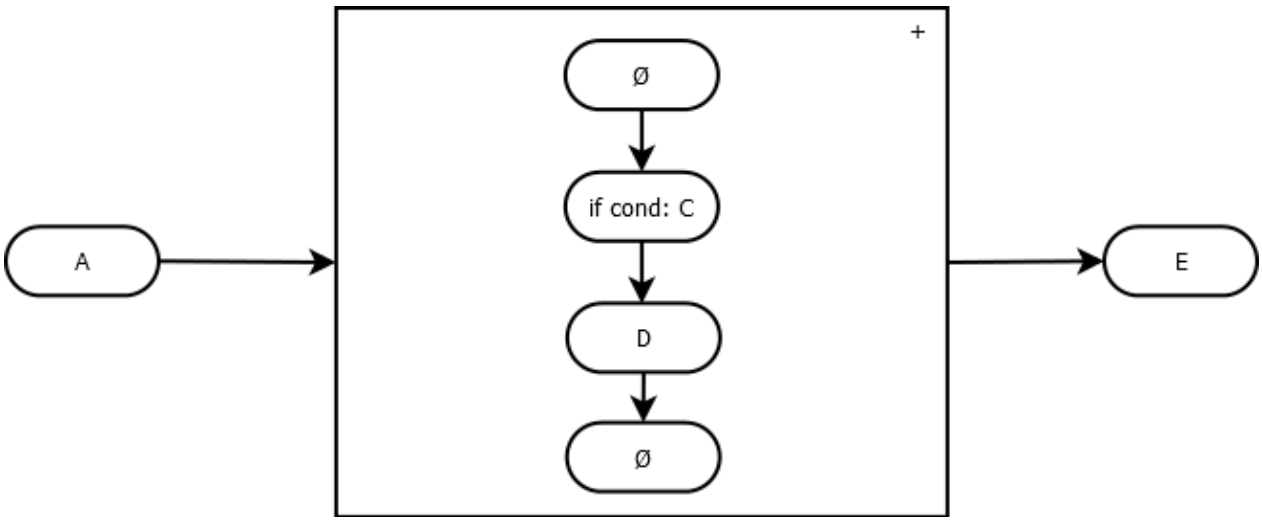


5.48. att. Cikls bez izejas punktu dublikātiem

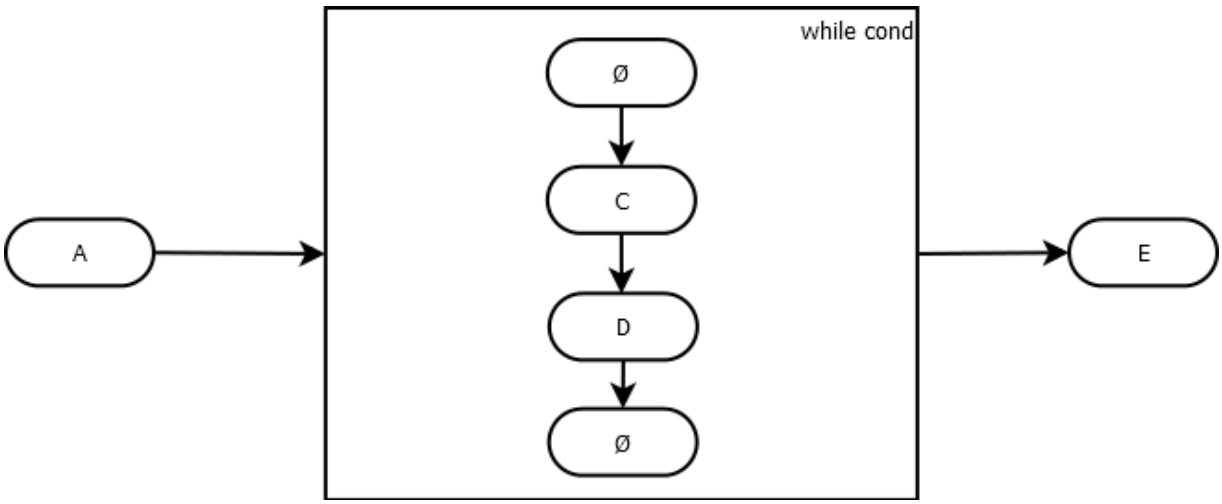
Hipotēze: ja cikla apakšgrafa pirmajam procesam ir definēts nosacījums, tad šo nosacījumu ir iespējams definēt kā dotā cikla nosacījumu.

5.49. att. ir parādīts grafs, kas sākas ar virsotni `C`, kurai ir definēts izpildes nosacījums `cond`. Gadījumā, kad nosacījums netiek izpildīts, virsotne `C` arī neizpildās, tātad arī nav

iespējams veikt pārējo virsotņu, kas atrodas aiz tās, apmeklēšanu – vienīgais ceļš, kas ieiet virsotnē `D`, prasa virsotnes `C` izpildi. Tā kā virsotne `C` ir pirmā virsotne dotā cikla apakšgrafā, var teikt, ka nosacījumam neizpildoties, neizpildās arī viss cikls. Tātad, šo ciklu var pierakstīt kā `A → (if cond: (C → D))+ → E`. Tas atbilst ciklam ar priekšnosacījumu, kas savukārt nozīmē, ka ir iespējams doto grafa fragmentu aizvietot tā, kā ir parādīts 5.50. att.



5.49. att. Cikls, kur pirmā virsotne satur izpildes nosacījumu



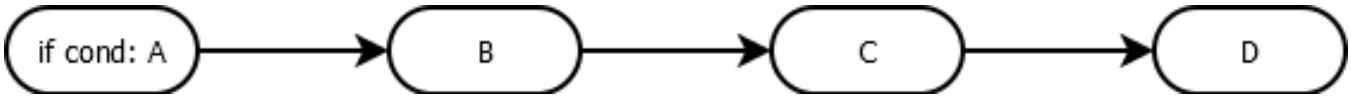
5.50. att. Cikls ar priekšnosacījumu

Var redzēt, ka aizvietošanas rezultātā oriģinālā virsotņu apmeklēšanas secība, `A → (if cond: (C → D))+ → E`, ir saglabāta. Tātad, šāda aizvietošanas operācija ir atļauta.

Šādu aizvietošanu ir iespējams pielietot arī citos gadījumos. Hipotēze: ja eksistē darbību secība, kas sākas ar virsotni, kurai ir definēts izpildes nosacījums, tad šim nosacījumam neizpildoties, netiek izpildīta arī visa secība. Tātad, šīs virsotnes nosacījumu ir iespējams definēt kā visas virsotņu secības izpildes nosacījumu. 5.51. att. ir parādīts grafa fragments, kas

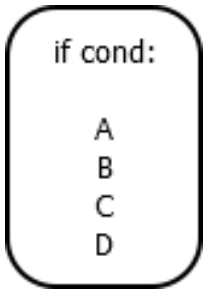
satur vairāku virsotņu secību. Pirmajai virsotnei – `A` – ir definēts izpildes nosacījums `cond`. Analizējot iespējamos ceļus šajā grafa fragmentā, ir iespējams definēt divus variantus:

1. Neizpildoties nosacījumam, netiek apmeklēta neviena virsotne.
2. `A → B → C → D`, nosacījumam izpildoties.



5.51. att. Secība ar nosacījumu pirms aizvietošanas

Šīs divas situācijas ir iespējams apvienot, definējot kā `(if cond: (A → B → C → D))`. Līdz ar to, šo grafa fragmentu var aizvietot ar fragmentu, kas ir parādīts 5.52. att.



5.52. att. Secība ar nosacījumu pēc aizvietošanas

Šādas aizvietošanas var tikt izmantotas atsevišķu virsotņu secību, kas izpildās pēc nosacījuma, definēšanai.

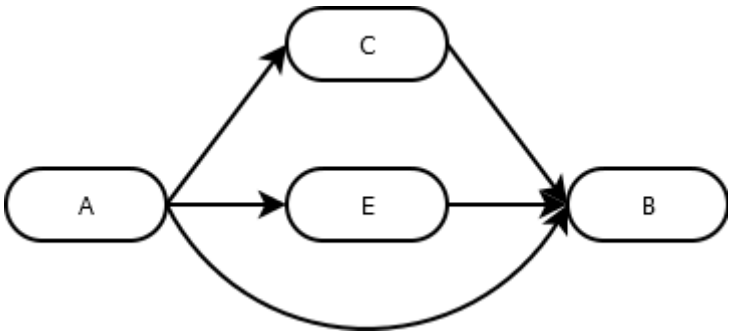
5.4.4.6. Disjunkciju definēšana

Vēl viena procesu izsaukumu grafa apstrādes operācija, ko definē promocijas darba autors, ir disjunkciju definēšana.

Disjunkcija tiek definēta, kad starp divām virsotnēm eksistē vairāki ceļi. Pie tam visiem šiem ceļiem ir jāsatur dažādas virsotnes. Šādas situācijas piemērs ir parādīts 5.53. att. Piemērā starp grafa virsotnēm `A` un `B` eksistē trīs dažādi ceļi. Tie ir:

1. `A → B`
2. `A → C → B`
3. `A → E → B`

Var redzēt, ka viens no šiem ceļiem ir "tukšs", jeb nesatur nekādas virsotnes. Šādus ceļus ir iespējams apvienot disjunkcijas ietvaros kā `(C || E || Ø)`, kur `||` nozīmē "vai". Rezultātā tiek iegūts grafs, kas ir parādīts 5.54. att. Var redzēt, ka visi trīs iespējamie ceļi ir saglabāti. Tātad, šāda aizvietošana ir atļauta. Attiecīga algoritma pseidokods ir parādīts 5.55. att.



5.53. att. Disjunkcijas situācija grafā

5.54. att. Disjunkcijas apstrādes rezultāts

```
for s in graph.vertices:
	for f in graph.vertices:
		routes = graph.getAllRoutesBetween(s, f)
		if routes.size > 1:
			if canProcess(routes):
				disjoint(graph, s, f, routes)

function canProcess(routes):
	return ∀ r ∈ routes: is different

function disjoint(graph, s, f, routes):
	replacement = new Disjoint(routes)
	hasEmptyRoute = false

	for route in routes:
		for v in route:
			graph.removeVertex(v)
		if route is empty:
			hasEmptyRoute = true

	if hasEmptyRoute:
		toRemove = graph.findEdgeSet(s, f)
		for e in toRemove:
			graph.removeEdge(e)

	graph.addEdge(s, replacement, new Edge());
	graph.addEdge(replacement, f, new Edge());
```

5.55. att. Disjunkcijas definēšanas algoritms

5.4.4.7. Grafa minimizēšanas algoritmu pielietošana

Analizējot visus promocijas darba autora definētos algoritmus, var redzēt, ka to sekmīgai pielietošanai ir svarīgi definēt pareizu secību, kurā šie algoritmi mēģina apstrādāt grafu. Piemēram, Džonsona algoritms [53] nevar tikt izmantots, ja grafs satur cilpas. Tapāt, disjunkcijas definēšanas algoritmu būtu vēlams pielietot gadījumos, kad visi iespējamie ceļi starp grafa virsotņu pāriem ir minimāli – tas ļauj definēt disjunkcijas, kuru sastāvdaļas satur minimālo virsotņu skaitu.

Līdz ar to autors piedāvā visus definētos algoritmus sadalīt vairākās grupās un veikt grafa apstrādi iteratīvi – pielietojot algoritmus no katras grupas, kamēr tas ir iespējams. Gadījumā, kad neviens algoritms no algoritmu grupas vairs nespēj veikt grafa modifikācijas, tiek pielietota nākamā algoritmu grupa. Process tiek atkārtots iteratīvi, kamēr aizvietošanas ir iespējamas.

Šo procesu ir iespējams aprakstīt ar pseidokodu, kas ir parādīts 5.56. att. Pseidokods ietver trīs funkcijas – `applyAlgorithmSet()` pielieto visus algoritmus no saņemtās algoritmu grupas pēc kārtas, pārtraucot savu darbu gadījumā, kad neviena algoritmu grupa vairs netika pielietota veiksmīgi. Otrā funkcija `applyAlgorithmFromSet()` savukārt pielieto dotos

algoritmus no vienas algoritmu grupas līdz visi algoritmi no dotās grupas vairs nespēj veikt grafa minimizēšanu. Tātad grafa minimizēšanas procesu var aprakstīt kā iteratīvu funkcijas `applyAlgorithmSet()` pielietošanu, kamēr tas ir iespējams. Šo iteratīvo procesu veic pēdējā funkcija – `minimize()`.

Tātad, izstrādāto algoritmu pielietošanai ir nepieciešams definēt algoritmu grupas. Pirmā algoritmu grupa ietver tikai vienu algoritmu – cilpu aizvietošanu. Cilpu aizvietošanas algoritms tika izvēlēts tāpēc, ka vairāki no piedāvātajiem grafu apstrādes algoritmiem nespēj apstrādāt cikliskus grafus, bet Džonsona algoritms [53] nevar tikt izmantots, ja grafs satur cilpas. Līdz ar to ir nepieciešams sākumā izvairīties no visiem cikliem, pārveidojot tos par jauniem grafiem, bet ciklu aizvietošanas nolūkiem vispirms ir nepieciešams izvairīties no cilpām.

```
function applyAlgorithmFromSet(graph, algorithmSet):
	atLeastOnce = false
	minimized = true

	while minimized:
		minimized = false
		for algorithm in algorithmSet:
			if applyAlgorithm(graph, algorithm):
				minimized = true
				atLeastOnce = true
				break

	return atLeastOnce

function applyAlgorithmSet(graph, algorithmSets):
	atLeastOnce = false
	minimized = true

	while minimized:
		minimized = false
		for algorithmSet in algorithmSets:
			if applyAlgorithmFromSet(graph, algorithmSet):
				minimized = true
				atLeastOnce = true
				break

	return atLeastOnce

function minimize(graph, algorithmSets):
	while applyAlgorithmSet(graph, algorithmSets):
		// Turpināt apstrādi
```

5.56. att. Algoritmu pielietošanas funkcijas

Nākamā algoritmu grupa satur ciklu aizvietošanas algoritmu. Pēc šīs grupas pielietošanas tiek iegūts grafs, kas nesatur ciklus. Līdz ar to tā apstrāde kļūst vienkāršāka.

Pēc cilpu un ciklu aizvietošanas ir iespējams veikt to ieejas punktu apstrādi, pārkārtojot un vienkāršojot tos. Līdz ar to trešā algoritmu grupa satur divus algoritmus: ieejas punktu pārkārtošanas un ieejas alternatīvu apstrādes algoritmu (5.40. att.).

Nākamā algoritmu grupa satur vienu algoritmu – loku dublikātu likvidēšanu. Tā pielietošana nodrošina to, ka apstrādājamais grafs nesatur liekus papildus lokus, kas savukārt var traucēt pārējo algoritmu pielietošanu.

Nākamā algoritmu grupa ietver divus algoritmus – secīgu virsotņu apvienošanu un cikla izejas punktu alternatīvu apstrādes algoritmu (5.40. att.). Šo divu algoritmu pielietošana ļauj minimizēt virsotņu skaitu apstrādājamajā grafā, kā arī vienkāršot cikla izejas alternatīvas. Līdz ar to grafs, kas nonāk nākamajos apstrādes soļos, saturēs tikai apvienotas virsotņu secības.

Pēc šo algoritmu pielietošanas ir iespējams pielietot disjunkciju definēšanas algoritmu, kas ļauj dažādus ceļus starp grafa virsotnēm apvienot vienā. Šajā brīdī apvienojamie ceļi (jeb disjunkcijas alternatīvas) satur minimālo virsotņu skaitu. Līdz ar to nākamā algoritmu grupa satur vienu algoritmu – disjunkciju definēšanas algoritmu.

5.3. tabula

Algoritmu grupas

Pielietošanas secība	Algoritmi
1	Cilpu aizvietošana
2	Ciklu aizvietošana
3	Cikla ieejas punktu pārkārtošana Cikla ieejas alternatīvu apstrāde
4	Loku dublikātu likvidēšana
5	Secīga virsotņu apvienošana Cikla izejas alternatīvu apstrāde
6	Disjunkciju definēšana
7	Cikla ieejas punktu likvidēšana Cikla izejas mērķu vienkāršošana Cikla izejas punktu likvidēšana Cikla nosacījumu definēšana Secību izpildes nosacījumu definēšana

Pēdējā algoritmu grupa satur visus pārējos algoritmus, kas ir cikla ieejas punktu likvidēšanas algoritms (skat. 5.41. att. un 5.42. att.), cikla izejas mērķu vienkāršošanas algoritms (skat. 5.45. att. un 5.46. att.), lieko cikla izejas punktu likvidēšanas algoritmus (5.43.

att., 5.47. att. un 5.48. att.), cikla nosacījumu definēšanas algoritmu (5.49. att. un 5.50. att.), kā arī secību izpildes nosacījumu definēšanas algoritmu (5.51. att. un 5.52. att.). Informācija par definētajām algoritmu grupām ir apkopota 5.3. tabulā.

Dotās algoritmu grupas promocijas darba autors ir definējis gan analizējot darbības, kuras veic katrs no definētajiem algoritmiem, gan arī veicot vairākus eksperimentus grupu noteikšanai. Algoritmu pielietošanas rezultātā procesu izsaukumu grafs tiek samazināts, jo virsotnes tiek vai nu apvienotas kopā, vai arī izņemtas no tā. Pēc autora hipotēzes šādas minimizēšanas rezultātā ir jābūt iespējai jebkuru procesu izsaukumu grafu transformēt līdz brīdim, kad tas sastāvēs no vienas virsotnes, kuras saturs apraksta visus procesu izsaukumu noteikumus, kas sevī ietver gan informāciju par procesu izsaukumu secību, gan arī informāciju par nosacījumiem procesu vai arī procesu secību izpildei.

Lai pārbaudītu šo hipotēzi, autors ir veicis 10 000 eksperimentus ar nejauši uzģenerētiem grafiem, kas ietvēra dažādu virsotņu daudzumu – no 10 līdz 100. Grafu ģenerēšanas laikā tika definēti vairāki nosacījumi:

- Grafam bija jāsatur vismaz viena cilpa.
- Grafam bija jāsatur vismaz viens cikls.
- Vismaz 10% no grafa virsotnēm bija jābūt savienotām ar vairākām citām virsotnēm – gan caur ieejošiem, gan caur izejošiem lokiem.
- Grafam bija jāsatur vismaz viena sākuma un vismaz viena beigu virsotne. Pie tam, katrai no beigu virsotnēm bija jābūt sasniedzamai no vismaz vienas sākumu virsotnes. Tapāt, starp jebkuru sākuma un beigu virsotni ir jāeksistē vismaz vienam ceļam.
- Grafam bija jābūt vāji saistītam.

Eksperimentu veikšanas rezultātā tika definētas vairākas korekcijas algoritmu pielietošanas secībai, rezultātā iegūstot 5.3. tabulā parādītās algoritmu grupas ar noteiktu pielietošanas secību. Gadījumā, ja kāds no uzģenerētajiem grafiem nevarēja tikt minimizēts līdz vienai virsotnei, tika veiktas attiecīgas korekcijas, un eksperiments tika atkārtots. Rezultātā, promocijas darba autoram neizdevās uzģenerēt tādus grafus, kurus nevar minimizēt (visi grafi bija minimizējami). Līdz ar to tiek pieņemts, ka hipotēze, ka definētie algoritmi ļauj minimizēt grafu, tika pierādīta.

5.5. Minimizēta procesu izsaukumu grafa apstrāde

Procesu izsaukumu grafa minimizēšanas rezultātā tiek iegūta virsotne, kas satur informāciju par visu minimizēto procesu izsaukumu grafu. Var redzēt, ka šī informācija jau ir definēta secības veidā – jo virsotnes saturs ir virsotņu apvienošanas rezultāts, kur visi grafā esošie ceļi faktiski tiek pārrakstīti izmantojot 5.2. tabulā definētos pierakstus.

Pie tam, katram procesam, kas tika definēts avota biznesa procesu modelī, jau ir definēta attiecīgās metodes signatūra. Lai gan, šajā brīdī vēl nav zināms, kurai no klasēm piederēs attiecīgā metode, ir zināms, kādi ir šīs metodes parametri un atgriežamais rezultāts.

Tātad, ir nepieciešams iegūto procesu izsaukumu secību pārveidot atbilstoši formulām (5.23) un (5.24). Dotā secība satur virsotņu apvienošanas rezultātus, kas ir definēti 5.2. tabulā. Ir nepieciešams definēt likumus attiecīgo elementu pārveidošanai. Tomēr ir būtiski veikt arī avota modeļa analīzi papildu informācijas izgūšanai un definēšanai. Ir jādefinē mainīgie, kas ir nepieciešami attiecīgā biznesa procesa izpildei. Šie mainīgie var tikt iegūti četros veidos:

1. No datu plūsmām biznesa procesu diagrammā – tā kā katra datu plūsma var saturēt datus, kas ir definēti konceptu vai primitīvu datu tipu veidā, ir iespējams apvienot informāciju no visām datu plūsmām, definējot nepieciešamos mainīgos.
2. No procesu vai procesu secību izpildes nosacījumiem – aplūkojot formulas (5.19)-(5.21), var redzēt, ka izpildes nosacījumu pārbaudei ir nepieciešams papildu mainīgais, kas tiek pārbaudīts.
3. Gadījumos, kad eksistē cikla ieejas un cikla izejas punkti (5.2. tabula), ir iespējams, ka pārejas no/uz šiem punktiem ne vienmēr var tikt izpildītas. Ir iespējams gan, ka šajos gadījumos izpildes nosacījumi ir definēti avota modelī, gan arī, ka nosacījumi nav definēti. Abos gadījumos, promocijas darba autors piedāvā definēt attiecīgos mainīgos.
4. Gadījumos, kad eksistē cikls, ir iespējams, ka tam ir definēts izpildes nosacījums. Tad tiek izpildīts šī saraksta 2. punkts. Citādi, ir nepieciešams definēt arī cikla izpildes nosacījuma mainīgos.

Tā kā katru mainīgo var raksturot ar tā nosaukumu un datu tipu, ir nepieciešams šo informāciju saglabāt, lai turpmāk izmantotu koda ģenerēšanas gaitā. Šī informācija var tikt izmantota dažādos veidos atkarībā no izvēlētās mērķa programmēšanas valodas un tās sintakses, kā arī pieņemtajiem mainīgo nosaukumu formātiem – tā, piemērām, programmēšanas

valodā Java [50] mainīgo nosaukumi parasti tiek definēti tā saucamā *camel-case* notācijā (piemēram, `myVariable`), kamēr programmēšanas valodā C [95] tiek pieņemta cita mainīgo vārdu notācija (piemēram, `my_variable`). Līdz ar to, veicot minimizēta procesu izsaukumu grafa pārveidošanu atbilstoši formulām (5.23) un (5.24), mainīgo nosaukumi netiek definēti – tiek saglabāta nepieciešamā informācija par mainīgā iegūšanas avotu.

5.57. att. ir parādīts algoritma, kas iegūst informāciju par nepieciešamajiem mainīgajiem no datu plūsmām, pseidokods. Var redzēt, ka definējot katru mainīgo, tiek saglabāta informācija par tā tipu, kā arī par attiecīgo datu plūsmu un tās parametru. Pie tam, katram mainīgajam tiek piešķirts unikāls identifikators, kas tiks izmantots turpmāk koda ģenerēšanas nolūkiem.

```
variableMap = {}
varIdx = 0
for d in processDiagram.dataFlows:
      for p in d.parameters:
            key = (d, p)
            variableMap[key] = (p.type, varIdx++)
```

5.57. att. Informācijas par mainīgajiem iegūšana no datu plūsmām

5.58. att. ir parādīts algoritms, kas analizē procesu un procesu secību izpildes nosacījumus, definējot nepieciešamos mainīgos. Algoritms pēc savas būtības sastāv no vienas rekursīvi izsaucamas funkcijas `checkSequence`, kas pārbauda katru no analizējamās secības elementiem. Gadījumos, kad attiecīgajam elementam ir definēts izpildes nosacījums, tiek definēts attiecīgais `boolean` tipa mainīgais. Gadījumos, kad attiecīgais elements satur citus elementus, tiek veikta to analīze. Analīze sākas ar minimizētā grafa vienīgās virsotnes saturu.

```
function checkSequence(seq):
      for el in seq:
            if el.hasGuard:
                  variableMap[el] = (boolean, varIdx++)

            if el.hasChildren:
                  checkSequence(el.children)

checkSequence(minimizedProcessGraph.resultingSeq)
```

5.58. att. Informācijas par mainīgajiem iegūšana no izpildes nosacījumiem

Līdzīgā veidā darbojas arī algoritms, kas veic cikla ieejas un izejas punktu analīzi, un ir parādīts 5.59. att. Algoritms rekursīvi analizē visus dotās secības elementus, pārbaudot, vai

attiecīgais elements ir cikla ieejas vai izejas punkts. Gadījumos, kad tas tā ir, tiek pievienots jauns `boolean` tipa mainīgais. Analīze sākas ar minimizētā grafa vienīgās virsotnes saturu.

5.60. att. ir parādīts pseidokods pēdējam algoritmam, kas veic ciklu nosacījumu izpildes pārbaudi mainīgo definēšanai. Tas ir līdzīgs iepriekš aprakstītajiem algoritmiem – tiek analizēti visi padotās secības elementi, pēc nepieciešamības apstrādājot arī šo elementu saturu. Analīze sākas ar minimizētā grafa vienīgās virsotnes saturu.

```
function checkLoopEntriesAndBreaks(seq):
       for el in seq:
              if el is LoopEntry:
                     variableMap[el] = (boolean, varIdx++)

              if el is LoopBreak:
                     variableMap[el] = (boolean, varIdx++)

              if el.hasChildren:
                     checkLoopEntriesAndBreaks(el.children)

checkLoopEntriesAndBreaks(minimizedProcessGraph.resultingSeq)
```

5.59. att. Informācijas par mainīgajiem iegūšana no ciklu ieejas un izejas punktiem

```
function checkLoops(seq):
       for el in seq:
              if el is Loop:
                     if not el.hasGuard:
                            variableMap[el] = (boolean, varIdx++)

              if el.hasChildren:
                     checkLoops(el.children)

checkLoops(minimizedProcessGraph.resultingSeq)
```

5.60. att. Informācijas par mainīgajiem iegūšana no ciklu izpildes nosacījumiem

Parādītie mainīgo definēšanas algoritmi ļauj pārveidot informāciju no biznesa procesu modeļa formulā (5.10). Faktiski, heštabula `variableMap` pēc to izpildes satur visu nepieciešamo mainīgo datu tipus un informāciju, kuru izmantojot, ir iespējams ģenerēt to nosaukumus. Mainīgo turpmāka ģenerēšana pēc savas būtības ir šīs heštabulas apstaigāšana, pārveidojot attiecīgos elementus par attiecīgiem mainīgajiem.

Nākamais ģenerējamo datu struktūras elements, ko var iegūt no eksistējošās informācijas, ir iezīme (formula (5.9)). Iezīmes ļauj realizēt zarošanos ģenerējamajā programmatūras kodā, līdz ar to viens no to iespējamajiem iegūšanas veidiem ir mainīgo heštabulas analīze – tā satur nosacījuma mainīgos, kas ir nepieciešami procesu vai procesu secību izpildei, kas savukārt nozīmē iespējamo zarošanos. Tomēr, šī informācija nav pietiekama visu nepieciešamo iezīmju definēšanai.

Hipotēze: ja procesu secību izpildei ir definēts nosacījums, tad ir nepieciešams definēt divas iezīmes – vienu šīs secības sākumā, otru – beigās. Lai pieradītu šo hipotēzi, promocijas darba autors piedāvā apskatīt procesu secību, kurai ir definēts nosacījums (skat. formulu (5.25)):

$$A \rightarrow (if\ cond: (B \rightarrow C)) \rightarrow D \tag{5.25}$$

Šeit pēc procesa `A` izpildes tiek pārbaudīts nosacījums `cond`, kuram izpildoties, tiek izpildīta procesu secība `B → C`, kurai seko process `D`. Gadījumā, ja dotais nosacījums netiek izpildīts, uzreiz pēc procesa `A` tiek izpildīts process `D`. Doto procesu secību var pierakstīt kā parādīts formulā (5.26):

$$\begin{gathered} A \\ \textbf{\textit{if}}\ cond\ \textbf{\textit{goto}}\ L1\ \textbf{\textit{else goto}}\ L2 \\ L1: (B \rightarrow C) \\ L2: D \end{gathered} \tag{5.26}$$

Var redzēt, ka aizvietošanas rezultātā procesu izpildes noteikumi tiek saglabāti – nosacījumam `cond` izpildoties, tiek izpildīta secība `A → B → C → D`, citādi – `A → D`. Protams, šajā gadījumā, var iztikt bez iezīmes `L1`, jo procesam `A` uzreiz seko procesi `B` un `C`, tomēr autors piedāvā ievietot abas iezīmes, jo veicot koda ģenerēšanu, var būt nepieciešamība pēc tā. Piemēram, programmēšanas valodas Java [50] gadījumā ir nepieciešamas abas iezīmes – jo tās veido tā saucamo baitu koda freimu, kas ir nepieciešams.

Līdzīgi ir iespējams ievietot iezīmes ciklu sākumā un beigās. Formulas (5.27) un (5.28) demonstrē iezīmju ievietošanu ciklam ar pirmsnosacījumu. Var redzēt, ka šajā gadījumā iezīme `L1` ir nepieciešama cikla izpildei.

$$A \rightarrow (while\ cond: (B \rightarrow C)) \rightarrow D \tag{5.27}$$

Tapāt, veicot cikla ar pēcnosacījumu apstrādi, tiek iegūts rezultāts, ko var redzēt formulās (5.29) un (5.30).

$$
\begin{gathered}
A \\
\boldsymbol{if}\ cond\ \boldsymbol{goto}\ L1\ \boldsymbol{else\ goto}\ L2 \\
L1\colon (B \to C) \\
\boldsymbol{goto}\ L1 \\
L2\colon D
\end{gathered}
\tag{5.28}
$$

$$
A \to (dowhile\ cond\colon (B \to C)) \to D \tag{5.29}
$$

$$
\begin{gathered}
A \\
L1\colon (B \to C) \\
\boldsymbol{if}\ cond\ \boldsymbol{goto}\ L1\ \boldsymbol{else\ goto}\ L2 \\
L2\colon D
\end{gathered}
\tag{5.30}
$$

Papildus autors piedāvā pievienot iezīmes visas minimizētās procesu secības sākumā un beigās – piemēram, programmēšanas valodas Java [50] gadījumā tas ir nepieciešams koda ģenerēšanai, jo šīs iezīmes definē baitu koda freimu, kas atbilst attiecīgās metodes robežām.

Papildu iezīmes ir nepieciešams ievietot arī pirms cikla ieejas punktiem, jo šādi punkti ir iespējamie zarošanās mērķi. Ir arī nepieciešams ievietot iezīmes cikla izejas mērķa punktos – jo ir nepieciešams nodot kontroli uz tiem.

Nepieciešams apstrādāt arī disjunkcijas – alternatīvos izpildes ceļus. Šajos gadījumos ir nepieciešams atrast elementu, kas seko disjunkcijai un ievietot iezīmi pirms tā – lai būtu iespējams veikt vienīgās alternatīvas apstrādi un pēc tam turpināt darbu.

Tātad, iezīmju ievietošanas punktu definēšanas algoritmu var aprakstīt ar algoritmu, kas ir definēts 5.61. att. Algoritms iekļauj trīs funkcijas – `defineLabel`, kas pārbauda, vai padotajam elementam jau ir definēta iezīme. Gadījumā, ja tas tā nav, jauna iezīme tiek definēta. Nākamā algoritma funkcija ir `findSequenceEnd`, kas atgriež elementu, kas seko uzreiz aiz padotās elementu secības – cikla gadījumā tas ir pirmais elements, kas seko ciklam, vienkāršas elementu secības gadījumā – pirmais elements pēc dotās secības. Par algoritma galveno funkciju, kalpo funkcija `findLabels`, kas veic padotās elementu secības analīzi, pielietojot visus iepriekš definētos likumus. Pēc nepieciešamības šī funkcija tiek rekursīvi pielietota esošā elementa saturam. Analīze sākas ar minimizētā grafa vienīgās virsotnes saturu.

```
labelMap = {}
labelIdx = 0

function defineLabel(el):
       if not el in labelMap:
              labelMap[el] = Label<labelIdx++>

function findLabels(seq):
       for el in seq:
              if el.hasGuard:
                     defineLabel(el)
                     guardedSequenceEnd = findSequenceEnd(el)
                     defineLabel(guardedSequenceEnd)

              if el is Disjoint:
                     disjointEnd = findSequenceEnd(el)
                     defineLabel(disjointEnd)

              if el is Loop:
                     defineLabel(el)
                     loopEnd = findSequenceEnd(el)
                     defineLabel(loopEnd)

              if el is LoopEntry:
                     defineLabel(el)

              if el is LoopBreak and el.hasTarget:
                     defineLabel(el.target)

              if el.hasChildren:
                     findLabels(el.children)

findLabels(minimizedProcessGraph.resultingSeq)
```

5.61. att. Iezīmju definēšanas algoritms

Turpmāk analizējot formulas (5.26)-(5.30), var redzēt, ka tās bez iezīmēm satur arī nosacījumu pārbaudes (formula (5.19)), kā arī nosacījumu un beznosacījumu zarošanās instrukcijas (formulas (5.18), (5.20) un (5.21)). Var redzēt, ka pirms katras nosacījumu zarošanās instrukcijas vienmēr tiek ievietota nosacījuma pārbaudes instrukcija. Tātad, šiem elementiem ir jābūt ģenerētiem kopā.

Lai definētu nosacījumu pārbaudes un zarošanās instrukciju ievietošanas vietas, promocijas darba autors analizē, kur ir iespējamas zarošanās. Beznosacījuma zarošanās ir iespējama ciklos ar pirmsnosacījumu – šādu ciklu beigās tiek veikta kontroles nodošana uz cikla sākumu. Vēl viena iespēja izpildīt nosacījuma zarošanos ir disjunkcijas – gadījumos, kad eksistē vairākas izpildāmo secību alternatīvas, pēc vienas alternatīvas apstrādes ir nepieciešams pāriet uz disjunkcijas beigām. Analizējot citus procesu izsaukumu secības elementus, var redzēt, ka citu iespēju beznosacījuma zarošanās definēšanai nav.

Nosacījuma zarošanās, savukārt, ir iespējama divos veidos – ja attiecīgais nosacījums izpildās, vai, ja tas netiek izpildīts. Sākumā autors piedāvā veikt otra gadījuma analīzi – jo tas raksturo mazāku skaitu gadījumu.

Pirmais gadījums, kad ir iespējama nosacījuma zarošanās ar neizpildītu nosacījumu, ir procesu izsaukumu secība, kurai ir definēts izpildes nosacījums – ir nepieciešams veikt šāda nosacījuma pārbaudi, un gadījumā, kad tas netiek izpildīts, veikt kontroles nodošanu uz secības elementu, kas atrodas aiz dotās procesu izsaukumu secības.

Otrais nosacījuma zarošanās gadījums nosacījumam neizpildoties, ir cikls ar pirmsnosacījumu – šāda cikla sākumā tiek pārbaudīts tā izpildes nosacījums. Gadījumā, kad nosacījums neizpildās, kontrole tie nodota uz procesu izsaukumu secības elementu, kas atrodas uzreiz aiz dotā cikla.

Nosacījuma zarošanās, kas notiek nosacījumam izpildoties, ir iespējama vairākos gadījumos. Pirmais gadījums ir cikla izejas punkts. Kā jau tika definēts, visiem cikla izejas punktiem tiek definēts to izpildes nosacījuma mainīgais, un tā pārbaude nosaka, vai notiek izeja no cikla. Cikla izejas punktam var eksistēt vai arī neeksistēt mērķa punkts, uz kuru tiek nodota kontrole, izejas punkta nosacījumam izpildoties. Gadījumā, ja tāds punkts eksistē, ir nepieciešams definēt nosacījuma zarošanos uz šo punktu. Citādi nosacījuma zarošanās nodod kontroli procesu izsaukumu secības elementam, kas atrodas uzreiz aiz konkrētā cikla.

```
jumpMap = {}

function findJumps(seq):
       for el in seq:
              if el is Disjoint:
                     disjointEnd = findSequenceEnd(el)
                     label = labelMap[disjointEnd]
                     for alt in el.alternatives:
                            lastElem = findSequenceEnd (alt)
                            jumpMap[lastElem] = Jump<disjointEnd>

              if el.hasGuard and not el is Loop:
                     guardedSequenceEnd = findSequenceEnd(el)
                     label = labelMap[guardedSequenceEnd]
                     var = variableMap[el][1]

                     jumpMap[el] = JumpIfNot<var, label>

              if el is WhileLoop:
                     loopEnd = findSequenceEnd(el)
                     loopStartLabel = labelMap[el]
                     loopEndLabel = labelMap[loopEnd]
                     var = variableMap[el][1]

                     jumpMap[el] = JumpIfNot<var, loopEndLabel>
                     jumpMap[loopEnd] = Jump<loopStartLabel>

              if el is LoopBreak:
                     if el.hasTarget:
                            targetLabel = labelMap[el.target]
                     else:
                            loopSeq = findLoop(el)
                            loopEnd = findSequenceEnd(loopSeq)
                            targetLabel = labelMap[loopEnd]

                     var = variableMap[el][1]
                     jumpMap[el] = JumpIf<var, targetLabel>

              if el is DoLoop:
                     loopEnd = findSequenceEnd(el)
                     loopStartLabel = labelMap[el]
                     var = variableMap[el][1]

                     jumpMap[loopEnd] = JumpIf<var, loopStartLabel>

              if el is LoopEntry:
                     src = el.source
                     jumpSourceEl = nextElement(src)
                     var = variableMap[el]
                     label = labelMap[el][1]

                     jumpMap[jumpSourceEl] = JumpIf<var, label>

              if el.hasChildren:
                     findJumps(el.children)

findJumps(minimizedProcessGraph.resultingSeq)
```

5.62. att. Zarošananās definēšanas algoritms

Otrais šādas nosacījuma zarošanās gadījums ir cikla ar pēcnosacījumu beigas. Gadījumā, ja cikla izpildes nosacījums, ir izpildīts, ir nepieciešams atgriezties attiecīgā cikla sākumā.

Pēdējā iespēja ievietot nosacījuma zarošanos ir gadījumos, kad eksistē cikla ieejas punkts.. Katram cikla ieejas punktam ir definēts procesu izsaukumu secības elements, no kura ir iespējama kontroles nodošana uz doto ieejas punktu. Tātad, pēc attiecīgā procesu izsaukumu secības elementa ir nepieciešams ievietot nosacījuma zarošanās instrukciju.

Šos gadījumus var apstrādāt ar algoritma, kura pseidokods ir parādīts 5.62. att. Šis algoritms izmanto iepriekš definētās mainīgo un iezīmju heštabulas (`variableMap`, `labelMap`), kā arī vairākas iepriekš aprakstītās funkcijas (`findLoop`, `findSequenceEnd`).

Bez tām, tiek definētas arī divas jaunas funkcijas – `findLoop`, kas atgriež ciklu, kas satur padoto elementu. Otrā jaunā funkcija ir `nextElement`, kas saņem procesu izsaukumu grafa elementu kā parametru, un atgriež kā rezultātu nākamo elementu, kas tam seko. Par algoritma galveno funkciju, savukārt, kalpo funkcija `findJumps`, kas rekursīvi analizē padoto procesu izsaukumu grafa elementu secību, pārbaudot visus iepriekš aprakstītos gadījumus. Funkcijas izpildes rezultātā tiek aizpildīta heštabula `jumpMap`, kas satur informāciju par zarošanās instrukciju ievietošanas punktiem. Analīze sākas ar minimizētā grafa vienīgās virsotnes saturu.

Pēc visu trīs iepriekš aprakstīto tabulu aizpildīšanas ir iespējams veikt koda ģenerēšanu – jo visa nepieciešamā informācija par mainīgajiem, zarošanās punktiem un nepieciešamajiem zarošanās nosacījumiem ir apkopota, un var veikt procesu izsaukumu secības apstaigāšanu, transformējot katru tās elementu pēc definētajiem likumiem, ņemot vērā iezīmju un zarošanās instrukciju ievietošanas punktus. Tomēr, pirms koda ģenerēšanas veikšanas ir nepieciešams definēt nepieciešamās klases, kas tiks izmantotas tās gaitā.

Var redzēt, ka pēc procesa izpildes var tikt iegūti viens vai vairāki rezultāti – tas nozīmē to, ka uzģenerētajai metodei būtu jāatgriež neviens vai vairāki koncepti vai primitīvie datu tipi. Eksistē programmēšanas valodas, kas ļauj to darīt – piemēram Python [119], Ruby [96] un citas. Tādas programmēšanas valodas, kā C [95], C++ [101], C# [12] un citas, ļauj arī ierakstīt atgriežamo vērtību vienā no metodes/funkcijas parametriem. Tomēr, citas valodas, tādas kā Java [50], JavaScript [52] un citas, to neļauj izdarīt bez speciālo klašu vai paņēmienu izmantošanas. Līdz ar to, lai panāktu transformācijas likumu maksimālu universālumu, promocijas darba autors piedāvā izveidot tā saucamās rezultātu klases, kas sevī apvieno attiecīgā biznesa procesa izpildes rezultātus. To var panākt, apstaigājot visus biznesa procesus, katram no tiem atrodot attiecīgo signatūru, un analizējot to. Gadījumos, kad attiecīgais biznesa process atgriež vairākas vērtības, ir nepieciešama rezultāta klases definēšana. Ir arī nepieciešams definēt attiecīgo mainīgo procesa izsaukuma rezultāta glabāšanai. Attiecīgā algoritma pseidokods ir parādīts 5.63. att.

Var redzēt, ka attiecīgā algoritma izpildes rezultātā tiek izveidotas vairākas klases, kas satur informāciju par attiecīgo metožu atgriežamajām vērtībām. Šīs klases veido apakškopu no visām metodes ģenerējamajām klasēm. Pie tam ir nepieciešams veikt arī konceptu no konceptu diagrammas pārveidošanu. Tā kā katrs koncepts ietver informāciju par tā atribūtiem, šī pārveidošana ir pietiekami vienkārša – katram konceptam tiek izveidota atbilstoša klase. Attiecīgā koncepta atribūti tiek pārveidoti par attiecīgās klases atribūtiem. Līdzīga pieeja tiek apskatīta arī [77],[82] un [84]. Attiecīgā algoritma pseidokods ir parādīts 5.64. att.

```
resultMap = {}
for p in processDiagram.processes:
	Set<DataFlow> outputs = processMapping[p].outputs
	resultingValues = []

	for df in outputs:
		for par in df.parameters:
			name = par.name
			type = par.type

			resultingValues += (name, type)

	if resultingValues.size == 1:
		resultMap[p] = resultingValues[0].type
	else if resultingValues.size > 1:
		resultClass = new Class()
		resultClass.name = generateClassName(p)

		for rv in resultingValues:
			resultClass.addAttribute(rv)

		resultMap[p] = resultClass
		addClass(resultClass)
		variableMap[p] = (resultClass, varIdx++)
```

5.63. att. Rezultātu klašu definēšanas algoritms

Izpildot abus šos algoritmus, tiek iegūta visu ģenerējamo klašu kopa, un ir iespējams veikt koda ģenerēšanu.

```
conceptClassMap = {}

for c in conceptDiagram.concepts:
	conceptClass = new Class()
	conceptClass.name = c.name

	for a in c.attributes:
		name = a.name
		type = a.type

		conceptClass.addAttribute((name, type))

	conceptClassMap[c] = conceptClass
	addClass(conceptClass)
```

5.64. att. Konceptu pārveidošana par klasēm

Koda ģenerēšanas galvenais uzdevums ir izveidot attiecīgo koda reprezentācijas elementu secību. Šīs secības sākumā autors piedāvā izvietot nepieciešamo mainīgo definīcijas atbilstoši formulai (5.10). Tas ir paredzēts, lai padarītu metodi universālāku – ne visas programmēšanas valodas ļauj definēt mainīgos jebkurā metodes vietā – piemēram, programmēšanas valodas C [95] gadījumā mainīgajiem ir jābūt definētiem to redzamības apgabala sākumā. Ne visas programmēšanas valodas definē šāda veida prasības, bet mainīgo definēšana ģenerējamā koda sākumā turpmāk ļauj to izmantot dažādām mērķa valodām.

```
instructions = []
for k -> (type, idx) in variableMap:
       variableName = generateName(k)
       instructions += Var<idx, variableName, type>
```

5.65. att. Mainīgo definēšanas instrukciju ģenerēšana

Nepieciešamo mainīgo definēšanai ir jāapstaigā iepriekš izveidotā mainīgo heštabula, pārveidojot katru no tās elementiem par mainīgā definīciju. Tā kā mainīgo heštabula jau satur visu nepieciešamo informāciju, viss, kas ir nepieciešams, ir šīs informācijas pievienošana ģenerējamajai koda elementu secībai (skat. 5.65. att.).

Kad mainīgo definēšanas instrukcijas ir pievienotas ģenerējamajai koda elementu secībai, ir iespējams veikt minimizētā procesu izsaukuma grafa vienīgās virsotnes satura rekursīvo apstaigāšanu. Veicot šo apstaigāšanu, ir nepieciešams analizēt, kas ir apskatāmais secības elements, vai tam ir definēta iezīme un zarošanās iespēja. Apskatāmā elementa apstrādes algoritmu var definēt šādi:

- Pārbaudīt, vai apskatāmajam elementam ir definēta iezīme. Ja tas tā ir, pievienot iezīmes ievietošanas instrukciju ģenerējamajai koda elementu secībai.
- Pārbaudīt, vai apskatāmajam elementam ir definēts izpildes nosacījums. Ja tas tā ir, pievienot nosacījuma pārbaudes instrukciju (5.19) ģenerējamajai koda elementu secībai.
- Pārbaudīt, vai ir iespējama zarošanās no apskatāmā elementa. Ja tas tā ir, pievienot attiecīgo zarošanās instrukciju ģenerējamajai koda elementu secībai.
- Pārbaudīt, vai apskatāmais elements ir procesa izsaukums. Ja tās tā ir, ir nepieciešams noteikt visus attiecīgā procesa parametrus, kas kļūst par attiecīgās instrukcijas (procesa izsaukums – (5.12)) parametru sarakstu. Tad ir nepieciešams noteikt, kurā mainīgajā tiks ierakstīts izsaukuma rezultāts. Šeit pastāv trīs iespējas:

- Metodes izsaukuma rezultātā tiek atgriezta viena datu plūsma ar vienu parametru. Šajā gadījumā ir iespējams atrast pareizo mainīgo attiecīgajā heštabulā un definēt to kā izsaukuma rezultātu.
- Citādi ir nepieciešams atrast pareizo rezultāta klasi, kā arī attiecīgo mainīgo heštabulā. Bez tā ir nepieciešams veikt arī rezultāta apstrādi – attiecīgos rezultāta atribūtus ierakstīt attiecīgajos mainīgos, definējot papildus instrukcijas ((5.14) – atribūta nolasīšana), kas tiks ievietotas pēc attiecīgā procesa izsaukuma instrukcijas.
- Procesa izsaukums neatgriež nekādu rezultātu. Šajā gadījumā nekādas papildus darbības nav nepieciešamas.

Rezultātā, ir zināms, kāds process ar kādiem parametriem tiks izsaukts, kā arī, kas notiks ar tā izpildes rezultātu, ja tāds eksistē. Līdz ar to ir iespējams ģenerējamajai instrukciju secībai pievienot visas nepieciešamās instrukcijas.

Visas apskatāmā elementa apstrādes rezultātā iegūtās instrukcijas tiek pievienotas ģenerējamo koda elementu secībai. Attiecīgais elementa apstrādes algoritms ir parādīts 5.66. att.

Pēc iepriekš definēto darbību izpildes tiek iegūti šādi ģenerējamā koda elementi:

- Ģenerējamo klašu saraksts, kas satur gan klases, kas tiek iegūtas konceptu modeļa elementu apstrādes rezultātā, gan arī klases, kas ir procesu izsaukumu izpildes rezultāti.
- Koda elementu secība, kas definē attiecīgās procesu diagrammas biznesa loģiku.

Faktiski, iegūtais rezultāts satur gandrīz visu informāciju, kas ir nepieciešama veiksmīgai koda ģenerēšanai. Rezultāts neiekļauj metožu, kuras atbilst konkrētajiem biznesa procesiem, signatūras. Bet, šo signatūru ģenerēšana jau tika izpildīta pirms tika veikta biznesa procesu diagrammas grafa minimizēšana. Līdz ar to, pēc visu aprakstīto algoritmu izpildes tiek iegūts nepieciešamais informācijas apjoms, kas atbilst ģenerējamajam programmatūras kodam.

```
function generateInstructions(seq):
       for el in seq:
              if el in labelMap:
                     instructions += labelMap[el]
              if el.hasGuard:
                     var = variableMap[el]
                     instructions += Check<var, el.guard>
              if el in jumpMap:
                     instructions += jumpMap[el]
              if el is ProcessInvocation:
                     process = processMapping[el.process]
                     params = []
                     for df in process.inputs:
                            for p in df.parameters:
                                   key = (d, p)
                                   var = variableMap[key][1]
                                   params += var

                     resultingValues = []
                     for df in process.outputs:
                            for p in df.parameters:
                                   resultingValues += (df, p.type, p.name)

                     resultVar = null
                     additionalInstructions = []
                     if resultingValues.size == 1:
                            key = (resultingValues[0][0],
                                   resultingValues[0][1])
                            resultVar = variableMap[key][1]
                     else if resultingValues.size != 0:
                            resultVar = variableMap[el.process][1]
                            for rv in resultingValues:
                                   key = (rv[0], rv[1])
                                   var = variableMap[key][1]
                                   attr = (rv[2], rv[1])
                                   additionalInstructions
                                          += GetField<resultVar, attr, var>
                     instructions += Invoke<process, params, resultVar?>
                     for ai in additionalInstructions:
                            instructions += ai

              if el.hasChildren:
                     generateInstructions(el.children)

generateInstructions(minimizedProcessGraph.resultingSeq)
```

5.66. att. Secības elementa apstrāde

5.6. Koda ģenerēšanas algoritms – kopsavilkums

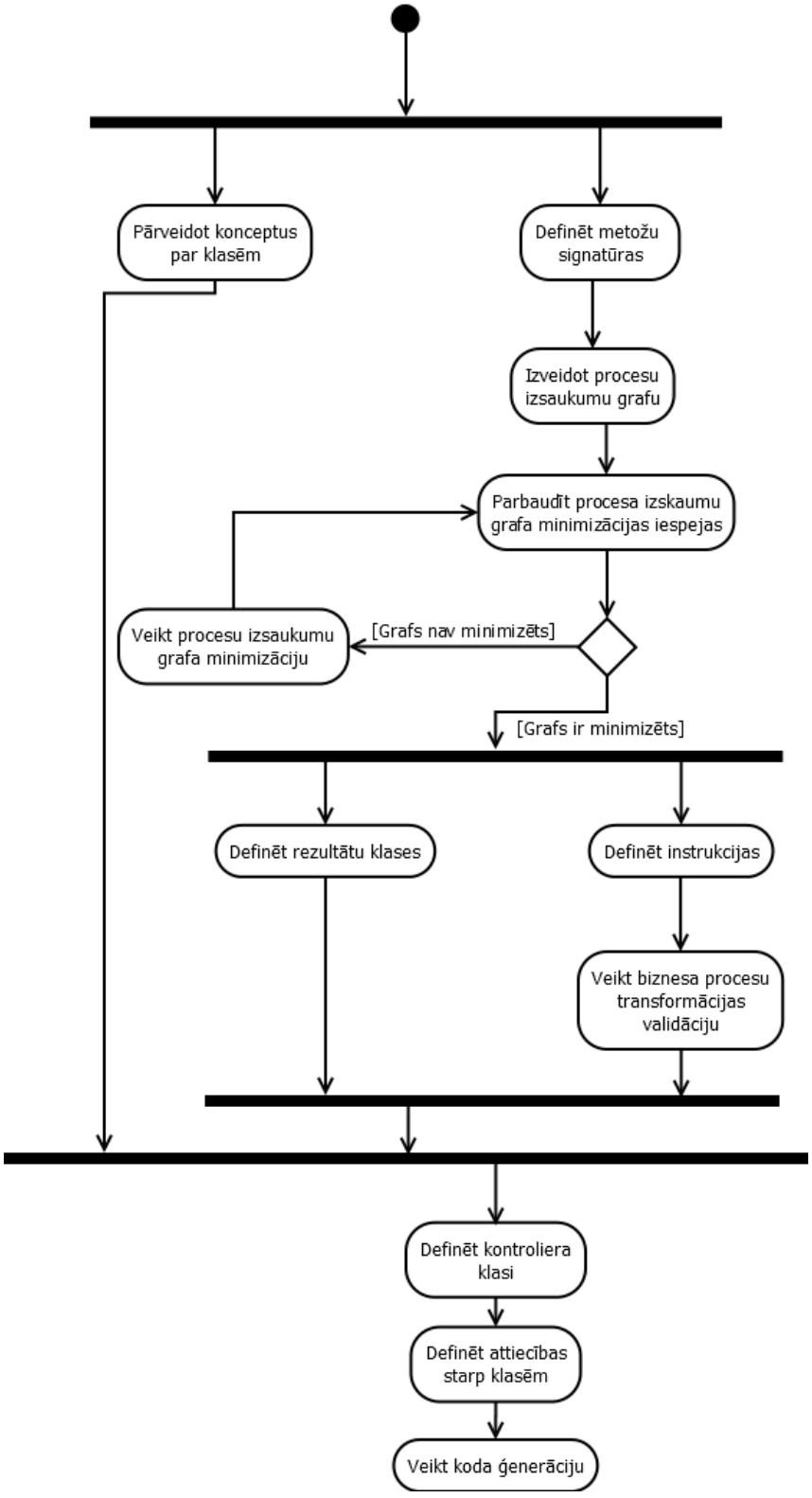
Šajā sadaļā ir aprakstīti transformācijas likumi, kas tiek pielietoti avota divpusložu modelim, iegūstot modelim atbilstošu instrukciju secību. Sākumā divpusložu modelis tiek sadalīts konceptu diagrammā un biznesa procesu diagrammu kopā. Katra no šīm sastāvdaļām tiek apstrādāta atsevišķi – koncepti no konceptu diagrammas var tikt uzreiz pārveidoti par klasēm, kas tiks pievienotas rezultāta klašu kopai. Katra biznesu procesu diagramma savukārt tiek analizēta, iegūstot metožu signatūras un tiek pārveidota par procesu izsaukumu grafu.

Rezultātā tiek iegūta procesu izsaukumu grafu kopa. Katrs no šiem procesu izsaukumu grafiem tiek minimizēts, izmantojot promocijas darba autora piedāvātās metodes. Pēc minimizēšanas tiek iegūti vairāki grafi, katrs no tiem satur vienu virsotni. Pie tam, šī virsotne satur visu nepieciešamo instrukciju secību, kas apraksta avota biznesa procesu. Šī informācija, savukārt, tiek izmantota instrukciju secību, kas apraksta attiecīgos biznesa procesus, definēšanai. Instrukciju definēšanas procesā tiek definētas arī tā saucamās rezultātu klases. Pēc šo procesu izpildes ir iespējams veikt biznesa procesu transformāciju rezultātu validāciju. Rezultātā tiek iegūti šādi artefakti:

- Uzģenerēto klašu kopa, kas satur domēna klases, kas tiek izveidotas no konceptiem, un rezultātu klases, iegūtas biznesa procesa diagrammu transformācijas rezultātā.
- Metožu signatūru definīcijas.
- Instrukciju secības.

Apvienojot metožu signatūras un instrukciju secības, ir iespējams iegūt klasi, kas realizē procesus, kas tiek definēti vienā biznesa procesu diagrammā. Autors sauc šo klasi par kontroliera klasi (pēc MVC [111] principiem). Kad visas nepieciešamās kontrolieru klases ir definētas, ir iespējams veikt iegūtās klašu kopas, kas pašreiz satur domēna, rezultātu un kontrolieru klases, apstrādi ar klašu attiecību noteikšanas algoritmu. Tas ļauj iegūt finmāla klašu kopu, kas tiks izmantota programmatūras koda ģenerēšanai. Ir vērts atzīmēt, ka ir iespējams arī sākumā uzģenerēt programmatūras kodu, un pēc tam izmantot klašu attiecību noteikšanas algoritmu, jo kā ir definēts, pats algoritms tika izveidots tā, lai varētu strādāt ar jebkāda veidā definētu klašu kopu. Transformācijas likumu pielietošanas process ir parādīts 5.67. att.

Nākamajā sadaļā tiek apskatīts transformācijas likumu pielietošanas piemērs, transformāciju rezultātu validācijas algoritms, kā arī Java [50] koda ģenerēšana.



5.67. att. Transformācijas likumu pielietošana

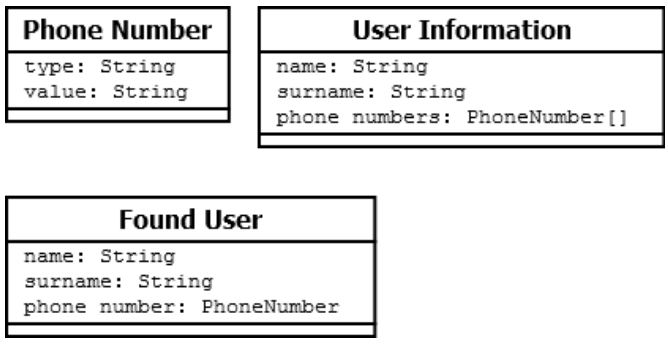
6. KODA ĢENERĒŠANAS ALGORITMA NOVĒRTĒŠANA

Iepriekšējās nodaļās ir apskatīti un pamatoti uzlabojumi divpusložu modelī un transformāciju algoritmi, kas var tikt izmantoti kopā ar uzlaboto modeli. Algoritmu izpildes rezultātā tiek iegūta klašu kopa, metožu signatūru definīcijas un instrukciju secības, kas atbilst avota procesu diagrammām. Izmantojot šo informāciju, ir nepieciešams ģenerēt programmatūras kodu un pārbaudīt tā atbilstību avota modelim.

Koda ģenerēšanas uzdevums nav jauns uzdevums programmatūras inženierijas vēsturē. Ar automātisku programmatūras koda ģenerēšanu no iepriekšdefinētiem sistēmas modeļiem vai prasībām programmatūras inženierijas speciālisti sāka nodarboties jau 1980-jos gados (piemēram, [61]). Koda ģenerēšanu atbalsta arī vairākas modernas izstrādes vides, piemēram [22],[45] un citas.

Šajā nodaļā promocijas darba autors demonstrē piedāvātā algoritma darbību ar praktisku piemēru, specificējot algoritma darbības galvenos principus, apraksta ar koda ģenerēšanu saistītos pētījumus, parādot izstrādātā algoritma vietu starp pētījumiem un izceļ izstrādātā algoritma priekšrocības, kā arī validē izstrādātā algoritma darbības rezultātu pret avota modeli, parādot, ka visa informācija, kas tika reprezentēta tajā, ir saglabāta.

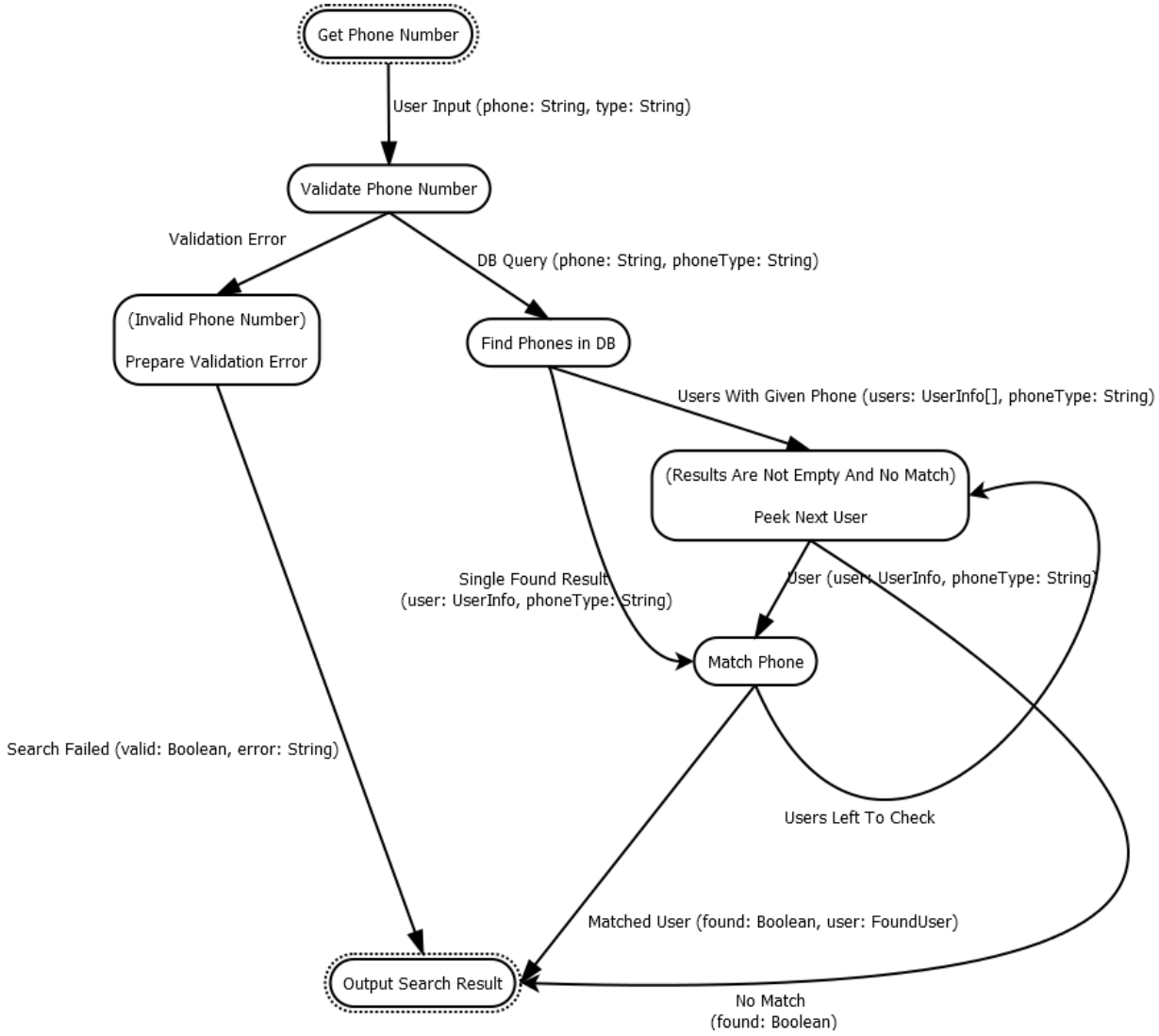
6.1. Algoritma darbības piemērs



6.1. att. Piemēra divpusložu modelis – konceptu diagramma

Lai būtu iespējams apskatīt izstrādātā transformācijas algoritma darbību, tiek piedāvāts apskatīt šādu divpusložu modeli (6.1. att. un 6.2. att.). Koda ģenerēšanas nolūkiem tas tiek definēts angļu valodā. Šis divpusložu modelis apraksta kontaktpersonas meklēšanas procesu pēc telefona numura un tā tipa (piemēram, mobilais, darba, vai cits). Ir definēti trīs koncepti:

- `Phone Number` – informācija par noteiktu kontaktpersonas telefona numuru – numura tips un numurs.
- `User Information` – informācija par kontaktpersonu – vārds, uzvārds un telefona numuru saraksts.
- `Found User` – informācija par atrasto kontaktpersonu – vārds, uzvārds un atbilstošais telefona numurs.



6.2. att. Piemēra divpusložu modelis – biznesa procesu diagramma

Biznesa procesu diagramma sākas ar telefona numura ievades procesu, kas atbilst kontaktpersonas datu ievadei – `Get Phone Number`. Dotais process atgriež divas vērtības – meklējamo telefona numuru un tā tipu.

Pēc tam tiek veikta lietotāja datu ievades validācija. Par to atbild process ar nosaukumu `Validate Phone Number`. Atkarība no validācijas rezultāta, process var vai nu atgriezt

validācijas kļūdu, nododot kontroli `Prepare Validation Error` procesam, vai nu nodot kontroli tālāk – `Find Phones in DB` procesam, kas savukārt veic kontaktpersonas ar doto telefona numuru meklēšanu datu bāzē. Ir svarīgi atzīmēt faktu, ka datu plūsma, kas iet no validācijas procesa uz kļūdas ziņojuma veidošanas procesu ir tukša – tā nesatur nekādus datus, kas tiek padoti procesam.

`Prepare Validation Error` process satur izpildes nosacījumu (`Invalid Phone Number`), kas definē to, ka šis process tiks izpildīts, tikai saņemot nepareizu datu ievadi no lietotāja – piemēram, neveiksmīgas datu validācijas gadījumā. Process atgriež divas vērtības – kļūdas paziņojumu un neveiksmīgas meklēšanas pazīme. Šie dati tiek nodoti beigu procesam šajā diagrammā – `Output Search Result`, kas ir paredzēts meklēšanas rezultāta prezentācijai lietotājam, piemēram, izvadot meklēšanas rezultātus ekrānā.

Process, kas ir nosaukts `Find Phones in DB` kā parametrus saņem divas vērtības – telefona numuru un tā tipu un veic datu meklēšanu datu bāzē. Rezultātā tiek iegūts kontaktpersonu saraksts, kas var būt tukšs, saturēt vienu vai vairākus elementus. Papildus tiek atgriezts arī meklējamā telefona numura tips, kas faktiski tiek padots cauri visiem meklēšanas procesiem, un ir saņemts no lietotāja. Procesa izpildes rezultāti var tikt padoti diviem nākamajiem procesiem. Pirmais no tiem ir `Match Phone`, kas pārbauda, vai noteiktais ieraksts atbilst meklēšanas kritērijiem. Tas saņem telefona numura tipu un kontaktpersonas informācijas ierakstu. Otrais ir `Peek Next User`, kas saņem meklējamā telefona numura tipu un kontaktpersonu sarakstu.

`Match Phone` process pārbauda, vai tam padotais kontaktpersonas informācijas ieraksts atbilst meklēšanas kritērijiem, gadījumā ja tas tā ir, tiek atgriezts rezultāts, kas ir meklēšanas pazīme un kontaktpersonas informācija `Output Search Result` procesam. Pretējā gadījumā tiek izsaukts `Peek Next User` process, pieprasot nākamo datubāzes pārmeklēšanas rezultātu.

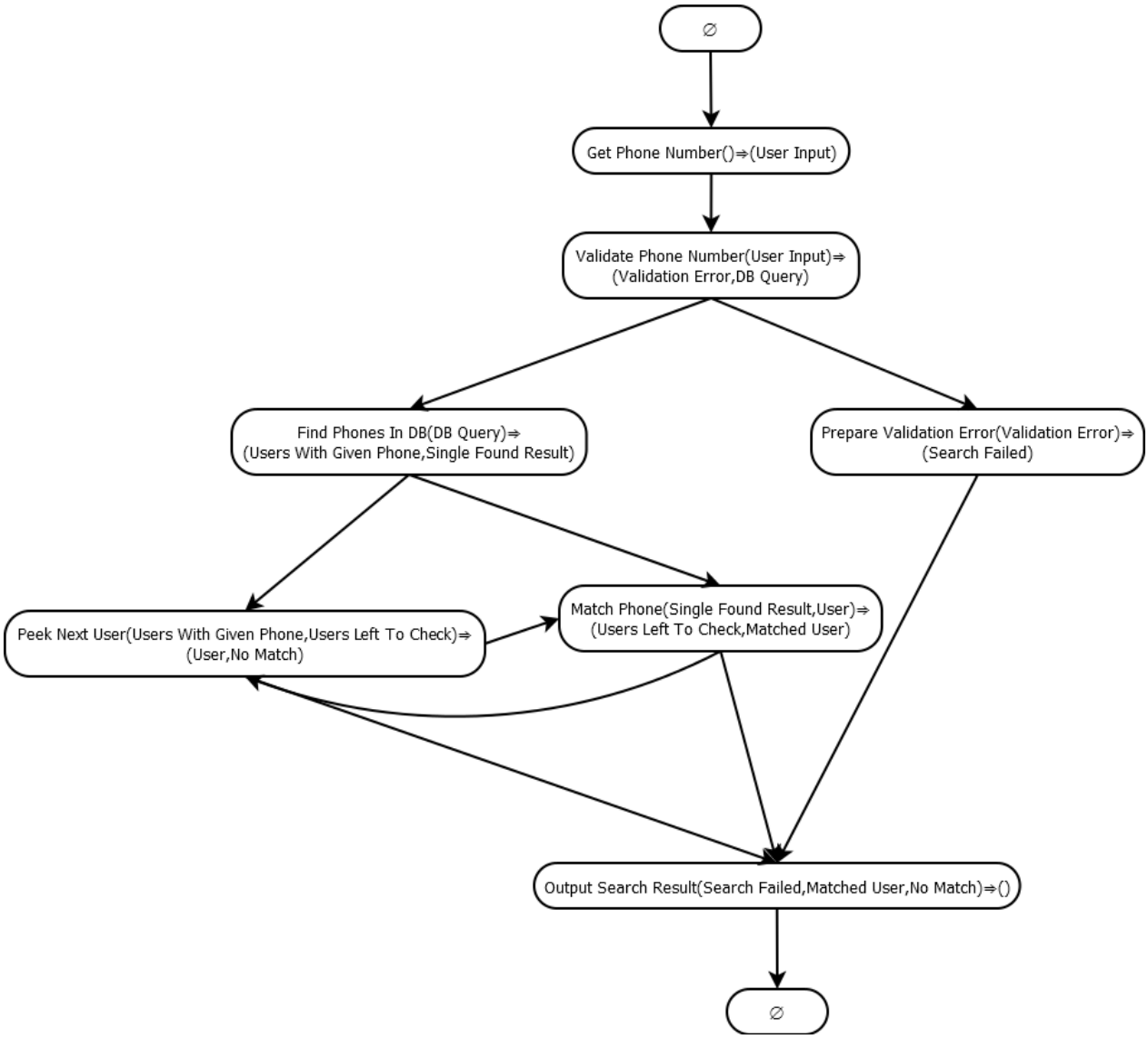
Process `Peek Next User` izpildās, ja izpildās attiecīgais priekšnosacījums (`Results Are Not Empty And No Match`), tas ir, ja vēl eksistē kontaktpersonu ieraksti, kas vēl nav apstrādāti, un rezultāts vēl nav atrasts. Tas var nodot kontroli diviem procesiem. `Match Phoneprocesam tiek padots` nākamais kontaktpersonas ieraksts un meklējamā telefona numura tips. `Output Search Result tiek padota` neveiksmīgas meklēšanas pazīme.

Procesi `Match Phone` un `Peek Next User` veido ciklu, kam eksistē divi ieejas punkti – tajā ir iespējams ieiet, sākot ar jebkuru no šiem diviem procesiem. Ciklam arī ir divas iespējamās izejas – ir iespējams iziet no cikla pēc jebkura no šo divu procesa izpildes.

Piemēra modeļa apraksts divpusložu modeli aprakstošā domēna-specifiskā valodā ir dots 4. pielikumā.

Pirmā transformācija, kas tiek veikta, ir metožu signatūru definēšana, kas ietver visu biznesa procesu diagrammas procesu apstrādi, analizējot katra procesa ieejas un izejas. Pēc šīs transformācijas izpildes tiek iegūti šādi rezultāti:

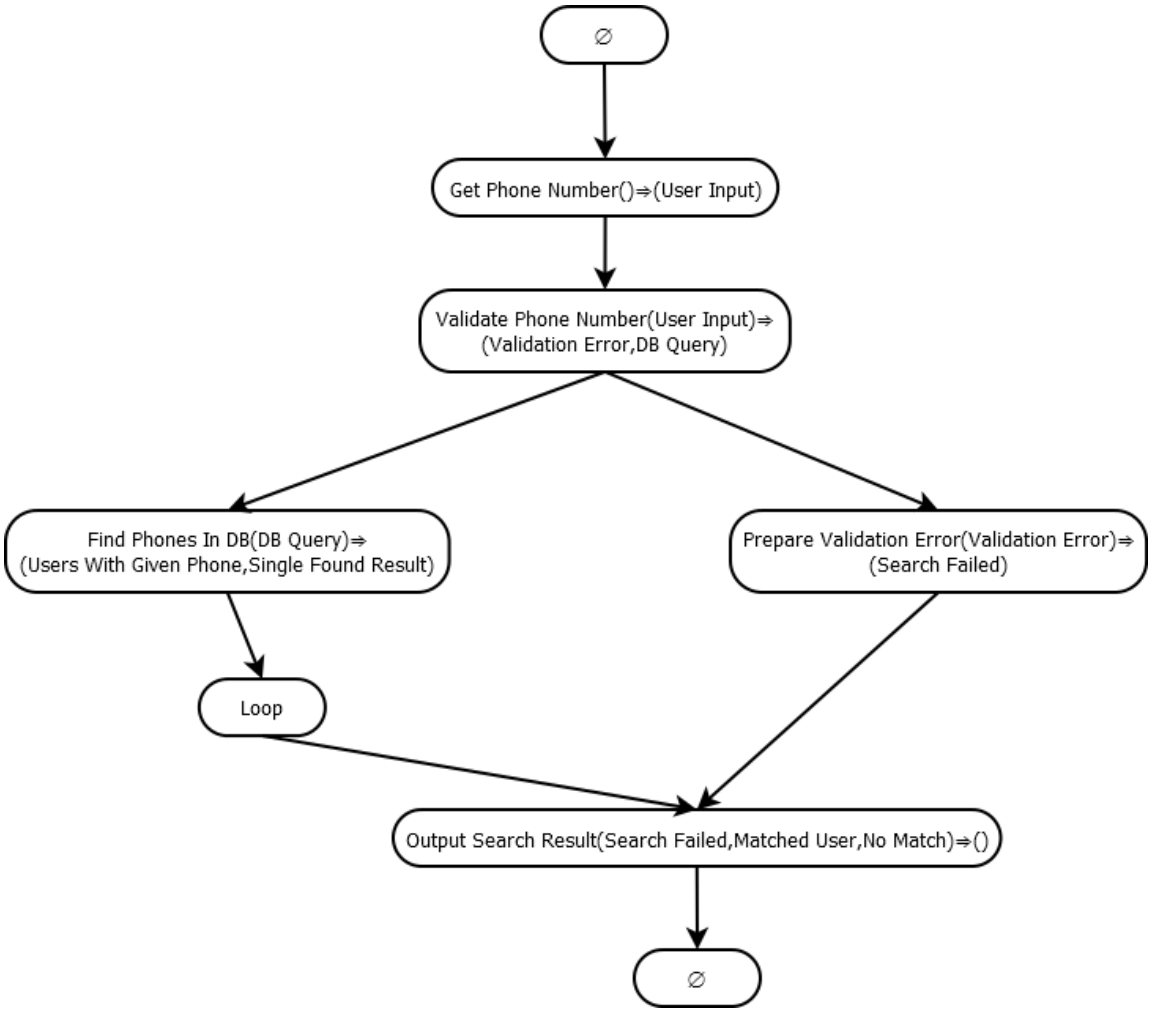
- `Get Phone Number()⇒(User Input)` – attiecīgais process nesaņem nekādus datus, atgriež lietotāja ievadīto meklēšanas pieprasījumu.
- `Validate Phone Number(User Input)⇒(Validation Error,DB Query)` – attiecīgais process saņem lietotāja ievadīto meklēšanas pieprasījumu un atgriež vai nu validācijas kļūdu, vai pieprasījumu datu bāzei.
- `Prepare Validation Error(Validation Error)⇒(Search Failed)` – process saņem validācijas kļūdu un atgriež neveiksmīgās meklēšanas rezultātu.
- `(Find Phones In DB(DB Query)⇒(Users With Given Phone,Single Found Result)` – process saņem pieprasījumu datu bāzei un var atgriezt vai nu atrasto kontaktpersonu sarakstu, vai vienu atrasto rezultātu.
- `Peek Next User(Users With Given Phone,Users Left To Check)⇒(User,No Match)` – process var saņemt vai nu kontaktpersonu sarakstu, vai datu plūsmu `Users Left To Check`, kas pēc savas būtības nesatur nekādus datus. Tiek atgriezts vai nu nākamā kontaktpersona, vai neveiksmīgas meklēšanas rezultāts.
- `Match Phone(Single Found Result,User)⇒(Users Left To Check,Matched User)` – process var saņemt divas datu plūsmas, kas pārnes vienu un to pašu informāciju – kontaktpersonu, kuru ir nepieciešams pārbaudīt un meklējamo telefona tipu. Ir iespējami divi procesa izpildes rezultāti – vai nu veiksmīgas meklēšanas rezultāts `Matched User`, vai datu plūsma `Users Left To Check`, kas ir aprakstīta iepriekš.
- `Output Search Result(Search Failed,Matched User,No Match)⇒()` – tas ir pēdējais process, kas izvada meklēšanas rezultātus, attiecīgi saņemot tos vienā no trim veidiem: `Search Failed` – kas norāda uz kļūdainu datu ievadi, `Matched User` – atrasta kontaktpersona, vai `No Match` – kas norāda , ka meklēšanas rezultātā nekas netika atrasts. Dotais process neatgriež rezultātus.



6.3. att. Iegūtais procesu izsaukumu grafs

Nākamais solis ir procesu izsaukumu grafa izveidošana no sākotnējā biznesa procesu modeļa. Izpildot šo soli, tiek iegūts grafs, kas ir parādīts 6.3. att. Var redzēt, ka divi ārējie procesi pašreiz ir savienoti ar sākuma un beigu stāvokļiem. Citādi, šī grafa struktūra atbilst biznesa procesu modelim.

Līdzīgi kā sākotnējais biznesa procesu modelis, procesu izsaukumu grafs satur ciklu, kas tiek apstrādāts nākamajā transformācijas solī. Rezultātā tiek iegūts grafs, kas ir parādīts 6.4. att. Var redzēt, ka cikla aizvietošanas rezultātā procesu izsaukumu grafs ir vienkāršots, un to var minimizēt, izmantojot pārējos definētos algoritmus.



6.4. att. Procesu izsaukumu grafs pēc cikla aizvietošanas

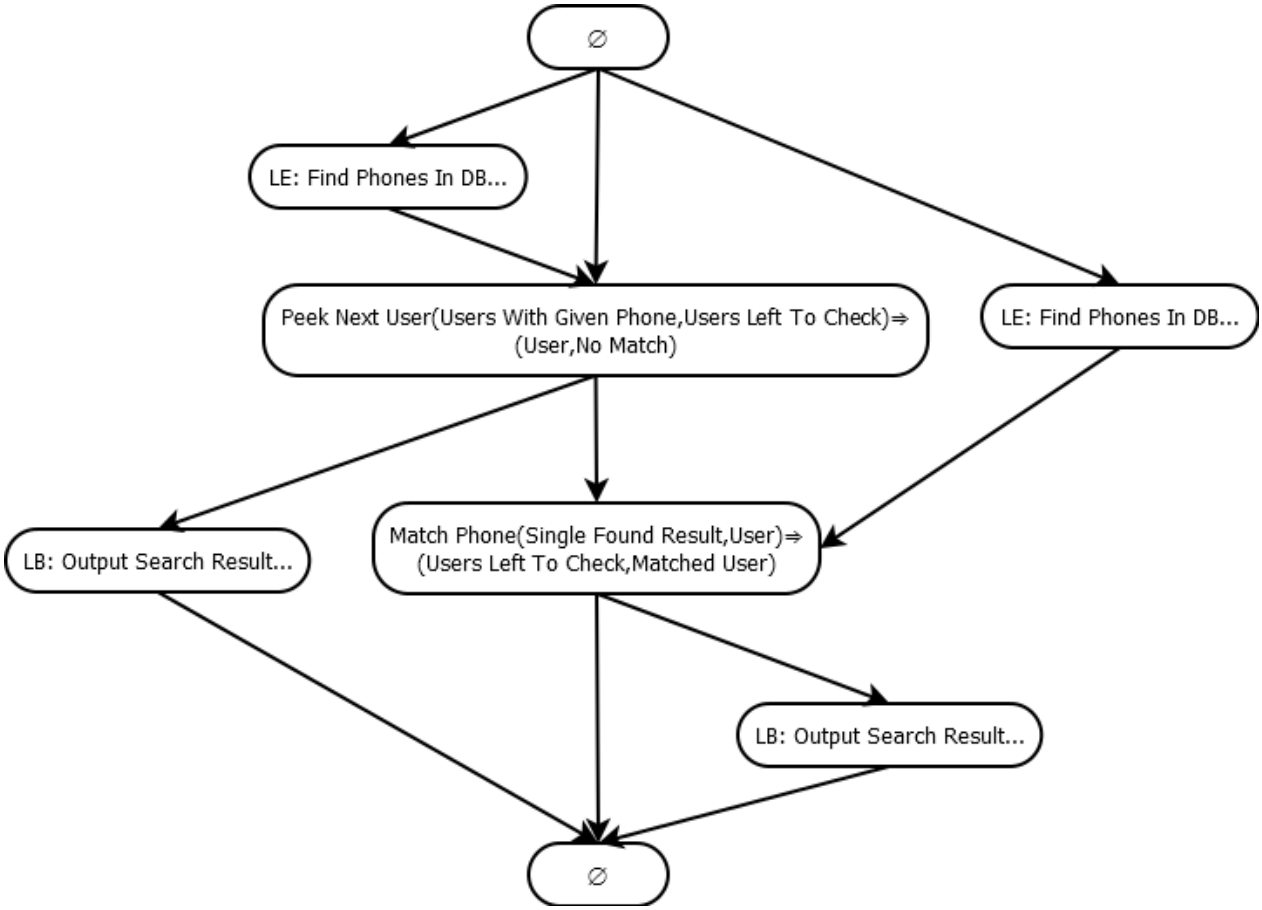
Savukārt, cikla saturs tiek reprezentēts ar procesa izsaukuma grafa, kas ir parādīts 6.5. att., palīdzību. Var redzēt, ka pats cikls sastāv no diviem procesiem, pie tam ir iespējams ieiet ciklā, sākot ar jebkuru no tiem. Tapāt, pēc jebkura cikla procesa izpildes ir iespējams no cikla iziet.

Apzīmējot:

1. Get Phone Number
2. Validate Phone Number
3. Find Phones In DB
4. Prepare Validation Error
5. Loop
6. Output Search Result
7. Peek Next User
8. Match Phone

Galveno procesu izsaukumu grafu var pierakstīt kā `1 → 2 → ((3 → 5) || 4) → 6`, kas atbilst tā minimizētajai struktūrai. Savukārt, minimizējot cikla procesa izsaukumu grafu,

tiek iegūts šāds rezultāts: `while: (7 → LB: ∅ → LE: 3 → 8)`, kas var tikt ievietots minimizētajā procesa izsaukumu grafā, rezultātā iegūstot: `1 → 2 → ((3 → (while: (7 → LB: ∅ → LE: 3 → 8))) || 4) → 6`.



6.5. att. Cikla procesu izsaukumu grafs

Dotais pieraksts nesatur procesu izpildes nosacījumus, kas ir ar mērķi, lai padarītu šo pierakstu vieglāk uztveramu. Veicot transformācijas, visa šī informācija tiek saglabāta un izmantota. Var redzēt, kā pēc grafa minimizēšana, tas tiek pārveidots lineārā datu struktūrā, ko tagad ir iespējams apstrādāt, definējot attiecīgos ģenerējamā koda elementus.

Kā jau tika definēts iepriekš, pirmais koda ģenerēšanas solis ir mainīgo tabulas veidošana. Sākumā mainīgo tabula ir tukša. Pēc tam tiek veikta visu datu plūsmu apstaigāšana mainīgo noteikšanai. Apzīmējot:

```
D1: User Input
D2: Validation Error
D3: Search Failed
D4: DB Query
D5: Users With Given Phone
```

```
D6: Single Found Result
D7: User
D8: Users Left To Check
D9: Matched User
D10: No Match
```

Un veicot datu plūsmu analīzi, tiek iegūti šādi mainīgie, kas ir parādīti 6.1. tabulā. Šeit katram mainīgajam ir definēta atslēga, kas nosaka, no kura elementa tika ģenerēts attiecīgais mainīgais, tā tips un indekss, kas tiks izmantots koda ģenerēšanas laikā.

6.1. tabula

Mainīgie, kas tika iegūti no datu plūsmām

Atslēga	Tips	Indekss
(D4, phone)	String	1
(D4, phoneType)	String	2
(D5, users)	User Information[]	3
(D5, phoneType)	String	4
(D6, user)	User Information	5
(D6, phoneType)	String	6
(D1, phone)	String	7
(D1, type)	String	8
(D3, error)	String	9
(D3, found)	Boolean	10
(D9, user)	Found User	11
(D9, found)	Boolean	12
(D10, found)	Boolean	13
(D7, user)	User Information	14
(D7, phoneType)	String	15

Pēc datu plūsmu satura analīzes ir nepieciešams veikt arī procesa izsaukuma grafa apstrādi ar mērķi noteikt iespējamos ar zarošanos saistītos mainīgos, kā arī procesu izpildes rezultātu klases. Analizējot procesu izsaukumus, tiek definētas šādas papildu klases, kas satur procesu izsaukumu rezultātus (skat. 6.6. att.). Var redzēt, ka papildu rezultātu klases tika uzģenerētas procesiem, kas atgriež vairāk nekā vienu vērtību, pat ja no attiecīgā procesa iziet tikai viena datu plūsma.

```
class GetPhoneNumberResult {
    String phone
    String type
}

class PrepareValidationErrorResult {
    String error
    Boolean found
}

class MatchPhoneResult {
    (Found User) user
    Boolean found
}

class FindPhonesInDBResult {
    (User Information)[] users
    String phoneType
    (User Information) user
    String phoneType
}

class ValidatePhoneNumberResult {
    String phoneType
    String phone
}

class PeekNextUserResult {
    (User Information) user
    String phoneType
    Boolean found
}
```

6.6. att. Procesa izsaukumu rezultātu klases

Rezultātā pēc procesu izsaukumu grafa analīzes tiek definēti papildu mainīgie (skat. 6.2. tabulu). Par atslēgu kalpo vai nu norāde uz attiecīgo procesu, kura rezultāts tiek izmantots, vai procesu izsaukumu grafa elements, kas ir viens no pieciem variantiem:

- `Invoke X` – procesa X izsaukums;
- `LE: X` – cikla ieejas punkts;
- `Loop X` – cikls X;
- `LB: X` – cikla izejas punkts.
- `LS: X` – cikla X sākums.

6.2. tabula

Mainīgie, kas tika iegūti procesu izsaukumu grafa analīzes rezultātā

Atslēga	Tips	Indekss
`Process: 1`	`GetPhoneNumberResult`	`16`
`Process: 2`	`ValidatePhoneNumberResult`	`17`
`Invoke 4`	`Boolean`	`18`
`Process: 4`	`PrepareValidationErrorResult`	`19`

6.2.tabula (turpinājums)

Process: 3	FindPhonesInDBResult	20
LE: 3	Boolean	21
Loop 1	Boolean	22
Process: 7	PeekNextUserResult	23
LB: ∅	Boolean	24
Process: 8	MatchPhoneResult	25

Mainīgo izveidošanas rezultātā tika definēti 25 mainīgie, kas var tikt izmantoti koda ģenerēšanas laikā. Ir svarīgi atzīmēt, ka definētie mainīgie, kas ir aprakstīti šajā darbā nav sakārtoti kādā noteiktā secībā, jo to saraksts ir iegūts no speciāli izveidotas programmas, kas veic doto analīzi, un veic apstrādi nenoteiktā secībā.

Nākamais koda ģenerēšanas solis ir iezīmju definēšana. Kā bija parādīts iepriekš, iezīmju definēšana notiek, analizējot procesu izsaukumu grafu. Rezultātā tiek iegūta tabula, kas satur attiecīgo procesa izsaukuma grafa elementu kā atslēgu un iezīmi kā vērtību. Apstrādājot piemēra izsaukumu grafu, tiek iegūtas iezīmes, kas ir parādītas 6.3. tabulā.

6.3. tabula

Definētās iezīmes

Atslēga	Iezīme
Process: 3	L1
Process: 6	L2
Process: 8	L3
LS: 1	L4

Kopumā tiek definētas 4 iezīmes, kas norāda uz procesa izsaukumu grafa elementiem, uz kuriem var notikt pārejas. Tā kā iezīmju definēšanas algoritms neļauj vienam un tam pašam grafa elementam definēt vairākas iezīmes, to rezultējošais skaits ir mazs.

Pēc iezīmju definēšanas notiek zarošanās definēšana, kuras izpildes rezultāts ir parādīts 6.4. tabulā. Zarošanās tiek definētas kā atslēga, kas ir procesu izsaukumu grafa elements, zarošanās tips – nosacījuma vai beznosacījuma un iezīme, kas ir zarošanās mērķis. Nosacījuma zarošanās gadījumā tiek norādīts arī pārbaudāmā mainīgā indekss.

6.4. tabula

Definētās zarošanās

Atslēga	Zarošanas tips	Mainīgais	Iezīme
Process: 4	JumpIfNot	18	L1
Process: 3	Jump	-	L2
LS: 1	JumpIf	21	L3
Process: 7	JumpIfNot	22	L2
Process: 8	JumpIf	24	L2
Process: 6	Jump	-	L4

Kopumā tiek definētas 6 zarošanās. No tām divas ir beznosacījuma, četras – nosacījuma. Pēc zarošanās definēšanas ir iespējams apstrādāt visu iegūto informāciju, veicot koda ģenerēšanu. Sākumā tiek apstrādāta mainīgo tabula, un tiek definētas mainīgo definēšanas instrukcijas, kas šajā gadījumā ir divdesmit piecas. Šī procesa izpildes rezultāts ir parādīts 6.7. att.

```
Var<1, (D4, phone), String>
Var<2, (D4, phoneType), String>
Var<3, (D5, users), User Information[]>
Var<4, (D5, phoneType), String>
Var<5, (D6, user), User Information>
Var<6, (D6, phoneType), String>
Var<7, (D1, phone), String>
Var<8, (D1, type), String>
Var<9, (D3, error), String>
Var<10, (D3, found), Boolean>
Var<11, (D9, user), Found User>
Var<12, (D9, found), Boolean>
Var<13, (D10, found), Boolean>
Var<14, (D7, user), User Information>
Var<15, (D7, phoneType), String>
Var<16, (Process: 1), GetPhoneNumberResult>
Var<17, (Process: 2), ValidatePhoneNumberResult>
Var<18, (Invoke 4), Boolean>
Var<19, (Process: 4), PrepareValidationErrorResult>
Var<20, (Process: 3), FindPhonesInDBResult>
Var<21, (LE: 3), Boolean>
Var<22, (Loop 1), Boolean>
Var<23, (Process: 7), PeekNextUserResult>
Var<24, (LB: ∅), Boolean>
Var<25, (Process: 8), MatchPhoneResult>
```

6.7. att. Mainīgo definēšanas instrukcijas

Var redzēt, ka mainīgo nosaukumu vietā tiek definēta informācija par to, kas ir šī mainīgā avots. Tas ir veikts ar nolūku, jo atkarībā no izvēlētās mērķa programmēšanas valodas var tikt izmantota atšķirīga notācija. Papildus, tas dod iespēju piedāvāt lietotājam definēt

mainīgo nosaukumus, uzdodot tam attiecīgos jautājumus, kas, piemēram, var būt formulēti kā "Kā ir jānosauc mainīgais, kas atbild par procesa X izpildi". Ir iespējams definēt mainīgo nosaukumus, arī no procesu izpildes nosacījumiem – pārveidojot tos par attiecīgajiem nosaukumiem. Tā kā ir pieņemami vairāki varianti mainīgo nosaukumu noteikšanai, tiek piedāvāts šajā solī mainīgo nosaukumus nedefinēt, bet ievietot visu nepieciešamo informāciju, lai to būtu iespējams izdarīt vēlāk.

Kad visi mainīgie ir definēti, tiek veikta instrukciju secības ģenerēšana pēc autora iepriekš definētā algoritma, kas ņem vērā informāciju par iezīmēm un iespējamām zarošanām un apvieno to ar minimizētā procesa izsaukumu grafa saturu. Šī soļa izpildes rezultāts ir parādīts 6.8. att.

```
        Invoke<Get Phone Number, [], 16>
        GetField<16, phone, 7>
        GetField<16, type, 8>
        Invoke<Validate Phone Number, [7, 8], 17>
        GetField<17, type, 1>
        GetField<17, type, 2>
        Check<18, Invalid Phone Number>
        JumpIfNot<18, L1>
        Invoke<Prepare Validation Error, [], 19>
        GetField<19, error, 9>
        GetField<19, found, 10>
        Jump<L2>

Label<L1>
        Invoke<Find Phones In DB, [1, 2], 20>
        GetField<20, users, 3>
        GetField<20, phoneType, 4>
        GetField<20, user, 5>
        GetField<20, phoneType, 6>
        Check<21, (LE: 3)>
        JumpIf<21, L3>

Label<L4>
        Check<22, Results are not empty and no match>
        JumpIf<22, L2>
        Invoke<Peek Next User, [3, 4], 23>
        GetField<23, found, 13>
        GetField<23, user, 14>
        GetField<23, phoneType, 15>
        Check<24, LB: ∅>
        JumpIf<24, L2>

Label<L3>
        Invoke<Match Phone, [5, 6, 14, 15], 25>
        GetField<25, user, 11>
        GetField<25, found, 12>
        Jump<L4>

Label<L2>
        Invoke<Output Search Result, [9, 10, 11, 12, 13], []>
```

6.8. att. Uzģenerētā instrukciju secība

Promocijas darba autors piedāvā veikt šīs instrukciju secības analīzi. Tā sākas ar instrukcijām:

```
Invoke<Get Phone Number, [], 16>
GetField<16, phone, 7>
GetField<16, type, 8>
```

Šīs instrukcijas atbilst procesa `Get Phone Number` izpildei un tā izpildes rezultāta ievietošanai attiecīgajos mainīgajos. Var redzēt, ka procesa izpildes rezultāts sākumā tiek ierakstīts pagaidu mainīgajā, kas ir noteiktas rezultāta klases vērtība. Pēc tam no šī pagaidu mainīgā tiek iegūtas attiecīgo atribūtu vērtības, kas tiek ierakstītas mainīgajos, kas atbilst attiecīgajām datu plūsmām. Var redzēt, ka process nesaņem nekādus parametrus savai izpildei, bet atgriež rezultātu.

Nākamās instrukcijas ir

```
Invoke<Validate Phone Number, [7, 8], 17>
GetField<17, type, 1>
GetField<17, type, 2>
```

Un tās atbilst nākamā procesa izpildei. Šeit process gan saņem, gan arī atgriež datus, kas tiek ņemti (vai ierakstīti) no attiecīgajiem mainīgajiem, kas atbilst konkrētām datu plūsmām.

Divas nākamas instrukcijas:

```
Check<18, Invalid Phone Number>
JumpIfNot<18, L1>
```

Veic procesa izpildes nosacījuma pārbaudi un, ja tas netiek izpildīts, veic kontroles nodošanu. Šeit procesa izpildes nosacījums ir definēts pašā modelī, un līdz ar to, arī instrukciju līmenī tas ir norādīts. Gadījumā, kad attiecīgais nosacījums izpildās, tiek izpildītas nākamās instrukcijas, kas ir:

```
Invoke<Prepare Validation Error, [], 19>
GetField<19, error, 9>
GetField<19, found, 10>
Jump<L2>
```

Var redzēt kartējā procesa izsaukumu, tā izpildes rezultāta ierakstu attiecīgajos mainīgajos un kontroles nodošanu bez nosacījuma uz iezīmi `L2`. Līdzīgi ir iespējams veikt arī turpmāko instrukciju analīzi, tomēr promocijas darba autors vēlas pievērst uzmanību divām nosacījumu pārbaudēm:

```
Check<21, (LE: 3)>
Check<24, LB: ∅>
```

Šīs pārbaudes tika uzģenerētas automātiski, un tās izriet no biznesa procesu modeļa un tam atbilstošā procesa izsaukumu grafa struktūras. Līdz ar to šīm nosacījumu pārbaudēm tiek definēts attiecīgais merķa biznesa procesu izsaukumu grafa elements, uz kuru tiek nodota

kontrole. Pārveidojot instrukciju secību programmatūras kodā, būs iespējams šajās vietās veikt noteiktas aizvietošanas, kas nozīmē nepieciešamību pēc uzģenerētā koda labošanas vai arī modeļa papildināšanas. Tas nozīmē to, ka koda ģenerēšanas laikā var tikt atklātas avota modeļa nepilnības, kas savukārt var norādīt uz nepieciešamību veikt papildus analīzi un modelēšanu.

6.2. Algoritma darbības rezultātu validācija

Lai pārliecinātos, ka uzģenerētā instrukciju secība atbilst avota modelim, ir nepieciešams veikt dotā algoritma rezultātu validāciju. Lai gan to ir iespējams veikt dažādos veidos – piemēram, salīdzināt uzģenerēto kodu ar kodu, ko uzrakstītu izstrādātājs, vai arī analizēt instrukciju secības, promocijas darba autors piedāvā izmantot risinājumu, kas balstās uz divu grafu savstarpēju salīdzināšanu. Kā pirmais grafs tiek izmantots procesu izsaukumu grafs, kas satur visu informāciju no sākotnējā biznesa procesu modeļa. Otro grafu tiek piedāvāts izveidot no uzģenerētās instrukciju secības, un to ir piedāvāts nosaukt par transformācijas validācijas grafu.

Transformācijas validācijas grafa izveidošanai ir nepieciešams veikt instrukciju secības analīzi. Veicot šo analīzi, sākumā tiek izveidotas visas nepieciešamās grafa virsotnes, kas ir vai nu procesu izsaukumu instrukcijas, vai iezīmes. Šo darbību ir iespējams aprakstīt ar pseidokoda, kas ir parādīts 6.9. att., palīdzību.

```
validationGraph = new Graph()

for inst in instructions:
       if inst is Invoke or inst is Label:
              validationGraph.addVertex(inst)
```

6.9. att. Transformācijas validācijas grafa virsotņu definēšana

```
currentEl = ∅

for inst in instructions:
       if inst is Invoke or inst is Label:
              validationGraph.addEdge(currentEl, inst)
              currentEl = inst

       if inst is Jump or inst is JumpIf or inst is JumpIfNot:
              l = inst.target
              validationGraph.addEdge(currentEl, l)
```

6.10. att. Transformācijas validācijas grafa loku definēšana

Kad visas šī grafa virsotnes ir definētas, ir nepieciešams definēt tā lokus. Šeit loki ir ne kas cits, kā iespējamās kontroles nodošana – ja procesa B izsaukums seko procesa A izsaukumam, tad grafam tiek pievienots loks A → B. Līdzīgi tiek apstrādātas arī zarošanās instrukcijas – ja ir iespējama zarošanās uz iezīmi X, tad tiek pievienots attiecīgais loks. Apstrādes gaitā ir nepieciešamas saglabāt norādi uz pēdējo analizēto instrukciju, lai būtu iespējams noteikt ne tikai uz kuru iezīmi notiek zarošanās, bet arī, no kuras vietas. Attiecīgā algoritma pseidokods ir parādīts 6.10. att.

Pēc uzģenerētās instrukciju secības analīzes tiek iegūts transformācijas validācijas grafs, kas satur informāciju par visiem iespējamajiem ceļiem jeb procesu izsaukumu secībām, kas ir iespējamas, veicot attiecīga biznesa procesa apstrādi. Var redzēt, ka šim grafam ir jāatbilst sākotnējam procesu izsaukumu grafam – ja procesu izsaukumu grafā eksistē divas virsotnes – A un B, un starp tām eksistē divi iespējamie ceļi – A → C → B un A → D → B, tad abām šīm virsotnēm ir jāeksistē arī validācijas grafā, un abiem ceļiem ir jābūt iespējamiem. Gadījumā, ja tas netiek izpildīts, transformācija tiek uzskatīta par nekorektu.

Veicot abu grafu salīdzināšanu, ir arī jāņem vērā fakts, ka validācijas grafs satur papildus virsotnes, kas atbilst iezīmēm. Līdz ar to nevar veikt salīdzināšanu, izmantojot primitīvu pieeju – pārbaudīt, vai katram procesu izsaukumu grafa lokam atbilst validācijas grafa loks. Ir nepieciešams sākumā pārbaudīt, vai starp divām virsotnēm, kuras savieno attiecīgais loks, vispār eksistē ceļš. Pēc tam ir nepieciešams pārbaudīt, vai šis ceļš satur tikai attiecīgos sākuma un beigu procesus un, iespējams, iezīmes. Gadījumā, kad dotajā ceļā parādās kāds cits process, transformācija tiek uzskatīta par neveiksmīgu.

Tātad, ir nepieciešams atrisināt divus uzdevumus. Lai noteiktu, vai starp divām grafa virsotnēm eksistē ceļš, var, piemēram, izmantot Floida-Uoršella algoritmu [25], kas izveidojot īsāko ceļu matricu, ļauj pārbaudīt doto prasību. Ir iespējams izmantot arī citus algoritmus – piemēram, IDDFS (angl. *Iterative Deepening Depth-First Search* – iteratīvi padziļinošo pārmeklēšanu dziļumā) [56], kas var būt noderīga, veicot lielu grafu apstrādi, jo Floida-Uoršella algoritmam ir raksturīga $O(n^3)$ sarežģītība, kamēr IDDFS ļauj ierobežot pārmeklēšanas dziļumu, tāda veidā samazinot nepieciešamo pārbaužu skaitu. Bet Floida-Uoršella algoritms tiek izpildīts vienu reizi, izveidojot matricu, kas turpmāk var tikt izmantota visa grafa analīzei.

Var turpināt IDDFS un Floida-Uoršella algoritmu priekšrocību un trūkumu salīdzinājumu, tomēr promocijas darba autors vēlas uzsvērt, ka šajā gadījumā konkrētais izmantotais algoritms nav svarīgs – galvenais ir tā iespējamība noteikt, vai starp divām grafa virsotnēm eksistē ceļš. Līdz ar to abi algoritmi tiek uzskatīti par piemērotiem.

Kad ir noteikts, ka starp divām validācijas grafa virsotnēm eksistē ceļš, ir nepieciešams analizēt, vai šis ceļš satur tikai dotās virsotnes un, iespējams, vairākas iezīmes. Lai atrisināto šo uzdevumu, var izmantot atgriezmeklēšanas (angl. *backtracking*) [55] paņēmienu, kas ļauj pārbaudīt, vai ir iespējams no gala virsotnes nonākt līdz sākotnējai, apmeklējot tikai iezīmju virsotnes. Attiecīgā algoritma pseidokods ir parādīts 6.11. att.

```
function backtrack(graph, from, target):
	vertexQueue = []
	closed = []

	vertexQueue += from

	while vertexQueue is not empty:
		v = vertexQueue.removeFirst()

		closed += v
		if v is target:
			return true

		for v1 in graph.getPredecessors(v):
			if v1 is target:
				return true

			if not v1 in closed and v1 is Label:
				vertexQueue += v1

	return false
```

6.11. att. Atgriezmeklēšana, apmeklējot tikai iezīmes

Pēc abu uzdevumu – ceļa eksistēšanas pārbaudes un tā validācijas – atrisināšanas ir iespējams definēt validācijas algoritmu, kas veic procesu izsaukumu grafa un transformācijas validācijas grafa salīdzināšanu ar mērķi noteikt, vai visa informācija, kas bija atrodama sākumā, tiek saglabāta pēc transformācijas.

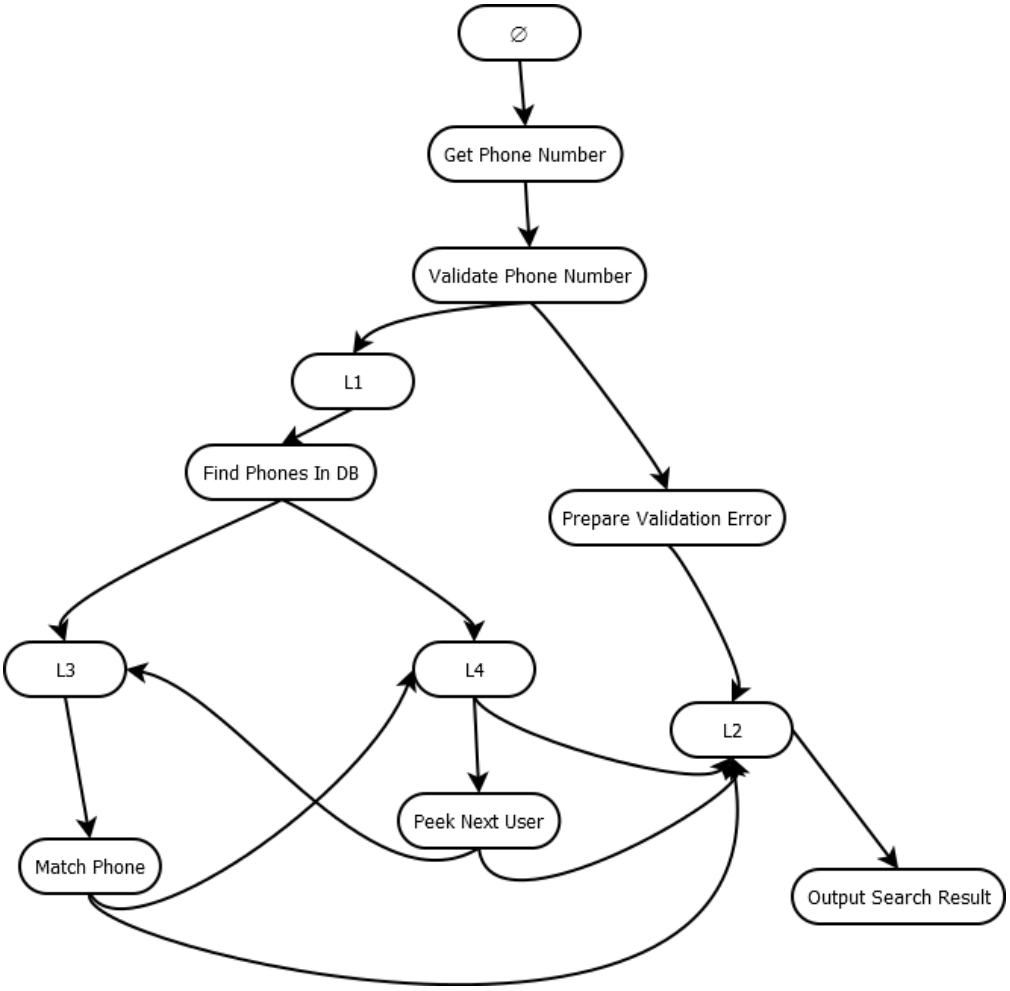
```
function validateTransformation(processInvocationGraph, validationGraph):
	for e in processInvocationGraph.edges:
		src = e.source
		dst = e.dest

		if not src in validationGraph or not dst in validationGraph:
			return false

		if not path exists in validationGraph: src → dst:
			return false

		if not backtrack(validationGraph, dst, src):
			return false

	return true
```

6.12. att. Transformācijas validācijas algoritms

Šāds algoritms ir parādīts 6.12. att., un tas veic procesa izsaukumu loku analīzi. Katrs loks tiek analizēts šādi:

- Ja transformācijas validācijas grafā nav atrodama attiecīgā loka sākuma vai beigu virsotne, tad transformācija ir neveiksmīga.
- Ja transformācijas validācijas grafā neeksistē ceļš starp loka sākuma un loka beigu virsotni, tad transformācija ir neveiksmīga.
- Ja ceļš starp loka sākumu un loka beigu virsotnēm satur ne tikai iezīmes, tad transformācija ir neveiksmīga.

Gadījumā, ja visi procesu izsaukuma grafa loki tika apstrādāti, un netika atrasta neviena iepriekš aprakstītā kļūda, transformācija tiek uzskatīta par veiksmīgu.



6.13. att. Piemēra transformācijas validācijas grafs

6.13. att. ir parādīts transformācijas validācijas grafs, kas ir izveidots analizējot piedāvāto piemēru. Salīdzinot to ar 6.3. att., var redzēt, ka visas validācijas prasības ir izpildītas, līdz ar to var secināt, ka transformācijas ir korektas..

6.3. Java koda iegūšana no instrukciju secības

Pēc instrukciju secības ģenerēšanas to var izmantot programmatūras koda iegūšanai jebkurā programmēšanas valodā. Kā tika minēts 5.1. sadaļā, šajā darbā par mērķa programmēšanas valodu tika izvelēta Java [50]. Šajā promocijas darba sadaļā tiek aprakstīta viena no Java valodas koda iegūšanas iespējām.

Programmēšanas valoda Java tiek kompilēta baitu kodā, pēc kura analīzes tika izveidota iepriekš aprakstītā instrukciju kopa. Tātad, ir iespējama vismaz viena transformācija – aizvietojot katru uzģenerēto instrukciju ar vienu vai vairākiem JVM baitu koda elementiem, ir iespējams iegūt to pašu rezultātu, kas tiek iegūts pēc valodas Java kompilācijas. Rezultātā ir iespējams iegūt kompilētas Java valodas klases, kas gan nav programmatūras kods.

Tomēr JVM baitu kodu var izmantot arī programmatūras koda iegūšanai. Pateicoties tam, ka JVM baitu kods ir salīdzinoši vienkāršāks par tipiskām asambleru valodām – piemēram, JVM baitu kods satur ap 200 instrukcijām [16], kamēr Intel procesoru asamblera valoda [46] satur ap 2000 – to ir iespējams dekompilēt, iegūstot sākotnējo programmatūras kodu.

Dekompilācija ir viena no reversās inženierijas tehnoloģijām, kas ir paredzēta sākotnējā programmatūras koda iegūšanai no kompilētiem artefaktiem [21]. Plašākā nozīmē tas ir zināšanu vai arī sākotnējo artefaktu izgūšanas process no jebkā, ko ir izveidojis cilvēks. Lai gan sākumā reversā inženierija un dekompilācija var likties nelikumīga – jo tā, iespējams, ir autortiesību pārkāpums, eksistē vairāki tās likumīgās pielietošanas piemēri:

- Tā var būt nepieciešama defektu labošanai programmatūrā, kura tika izstrādāta pirms kāda laika un tās pirmkods vairs nav pieejams.
- Nepieciešamība pēc reversās inženierijas var rasties arī, ja ir vajadzīgs iegūt informāciju no programmatūras artefakta (piemēram, kriptogrāfiskās atslēgas). Ir būtisks fakts, ka runa ir par pašizstrādātiem artefaktiem, kuriem jebkādu iemeslu dēļ nav pieejams pirmkods.
- Cits likumīgas reversās inženierijas piemērs ir datora vīrusu analīze, ko var veikt antivīrusu programmatūras izstrādātāji ar mērķi saprast, kā ļaunprātīgā programmatūra darbojas, un kādas darbības ir nepieciešamas cīņai ar to.

Eksistē arī citi piemēri, tomēr šajā darbā promocijas darba autors vēlas izmantot dekompilācijas iespējas programmatūras koda iegūšanai no uzģenerētās instrukciju secības.

Pirms dekompilācijas veikšanas ir nepieciešams izveidot bināro artefaktu (jeb kompilēto Java valodas klasi), kas atbilst uzģenerētajai instrukciju secībai. Tā kā autors ir izveidojis definēto instrukciju kopu balstoties uz JVM baitu kodu, ir iespējams izveidot vienkāršus likumus instrukciju pārveidošanai baitu koda elementos. Faktiski katra no piedāvātajām instrukcijām atbilst baitu koda elementu secībai. Šī informācija ir apkopota 6.5. tabulā. Var redzēt, ka vairākos gadījumos instrukciju var pārveidot par vienu JVM baitu koda elementu, tomēr eksistē arī tādas instrukcijas, kurām vispār neatbilst JVM baitu koda elements, vai arī ir nepieciešams definēt baitu koda elementu secību. Sīkāku informāciju par JVM baitu kodu var atrast [16] – šajā darbā autors tikai īsi apraksta ģenerējamo baitu kodu.

6.5. tabula

Instrukciju kartēšana uz JVM baitu kodu

Instrukcija	JVM baitu kods
`Label<Name>`	Iezīmes ievietošana – tai neatbilst specifiska baitu koda instrukcija. Faktiski JVM gadījumā iezīmes var saukt par virtuālām – tām atbilst nobīdes no metožu sākumiem [16]. Tomēr daudzas baitu koda manipulēšanas bibliotēkas, piemēram, ASM [5] ļauj tās izmantot.
`Var<Id, Name, Type>`	Līdzīgi kā iepriekš, speciālas JVM baitu koda instrukcijas neeksistē, tiek izmantotas speciālas tabulas [15]. Atkal, baitu koda manipulēšanas bibliotēkas ļauj šo informāciju definēt.
`GetField<Object, Field, Target>`	Sākumā ir nepieciešams ielādēt norādi uz `Object` objektu: `ALOAD` *`Object`* Tad tiek iegūts tā atribūts `Field`: `GETFIELD` *`Field`* Kas tiek saglabāts `Target` mainīgajā: `ASTORE` *`Target`*
`PutField<Object, Field, Source>`	Sākumā ir nepieciešams ielādēt norādi uz `Object` objektu: `ALOAD` *`Object`* Tad tiek ielādēts `Source` mainīgais: `ALOAD` *`Source`* Pēc tam tiek ierakstīts attiecīgais atribūts `Field`: `PUTFIELD` *`Field`*

6.5. tabula (turpinājums)

`Invoke<Method, Inputs, Output?>`	Sākumā ir nepieciešams ielādēt norādi uz `this` objektu:
	`ALOAD 0`
	Tad ir nepieciešams ielādēt norādes uz visiem argumentiem:
	`∀ v ∈ Inputs: ALOAD v`
	Pēc tam tiek ievietota attiecīgās metodes izsaukuma instrukcija:
	`INVOKESPECIAL Method …`
	Ja metode atgriež vienu vai vairākas vērtības, tiek ievietota rezultāta saglabāšanas instrukcija:
	`ASTORE Output`
	Gadījumā, kad tiek atgriezts objekts, ir nepieciešams veikt tā atribūtu saglabāšanu lokālajos mainīgajos:
	`∀ f ∈ Output.fields:`
	`  ALOAD Output`
	`  GETFIELD f`
	`  ASTORE var`
	Kur `var` ir attiecīga lokālā mainīgā indekss.

`Jump<Label>`	`GOTO Label`

`Check<Var, Guard>`	Tā kā metožu izpildes nosacījumi tiek definēti rakstiski, ir nepieciešams definēt komentāru, kas varētu palīdzēt pareiza nosacījuma definēšanai turpmāk. JVM baitu kods komentārus nesatur, līdz ar to ir nepieciešams kādā veidā šo informāciju pierakstīt. Promocijas darba autors piedāvā šo informāciju saglabāt izmantojot `Boolean` klases metodi `valueOf()` un pēc tam šādus gadījumus apstrādāt – jo ir zināms, ka citās vietās šādas konstrukcijas neparādīsies. Tātad uzdevums ir ģenerēt secību:
	`LDC Guard`
	`INVOKESTATIC Boolean.valueOf`
	`ISTORE Var`

`JumpIf<Var, Label>`	Ir nepieciešami divi soļi – mainīgā ielādēšana:
	`ILOAD Var`
	Un pārbaude, vai tas ir patiess, kam seko kontroles nodošana:
	`IFEQ Label`

`JumpIfNot<Var, Label>`	`ILOAD Var` `IFNE Label`

`Return<Var?>`	Ja `Var` ir definēts, tad vajag to ielādēt:
	`ALOAD Var`
	Un atgriezt:
	`ARETURN`
	Citādi, iziet no metodes:
	`RETURN`

Lai būtu iespējams instrukciju secību pārveidot par baitu koda elementiem, var izmantot vairākas bibliotēkas, kas ir paredzētas darbam ar JVM baitu kodu. Var arī definēt visus nepieciešamos elementus bez tam – izveidojot nepieciešamo pārveidotāju, jo kopējais izmantoto elementu skaits ir salīdzinoši mazs. Promocijas darba autors šiem nolūkiem izmantoja bibliotēku ASM [5], kas ir plaši izmantota bibliotēka – piemēram, tās kods ir sastopams arī vienā no JVM implementācijām – OpenJDK [89], kā arī citos projektos.

```
LOCALVARIABLE this LFindUserByPhoneNumber; L0 L6 0
LOCALVARIABLE dBQueryPhone LString; L0 L6 1
LOCALVARIABLE dBQueryPhoneType LString; L0 L6 2
LOCALVARIABLE usersWithGivenPhonePhoneType LString; L0 L6 3
LOCALVARIABLE usersWithGivenPhoneUsers LUserInformation[]; L0 L6 4
LOCALVARIABLE singleFoundResultUser LUserInformation; L0 L6 5
LOCALVARIABLE singleFoundResultPhoneType LString; L0 L6 6
LOCALVARIABLE userInputType LString; L0 L6 7
LOCALVARIABLE userInputPhone LString; L0 L6 8
LOCALVARIABLE searchFailedFound LBoolean; L0 L6 9
LOCALVARIABLE searchFailedError LString; L0 L6 10
LOCALVARIABLE matchedUserUser LFoundUser; L0 L6 11
LOCALVARIABLE matchedUserFound LBoolean; L0 L6 12
LOCALVARIABLE noMatchFound LBoolean; L0 L6 13
LOCALVARIABLE userUser LUserInformation; L0 L6 14
LOCALVARIABLE userPhoneType LString; L0 L6 15
LOCALVARIABLE getPhoneNumberResult LGetPhoneNumberResult; L0 L6 16
LOCALVARIABLE validatePhoneNumberResult LValidatePhoneNumberResult; L0 L6 17
LOCALVARIABLE Invalid_Phone_Number Z L0 L6 18
LOCALVARIABLE prepareValidationErrorResult LPrepareValidationErrorResult; L0 L6 19
LOCALVARIABLE findPhonesInDBResult LFindPhonesInDBResult; L0 L6 20
LOCALVARIABLE should_jump_further Z L0 L6 21
LOCALVARIABLE Results_are_not_empty Z L0 L6 22
LOCALVARIABLE peekNextUserResult LPeekNextUserResult; L0 L6 23
LOCALVARIABLE should_exit_loop Z L0 L6 24
LOCALVARIABLE matchPhoneResult LMatchPhoneResult; L0 L6 25
```

6.14. att. Mainīgo definēšana JVM baitu kodā

Pielietojot dotos kartēšanas likumus uzģenerētajai instrukciju secībai, kas tika apskatīta iepriekšējā piemēra ietvaros, tiek iegūts JVM baitu kods, ko ir iespējams sadalīt vairākās sekcijās analīzes vienkāršošanai. 6.14. att. ir parādīta pirmā no analizējamām baitu koda sekcijām – mainīgo definēšana, kas nav sastāvdaļa no metodes izpildāmā koda, bet ir daļa no tā metadatiem. Var redzēt, ka mainīgo nosaukumi tika uzģenerēti, izmantojot informāciju par attiecīgo datu plūsmu, kas ir mainīgā ģenerēšanas avots, vai arī balstoties uz metožu izpildes nosacījumiem. Autors vēlas pievērst uzmanību, ka dotais mainīgo nosaukumu definēšanas algoritms nav vienīgais iespējamais, un to vienmēr var modificēt vai pielāgot atkarībā no papildu prasībām. Var redzēt, ka tika definēti arī divi mainīgie - `should_jump_further` un `should_exit_loop`, kas ir zarošanās analīzes rezultāts. Šie mainīgie atbilst vietām, kur ir

iespējama zarošanās, bet nav pietiekami daudz informācijas konkrēta nosaukuma ģenerēšanai. Mainīgie ir pierakstīti formā:

```
LOCALVARIABLE Name Type; LS LE Idx
```

Kur `Name` ir mainīgā nosaukums, `Type` – tā datu tips, `LS` un `LE` nosaka mainīgā redzamības apgabalu, tās ir iezīmes, starp kurām mainīgais ir pieejams. `Idx` savukārt ir attiecīgā mainīgā indekss.

Nākamā uzģenerētā baitu koda sekcija ir parādīta 6.15. att., un to var nosaukt par metodes prologu. Šeit tiek definēta metodes sākuma iezīme `L0`, kas tiek izmantota mainīgo redzamības noteikšanai. Tālāk notiek visu lokālo mainīgo inicializācija, kas var tikt veikta vienā no diviem veidiem:

- Gadījumā, kad lokālais mainīgais atbilst datiem, kas tiek izmantoti attiecīgas biznesa loģikas realizācijai, tam tiek piešķirta `NULL` vērtība. Lai to izdarītu, ir nepieciešamas divas baitu koda instrukcijas:
  - `ACONST_NULL` ielādē `NULL` vērtību;
  - `ASTORE X` saglabā to mainīgajā `X`.
- Ja mainīgais ir zarošanās nosacījums, tam atbilst `boolean` tipa vērtība, kas tiek inicializēta ar vērtību `false`. Tam ir nepieciešamas divas baitu koda instrukcijas:
  - `ICONST_0` ielādē `0` vērtību;
  - `ISTORE X` saglabā to mainīgajā `X`.

Var redzēt, ka JVM baitu koda `boolean` tipa vērtības tiek uzglabātas kā skaitļi jeb viena baita mainīgie. Prologa izpildes rezultātā visi lokālie mainīgie ir inicializēti un izmantojami.

```
    L0
     ACONST_NULL
     ASTORE 1
     ACONST_NULL
     ASTORE 2
     ACONST_NULL
     ASTORE 3
     ACONST_NULL
     ASTORE 4
     ACONST_NULL
     ASTORE 5
     ACONST_NULL
     ASTORE 6
     ACONST_NULL
     ASTORE 7
     ACONST_NULL
     ASTORE 8
     ACONST_NULL
     ASTORE 9
     ACONST_NULL
     ASTORE 10
     ACONST_NULL
     ASTORE 11
     ACONST_NULL
     ASTORE 12
     ACONST_NULL
     ASTORE 13
     ACONST_NULL
     ASTORE 14
     ACONST_NULL
     ASTORE 15
     ACONST_NULL
     ASTORE 16
     ACONST_NULL
     ASTORE 17
     ICONST_0
     ISTORE 18
     ACONST_NULL
     ASTORE 19
     ACONST_NULL
     ASTORE 20
     ICONST_0
     ISTORE 21
     ICONST_0
     ISTORE 22
     ACONST_NULL
     ASTORE 23
     ICONST_0
     ISTORE 24
     ACONST_NULL
     ASTORE 25
```

6.15. att. Metodes prologs

Pēc metodes prologa seko pats metodes baitu kods, kas atbilst uzģenerētajai instrukciju secībai, un ko var redzēt 6.16. att. un 6.17. att. Var salīdzināt doto baitu kodu un uzģenerēto instrukciju secību un redzēt, ka abi artefakti atbilst viens otram (ņemot vērā kartēšanas likumus).

```
ALOAD 0
INVOKESPECIAL FindUserByPhoneNumber.getPhoneNumber ()LGetPhoneNumberResult;
ASTORE 16
ALOAD 16
GETFIELD GetPhoneNumberResult.userInputType : LString;
ASTORE 7
ALOAD 16
GETFIELD GetPhoneNumberResult.userInputPhone : LString;
ASTORE 8
ALOAD 0
ALOAD 7
ALOAD 8
INVOKESPECIAL FindUserByPhoneNumber.validatePhoneNumber
   (LString;LString;)LValidatePhoneNumberResult;
ASTORE 17
ALOAD 17
GETFIELD ValidatePhoneNumberResult.dBQueryPhone : LString;
ASTORE 1
ALOAD 17
GETFIELD ValidatePhoneNumberResult.dBQueryPhoneType : LString;
ASTORE 2
LDC "Invalid Phone Number"
INVOKESTATIC java/lang/Boolean.parseBoolean (Ljava/lang/String;)Z
ISTORE 18
ILOAD 18
IFEQ L1
ALOAD 0
INVOKESPECIAL FindUserByPhoneNumber.prepareValidationError ()
   LPrepareValidationErrorResult;
ASTORE 19
ALOAD 19
GETFIELD PrepareValidationErrorResult.searchFailedFound : LBoolean;
ASTORE 9
ALOAD 19
GETFIELD PrepareValidationErrorResult.searchFailedError : LString;
ASTORE 10
GOTO L2
L1
ALOAD 0
ALOAD 1
ALOAD 2
INVOKESPECIAL FindUserByPhoneNumber.findPhonesInDB
   (LString;LString;)LFindPhonesInDBResult;
ASTORE 20
ALOAD 20
GETFIELD FindPhonesInDBResult.usersWithGivenPhonePhoneType : LString;
ASTORE 3
ALOAD 20
GETFIELD FindPhonesInDBResult.usersWithGivenPhoneUsers : LUserInformation[];
ASTORE 4
ALOAD 20
GETFIELD FindPhonesInDBResult.singleFoundResultUser : LUserInformation;
ASTORE 5
ALOAD 20
GETFIELD FindPhonesInDBResult.singleFoundResultPhoneType : LString;
ASTORE 6
LDC "should jump further"
```

6.16. att. Metodes baitu kods

```
    INVOKESTATIC java/lang/Boolean.parseBoolean (Ljava/lang/String;)Z
    ISTORE 21
    ILOAD 21
    IFNE L3
   L4
    LDC "Results are not empty"
    INVOKESTATIC java/lang/Boolean.parseBoolean (Ljava/lang/String;)Z
    ISTORE 22
    ILOAD 22
    IFEQ L2
    ALOAD 0
    ALOAD 3
    ALOAD 4
    INVOKESPECIAL FindUserByPhoneNumber.peekNextUser
       (LString;[LUserInformation;)LPeekNextUserResult;
    ASTORE 23
    ALOAD 23
    GETFIELD PeekNextUserResult.userUser : LUserInformation;
    ASTORE 14
    ALOAD 23
    GETFIELD PeekNextUserResult.userPhoneType : LString;
    ASTORE 15
    ALOAD 23
    GETFIELD PeekNextUserResult.noMatchFound : LBoolean;
    ASTORE 13
    LDC "should exit loop"
    INVOKESTATIC java/lang/Boolean.parseBoolean (Ljava/lang/String;)Z
    ISTORE 24
    ILOAD 24
    IFNE L2
   L3
    ALOAD 0
    ALOAD 5
    ALOAD 6
    ALOAD 14
    ALOAD 15
    INVOKESPECIAL FindUserByPhoneNumber.matchPhone
       (LUserInformation;LString;LUserInformation;LString;)LMatchPhoneResult;
    ASTORE 25
    ALOAD 25
    GETFIELD MatchPhoneResult.matchedUserUser : LFoundUser;
    ASTORE 11
    ALOAD 25
    GETFIELD MatchPhoneResult.matchedUserFound : LBoolean;
    ASTORE 12
    GOTO L4
   L2
    ALOAD 0
    ALOAD 9
    ALOAD 10
    ALOAD 11
    ALOAD 12
    ALOAD 13
    INVOKESPECIAL FindUserByPhoneNumber.outputSearchResult
       (LBoolean;LString;LFoundUser;LBoolean;LBoolean;)V
    RETURN
   L6
```

6.17. att. Metodes baitu kods (turpinājums)

Papildus JVM baitu kods satur divas iezīmes – metodes sākuma un metodes beigu iezīmes, kas nosaka lokālo mainīgo redzamības apgabalu. Var redzēt, ka visi definētie mainīgie ir pieejami visā metodē.

Pēc baitu koda ģenerēšanas ir iespējams veikt tā dekompilāciju, lai iegūtu Java programmatūras kodu. Kā jau tika minēts, dekompilācijas veikšanai ir nepieciešams izmantot speciālas programmas, kas tiek sauktas par dekompilatoriem. Java valodas gadījumā eksistē vairākas šādas programmas, un promocijas darba autors ir veicis pētījumu [36], kura gaitā tika salīdzināti 4 dekompilatori – JAD [51], CFR [14], Procyon [64] un FernFlower [30]. Pētījuma laikā tika pārbaudīts, kā dotās programmas ir spējīgas dekompilēt jaunākās JVM instrukcijas, kā dotais dekompilators tiek licencēts, kā arī, vai ir iespējams paplašināt doto dekompilatoru, pievienojot paša definētu loģiku. Rezultātā, tika izvelēts Procyon dekompilators, kas arī tiek izmantots Java koda iegūšanai.

```
public AstNode visitAssignmentExpression(final AssignmentExpression node,
                 final Void data) {

    final Expression val = node.getRight();
    if (val instanceof InvocationExpression) {
        final MemberReferenceExpression target = (MemberReferenceExpression)
                 ((InvocationExpression) val).getTarget();

        final String memberName = target.getMemberName();
        if (Objects.equals("parseBoolean", memberName)) {
            final AstNode nextSibling = target.getNextSibling();
            if (nextSibling instanceof PrimitiveExpression) {
                final Object value = ((PrimitiveExpression) nextSibling)
                 .getValue();

                if (value != null) {
                    final Comment comment = new Comment(String.valueOf(value));
                    final PrimitiveExpression ex = new
                          PrimitiveExpression(Expression.MYSTERY_OFFSET, false);

                    node.setRight(ex);
                    final AstNode pa = node.getParent();

                    pa.insertChildBefore(node, comment, Roles.COMMENT);
                    pa.insertChildBefore(comment, NewLineNode.create(),
                          Roles.NEW_LINE);
                }
            }
        }
    }

    return super.visitAssignmentExpression(node, data);
}
```

6.18. att. Nosacījumu aizvietošana pēc dekompilācijas

Var redzēt, ka uzģenerētajā baitu kodā noteikumu pārbaude tiek realizēta izmantojot Java valodas objekta `Boolean` iebūvēto metodi `valueOf()`, kas saņem simbolu virkni kā parametru un atgriež attiecīgo Būla mainīgā vērtību atkarībā no padotās simbolu virknes. Lai gan tas ļauj iegūt sintaktiski korektu Java kodu pēc dekompilācijas, rezultātā tiek pazaudēta semantika, jo minētā metode var tikai un vienīgi apstrādāt `"true"` simbolu virkni. Līdz ar to

autors piedāvā veikt šo izteiksmju aizvietošanu ar vērtību `false`, pievienojot komentāru ar oriģinālo pārbaudāmo priekšnosacījumu. Procyon gadījumā šādas loģikas realizācijai ir nepieciešams realizēt Apmeklētāja (angl. *Visitor*) [28] klasi, kas varēs apstrādāt dotās iegūtā AST sastāvdaļas. Šādas klases realizācija izvēlētā dekompilatora gadījumā ir pietiekami primitīvs uzdevums. Daļa no realizētās klases koda, kas atbild par nosacījumu aizvietošanu, ir parādīta 6.18. att. Dotais kods pārbauda, vai tiek apstrādāta nepieciešamā konstrukcija, un gadījumā, kad tas tā ir, veic apstrādājamās AST sastāvdaļas modifikācijas, pievienojot komentāru un likvidējot metodes izsaukumu. Nepieciešamības gadījumā ir iespējams pievienot arī citas klases, kas veiks dekompilēta koda apstrādi un modificēšanu.

Pēc dekompilācijas veikšanas ir iespējams apvienot tās rezultātus ar procesu izsaukumu rezultātu klasēm, kā arī klasēm, kas tiek iegūtas no konceptiem, iegūstot programmatūras kodu Java programmēšanas valodā.

```
public class FoundUser {
    public String name;
    public String surname;
    public PhoneNumber phoneNumber;
}

public class PhoneNumber {
    public String type;
    public String value;
}

public class UserInformation {
    public String name;
    public String surname;
    public PhoneNumber[] phoneNumbers;
}
```

6.19. att. Domēna klases, kas tika iegūtas no konceptiem

6.19. att. ir parādītas domēna klases, kas tika iegūtas no konceptiem, var redzēt, ka to struktūra pilnībā atbilst sākotnējam modelim, un šādas domēna klases var tikt iegūtas vairākos veidos, kā ir parādīts iepriekšējos pētījumos – piemēram, [77],[82],[84]. Šeit domēna klases ir izveidotas tā, ka konceptu atribūti tiek pārveidoti par attiecīgo klašu publiskajiem atribūtiem. Lai gan tas var neatbilst pieņemtajiem programmēšanas valodas Java stila principiem (šeit netiek izmantotas tā saucamās *getter* un *setter* jeb *accessor* un *mutator* metodes), modernas izstrādes vides (piemēram, Eclipse [22] un Intellij IDEA[45]) ļauj šo trūkumu novērst, izmantojot iebūvētās koda ģenerēšanas un transformācijas iespējas. Pie tam, pēc promocijas darba autora pieredzes, dažādos projektos tiek pieņemtas dažādas vienošanās par koda stilu.

Līdz ar to, promocijas darba autors uzskata, ka dotās domēna klases ir iespējams izmantot, un, ja nepieciešams, modificēt atbilstoši koda stila prasībām.

6.20. att. ir parādītas klases, kas tika iegūtas, apstrādājot procesu izsaukumu rezultātus. Var redzēt, ka iegūtās Java klases atbilst sākotnēji iegūtajām klasēm, kas tika iegūtas, analizējot procesu izsaukumus (6.6. att.).

```
public class GetPhoneNumberResult {
    public String userInputType;
    public String userInputPhone;
}

public class PrepareValidationErrorResult {
    public Boolean searchFailedFound;
    public String searchFailedError;
}

public class MatchPhoneResult {
    public Boolean matchedUserFound;
    public FoundUser matchedUserUser;
}

public class FindPhonesInDBResult {
    public UserInformation[] usersWithGivenPhoneUsers;
    public String usersWithGivenPhonePhoneType;
    public UserInformation singleFoundResultUser;
    public String singleFoundResultPhoneType;
}

public class ValidatePhoneNumberResult {
    public String dBQueryPhone;
    public String dBQueryPhoneType;
}

public class PeekNextUserResult {
    public UserInformation userUser;
    public String userPhoneType;
    public Boolean noMatchFound;
}
```

6.20. att. Procesu izsaukumu rezultātu klases Java valodā

Sākotnējie biznesa procesi tika pārveidoti par metodēm, kas ir parādītas 6.21. att. Var redzēt, ka iegūtās metodes pagaidām nesatur nekādu biznesa loģiku, tomēr tām ir definētas pareizas signatūras, kas atbilst sākotnēji definētajai informācijai. Dotās metodes ir iespējams pievienot klasei, kas atbild par attiecīgajā biznesa procesu diagrammā definētās loģikas realizāciju.

```
private FindPhonesInDBResult findPhonesInDB(final String dBQueryPhoneType,
                                            final String dBQueryPhone) {
    return null;
}

private ValidatePhoneNumberResult validatePhoneNumber(final String userInputType,
                                                      final String userInputPhone)
{

    return null;
}

private void outputSearchResult(final Boolean searchFailedFound,
                                final String searchFailedError,
                                final FoundUser matchedUserUser,
                                final Boolean matchedUserFound,
                                final Boolean noMatchFound) {
}

private MatchPhoneResult matchPhone(final UserInformation singleFoundResultUser,
                                    final String singleFoundResultPhoneType,
                                    final UserInformation userUser,
                                    final String userPhoneType) {
    return null;
}

private GetPhoneNumberResult getPhoneNumber() {
    return null;
}

private PrepareValidationErrorResult prepareValidationError() {
    return null;
}

private PeekNextUserResult peekNextUser(final String usersWithGivenPhonePhoneType,
                                    final UserInformation[] usersWithGivenPhoneUsers)
{
    return null;
}
```

6.21. att. Metodes, kas tika iegūtas procesu transformācijas rezultātā

Pēdējais transformācijas rezultāts ir iegūts uzģenerētā baitu koda dekompilācijas procesa rezultātā. To var loģiski sadalīt vairākas daļās. 6.22. att. ir parādītas Java programmēšanas valodas mainīgo definīcijas, kas tiek izmantotas biznesa loģikas realizācijai. Var redzēt, ka dotie mainīgie atbilst sakontēji uzģenerētajiem (6.7. att. un 6.14. att.). Visi mainīgie tiek inicializēti ar sākotnējām vērtībām, kas ir `null` objektu gadījumā un `false Boolean` tipa mainīgo gadījumā.

```
String dBQueryPhone = null;
String dBQueryPhoneType = null;
String usersWithGivenPhonePhoneType = null;
UserInformation[] usersWithGivenPhoneUsers = null;
UserInformation singleFoundResultUser = null;
String singleFoundResultPhoneType = null;
String userInputType = null;
String userInputPhone = null;
Boolean searchFailedFound = null;
String searchFailedError = null;
FoundUser matchedUserUser = null;
Boolean matchedUserFound = null;
Boolean noMatchFound = null;
UserInformation userUser = null;
String userPhoneType = null;
GetPhoneNumberResult getPhoneNumberResult = null;
ValidatePhoneNumberResult validatePhoneNumberResult = null;
boolean Invalid_Phone_Number = false;
PrepareValidationErrorResult prepareValidationErrorResult = null;
FindPhonesInDBResult findPhonesInDBResult = null;
boolean should_jump_further = false;
boolean Results_are_not_empty = false;
PeekNextUserResult peekNextUserResult = null;
boolean should_exit_loop = false;
MatchPhoneResult matchPhoneResult = null;
```

6.22. att. Java mainīgo definīcijas

6.23. att. ir parādīts kods, kas tika iegūts pēc uzģenerētā baitu koda dekompilācijas. Var redzēt, ka Procyon dekompilators ir izveidojis divas iezīmes, kas tiek izmantotas cikla ieejas un izejas punkta realizācijas nolūkiem. Promocijas darba autors piedāvā doto Java valodas kodu salīdzināt ar FernFlower dekompilācijas rezultātu, kas ir parādīts 6.24. att. Lai gan no sākuma var secināt par to, ka Procyon iegūtais rezultāts ir sarežģītāks pēc struktūras, var redzēt, ka FernFlower tās pašas loģikas realizācijai ir izmantojis koda dublicēšanu, kas noved pie atkārtojumiem. Papildus var redzēt, ka Procyon izmantošanas gadījumā nosacījumi ir definēti ar komentāru palīdzību, kas ir papildus koda apstrādes rezultāts, ko FernFlower nepiedāvā. No biznesa loģikas abi koda fragmenti ir identiski. Ir arī jāņem vērā fakts, ka dotais divpusložu modelis tika ar nolūku izstrādāts visu iespējamo zarošanās gadījumu pārbaudei, un to ir iespējams vienkāršot, iegūstot citu Java valodas kodu, kas realizē to pašu biznesa loģiku.

Tagad, ir iespējams izveidot Java klasi, kas saturēs visas nepieciešamās metodes (6.21. att.) un galveno izpildāmo metodi (6.22. att. un 6.23. att.). Tā kā dotā klase faktiski pievienos jau esošajam kodam tikai sintakses elementus, kas ir paredzēti sintaktiski pareizas Java valodas klases definēšanai, šis solis darbā netiek apskatīts.

```
getPhoneNumberResult = getPhoneNumber();
userInputType = getPhoneNumberResult.userInputType;
userInputPhone = getPhoneNumberResult.userInputPhone;
validatePhoneNumberResult = validatePhoneNumber(userInputType, userInputPhone);
dBQueryPhone = validatePhoneNumberResult.dBQueryPhone;
dBQueryPhoneType = validatePhoneNumberResult.dBQueryPhoneType;

//Invalid Phone Number
Invalid_Phone_Number = false;
Label_0281: {
    if (!Invalid_Phone_Number) {
        findPhonesInDBResult = findPhonesInDB(dBQueryPhone, dBQueryPhoneType);
        usersWithGivenPhonePhoneType =
                findPhonesInDBResult.usersWithGivenPhonePhoneType;
        usersWithGivenPhoneUsers = findPhonesInDBResult.usersWithGivenPhoneUsers;
        singleFoundResultUser = findPhonesInDBResult.singleFoundResultUser;
        singleFoundResultPhoneType =
                findPhonesInDBResult.singleFoundResultPhoneType;

        //should jump further
        should_jump_further = false;
        while (true) {
            Label_0250: {
                if (should_jump_further) {
                    break Label_0250;
                }
                //Results are not empty
                Results_are_not_empty = false;
                if (!Results_are_not_empty) {
                    break Label_0281;
                }
                peekNextUserResult = peekNextUser(
                          usersWithGivenPhonePhoneType, usersWithGivenPhoneUsers);
                userUser = peekNextUserResult.userUser;
                userPhoneType = peekNextUserResult.userPhoneType;
                noMatchFound = peekNextUserResult.noMatchFound;
                //should exit loop
                should_exit_loop = false;
                if (should_exit_loop) {
                    break Label_0281;
                }
            }
            matchPhoneResult = matchPhone(singleFoundResultUser,
                 singleFoundResultPhoneType, userUser, userPhoneType);
            matchedUserUser = matchPhoneResult.matchedUserUser;
            matchedUserFound = matchPhoneResult.matchedUserFound;
        }
    }
    prepareValidationErrorResult = prepareValidationError();
    searchFailedFound = prepareValidationErrorResult.searchFailedFound;
    searchFailedError = prepareValidationErrorResult.searchFailedError;
}
outputSearchResult(searchFailedFound, searchFailedError, matchedUserUser,
         matchedUserFound, noMatchFound);
```

6.23. att. Java biznesa loģikas kods

```
getPhoneNumberResult = getPhoneNumber();
userInputType = getPhoneNumberResult.userInputType;
userInputPhone = getPhoneNumberResult.userInputPhone;
validatePhoneNumberResult = validatePhoneNumber(userInputType, userInputPhone);
dBQueryPhone = validatePhoneNumberResult.dBQueryPhone;
dBQueryPhoneType = validatePhoneNumberResult.dBQueryPhoneType;
Invalid_Phone_Number = java.lang.Boolean.parseBoolean("Invalid Phone Number");
if (Invalid_Phone_Number) {
    prepareValidationErrorResult = prepareValidationError();
    searchFailedFound = prepareValidationErrorResult.searchFailedFound;
    searchFailedError = prepareValidationErrorResult.searchFailedError;
} else {
    findPhonesInDBResult = findPhonesInDB(dBQueryPhone, dBQueryPhoneType);
    usersWithGivenPhonePhoneType =
         findPhonesInDBResult.usersWithGivenPhonePhoneType;
    usersWithGivenPhoneUsers = findPhonesInDBResult.usersWithGivenPhoneUsers;
    singleFoundResultUser = findPhonesInDBResult.singleFoundResultUser;
    singleFoundResultPhoneType = findPhonesInDBResult.singleFoundResultPhoneType;
    should_jump_further = java.lang.Boolean.parseBoolean("should jump further");
    if (should_jump_further) {
        matchPhoneResult = matchPhone(singleFoundResultUser,
                 singleFoundResultPhoneType, userUser, userPhoneType);
        matchedUserUser = matchPhoneResult.matchedUserUser;
        matchedUserFound = matchPhoneResult.matchedUserFound;
    }

    while(true) {
        Results_are_not_empty = java.lang.Boolean
                 .parseBoolean("Results are not empty");
        if (!Results_are_not_empty) {
            break;
        }

        peekNextUserResult = peekNextUser(usersWithGivenPhonePhoneType,
                 usersWithGivenPhoneUsers);
        userUser = peekNextUserResult.userUser;
        userPhoneType = peekNextUserResult.userPhoneType;
        noMatchFound = peekNextUserResult.noMatchFound;
        should_exit_loop = java.lang.Boolean.parseBoolean("should exit loop");
        if (should_exit_loop) {
            break;
        }

        matchPhoneResult = matchPhone(singleFoundResultUser,
                 singleFoundResultPhoneType, userUser, userPhoneType);
        matchedUserUser = matchPhoneResult.matchedUserUser;
        matchedUserFound = matchPhoneResult.matchedUserFound;
    }
}

outputSearchResult(searchFailedFound, searchFailedError, matchedUserUser,
         matchedUserFound, noMatchFound);
```

6.24. att. FernFlower dekompilācijas rezultāts

Pēc visu transformāciju izpildes tiek iegūta klašu kopa, kas satur nepieciešamās domēna klases, kā arī biznesa loģikas realizācijas klasi. Tagad ir iespējams veikt šīs kopas apstrādi ar algoritmu, kas ir aprakstīts 4. promocijas darba sadaļā. Uzlabotais klašu attiecību noteikšanas algoritms ir definēts kā pusautomātisks un prasa cilvēka piedalīšanos gan lēmumu pieņemšanā, gan arī definējot klašu un interfeisu nosaukumus. Līdz ar to tā izpildes rezultāts nav strikti

definēts un var atšķirties atkarībā no cilvēka pieņemtajiem lēmumiem. Tāpēc arī tā izpildes rezultāts šeit parādīts netiek, šī algoritma darbības piemērs ir apskatīts sadaļā 4.5.

Papildus autors vēlas uzsvērt, ka iegūtais programmatūras kods atbilst anēmiskā datu modeļa definētajiem principiem – dati jeb domēna objekti ir atdalīti no to apstrādes loģikas [24],[26],[59]. Līdz ar to balstoties uz savu pieredzi programmatūras izstrādē industrijas kontekstā, promocijas darba autors uzskata, ka šādā veidā uzģenerētais kods ir labāk piemērots mūsdienu biznesa atbalsta sistēmu izstrādei. Tomēr, anēmiskais datu modelis nav vienīgais iespējamas iegūtā koda variants – tā kā transformācijas rezultātā tiek iegūtas metožu signatūras un galvenās loģikas programmatūras kods, var arī ievietot šīs metodes domēna klasēs (piemēram, pielietojot pieeju no [77] vai [82]) un iegūstot "bagāto" datu modeli.

6.4. Saistīto pētījumu apskats

Tika veikta arī saistīto pētījumu analīze. Kā jau tika minēts darba ievadā, mūsdienās eksistē vairāki pētījumi koda ģenerēšanas jomā. Šādi pētījumi tiek apskatīti un analizēti tālāk.

Tā kā liela pētījumu daļa ir virzīta uz koda ģenerēšanu no UML [113] diagrammām, kas pēc savas būtības apraksta jau gatavus risinājumus, promocijas darba autors salīdzinošai analīzei izvēlējās pētījumus, kas neizmanto UML klašu, secību un aktivitāšu diagrammas – faktiski šīs diagrammas var tikt pārveidotas par programmatūras kodu, izmantojot vienkāršus likumus. Piemēram, programmatūras koda iegūšanai no UML klašu diagrammas ir nepieciešams pārveidot UML klašu definīcijas par programmatūras koda definīcijām un izmantot informāciju par klašu attiecībām mantošanas un interfeisu realizācijas definēšanai. Līdzīgi, secību diagrammu transformācija nozīmē ziņojumu pārveidošanu par attiecīgo metožu izsaukumiem, ņemot vērā arī papildu secību diagrammu elementus (tādus, kā mijiedarbības fragmentu). UML aktivitāšu diagrammas, savukārt, var salīdzināt ar blokshēmām – abi artefakti tiek izmantoti līdzīgiem uzdevumiem un satur līdzīgu informāciju. Aktivitāšu diagrammu pārveidošana par programmatūras kodu ir pietiekami vienkāršs uzdevums, kas ietver aktivitāšu pārveidošanu par metožu izsaukumiem un papildus diagrammas elementu izmantošanu zarošanās un ciklu definēšanai. Eksistē arī speciāli rīki, piemēram Blu Age [42], kas izmanto UML aktivitāšu diagrammas programmatūras koda ģenerēšanai. Promocijas darba autors bija strādājis ar Blu Age rīku, pārveidojot tehnoloģiski novecojušas (angļ. *legacy*) sistēmas, kas bija izveidotas COBOL programmēšanas valodā, uz Java [50] programmēšanas valodu. Galvenais šī darba uzdevums bija UML aktivitāšu diagrammu veidošana, kas pēc tam tika pārveidotas par

programmatūras kodu. Faktiski, bija nepieciešams "rakstīt kodu diagrammas veidā", kas apliecina faktu, ka dotais modelis nav piemērots izmantošanai MDSD kontekstā.

Protams, koda ģenerēšana no UML [113] diagrammām ir pietiekami aktuāls uzdevums – var iedomāties situāciju, kad, piemēram, ir nepieciešams pārveidot tehnoloģiski novecojušu sistēmu, pārrakstot to no programmēšanas valodas A uz programmēšanas valodu B. Ja abas šīs valodas ir objektorientētas, ir iespējams uzģenerēt nepieciešamās UML diagrammas no esoša koda un no šīm diagrammām, savukārt, uzģenerēt jaunu kodu. Protams, šis nav vienīgais UML izmantošanas piemērs, bet pēc promocijas darba autora domām, kuras arī atbalsta programmatūras izstrādātāju, arhitektu un biznesa analītiķu aptaujas rezultāti, UML diagrammas nav pietiekami vienkāršs avota modelis MDSD ieviešanai.

Līdz ar to tika definēta salīdzinoši "šauru" saistīto pētījumu joma – tika apskatīti darbi, kas par avota modeli izmanto stāvokļu diagrammas. Šāda pētījumu izvēle var būt pamatota ar to, ka, kā ir parādīts [39],[40] un [79], ir iespējams biznesa procesu modeli salīdzināt ar galīgo automātu [57],[100],[121] jeb stāvokļu mašīnu. Stāvokļiem šajā gadījumā atbilst attiecīgo procesu izsaukumi, bet pārējām – to izpildes secība. Tātad koda ģenerēšanas process no biznesa procesu diagrammas ir līdzīgs koda ģenerēšanas procesam no stāvokļu diagrammas.

Pirmais no apskatītajiem pētījumiem ir [7]. Šeit autori kā avotu modeli izmanto kompleksas stāvokļu diagrammas, kur jebkurš stāvoklis var saturēt citu stāvokļu mašīnu. Šādas stāvokļu diagrammas pēc savas būtības ir ļoti līdzīgas procesu izsaukumu grafam pēc ciklu aizvietošanas procesa (tajā arī ir iespējamas situācijas, kad viena virsotne satur citu procesu izsaukumu grafu, kas atbilst aizvietotam ciklam). Rezultātā darba [7] autori ir ieguvuši programmatūras kodu, kas sastāv no vienas izpildāmās klases, kura satur informāciju par visiem stāvokļiem un iekšējām stāvokļu mašīnām (kuras ir iekļautas citos stāvokļos). Šīs klases kods izmanto sintaktisko operatoru `switch`, lai realizētu pāreju starp stāvokļiem un stāvokļu mašīnām. Šāda pieeja stāvokļu mašīnas realizācijas gadījumā ir viens no standarta risinājumiem, kas tomēr ne vienmēr ir pietiekami saprotams cilvēkam [100],[121]. Tas nozīme to, ka piedāvātais risinājums ļauj iegūt programmatūras kodu, kas pareizi realizē biznesa loģiku, bet kuru ir salīdzinoši grūti atbalstīt. Kā vēl vienu aspektu, promocijas darba autors vēlas atzīmēt [7] autoru mērķi iegūt iespējami pēc apjoma mazāku programmatūras kodu. Šāda metrika tika izvēlēta kā netiešs rādītājs uz iegūtā programmatūras koda sarežģītību.

Darba [103] autori savā pētījumā izmanto citu pieeju stāvokļu mašīnu reprezentēšanai programmatūras koda veidā. Viņu piedāvātais risinājums ir balstīts uz projektēšanas šablonu izmantošanu (specifiski runa ir par *State* šablonu [28]). Darba autoru piedāvātais risinājums ietver vairāku klašu definēšanu – klases tiek ģenerētas katram stāvoklim un katram notikumam,

kas var novest pie sistēmas stāvokļa izmaiņas. Papildus tiek ģenerēta arī konteksta klase, kas atbild par pašreizējā stāvokļa informācijas uzturēšanu, notikumu apstrādi un stāvokļu izmaiņu. Šis risinājums izmanto standarta projektēšanas šablonu kā realizācijas veidu, kas ir tā priekšrocība. Tomēr, promocijas darba autors vēlas atzīmēt arī šīs metodes ierobežojumus divpusložu modeļa apstrādes kontekstā. Biznesa procesu diagrammas gadījumā notikumi ir ne kas cits, kā procesa izsaukums. Tas nozīmē nepieciešamību pēc vairāku papildu klašu definēšanas. Arī katram biznesa procesam būs nepieciešams definēt tā realizējošo klasi. Tas gan nav ierobežojums, jo šāda pieeja atbilst vienas atbildības (angl. *single responsibility* [62]) principam, tomēr ne vienmēr ir nepieciešams šo principu pielietot, jo tas var sarežģīt sistēmas arhitektūru un uzturēšanu [62].

Vēl viens pētījums, kas ir veltīts koda ģenerēšanai no stāvokļu diagrammas, ir [114]. Tā autori piedāvā algoritmu, kas ģenerē vienu metodi katram atbilstošajam notikumam, kas var novest pie sistēmas stāvokļa izmaiņām. Pētījuma autori koncentrējas uz vairāku pavedienu izmantošanu maksimālas apstrādes ātruma iegūšanai. Līdz ar to attiecīgā notikuma izpilde var palaist vai apstādināt vienu vai vairākus pavedienus. Darba autori izmanto vēl vienu stāvokļu mašīnas realizācijas iespēju – stāvokļi un pārejas starp tiem tiek reprezentēti ar speciālas tabulas palīdzību, kur tiek definēti visi iespējamie stāvokļi un pārejas starp tiem. Autoru piedāvātais risinājums ļauj iegūt korekti strādājošu programmatūras kodu, ko var būt grūti uzturēt – pielietojot šādu metodi biznesa procesa modeļa apstrādei būtu nepieciešams definēt visas iespējamās datu plūsmas kā globālos mainīgos (vai arī attiecīgas kontroliera klases atribūtus) un uzturēt pāreju tabulu. Attiecīgi šāda klase saturētu metodes, kas mainītu norādi uz sistēmas pašreizējo stāvokli. Līdzīgi kā darbā [7], pētījuma [114] autori kā vienu no mērķiem izvirzīja pēc iespējas apjomā mazāku kodu.

Pētījumā [66] autori piedāvā stāvokļu klašu ģenerēšanu (līdzīgi kā to piedāvā [28] autori *State* projektēšanas šablona realizācijai). Pārejas starp stāvokļiem autoru piedāvātajā risinājumā tiek realizētas ar zarošanās valodas konstrukciju palīdzību. Tiek piedāvāts arī definēt klases katram notikumam, kas var mainīt sistēmas stāvokli un konteksta klasi, kas satur visu nepieciešamo informāciju par sistēmas tekošo stāvokli. Pielāgojot šādu pieeju biznesa procesu diagrammas apstrādei, būtu nepieciešams definēt klasi katram procesam, bet datu plūsmu atribūtus definēt kā konteksta klases atribūtus. Tomēr piedāvātai metodei ir raksturīga priekšrocība, ka informācija par pārejām starp stāvokļiem ir faktiski programmatūras koda sastāvdaļa – viss tiek realizēts ar zarošanās instrukciju palīdzību.

Var redzēt, ka pētījumi, kas ir veltīti programmatūras koda ģenerēšanai no stāvokļu diagrammām, galvenokārt ir paredzēti korekta izpildāmā koda definēšanai. Tomēr to autori

parasti kā galveno mērķi uzstāda pēc apjoma mazāka koda ar lielāku veiktspēju iegūšanu. Tas tiek panākts, izmantojot dažādas stāvokļu mašīnas reprezentācijas, pielietojot vairākus pavedienus paralēlai apstrādei, vai minimizējot izpildāmā koda apjomu. Lai gan tas ļauj sasniegt mērķus, ko ir definējuši šo pētījumu autori, tas noved pie grūti lasāma un uzturama koda iegūšanas. Faktiski kļūdu labošanai un jaunu biznesa prasību atbalstam būtu nepieciešams pārdefinēt avota modeli un veikt atkārtotu koda ģenerēšanu. Tas var novest pie tā saucamās regresijas [32] jeb kļūdām jau pārbaudītajā programmas daļā. Protams, ja eksistē regresijas testu kopa, ir iespējams šādas situācijas novērst. Tomēr tas izvirza papildus prasības koda uzturēšanai – definēt visus nepieciešamos funkcionalitātes testus kopā ar koda ģenerēšanu no modeļa un uzturēt tos.

Papildus promocijas darba autors vēlas pieminēt šādus papildus pētījumus, kas nav cieši saistīti ar koda ģenerēšanu no stāvokļu diagrammām. Šie pētījumi ir veltīti staprmodeļu definēšanai, kas var tikt izmantoti programmatūras koda ģenerēšanai. Tā kā šajā darbā autors ir arī izmantojis starpmodeli, kas kalpo par tiltu starp avota divpusložu modeli un mērķa programmatūras kodu, bija nepieciešams noteikt, kas tiek izmantots šajā jomā.

Darba [88] autori piedāvā paplašināmu starpmodeli, kas var tikt izmantots programmatūras koda ģenerēšanai no UML [113] secību diagrammām. Pētījuma ietvaros tiek aprakstīts šī modeļa metamodelis un tā iespējamie paplašinājumi. Darba autori piedāvā modeļa definīciju un transformācijas likumus, kas atbalsta MVC [111] pamatprincipus un ļauj ģenerēt programmatūras kodu, kas šo arhitektūru atbalsta. Pieeja var tikt izmantota dažādām mērķa programmēšanas valodām, tomēr šīm valodām ir jābūt objektorientētām. Pētījuma [88] rezultāti parāda iespējamību definēt universālo starpmodeli, ko ir iespējams izmantot programmatūras koda turpmākai ģenerēšanai. Tomēr promocijas darba autors vēlas atzīmēt arī dotā pētījuma ierobežojumus – tiek atbalstīta konkrēta arhitektūra (MVC), kā arī mērķa programmēšanas valodai ir jābūt objektorientētai.

Arī pētījums [118] ir veltīts starpmodeļa definēšanai MDSD kontekstā. Tā autors piedāvā tā saucamo Hierarhisko Sintakses Diagrammu (angl. *Hierarchical Syntax Chart* – HSC), kas tiek izmantota UML [113] aktivitāšu diagrammu pārveidošanai programmatūras kodā. Kā mērķa programmēšanas valoda tika izvelēta Java [50]. Piedāvātā metode var tikt izmantota programmatūras koda iegūšanai jebkurā objektorientētā programmēšanas valodā, tomēr tās pielietošana prasa izmaiņu ieviešanu UML aktivitāšu diagrammu notācijā, ko šī darba autors uzskata par metodes ierobežojumu. Faktiski, [118] autors piedāvā metodi, kas ir līdzīga Blu Age rīkā izmantotajai [42], kas prasīja koda fragmentu definēšanu UML aktivitātes diagrammas ietvaros.

Pētījuma [58] autori piedāvā starpmodeli programmatūras koda ģenerēšanai no UML [113] diagrammu kopas. Šeit autori piedāvā modeli, kas turpmāk var tikt transformēts kodā, kas satur noteiktus arhitektūras elementus. Līdzīgi kā iepriekšējos pētījumos, pētījuma autori kā mērķa valodu izvēlējas programmēšanas valodu Java [50]. To piedāvātais modelis, savukārt, ir no mērķa valodas neatkarīgs, definējot tai tikai vienu prasību – būt objektorientētai. Pietam iegūtais kods tiek definēts, izmantojot konkrētos arhitektūras risinājumus, kas var būt ierobežojums.

Analizējot pētījumus, kas ir veltīti programmatūras koda ģenerēšanai, var redzēt, ka tie parasti izmanto objektorientētas valodas kā mērķa valodas. Pēc promocijas darba autora domām, tas ir skaidrojams ar UML [113] izmantošanu kā avota modeli. Lai gan šajā darbā apskatītais piemērs arī definē objektorientētu programmatūras kodu, kā ir parādīts [38], izstrādātais starpmodelis ir no paradigmas neatkarīgs, un, veicot transformācijas likumu modificēšanu, ir iespējams iegūt ne tikai klases, bet arī cita veida datu struktūras. Arī ir vērts atzīmēt, ka lielākā pētnieku daļa kā mērķa programmēšanas valodu izvēlas Java [50], kas liecina arī par paša autora pareizo izvēli. Arī starpmodeļa izmantošana koda ģenerēšanas nolūkiem tiek plaši izmantota, veicot jauno MDSD metožu izstrādi un eksistējošo metožu uzlabošanu.

6.5. Koda ģenerēšanas algoritma novērtēšanas rezultāts

Kā ir parādīts šajā sadaļā, piedāvātais koda ģenerēšanas algoritms ļauj no starpmodeļa, kas, savukārt, tiek iegūts no divpusložu modeļa, iegūt programmatūras kodu izvēlētajā programmēšanas valodā. Lai pārbaudītu šādas transformācijas pareizību, promocijas darba autors ir piedāvājis arī transformācijas likumu darbības rezultātu validācijas algoritmu, kas ir balstīts uz iegūtā starpmodeļa salīdzināšanu ar avota modeli.

Programmatūras koda iegūšanai tika izmantots dekompilācijas paņēmiens [21]. Lai būtu iespējams to veikt, sākumā starpmodeļa instrukcijas tiek pārveidotas par JVM instrukcijām [16], no kurām pēc tam tiek iegūts Java kods. Fakts, ka ir iespējama iegūtā baitu koda korekta dekompilācija, arī liecina par transformācijas likumu pareizību.

Promocijas darba autors apskata piemēru, kas tika izveidots tā, lai tas saturētu pēc iespējas vairāk īpaši apstrādājamu gadījumu – tas ir cikli ar vairākiem ieejas un izejas punktiem; zarošanās; datu plūsmas, kas nepārnes nekādu informāciju. Rezultātā, no šāda biznesa procesu modeļa tika iegūts korekts Java kods, ko var turpmāk modificēt.

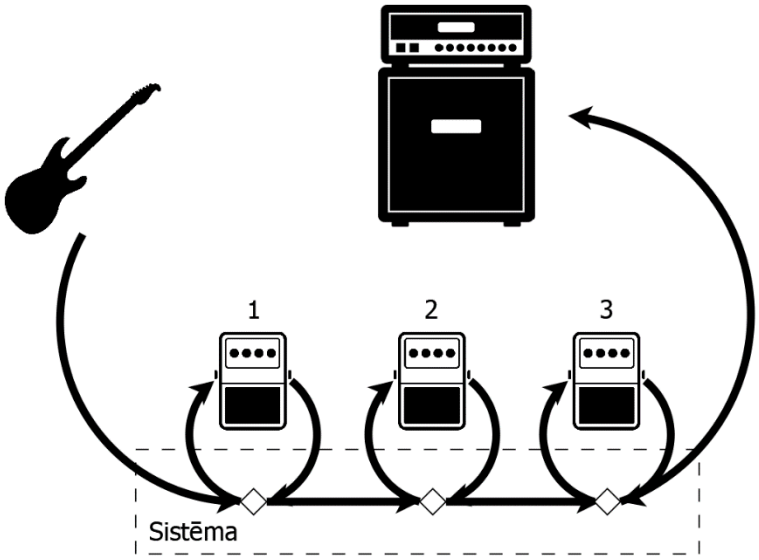
Šajā sadaļā tiek apskatīti arī saistītie darbi un tiek definēti galvenie saistīto pētījumu ierobežojumi – to autori galvenokārt izmanto UML [113] kā avota modeli un objektorientētas programmēšanas valodas kā mērķa modeli. Pētījumi, kas ir veltīti, programmatūras koda iegūšanai no stāvokļu diagrammām, savukārt apskata dažādus stāvokļu mašīnas reprezentācijas un optimizācijas veidus. To autori nemēģina iegūt cita veida programmatūras kodu.

7. DARBA REZULTĀTU PRAKTISKAIS PIELIETOJUMS

Promocijas darba izstrādes laikā tā autoram tika piedāvāts izstrādāt programmu PIC18F45Q10 [90] mikrokontrolierim. Dotais mikrokontrolieris tiks izmantots elektroģitāras efektu pārslēgšanas sistēmā, kas tiks aprakstīta zemāk. Kopā ar pasūtītāju tika nolemts šīm nolūkam izmantot šajā darbā aprakstīto pieeju un sākumā definēt divpusložu modeli, no kuras tiks ģenerēts programmatūras kods. Šajā sadaļa tiek aprakstīta šīs programmas izstrāde.

7.1. Elektroģitāras efektu pārslēgšanas sistēma

Elektroģitāras efekti ir ierīces, kas ļauj mainīt elektroģitāras skaņu [44], piemēram pievienot atbalsu, vai citā veidā mainīt audiosignālu. Šādi efekti var tikt realizēti, ka atsevišķas ierīce, vai arī kā skaņas procesori [44], kas pārveido signālu digitālajā formā un veic tā apstrādi, pēc tam pārveidojot apstrādes rezultātus par analogo signālu.

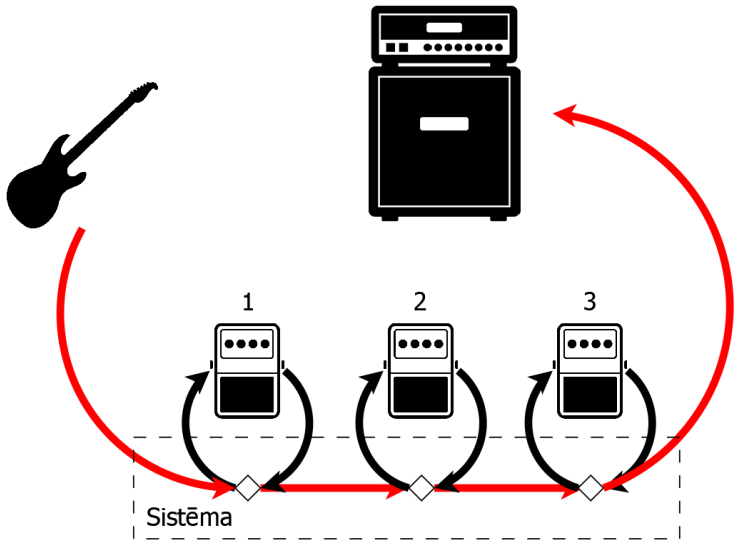


7.1. att. Elektroģitāras efektu pārslēgšanas sistēmas shēma

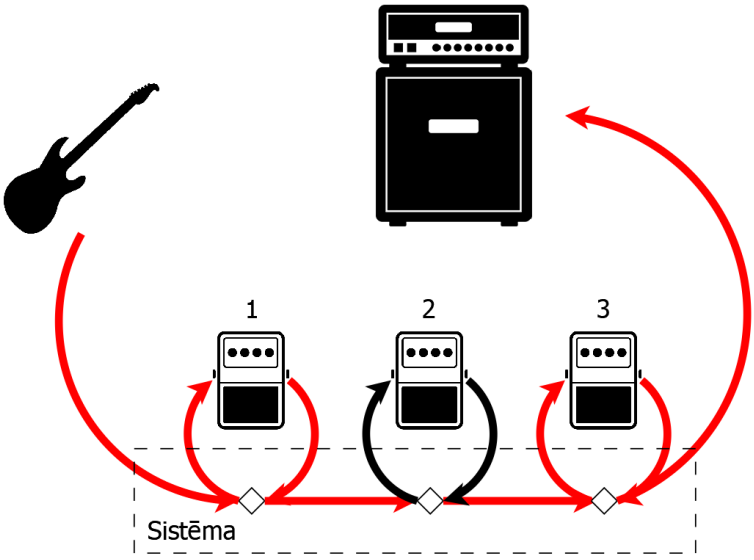
Pārslēgšanas sistēma, kas tiek aprakstīta šajā sadaļa, ļauj veidot tā saucamas efektu cilpas, kas ļauj veikt dažādu efektu savstarpējo komutāciju, pieslēdzot vai izslēdzot no signāla pārraides ceļa tos. 7.1. att. ir paradīta šādas sistēmas vienkāršota shēma. Signāla pārraides ceļš sākas ar elektroģitāru un beidzas ar ģitāras pastiprinātāju. Tajā var tikt ievietoti efekti, kas šeit

tiek atzīmēti ar `1,2,3`. Gadījumā, kad visi efekti ir izslēgti no signāla pārraides ceļa, šo ceļu var aprakstīt kā: `Elektroģitāra → ģitāras pastiprinātājs`.

Šī situācija ir paradīta 7.2. att.



7.2. att. Signāla pārraides ceļš ar izslēgtiem efektiem



7.3. att. Signāla pārraides ceļš ar vairākiem ieslēgtiem efektiem

7.3. att. ir paradīts cits iespējamas situācijas piemērs, kur signāla pārraides ceļš ir: `Elektroģitāra → Efekts 1 → Efekts 3 → ģitāras pastiprinātājs`. Protams, ir iespējams veidot arī citus signāla pārraides ceļus. Lai būtu iespējams uzglabāt informāciju par efektu komutācijas veidiem, sistēmas ietvaros tiek izmantots PIC18F45Q10 [90]

mikrokontrolieris kopā ar citām elektriskām komponentēm, piemēram, atmiņas mikroshēmām. Dotais mikrokontrolieris veic arī komutācijas ceļu pārslēgšanu, ieslēdzot dažādus no saglabātiem priekšiestatījumiem. Priekšiestatījumi tiek organizēti vairākas bankās, kur katra no bankām satur 8 priekšiestatījumus. Kopā sistēmai ir iespējams pieslēgt 8 dažādus efektus.

Sistēmas kontrolēšanai tiek izmantotas 11 pogas, kas tiek pārskaitītas zemāk:

- `Edit/Save` – rediģēšanas režīma pārslēgšana. Sākumā sistēma darbojas pārslēgšanas režīmā. Nospiežot šo pogu, ir iespējams sistēmu pārslēgt rediģēšanas režīmā, kas ļauj veidot priekšiestatījumus pašreiz izvēlētājā bankā.
- `Bank Up` – rediģēšanas režīmā pogas nospiešana neizraisa nekādas darbības, pārslēgšanas režīmā tiek izvēlēta nākama efektu banka.
- `Bank Down` – rediģēšanas režīmā pogas nospiešana neizraisa nekādas darbības, pārslēgšanas režīmā tiek izvēlēta iepriekšēja efektu banka.
- `1-8` – rediģēšanas režīmā pogas nospiešana pievieno vai noņem no signāla pārraides ceļa efektu ar attiecīgo numuru (1-8). Pārslēgšanas režīmā izvēlas attiecīgo priekšiestatījumu (1-8) no pašreiz izvēlētas bankas.

Nospiežot vairākas pogas vienlaicīgi, tiks apstrādāta tikai viena no tām. Mikrokontroliera programmas uzdevums ir pārbaudīt, kura no pogām tika nospiesta, kura režīmā pašreiz darbojas sistēma, un atkarībā no tā veikt dažādas darbības.

7.2. Sistēmas divpusložu modelis

State	Input
current bank: int current preset: int mode: int current loops: int[]	up pressed: boolean down pressed: boolean edit pressed: boolean preset pressed: boolean[]

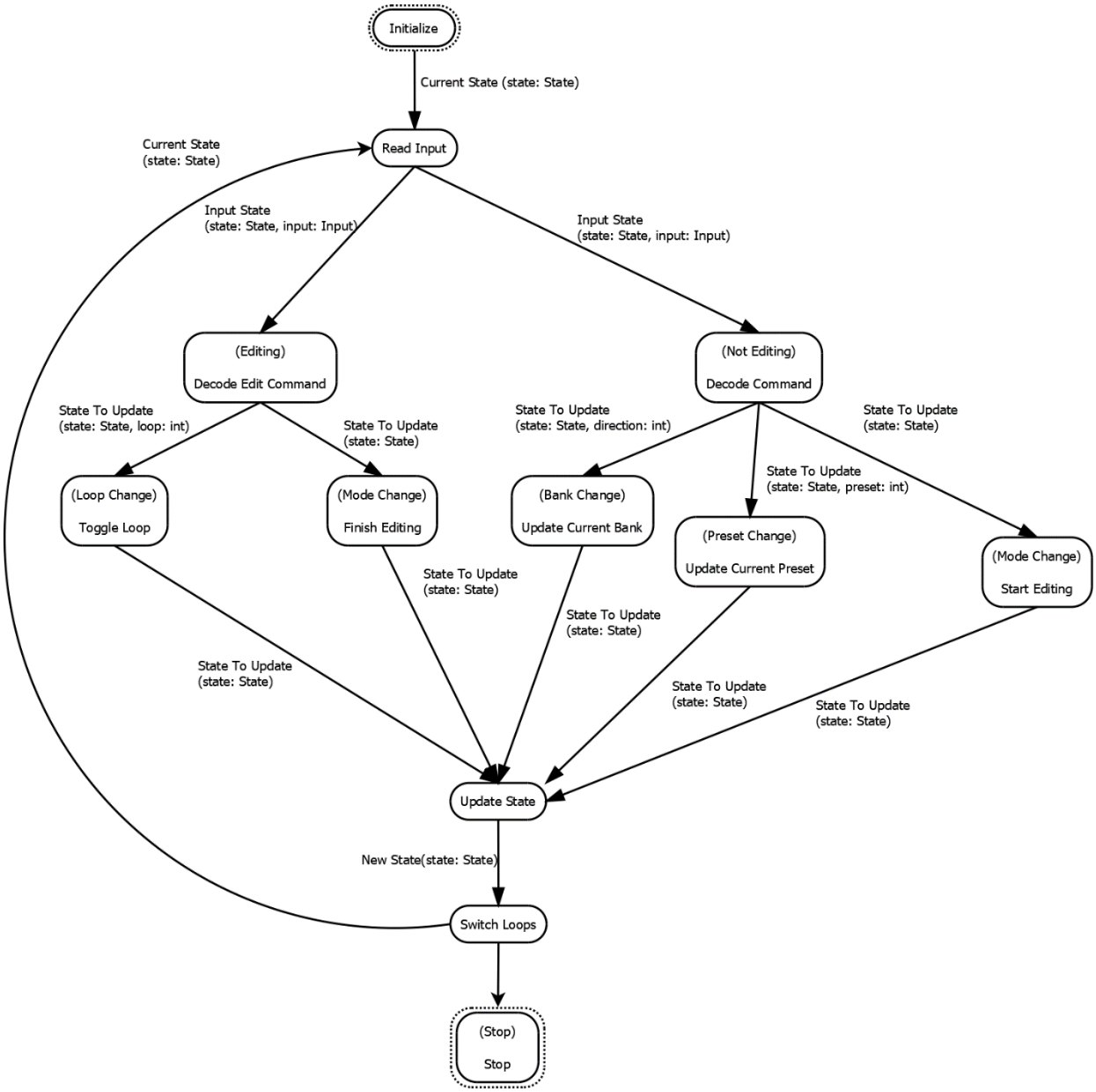
7.4. att. Definētas konceptu modelis

Iepriekš aprakstītas sistēmas darbības atbalstām tika izveidots divpusložu modelis, kas apraksta mikrokotroliera darbību. Koda ģenerēšanas un komunikācijas ar pasūtītāju nolūkiem tas ir definēts angļu valodā. 7.4. att. ir paradīts sistēmas konceptu modelis, kas satur divus konceptus:

- Koncepts `State` apraksta sistēmas esošo stāvokli:
  - Tā atribūts `current bank` satur informāciju par pašreiz izvēlēto banku.

    - Atribūts `current preset` satur izvēlēta priekšiestatījuma numuru.
    - Atribūts `mode` satur informāciju par sistēmas darbības režīmu – pārslēgšanas vai rediģēšanas.
    - Atribūts `current loop` satur informāciju par pašreiz ieslēgtiem efektiem.
- Koncepts `Input` apraksta ievades notikumu, definējot, kādas pogas tika nospiestas:
    - Atribūts `up pressed` norāda uz pogas `Bank Up` nospiešanu.
    - Atribūts `down pressed` norāda uz pogas `Bank Down` nospiešanu.
    - Atribūts `edit pressed` norāda uz pogas `Edit/Save` nospiešanu.
    - Atribūts `preset pressed` norāda uz vienas no pogām `1-8` nospiešanu.

7.5. att. ir paradīts izstrādātais procesu modelis, kas apraksta mikrokontroliera darbību. Sākumā no atmiņas tiek nolasīta informācija par iepriekš saglabāto sistēmas stāvokli – izvēlētā banka un izvēlētais priekšiestatījums. Par to atbild ārējais process `Initialize`. Pēc inicializācijas sākas notikumu apstrādes cikls, kura mikrokontrolieris periodiski pārbauda ievada stāvokli. Par to atbild process `Read Input`. Gadījumā, ja tas ir izmainījies, ir nepieciešams pārbaudīt, kurā režīmā pašreiz sistēma strādā un veikt attiecīgas komandas dekodēšanu – šīm nolūkam ir domāti procesi `Decode Edit Command` un `Decode Command`. Rediģēšanas režīmā ir iespējams saņemt režīma maiņas komandu, ko apstrādā process `Finish Editing`, vai arī noteikta efekta ieslēgšanas/izslēgšanas komandu, ko apstrādā process `Toggle Loop`. Pārslēgšanas režīmā ir iespējamas trīs komandas – bankas maiņas komandu, ko apstrādā process `Update Current Bank`, priekšiestatījuma maiņas komandu, ko apstrādā process `Update Current Preset`, vai arī režīma maiņas komandu, kas ar procesa `Start Editing` palīdzību ieslēdz rediģēšanas režīmu. Pēc jebkuras komandas apstrādes esošais stāvoklis tiek atjaunināts (`Update State`) un tiek veikta efektu pārslēgšana (`Switch Loops`). Izejot no ievada apstrādes cikla, tiek izpildīts process `Stop`, kas nodrošina beigu stāvokļa saglabāšanu.



7.5. att. Definētas procesu modelis

7.3. Koda ģenerēšanas procesa pielāgošana

Tā kā mikrokontroliera programma tiek definēta programmēšanas valodā C [95], bija nepieciešams veikt izmaiņas koda ģenerēšanas algoritmā. Darba autors vēlētos atzīmēt, ka šādas izmaiņas bija minimālas, un kopumā tas liecina par to faktu, ka darbā aprakstīta metode var tikt izmantota ne tikai objektorientēta koda ģenerēšanai. Kopumā tika veiktas šādas izmaiņas:

- Lokālo mainīgo definēšana – mikrokontrolierim ir pieejams salīdzinoši mazs resursu (operatīva atmiņa, steka atmiņa) apjoms. Līdz ar to būtu vēlams minimizēt izmantojam lokālo mainīgo daudzumu. Lai atbalstītu šo prasību, visi datu plūsmu parametri ar vienādu nosaukumu un tipu tiek pārveidoti par vienu lokālo mainīgo ar tādu pašu nosaukumu un tipu.
- Koncepti tiek pārveidoti nevis klasēs, bet valodas C [95] struktūrās (`struct`).
- Procesi tiek pārveidoti par funkcijām, nevis klašu metodēm.
- Tiek modificēts dekompilēta baitu koda apstrādes algoritms, kas ļauj Java [50] klases vietā iegūt C failu ar attiecīgām funkcijām. Pateicoties Procyon [64] iespējam, šāda modifikācija prasa minimālas koda izmaiņas, kas vēlreiz liecina par pareizo dekompilatora izvēli.
- Tiek modificēts metožu un mainīgo nosaukumu ģenerēšanas algoritms atbilstoši valodā C [95] pieņemtajām stilam.
- `boolean` un `integer` tipa mainīgie tiek aizvietoti ar `unsigned char` tipa mainīgiem.

7.4. Iegūtie rezultāti

Koda ģenerēšanas procesā tika izveidotas datu struktūras, kas ir paradītas 7.6. att. Bez datu struktūrām, kas tika iegūtas konceptu transformēšanas rezultātā, tika definētas arī trīs papildus datu struktūras, kas reprezentē atsevišķo metožu izpildes rezultātus. Iegūtais kods ir kompilējams un izmantojams ar minimālo modifikāciju – ir nepieciešams struktūras `input_t` laukam `preset_pressed[]` un struktūras `state_t` laukam `current_loop` definēt masīva garumu, rezultātā iegūstot:

```
unsigned char preset_pressed[8]
unsigned char current_loop[8]
```

Šāda modifikācija ir nepieciešama tāpēc, ka avota modelis nesatur informāciju par masīva izmēru, lai gan to ir iespējams pievienot modeļa notācijai.

```
typedef struct {
    unsigned char current_bank;
    unsigned char current_preset;
    unsigned char mode;
    unsigned char current_loop[];
} state_t;

typedef struct {
    unsigned char up_pressed;
    unsigned char down_pressed;
    unsigned char edit_pressed;
    unsigned char preset_pressed[];
} input_t;

typedef struct {
    state_t state;
    input_t input;
} read_input_result_t;

typedef struct {
    unsigned char direction;
    state_t state;
    unsigned char preset;
} decode_command_result_t;

typedef struct {
    unsigned char loop;
    state_t state;
} decode_edit_command_result_t;
```

7.6. att. Uzģenerētas datu struktūras

7.7. att. ir paradītas funkcijas, kas tika uzģenerētas procesu transformāciju rezultātā. Tika iegūts kompilējamas valodas C [95] kods, kas izmanto iepriekš definētas datu struktūras. 7.8. att. ir paradīts uzģenerētais programmas galvenās funkcijas kods. Iegūtais programmatūras kods ir kompilējams un prasa minimālas modifikācijas, lai pilnīgi atbalstītu visas prasības. Modifikācijas ir nepieciešamas divos gadījumos:

- Ir nepieciešams definēt metožu, kas tika iegūti no biznesa procesiem, ķermeņus.
- Ir nepieciešams precizēt zarošanas nosacījumus.

Modificētais programmatūras kods ir sniegts 5. pielikumā. Var redzēt, ka pēc minimālo modifikāciju veikšanas tika iegūts strādājošā mikrokontroliera programma, kas arī tiek izmantota iepriekš aprakstītas sistēmas funkcionēšanas nodrošināšanai. Veiktas modifikācijas ir galvenokārt veltītas zema līmeņa funkciju izsaukumiem un mikrokontroliera ievades/izvades apstrādei. Galvenās programmas funkcijas kods tika papildināts vienīgi ar modificētiem zarošanas nosacījumiem.

```
read_input_result_t read_input(state_t state) {
    read_input_result_t result;
    return result;
}
state_t toggle_loop(unsigned char loop, state_t state) {
    state_t result;
    return result;
}
decode_command_result_t decode_command(state_t state, input_t input) {
    decode_command_result_t result;
    return result;
}
state_t finish_editing(state_t state) {
    state_t result;
    return result;
}
state_t update_state(state_t state) {
    state_t result;
    return result;
}
decode_edit_command_result_t decode_edit_command(state_t state, input_t input) {
    decode_edit_command_result_t result;
    return result;
}
state_t update_current_bank(unsigned char direction, state_t state) {
    state_t result;
    return result;
}
state_t switch_loops(state_t state) {
    state_t result;
    return result;
}
state_t update_current_preset(state_t state, unsigned char preset) {
    state_t result;
    return result;
}
void stop(void) {
}
state_t initialize(void) {
    state_t result;
    return result;
}
state_t start_editing(state_t state) {
    state_t result;
    return result;
}
```

7.7. att. Uzģenerētas funkcijas

```
void main(void) {
    state_t state;
    input_t input;
    unsigned char loop;
    unsigned char direction;
    unsigned char preset;
    unsigned char should_loop;
    read_input_result_t read_input_result;
    unsigned char not_editing;
    decode_command_result_t decode_command_result;
    unsigned char mode_change;
    unsigned char preset_change;
    unsigned char bank_change;
    unsigned char loop_change;
    unsigned char editing;
    decode_edit_command_result_t decode_edit_command_result;
    state = initialize();
    for (should_loop = 0/* should loop */; should_loop;
                 state = switch_loops(state)) {
        read_input_result = read_input(state);
        state = read_input_result.state;
        input = read_input_result.input;
        not_editing = 0/* Not Editing */;
        if (not_editing) {
            decode_command_result = decode_command(state, input);
            direction = decode_command_result.direction;
            state = decode_command_result.state;
            preset = decode_command_result.preset;
            mode_change = 0/* Mode change */;
            if (mode_change) {
                state = start_editing(state);
            }
            preset_change = 0/* Preset change */;
            if (preset_change) {
                state = update_current_preset(state, preset);
            }
            bank_change = 0/* Bank change */;
            if (bank_change) {
                state = update_current_bank(direction, state);
            }
        }
        editing = 0/* Editing */;
        if (editing) {
            decode_edit_command_result = decode_edit_command(state, input);
            loop = decode_edit_command_result.loop;
            state = decode_edit_command_result.state;
            mode_change = 0/* Mode change */;
            if (mode_change) {
                state = finish_editing(state);
            }
            loop_change = 0/* Loop change */;
            if (loop_change) {
                state = toggle_loop(loop, state);
            }
        }
        state = update_state(state);
    }

    stop();
}
```

7.8. att. Uzģenerētais `main()` funkcijas kods

Var redzēt, ka izstrādāta divpusložu modeļa transformācijas metode var tikt izmantota koda ģenerēšanai dažādās programmēšanas valodās, pie tam mērķa valodas izmaiņa prasa

modifikācijas pēdējos soļos. Pamata metode paliek nemainīga, kas liecina, ka autora piedāvātais starpmodelis ir noderīgs un var kalpot par pamatu koda ģenerēšanai.

Piedāvātā metode tika aprobēta arī praktiski, un darba autora pieredze darbā ar sistēmas pasūtītāju liecina par to, ka to ir iespējams izmantot arī citos lielākos projektos.

Nobeigums

Promocijas darbs ir veltīts programatūras koda ģenerēšanai no avota modeļa MDSD kontekstā. Koda ģenerēšana tiek izmantota no 1980-iem gadiem dažādiem nolūkiem (piemēram, integrētās izstrādes vidēs), tomēr modeļvadāmas programmatūras izstrādes gadījumā tā ir nevis palīgrīks, bet metodoloģijas pamats. Ja ir iespējams iegūt programmatūras kodu no avota modeļa, kas ir saprotams gan programmatūras izstrādātājiem, gan arī problēmsfēras ekspertiem, tad ir iespējams runāt par pilnu MDSD atbalstu. Promocijas darbā ir analizētas avota modeļa notācijas, pamatota notācijas izvēle, kas jau ir pielietota modeļu transformācijās MDSD kontekstā dažādos abstrakcijas līmeņos, un esošajā pētījumā šī notācija ir modificēta koda ģenerēšanas uzdevuma realizācijai, tādējādi panākot MDSD atbalstu pārējo līmeņu ķēdē. Promocijas darba izstrādes laikā definētie uzdevumi tika izpildīti pilnībā:

1. Promocijas darba izstrādes laikā tika veiktā divpusložu modeļa priekšrocību un ierobežojumu analīze koda ģenerēšanas kontekstā. Tika identificēti vairāki ierobežojumi, kurus bija nepieciešams uzlabot, lai padarītu koda ģenerēšanas uzdevumu iespējamu. Ierobežojumi tika identificēti gan divpusložu modeļa notācijā, gan arī esošajos transformāciju likumos. Visiem atrastajiem ierobežojumiem tika piedāvāti risinājumi, kas arī kļuva par pamatu turpmākam darbam.
2. Promocijas darba autors ir veicis mērķa programmēšanas valodas izvēli, pamatojoties uz TIOBE reitinga datiem par programmēšanas valodu popularitāti. Rezultātā tika izvēlēta programmēšanas valoda Java.
3. Tika izstrādāti transformācijas līkumi, kas ļauj no divpusložu modeļa iegūt programmatūras kodu. Dotie līkumi ar starpmodeļa palīdzību ir pielietojami ne tikai objektorientētas programmatūras veidošanai, kas arī tika pārbaudīts praktiski.
4. Tika izstrādāta validācijas metodoloģija transformāciju pareizību pārbaudei, kas ir balstīta uz avota modeļa un iegūto rezultātu savstarpējo salīdzināšanu.
5. Pielietojot izstrādāto algoritmu tika izstrādāta elektroģitāras efektu pārslēgšanas sistēma, kas ļāva šo algoritmu pārbaudīt praktiski.

Promocijas darba **galvenais rezultāts** ir piedāvātais algoritms, kas nodrošina programmatūras koda ģenerēšanu no divpusložu modeļa. Šī promocijas darba ietvaros ir apskatīta Java koda ģenerēšana, bet pats algoritms tika definēts vispārīgā veidā un ļauj iegūt kodu vairākās programmēšanas valodās, kas var būt vai nebūt objektorientētas. Lai to panāktu,

tika definēts starpmodelis, kas ir balstīts uz speciālo instrukciju kopas izmantošanu. Definējot šo modeli, promocijas darba autors ir veicis iespējamo programmatūras koda reprezentāciju analīzi un tika izvirzījis prasību par dotā starpmodeļa universālumu. Var redzēt, ka dotais modelis faktiski netiek piesaistīts objektorientētai paradigmai, bet operē ar funkciju/metožu izsaukumiem un kontroles novirzīšanas instrukcijām. Tas potenciāli dod iespēju šo starpmodeli izmantot plašam mērķa programmēšanas valodu klāstam. Var redzēt, ka dotais modelis pagaidām ir salīdzinoši primitīvs un neatbalsta vairākas papildu iespējas. Piemēram, nav iespējams tajā aprakstīt matemātiskas operācijas ar mainīgajiem, vai veikt mainīgo tipu pārbaudi un pārveidošanu. Tomēr autors uzskata, ka promocijas darbā izvirzītājiem uzdevumiem dotais modelis ir pietiekams, kas arī tiek parādīts šajā promocijas darbā. Ir vērts piezīmēt, ka ir iespējams veikt šī modeļa papildināšanu, pateicoties izvēlētajai modeļa notācijai – faktiski var pievienot jebkura veida instrukcijas un definēt likumus to pārveidošanai mērķa programmēšanas valodā.

Lai būtu iespējams pārveidot sākotnējo modeli par starpmodeli, ir nepieciešams definēt transformācijas likumus jeb transformācijas algoritmu. Šo uzdevumu promocijas darba autors arī ir izpildījis, definējot biznesa procesa modeļa pārveidošanas likumus, kas ir aprakstīti promocijas darbā. Definējot šos likumus, promocijas darba autors ir izmantojis faktu, ka biznesa procesu diagrammu var definēt un apskatīt kā orientētu multigrafu ar cilpām. Veicot transformācijas likumu definīciju, tika apskatīts sākotnējais grafa fragments, tā pārveidošanas iespējas un pārveidošanas rezultāts, pārbaudot, vai visa informācija tika saglabāta. Kopumā tika definēti 13 grafa transformācijas likumi, kā arī to pielietošanas noteikumi. Dotie transformācijas likumi ļauj biznesa procesu modeli minimizēt līdz vienas virsotnes grafam, kur dotā virsotne satur visu sākotnējo informāciju par biznesa procesiem, datiem, kas tiek nodoti starp tiem, kā arī to izpildes secību. Lai pārliecinātos, ka dotie likumi strādā, tika veikta testēšana ar pēc dotajiem likumiem nejauši uzģenerētiem grafiem. Testēšanas laikā netika atrasta neviena situācija, kad grafa pārveidošana nav iespējama.

Minimizētais grafa saturs faktiski atbilst starpmodelim jeb instrukciju secībai, tomēr ir nepieciešams izgūt arī papildus informāciju par mainīgajiem un definēt tā transformācijas algoritmu uz starpmodeli. Šis uzdevums arī tika izpildīts promocijas darba izstrādes laikā.

Lai pārliecinātos, ka iegūtais starpmodelis atbilst sākotnējam modelim, tika izstrādāts tā validācijas algoritms, kas veic pārbaudi, pārveidojot starpmodeli par grafu, kas satur informāciju par iespējamo procesu izpildes secību. Šeit promocijas darba autors vēlas piezīmēt, ka doto algoritmu ir iespējams izmantot arī citiem nolūkiem – piemēram, biznesa procesu modeļa iegūšanai no programmatūras koda. Lai gan šāda iespēja netiek apskatīta promocijas

darba ietvaros, to ir iespējams attīstīt, lai nākotnē varētu izveidot divpusložu modeli no programmatūras koda, šādā veidā atbalstot atgriezenisko saiti starp sākotnējo un mērķa modeli.

Programmatūras koda iegūšanai no starpmodeļa tika izmantots reversās inženierijas paņēmiens, kas tiek saukts par dekompilāciju – starpmodelis jeb instrukciju secība sākumā tiek pārveidota JVM baitu kodā, no kura pēc tam tiek iegūts kods programmēšanas valodā Java. Šāda pieeja ļauj izmantot salīdzinoši vienkāršus starpmodeļa transformācijas likumus, bet nav vienīgā iespējamā – nākotnē ir iespējams definēt arī citus starpmodeļa pārveidošanas likumus, kas var būt izmantojami arī citām mērķa programmēšanas valodām.

Papildus tika izstrādāts arī klašu kopas apstrādes algoritms, kas ļauj definēt attiecības starp klasēm, balstoties gan uz to struktūru, gan arī uz informāciju par klašu savstarpējo mijiedarbību. Šo algoritmu ir iespējams pielietot gadījumos, kad mērķa programmēšanas valoda ir objektorientēta, un ir nepieciešams veikt iegūto rezultātu papildu apstrādi. Visi transformācijas likumi, kas tika izstrādāti ārpus šī promocijas darba ietvariem, klašu attiecību noteikšanai izmantoja informāciju, kas ir atrodama biznesa procesu modelī. Lai izvairītos no šī ierobežojuma, promocijas darba autors ir piedāvājis sākumā uzģenerēt visas rezultējošās klases, kas ir gan konceptu, gan arī biznesa procesu diagrammas transformācijas rezultāts, un tad veikt iegūtās klašu kopas analīzi attiecību noteikšanai. Lai būtu iespējams veikt šādu analīzi, autors ir definējis algoritmu, kas darbojas pusautomātiskā režīmā. Šāda darba režīma izvēle ir pamatota ar nepieciešamību pieņemt lēmumus par jaunizveidojamo klašu nosaukumiem, kas pilnīgi automātiskas darbības gadījumā varētu novest pie neuztveramu nosaukumu ģenerēšanas (it īpaši ja tiek definēti vairāki hierarhijas līmeņi – pie tam katrs nākamais līmenis izmanto iepriekšējas apstrādes rezultātus). Dotais algoritms arī ir aprakstīts promocijas darbā.

Tika iegūts arī **papildus rezultāts**, kas nebija uzstādīts sākotnējā uzdevuma definēšanā, bet bez tā algoritma izstrāde nebūtu iespējama – tā ir divpusložu modeļa modernizācija. Lai būtu iespējams atrisināt ierobežojumus divpusložu modeļa notācijā, ir nepieciešams doto notāciju papildināt. Tas nozīmē esošā modeļa modificēšanu, kas ietver gan hipotēžu par nepieciešamajiem uzlabojumiem izstrādi, gan arī šo hipotēžu pārbaudi. Tātad, notācijas uzlabošanai ir nepieciešams instruments, kas atbalsta iteratīvu modeļa modificēšanas procesu, veicot minimālas izmaiņas programmatūras kodā, kas spēj šo modeli apstrādāt. Kā tāds instruments, tika izvēlēta domēna-specifiska valoda, kas arī tika izstrādāta. Izstrādātā domēna-specifiskā valoda ir paredzēta divpusložu modeļa definēšanai un autors uzskata, ka to ir iespējams izmantot arī ārpus šī darba ietvariem. Piemēram, ir iespējams izmantot šo valodu kā universālu datu uzglabāšanas formātu nākotnes rīkiem, kas atbalstīs uz divpusloža modeļa izmantošanu balstītas modeļvadāmas metodes. Šī valoda ir aprakstīta promocijas darbā un tika

izmantota tā izstrādes laikā testa modeļu definēšanai. Ar dotās valodas palīdzību promocijas darba autors ir atrisinājis visus definētos divpusložu modeļa notācijas ierobežojumus, kas savukārt ļāva šo modeli izmantot koda ģenerēšanas nolūkiem.

Kopumā par promocijas **darba rezultātiem** var uzskatīt šādus:

1. Tika veikta divpusložu modeļa uzlabošana un papildināšana.
2. Tika definēti divpusložu modeļa transformācijas likumi, kas tiek piedāvāti pseidokoda veidā.
3. Tika definēts starpmodelis, ko var izmantot programmatūras koda ģenerēšanai ne tikai no divpusložu modeļa.
4. Tika piedāvāts klašu attiecību noteikšanas algoritms, kas var tikt izmantos kopā ar citām metodēm, kas arī var būt nesaistītas ar modeļvadāmo programmatūras izstrādi (piemēram, koda pārrakstīšana (angl. *refactoring*).
5. Tika pārbaudītas izstrādātas metodes praktiskās pielietošanas iespējas, izmantojot to programmatūras sistēmas izveidošanai.

Balstoties uz promocijas darba ietvaros veiktiem pētījumiem un iegūtajiem rezultātiem, ir izdarīti **šādi secinājumi**:

1. Lai gan MDSD jomā parasti tiek izmantota UML valoda artefaktu definēšanai, promocijas darba autors pēc veiktās analīzes un aptaujas uzskata to par nepiemērotu avota modeli programmatūras koda ģenerēšanai. Tas ir skaidrojams ar to, ka UML ir paredzēta gatavu risinājumu aprakstam un var būt nesaprotama problēmsfēras ekspertiem.
2. Divpusložu modelis ir saprotams gan programmatūras izstrādātājiem, gan arī citām iesaistītām pusēm (kas tika pārbaudīts veicot piedāvāta risinājuma praktisko pārbaudi – sistēmas pasūtītājs bija spējīgs ar minimālo palīdzību izveidot sistēmas modeli). Tas nozīmē to, ka to ir iespējams izmantot kā avota modeli sistēmu definēšanai.
3. Divpusložu var tikt izmantots ne tikai UML diagrammu, bet arī programmatūras koda ģenerēšanai.
4. Kā papildu promocijas darba ieguvumi tika definēti klašu attiecību noteikšanas algoritms un starpmodelis programmatūras koda ģenerēšanai. Tos ir iespējams izmantot ne tikai divpusložu modeļa transformācijām, bet arī strādājot ar cita veida avota modeļiem. Arī ir nepieciešams atzīmēt to, ka ir iespējama ne tikai objektorientētā programmatūras koda iegūšana.

5. Pēc promocijas darba laikā veiktajiem pētījumiem, divpusložu modeļa notācijas un transformācijas likumu veiktajiem uzlabojumiem ir iespējams bagātināt piedāvāto transformāciju algoritmu, kā arī veikt to atbalstoša rīka izstrādi.

Tātad, modificējot esošo divpusložu modeļa notāciju un adaptējot to transformācijām programmatūras kodā, promocijas darbā izstrādātais koda ģenerēšanas algoritms pēc tā implementācijas attiecīgajā modeļvadāmās pieejas atbalsta rīkā var būt ieviests industrijā, jo algoritms dot iespēju no problēmsfēras ekspertiem saprotamā modeļa iegūt programmatūras kodu. Kā arī pētījuma gaitā definētie spriedumi un iegūtie rezultāti var būt interesanti plaša loka speciālistiem gan industrijā, gan turpinot zinātniskos pētījumus modeļvadāmās programmatūras jomā.

Bibliogrāfija

1. Águila, I., Palma, J., Túnez, S. Milestones in Software Engineering and Knowledge Engineering History: A Comparative Review. *The Scientific World Journal*, 2014, vol. 14, 10 pages. Available from: doi:10.1155/2014/692510
2. Anderson, J.R. *Cognitive psychology and its implications*. 7th edition. USA: Worth Publishers, 2009. 469 p. ISBN 978-1429219488.
3. *AngularJS: Developer Guide: Conceptual Overview.* [skatīts 2019.g. 9.decembrī]. Pieejams: https://docs.angularjs.org/guide/concepts
4. *ANTLR*. [skatīts 2019.g. 9.decembrī]. Pieejams: http://www.antlr.org/
5. *ASM*. [skatīts 2019.g. 9.decembrī]. Pieejams: https://asm.ow2.io/
6. Asnina, E. The Formal Approach to Problem Domain Modeling Within Model Driven Architecture. In: *Proceedings of the 9th International Conference "Information Systems Implementation and Modeling" (ISIM 2006)*. Ostrava, Czech Republic, 2006, pp. 97-104.
7. Badreddin, O., Lethbridge, T., Forward, A., Elaasar, M., Aljamaan, H. Garzón, M. Enhanced Code Generation from UML Composite State Machines. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*. Lisbon, Portugal, 2014, pp. 235-245. ISBN 978-989-758-007-9. Available from: doi:10.5220/0004699602350245.
8. Baudoin, C.R. The Evolution and Ecosystem of the Unified Modeling Language. In: *Proceedings of Present and Ulterior Software Engineering (PAUSE) symposium*. USA: Springer International Publishing AG, 2017, pp. 37-46. ISBN 978-3-319-67424-7. Available from: doi:10.1007/978-3-319-67425-4_3.
9. Benoît, L., Jitia, C.-E., Jouenne, E. DSL classification. In: *Proceedings of OOPSLA 7th workshop on domain specific modeling*. USA: ACM, 2007.
10. *BPMN Specification - Business Process Model and Notation*. [skatīts 2019.g. 9.decembrī]. Pieejams: http://www.bpmn.org/
11. Brambilla, M., Cabot, J., Wimmer, M. *Model-Driven Software Engineering in Practice: Second Edition. 2nd edition*. USA: Morgan & Claypool Publishers, 2017. 208 p. ISBN 978-1627057080.
12. *C# Programming Guide*. [skatīts 2019.g. 12.decembrī]. Pieejams: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/
13. Cemus, K., Cerny, T., Matl, L., Donahoo, M.J. Aspect, Rich and Anemic Domain Models in Enterprise Information Systems. In: *Proceedings of the 42nd International Conference on SOFSEM 2016: Theory and Practice of Computer Science*. Berlin, Germany: Springer-Verlag, 2016, pp. 445-456. ISBN 978-3-662-49191-1. Available from: doi:10.1007/978-3-662-49192-8_36.
14. *CFR - yet another Java decompiler*. [skatīts 2019.g. 12.decembrī]. Pieejams: http://www.benf.org/other/cfr/
15. *Chapter 4. The class File Format*. [skatīts 2019.g. 12.decembrī]. Pieejams: https://docs.oracle.com/javase/specs/jvms/se13/html/jvms-4.html
16. *Chapter 6. The Java Virtual Machine Instruction Set*. [skatīts 2019.g. 12.decembrī]. Pieejams: https://docs.oracle.com/javase/specs/jvms/se13/html/jvms-6.html
17. Chen, P.P. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems (TODS) - Special issue: papers from the international conference on very large data bases: September 22–24*. 1976, Volume 1, Issue 1, pp. 9-36. Available from: doi:10.1145/320434.320440.

18. *Choose between .NET Core and .NET Framework for server apps*. [skatīts 2019.g. 12.decembrī]. Pieejams: https://docs.microsoft.com/en-us/dotnet/standard/choosing-core-framework-server
19. *Cucumber | Tools & techniques that elevate teams to greatness*. [skatīts 2019.g. 12.decembrī]. Pieejams: https://cucumber.io/
20. Daniel, F., Matera, M. *Mashups: Concepts, Models and Architectures*. Berlin, Germany: Springer-Verlag, 2014. 319 p. ISBN 978-3-642-55048-5.
21. Eilam, E. *Reversing: Secrets of Reverse Engineering*. 1st edition. USA: Wiley, 2005 - 624 p. ISBN 978-0764574818.
22. *Enabling Open Innovation & Collaboration | The Eclipse Foundation*. [skatīts 2019.g. 12.decembrī]. Pieejams: https://www.eclipse.org/
23. Epp, S.S. *Discrete Mathematics with Applications*. 4th Edition. USA: Cengage Learning, 2010. 984 p. ISBN 978-0495391326.
24. Evans, E. *Domain-driven design: tackling complexity in the heart of software*. USA: Addison-Wesley Professional, 2003. 560 p. ISBN 978-0321125217.
25. Floyd, R.W. Algorithm 97: Shortest path. *Communications of the ACM Magazine*, 1962, Volume 5, Issue 6, p. 345. Available from: doi:10.1145/367766.368168.
26. Fowler, M. *Anaemic Domain Model*. [skatīts 2019.g. 13.decembrī]. Pieejams: http://www.martinfowler.com/bliki/AnemicDomainModel.html
27. Fowler, M. *Refactoring: Improving the Design of Existing Code*. 2nd Edition. USA: Addison-Wesley Professional, 2018. 486 p. ISBN 978-0201485677.
28. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 1994. 416 p. ISBN 978-0201633610.
29. Gane, C., Sarson, T. *Structured systems analysis: tools and techniques*. USA: Prentice-Hall, 1979. 241 p. ISBN 978-0138545475.
30. *GitHub - fesh0r/fernflower: Unofficial mirror of FernFlower Java decompiler*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://github.com/fesh0r/fernflower
31. *Google Maps*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://maps.google.com/
32. Gopalaswamy, R., Desikan, S. *Software Testing: Principles and Practices*. 1 edition. Canada: Pearson Education, 2009. 480 p. ISBN 978-8177581218.
33. Grundspenkis, J. Causal Domain Model Driven Knowledge Acquisition for Expert Diagnosis. *System Development. Journal of Intelligent Manufacturing*, 1998, Volume 9, Issue 6, pp. 547-558. eISSN 1572-8145. ISSN 0956-5515. Available from: doi:10.1023/A:1008840303610.
34. Grune, D., Jacobs, C.J.H. *Parsing Techniques: A Practical Guide (Monographs in Computer Science)*. 2nd Edition. USA: Springer, 2010. 688 p. ISBN 978-1441919014.
35. Gurad, H.D., Mahalle, V.S. An Approach to Code Generation from UML Diagrams. *IJESRT - International Journal of Engineering Sciences & Research Technology*, 2014. pp. 421-423. ISSN 2277-9655.
36. Gusarovs, K. An Analysis on Java Programming Language Decompiler Capabilities. *Applied Computer Systems*, 2018, Volume 23, No. 2. pp. 109-117. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.2478/acss-2018-0014.
37. Gusarovs, K., Nikiforova, O., Jukss, M. A Prototype of Description Language for the Two-Hemisphere Model. *Applied Computer Systems*, 2015, Volume 18, Issue 1. pp. 15-20. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0014.
38. Gusarovs, K., Nikiforova, O. An Intermediate Model for the Code Generation from the Two-Hemisphere Model. In: *Proceedings of the 14th International Conference on Software Engineering Advances (ICSEA 2019)*, accepted for publishing. ISBN: 978-1-61208-752-8.

39. Gusarovs, K., Nikiforova, O., Giurca, A. Simplified Lisp Code Generation from the Two-hemisphere Model. *Procedia Computer Science*, 2017, Volume 104, Issue C. pp. 329-337. Available from: doi:10.1016/j.procs.2017.01.142.
40. Gusarovs, K., Nikiforova, O. Workflow Generation from the Two-Hemisphere Model. *Applied Computer Systems*, 2017, Volume 22, Issue 1. pp. 36-46. eISSN 2255-8691. ISSN 2255-8683. Available from: doi:10.1515/acss-2017-0016.
41. *Hibernate. Everything data. - Hibernate*. [skatīts 2019.g. 13.decembrī]. Pieejams: http://hibernate.org/
42. *Home | Automated Mainframe COBOL, PL/I, Natural, PowerBuilder, RPG, Delphi... modernization to Java and .NET by Blu Age*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://www.bluage.com/
43. *Home | Mono*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://www.mono-project.com
44. Hunter, D. *Guitar Effects Pedals - the Practical Handbook*. Canada: Backbeat Books, 2004. 192 p. ISBN 978-0879308063.
45. *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://www.jetbrains.com/idea/
46. *Intel® 64 and IA-32 Architectures Software Developer Manuals | Intel® Software*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://software.intel.com/en-us/articles/intel-sdm
47. *ISO/IEC 2382:2015(en), Information technology — Vocabulary*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en
48. *ISO - ISO/IEC 9075-1:2016 - Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://www.iso.org/standard/63555.html
49. *ISO - ISO/IEC 9899:2011 - Information technology — Programming languages — C*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://www.iso.org/standard/57853.html
50. *Java | Oracle*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://www.java.com/en/
51. *Java Decompiler*. [skatīts 2019.g. 13.decembrī]. Pieejams: http://java-decompiler.github.io/
52. *JavaScript | MDN*. [skatīts 2019.g. 13.decembrī]. Pieejams: https://developer.mozilla.org/en-US/docs/Web/JavaScript
53. Johnson, D.B. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 1975, Volume 4, pp. 77-84. Available from: doi: 10.1137/0204007.
54. Kleppe, A., Warmer, J., Bast W. *MDA Explained: The Model Driven Architecture – Practise and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. 170 p. ISBN 032119442X.
55. Knuth, D.E. *The Art of Computer Programming*. 1st Edition. USA: Addison-Wesley Professional, 2003. 9998 p. ISBN 978-0321751041.
56. Korf, R. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence Journal*, 1985, Volume 27, Issue 1, pp. 97-109. ISSN 0004-3702. Available from: doi:10.1016/0004-3702(85)90084-0.
57. Koshy, T. *Discrete Mathematics With Applications*. 1st Edition. USA: Academic Press, 2003. 1042 p. ISBN 978-0124211803.
58. Lasbahani, A., Chhiba, M., Tabyaoui, A. A UML Profile for Security and Code Generation. *International Journal of Electrical and Computer Engineering (IJECE)*, 2018, Volume 8, No. 6, 5278-5291 pp. ISSN 2088-8708. Available from: doi:10.11591/ijece.v8i6.pp5278-5291.
59. Luís Marques. *A defense of so-called anemic domain models. Slides of D-Lang-Silicon-Valley Meetup @ January 28, 2016*. [skatīts 2019.g. 24.decembrī]. Pieejams: http://files.meetup.com/18234529/luis_marques_anemic_domain_models.pdf

60. *macOS - Official Apple Support*. [skatīts 2019.g. 24.decembrī]. Pieejams: https://support.apple.com/macos
61. Manna, Z., Waldinger, R. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 1980, Volume 2, No. 1, 90-121 pp. ISSN 0164-0925. Available from: doi:10.1145/357084.357090.
62. Martin, R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st Edition. USA: Prentice Hall, 2008. 464 p. ISBN 978-0132350884.
63. Mernik, M., Heering, J., Sloane, A.M. When and how to develop domain-specific languages. *ACM Computing Surveys*, 2005, Volume 37, No. 4, 316-344 pp. ISSN 0360-0300. Available from: doi:10.1145/1118890.1118892.
64. *mstrobel / procyon – Bitbucket*. [skatīts 2019.g. 24.decembrī]. Pieejams: https://bitbucket.org/mstrobel/procyon
65. *.NET | Free. Cross-platform. Open Source*. [skatīts 2019.g. 24.decembrī]. Pieejams: https://dotnet.microsoft.com/
66. Niaz, I.A., Jiro, T. Mapping UML statecharts to Java code. In: *IASTED Conf. on Software Engineering*, 2004.
67. Nikiforova, O. Object Interaction as a Central Component of Object-Oriented System Analysis. In: *Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modeling Theory-Driven Development*. Portugal: SciTePress, 2010, pp 3-12. Available from: doi:10.5220/0003023000030012.
68. Nikiforova, O. System Modeling in UML with Two-Hemisphere Model Driven Approach. *Applied Computer Systems*, 2011, Volume 41, Issue 1, pp. 37-44. ISSN 2255-8691. Available from: doi:10.2478/v10143-010-0022-x.
69. Nikiforova, O. Two Hemisphere Model Driven Approach for Generation of UML Class Diagram in the Context of MDA. *e-Informatica Software Engineering Journal*, 2009, Volume 3, pp. 59-72.
70. Nikiforova, O., Ahilcenoka, D., Ungurs, D., Gusarovs, K., Kozacenko, L. Several Issues on the Layout of the UML Sequence and Class Diagram. In: *Proceedings of the 9th International Conference on Software Engineering Advances (ICSEA 2014)*, 2014, pp. 40-47.
71. Nikiforova, O., Bohomaz, Y., Gusarovs, K. A Comparison of the Implementation Means for Development of Modelling Tool. In: *Proceedings of the 2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS 2017)*, 2017, pp. 1-6. Available from: doi:10.1109/WITS.2017.7934623.
72. Nikiforova, O., El Marzouki, N., Gusarovs, K., Vangheluwe, H., Bures, T., Al-Ali, R., Iacono, M., Esquivel, P.O., Leon, F. The Two-Hemisphere Modelling Approach to the Composition of Cyber-Physical Systems. In: *In Proceedings of the 12th International Conference on Software Technologies*. Portugal: SciTePress, 2017, pp. 286-293. ISBN 978-989-758-262-2. Available from: doi:10.5220/0006424902860293.
73. Nikiforova, O., Gorbiks, O., Gusarovs, K., Ahilcenoka, D., Bajovs, A., Kozacenko, L., Skindere, N., Ungurs, D. Development of BrainTool for Generation of UML Diagrams from the Two-hemisphere Model Based on the Two-Hemisphere Model Transformation Itself. In: *Proceedings of the International Scientific Conference "Applied Information and Communication Technologies"*. Latvia: LLU, 2013, pp. 267-274. ISSN 2255-8586.
74. Nikiforova, O., Gusarovs, K. Comparison of BrainTool to Other UML Modeling and Model Transformation Tools. *AIP Conference Proceedings*, 2017, Volume 1863, No. 1, id. 330005. Available from: doi:10.1063/1.4992503.
75. Nikiforova, O., Gusarovs, K. Kozacenko, L., Ahilcenoka, D., Ungurs, D. An Approach to Compare UML Class Diagrams Based on Semantical Features of Their Elements. In:

Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA 2015). Wilmington: IARIA, 2015, pp. 147-153. ISBN 978-1-61208-438-1.

76. Nikiforova, O., Gusarovs, K. Anemic Domain Model vs Rich Domain Model to Improve the Two-Hemisphere Model-Driven Approach. *Applied Computer Systems*, 2020, Volume 25, Issue 1. (Accepted for publication).
77. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. BrainTool. A Tool for Generation of the UML Class Diagrams. In: *Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012. Lisbon: IARIA, 2012, pp. 60-69. ISBN 9781612082301.
78. Nikiforova, O., Gusarovs, K., Gorbiks, O., Pavlova, N. Improvement of the Two-Hemisphere Model-Driven Approach for Generation of the UML Class Diagram. *Applied Computer Systems*, 2013, Volume 14, Issue 1, pp. 19-30. ISSN 2255-8691. Available from: doi:10.2478/acss-2013-0003.
79. Nikiforova, O., Gusarovs, K., Ressin, A. An Approach to Generation of the UML Sequence Diagram from the Two-Hemisphere Model. In: *Proceedings of the 11th International Conference on Software Engineering Advances (ICSEA 2016)*. Wilmington: IARIA, 2016, pp. 142-149. ISBN 978-1-61208-498-5.
80. Nikiforova, O., Kirikova, M. Two-Hemisphere Model Driven Approach: Engineering Based Software Development. In: *Advanced Information Systems Engineering. CAiSE 2004. Lecture Notes in Computer Science*, Volume 3084. Berlin: Springer, 2004, pp. 219-233. ISBN 978-3-540-22151-7. Available from: doi:10.1007/978-3-540-25975-6_17.
81. Nikiforova, O., Kozacenko, L., Ahilcenoka, D. UML Sequence Diagram: Transformation from the Two-Hemisphere Model and Layout. *Applied Computer Systems*, 2013, Volume 14, Issue 1, pp. 31-41. ISSN 2255-8691. Available from: doi:10.2478/acss-2013-0004.
82. Nikiforova, O., Kozacenko, L., Ungurs, D., Ahilcenoka, D., Bajovs, A., Skindere, N., Gusarovs, K., Jukss, M. BrainTool v2.0 for Software Modeling in UML. *Applied Computer Systems*, 2015, Volume 16, Issue 1, pp 33-42. ISSN 2255-8691. Available from: doi:10.1515/acss-2014-0011.
83. Nikiforova, O., Kozacenko, L., Ahilcenoka, D., Gusarovs, K., Ungurs, D., Jukss, M. Comparison of the Two-Hemisphere Model-Driven Approach to Other Methods for Model-Driven Software Development. *Applied Computer Systems*, 2016, Volume 18, Issue 1, pp. 5-14. ISSN 2255-8691. Available from: doi:10.1515/acss-2015-0013.
84. Nikiforova, O., Pavlova, N. Development of the Tool for Generation of UML Class Diagram from Two-hemisphere model. In: *Proceedings of The Third International Conference on Software Engineering Advances (ICSEA)*. USA: IEEE, 2008, pp. 105-112. ISBN 978-1-4244-3218-9.
85. Nikiforova, O., Pavlova, N., Gusarovs, K., Gorbiks, O., Vorotilovs, J., Zaharovs, A., Umanovskis, D., Sejans, J. Development of the Tool for Transformation of the Two-Hemisphere Model to the UML Class Diagram: Technical Solutions and Lessons Learned. In: *Proceedings of the 5th Internation Scientific Conference "Applied Information and Communication Technology 2012"*. Latvia: LLU, 2012, pp. 11-19. ISBN 978-9984-48-065-7.
86. Nikiforova, O., Sukovskis, U., Gusarovs, K. Application of the Two-Hemisphere Model Supported by BrainTool: Football Game Simulation. *AIP Conference Proceedings*, 2015, Volume 1648, No. 1, id. 310004. Available from: doi:10.1063/1.4912557.
87. Noureen, A., Amjad, A., Azam, F. Model Driven Architecture - Issues, Challenges and Future Directions. *JSW*, 2016, Volume 11, pp. 924-933. Available from: 10.17706/jsw.11.9.924-933.

88. Omar, E.B., Brahim, B., Taoufiq, G. Automatic code generation by model transformation from sequence diagram of system's internal behavior. *International Journal of Computer and Information Technology*, 2012, Volume 1, Issue 2, pp. 129-146.
89. *OpenJDK Mercurial Repositories*. [skatīts 2019.g. 25.decembrī]. Pieejams: https://hg.openjdk.java.net/jdk
90. *Overview: PIC18F45Q10 - 8-bit Microcontrollers.* [skatīts 2020.g. 25.martā]. Pieejams: https://www.microchip.com/wwwproducts/en/PIC18F45Q10
91. Paige, R.F., Zolotas, A., Kolovos, D. The Changing Face of Model-Driven Engineering. In: *Proceedings of Present and Ulterior Software Engineering (PAUSE) symposium*. Germany: Springer, 2017, pp. 103-118. ISBN 978-3-319-67424-7.
92. Perisic, B. Model Driven Software Development – State of the Art and Perspectives. In: *Proceedings of INFOTEH 2014*, Volume 13. pp. 1237-1248.
93. *PHP: Hypertext Preprocessor*. [skatīts 2019.g. 25.decembrī]. Pieejams: https://www.php.net/
94. *Regular-Expressions.info - Regex Tutorial, Examples and Reference - Regexp Patterns*. [skatīts 2019.g. 25.decembrī]. Pieejams: https://www.regular-expressions.info/
95. Ritchie, D. M., Kernighan, B. W. *The C Programming Language*. Second Edition. USA: Prentice Hall, 1988. 272 p. ISBN 978-0131103627.
96. *Ruby Programming Language*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://www.ruby-lang.org/en/
97. Salomon, D. *Assemblers and Loaders*. USA: Prentice Hall, 1993. 308 p. ISBN 978-0130525642.
98. Sejans, J., Nikiforova, O. Practical Experiments with Code Generation from the UML Class Diagram. In: *Proceedings of MDA&MDSD 2011, 3rd International Workshop on Model Driven Architecture and Modeling Driven Software Development*. Lisbon: SciTePress, 2011, pp. 57-67. ISBN 978-989-8425-59-1.
99. Shiferaw, M.K., Jena, A.K. Code Generator for Model-Driven Software Development Using UML Models. *Proceedings of 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, 2018, pp. 1671-1678. Available from: doi:10.1109/iceca.2018.8474690.
100. Skiena, S.S. *The Algorithm Design Manual*. 2nd Edition. Germany: Springer, 2008. 748 p. ISBN 978-1849967204.
101. *Standard C++*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://isocpp.org/
102. Stevens, W.P., Myers, G.J., Constantine, L.L. Structured Design. *IBM Systems Journal*, 1974, Volume 13, Issue 2, pp. 115-139. Available from: doi:10.1147/sj.132.0115.
103. Sunitha, E.V., Philip, S. Automatic Code Generation From UML State Chart Diagrams. *IEEE Access*, 2019, Volume 7, pp. 8591-8606. ISSN 2169-3536. Available from: doi:10.1109/ACCESS.2018.2890791.
104. Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1972, Volume 1, Issue 2, pp. 146-160. Available from: doi:10.1137/0201010.
105. Terekhov, A., Bryksin, T., Litvinov, Y. How to Make Visual Modeling More Attractive to Software Developers. In: *Present and Ulterior Software Engineering*. Germany: Springer, 2017, pp. 139-152. ISBN 978-3-319-67424-7.
106. *The Anaemic Domain Model is no Anti-Pattern, it's a SOLID design | SAPM: Course Blog*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://blog.inf.ed.ac.uk/sapm/2014/02/04/the-anaemic-domain-model-is-no-anti-pattern-its-a-solid-design/
107. *The Java® Virtual Machine Specification*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://docs.oracle.com/javase/specs/jvms/se13/html/index.html

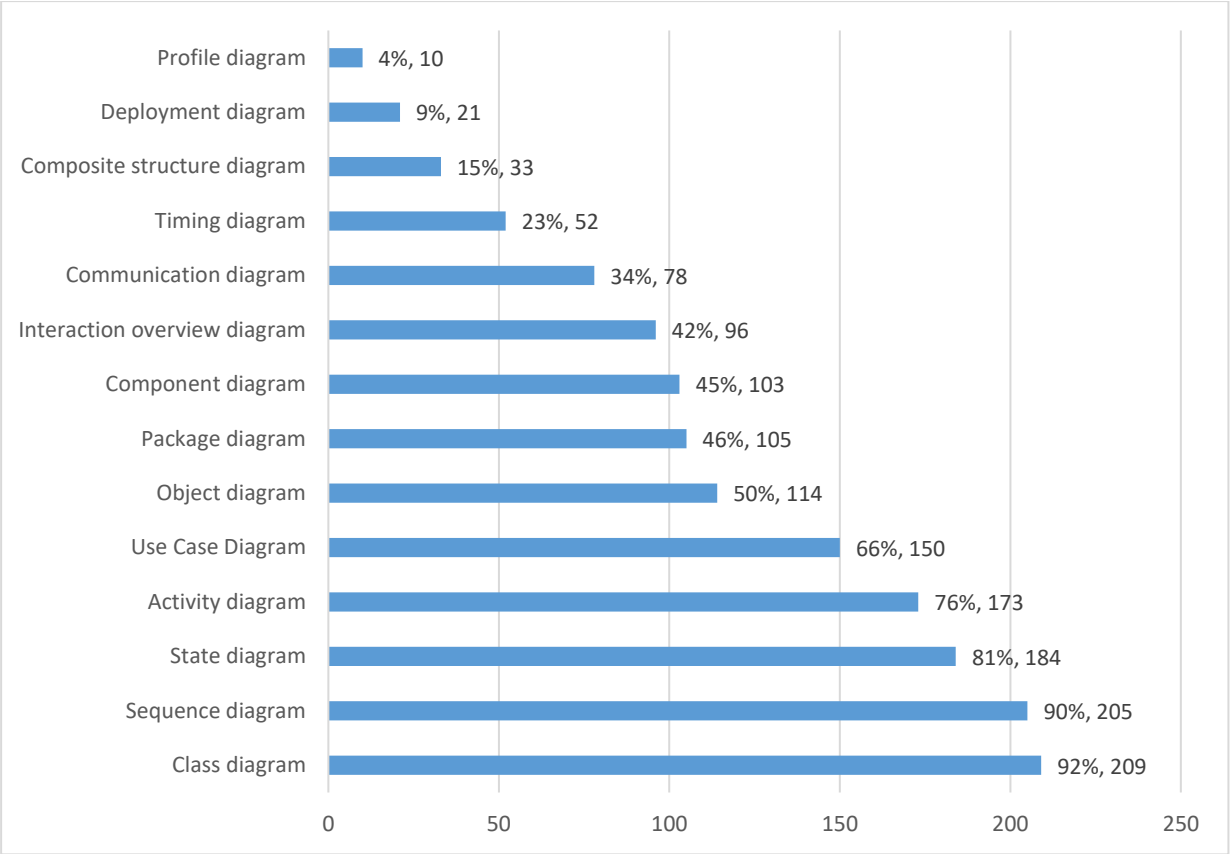
108. *The LEX & YACC Page*. [skatīts 2019.g. 26.decembrī]. Pieejams: http://dinosaur.compilertools.net/
109. *The Linux Kernel Archives*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://www.kernel.org/
110. *TIOBE Index | TIOBE - The Software Quality Company*. [skatīts 2018.g. 1.decembrī]. Pieejams: https://www.tiobe.com/tiobe-index/
111. *Trygve/MVC*. [skatīts 2019.g. 26.decembrī]. Pieejams: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html
112. *Unicode 12.1.0*. [skatīts 2019.g. 26.decembrī]. Pieejams: http://www.unicode.org/versions/Unicode12.1.0/
113. *Unified Modeling Language*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://www.omg.org/spec/UML/
114. Van Cam, P., Radermacher, A., Gérard, S., Shuai, L. Complete Code Generation from UML State Machine. *MODELSWARD 2017*, 2017, pp. 208-219. Available from: doi:10.5220/0006274502080219.
115. van der Aalst, W.P.M. Business process management: A comprehensive survey. *ISRN Software Engineering, 2013*, Volume 2013, id. 507984, pp. 1-37. Available from: doi:10.1155/2013/507984.
116. *Visual Basic docs - get started, tutorials, reference. | Microsoft Docs*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://docs.microsoft.com/en-us/dotnet/visual-basic/
117. *W3C XML Schema*. [skatīts 2019.g. 26.decembrī]. Pieejams: http://www.w3.org/XML/Schema
118. Wang, Z. A JAVA Code Generation Method based on XUML. *IOP Conference Series: Materials Science and Engineering*, 2019, Volume 563, id. 052001. Available from: doi:10.1088/1757-899x/563/5/052001.
119. *Welcome to Python.org*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://www.python.org/
120. *Windows 10 OS, Computers, Apps, & More | Microsoft*. [skatīts 2019.g. 26.decembrī]. Pieejams: https://www.microsoft.com/en-us/windows
121. Wright, D.R., *Finite State Machines (CSC215 Class Notes)*. [skatīts 2017.g. 20.septembrī]. Pieejams: http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf
122. Zusane, U.I., Nikiforova, O., Gusarovs, K. Several Issues on the Model Interchange Between Model-Driven Software Development Tools. In: *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA 2015)*. Wilmington: IARIA, 2015, pp. 451-457. ISBN 978-1-61208-438-1.

Pielikumi

1. pielikums

Programmatūras izstrādātāju un arhitektu aptaujas rezultāti

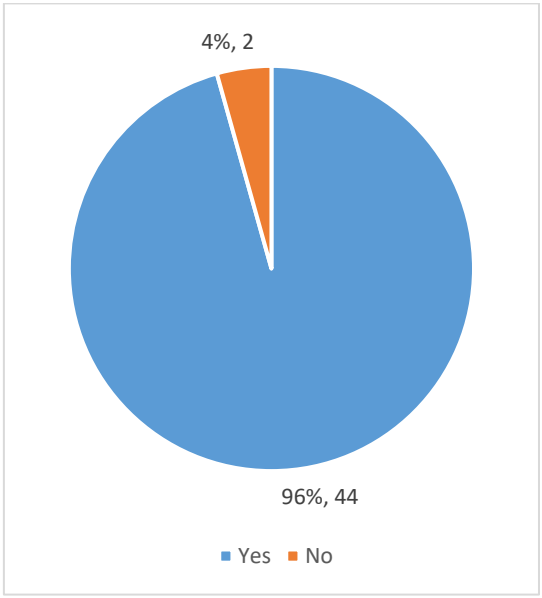
As a Software Developer/Software Architect I can define following UML Diagrams (0 or more):



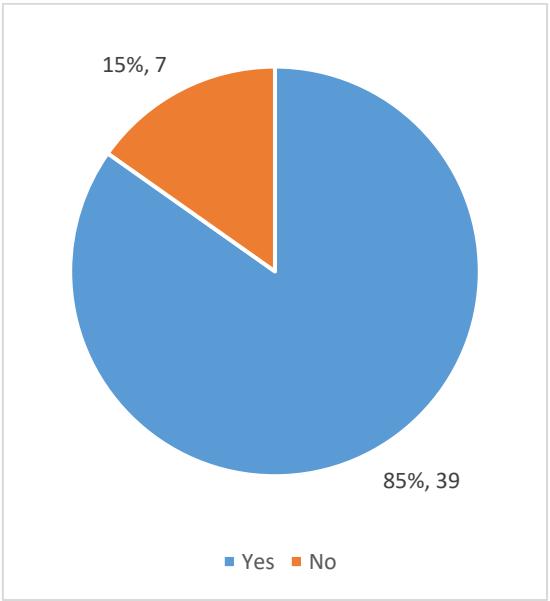
2. pielikums

Biznesa analītiķu aptaujas rezultāti

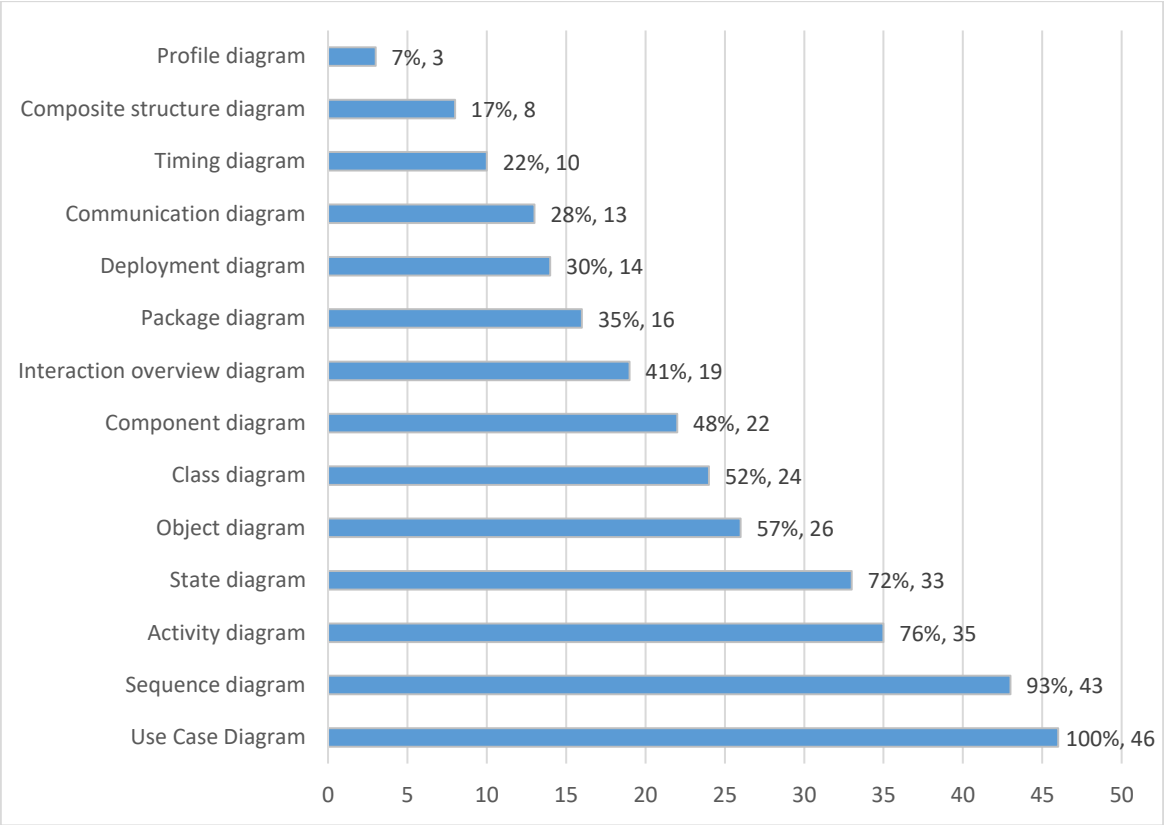
- I agree for my anonymous free text answers to be provided as a part of finished PhD thesis.



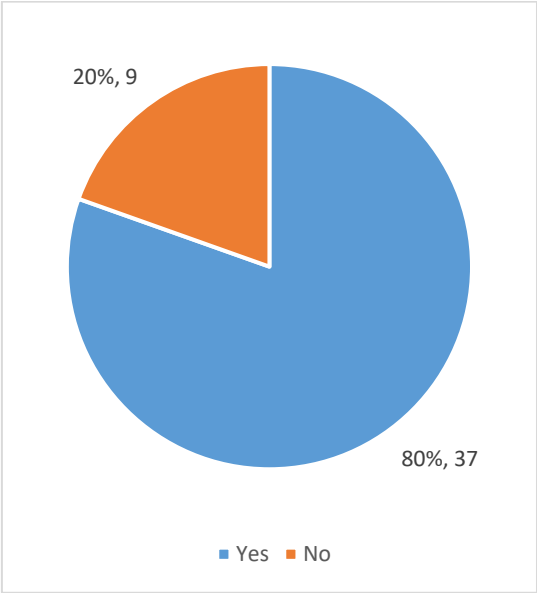
- As a Business Analyst I am familiar with business process modelling



- As a Business Analyst I can define following UML Diagrams (0 or more)



- I know, what ER-diagram is

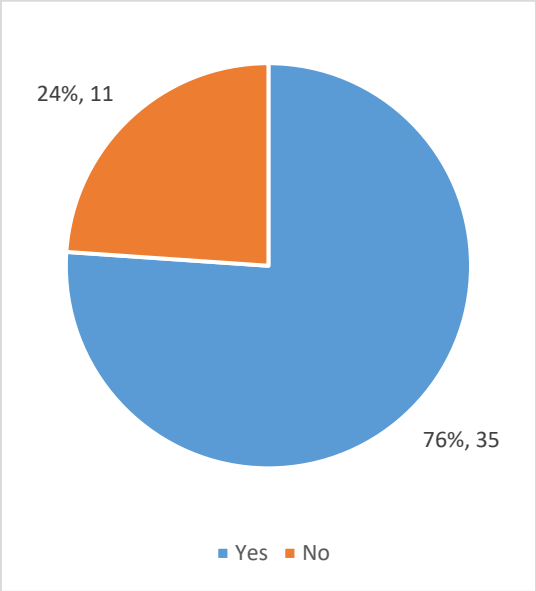


- If answered "Yes" in previous question, describe ER-diagram in 1-2 sentences
    - A way of representing data and relations between it.
    - Key diagram for database (key entities) modeling.

- ER-diagram describes entities in DB, relations between them and their properties.
- High level entity relationship model that can be used to illustrate, how people, objects or concepts relate to each other within a system.
- ER-diagram is data model representation stressing object relationships and cardinalities of those relationships, and possibly detailing various attributes of the objects.
- Entity Relationship diagrams basically represents how and what data is stored in a database. Represented differently based on model type - conceptual, logical or physical data.
- Graphical representation of subjects and their relation between each other in a specific domain. Originally, not only subjects, but also relationships are considered first-class citizens of the diagram.
- Entity relationship diagram shows data model definition on entity level (conceptual model) and relations between these entities
- ER diagram is way of describing entities and their relations. It also can include attributes/properties of each entity and can be used for physical data model design.
- Diagram for designing DB.
- Data model as entities and relationship.
- An easy way to represent data structures.
- Data model, that can be understood by people with different software development skills. Includes data structures, their attributes and relations.
- Can be also called concept diagram. Shows domain model of the system
- A way of describing data.
- Data model.
- Set of domain objects, their attributes and relations. Usable for database modeling.
- Physical or logical model of database.
- A way of presenting data to clients and developers by showing relations between different data types and their attributes.
- High level view of data structures used in a system.
- Entities, their properties, relations between them.

- Database diagram. Can be shown to the clients and developers.
- Also, can be called concept model. Used to describe different types of domain objects.
- Entity relationship diagram displays the relationships of entity set stored in a database.
- Model that reflects key entities and relations in a system.
- An overview to underlying data structures in a system.
- Visual representation of a database model. Can be used throughout all stages of system development.
- Something I can show both to clients and developers when describing data model.
- Domain design.
- Graphical representation of domain model in an analyzed system. Consists of entities, their properties and relations.

- I know, what Business Process Diagram is



- If answered "Yes" in previous question, describe Business Process Diagram in 1-2 sentences
  - Model for representing processes and data passed between them, as well as actors responsible for process execution.
  - If referred to BPMN it is more user-friendly way to represent processes. UML should never be used in communication to business people.
  - Business Process Diagram used to illustrate business process by showing, how the process starts, tasks necessary to complete this

process, their sequence and conditions, involved stakeholders and other systems, by representing business rules that should be followed to complete the shown business process.

- It is a visual representation of the business process.
- Business process diagram is business process model visual representation depicting the process flow with its various participants, events, constraints and outcomes.
- Graphical notation of a business process.
- Graphical representation of one of the business processes in the organization to be backed up by the IT System. Typically depicted as a sequence of activities within certain boundaries with conditional transitions.
- BPM gives an opportunity to describe business process in a visual way. Can be developed on a different levels - overview, detailed process models (including cross references) etc.
- Business process diagram describes some process happening in the enterprise/real world. It captures different details including related actors, resources, conditions for state changes and transitions between these elements.
- Process flow via activities and control flow elements.
- A way of representing, what happens in the system.
- Description of processes, that are happening in the system. As an ER-diagram, can be understood by almost everyone.
- A visual representation of the business process - events, data flows, sequence, conditions.
- Description of the business process, similar to the flowchart.
- Way of representing the processes, data passed between them and preconditions for the process execution.
- A synonym to data flow diagram, that shows, what data and in which way is being processed in the system.
- Sequence of activities executed by different actors with different outcomes.
- A way of representing the analyzed business process, i.e. event sequence in an easy and understandable form.

- Notation for the business process representation.
- Modern analogue of the flow chart - shows what and in which order happens in the system.
- Alternative to the UML sequence diagram. Consists of processes, data passed between them, responsible stakeholders and outcomes.
- A set of processes and transitions between them, including conditions and exchanged data.
- An easy way to represent, what actually happens in the system - events, activities.
- A diagram that depicts a directed flow of activities in the system being developed.
- Helpful diagram for business analysts that shows the parties and systems involved with a particular process, as well as the data used.

Divpusložu modeli aprakstošas domēna-specifiskās valodas ANTLR likumi

```
grammar Thml;

model
    :   statement* EOF
    ;

statement
    :   conceptDefine
    |   processDefine
    |   dataFlowDefine
    |   processModelDefine
    ;

processModelDefine
    :   DEFINE PROCESS MODEL Identifier processModelParameter* endProcessModel
    ;

processDefine
    :   DEFINE PROCESS Identifier guardParameter? processParameter* endProcess
    ;

conceptDefine
    :   DEFINE CONCEPT Identifier conceptParameter* endConcept
    ;

dataFlowDefine
    :   DEFINE DATAFLOW Identifier FROM Identifier TO Identifier dataFlowParameter*
endDataFlow
    ;

nameDefine
    :   glue NAME StringLiteral
    ;

descriptionDefine
    :   glue DESCRIPTION StringLiteral
    ;

commentDefine
    :   glue COMMENT StringLiteral
    ;

attributeDefine
    :   glue visibility* ATTRIBUTE dataTypeDefine
    ;

processRefDefine
    :   glue PROCESS Identifier
    ;

performerDefine
    :   glue PERFORMER StringLiteral
    ;

dataTypeDefine
    :   StringLiteral '(' Identifier ')' cardinalityDefine?
    ;

cardinalityDefine
```

```
    :   HAVING attributeCardinality CARDINALITY
    ;

processTypeDefine
    :   glue TYPE processType
    ;

glue
    :   WITH
    |   AND
    ;

processType
    :   INTERNAL
    |   EXTERNAL
    ;

attributeCardinality
    :   SINGLE
    |   ARRAY
    |   COLLECTION
    ;

visibility
    :   PRIVATE
    |   PROTECTED
    |   PUBLIC
    ;

conceptParameter
    :   nameDefine
    |   attributeDefine
    |   commentDefine
    |   descriptionDefine
    ;

processParameter
    :   nameDefine
    |   processTypeDefine
    |   performerDefine
    |   commentDefine
    |   descriptionDefine
    ;

guardParameter
    :   WITH GUARD StringLiteral
    ;

processModelParameter
    :   nameDefine
    |   processRefDefine
    |   commentDefine
    |   descriptionDefine
    ;

dataFlowParameter
    :   nameDefine
    |   glue PARAMETER dataTypeDefine
    |   commentDefine
    |   descriptionDefine
    ;

endConcept
    :   END CONCEPT
    ;

endProcess
    :   END PROCESS
```

```
    ;

endProcessModel
    :   END PROCESS MODEL
    ;

endDataFlow
    :   END DATAFLOW
    ;

PARAMETER
    :   'PARAMETER'
    ;

GUARD
    :   'GUARD'
    ;

DEFINE
    :   'DEFINE'
    ;

LET
    :   'LET'
    ;

PROCESS
    :   'PROCESS'
    ;

CONCEPT
    :   'CONCEPT'
    ;

END
    :   'END'
    ;

NAME
    :   'NAME'
    ;

ATTRIBUTE
    :   'ATTRIBUTE'
    ;

WITH
    :   'WITH'
    ;

AND
    :   'AND'
    ;

INTERNAL
    :   'INTERNAL'
    ;

EXTERNAL
    :   'EXTERNAL'
    ;

TYPE
    :   'TYPE'
    ;

SINGLE
    :   'SINGLE'
```

```
    ;

ARRAY
    :   'ARRAY'
    ;

COLLECTION
    :   'COLLECTION'
    ;

CARDINALITY
    :   'CARDINALITY'
    ;

HAVING
    :   'HAVING'
    ;

PERFORMER
    :   'PERFORMER'
    ;

DATAFLOW
    :   'DATA FLOW'
    ;

FROM
    :   'FROM'
    ;

TO
    :   'TO'
    ;

DESCRIPTION
    :   'DESCRIPTION'
    ;

COMMENT
    :   'COMMENT'
    ;

MODEL
    :   'MODEL'
    ;

PRIVATE
    :   'PRIVATE'
    ;

PROTECTED
    :   'PROTECTED'
    ;

PUBLIC
    :   'PUBLIC'
    ;

fragment
IdentifierLetter
   :  [a-zA-Z]
   ;

fragment
IdentifierLetterOrDigit
   :  [a-zA-Z0-9]
   ;
```

```
Identifier
   :  IdentifierLetter IdentifierLetterOrDigit*
   ;

fragment
EscapeSequence
   :  '\\' [btnfr"'\\]
   ;

fragment
StringCharacters
   :  StringCharacter+
   ;

fragment
StringCharacter
   :  ~["\\]
    |  EscapeSequence
   ;

StringLiteral
   :  '"' StringCharacters? '"'
   ;

WhiteSpace
   :  [ \t\r\n\u000C]+ -> skip
   ;
```

Piemēra divpusložu modelis

```
DEFINE CONCEPT PhoneNumber
    WITH NAME "Phone Number"
    AND PUBLIC ATTRIBUTE "type"(String)
    AND PUBLIC ATTRIBUTE "value"(String)
END CONCEPT

DEFINE CONCEPT UserInfo
    WITH NAME "User Information"
    AND PUBLIC ATTRIBUTE "name"(String)
    AND PUBLIC ATTRIBUTE "surname"(String)
    AND PUBLIC ATTRIBUTE "phone numbers"(PhoneNumber) HAVING ARRAY
CARDINALITY
END CONCEPT

DEFINE CONCEPT FoundUser
    WITH NAME "Found User"
    AND PUBLIC ATTRIBUTE "name"(String)
    AND PUBLIC ATTRIBUTE "surname"(String)
    AND PUBLIC ATTRIBUTE "phone number"(PhoneNumber)
END CONCEPT

DEFINE PROCESS GetPhoneNumber
    WITH NAME "Get Phone Number"
    AND TYPE EXTERNAL
END PROCESS

DEFINE PROCESS ValidatePhoneNumber
    WITH NAME "Validate Phone Number"
    AND TYPE INTERNAL
END PROCESS

DEFINE PROCESS OutputSearchResult
    WITH NAME "Output Search Result"
    AND TYPE EXTERNAL
END PROCESS

DEFINE PROCESS PrepareValidationError
    WITH GUARD "Invalid Phone Number"
    AND NAME "Prepare Validation Error"
    AND TYPE INTERNAL
END PROCESS

DEFINE DATA FLOW UserInput FROM GetPhoneNumber TO ValidatePhoneNumber
    WITH NAME "User Input"
    AND PARAMETER "phone"(String)
    AND PARAMETER "type"(String)
END DATA FLOW

DEFINE DATA FLOW ValidationError FROM ValidatePhoneNumber TO
PrepareValidationError
    WITH NAME "Validation Error"
END DATA FLOW
```

```
DEFINE DATA FLOW ErrorSearchResult FROM PrepareValidationError TO
OutputSearchResult
    WITH NAME "Search Failed"
    AND PARAMETER "found"(Boolean)
    AND PARAMETER "error"(String)
END DATA FLOW

DEFINE PROCESS FindPhonesInDatabase
    WITH NAME "Find Phones In DB"
    AND TYPE INTERNAL
END PROCESS

DEFINE DATA FLOW DbQuery FROM ValidatePhoneNumber TO FindPhonesInDatabase
    WITH NAME "DB Query"
    AND PARAMETER "phone"(String)
    AND PARAMETER "phoneType"(String)
END DATA FLOW

DEFINE PROCESS PeekNextUser
    WITH GUARD "Results are not empty"
    AND NAME "Peek Next User"
    AND TYPE INTERNAL
END PROCESS

DEFINE DATA FLOW DbQueryRes FROM FindPhonesInDatabase TO PeekNextUser
    WITH NAME "Users With Given Phone"
    AND PARAMETER "users"(UserInfo) HAVING ARRAY CARDINALITY
    AND PARAMETER "phoneType"(String)
END DATA FLOW

DEFINE PROCESS MatchPhone
    WITH NAME "Match Phone"
    AND TYPE INTERNAL
END PROCESS

DEFINE DATA FLOW SingleResult FROM FindPhonesInDatabase TO MatchPhone
    WITH NAME "Single Found Result"
    AND PARAMETER "user"(UserInfo)
    AND PARAMETER "phoneType"(String)
END DATA FLOW

DEFINE DATA FLOW UserToMatch FROM PeekNextUser TO MatchPhone
    WITH NAME "User"
    AND PARAMETER "user"(UserInfo)
    AND PARAMETER "phoneType"(String)
END DATA FLOW

DEFINE DATA FLOW UsersLeft FROM MatchPhone TO PeekNextUser
    WITH NAME "Users Left To Check"
END DATA FLOW

DEFINE DATA FLOW FoundResult FROM MatchPhone TO OutputSearchResult
    AND NAME "Matched User"
    AND PARAMETER "found"(Boolean)
    AND PARAMETER "user"(FoundUser)
END DATA FLOW

DEFINE DATA FLOW NotFoundResult FROM PeekNextUser TO OutputSearchResult
    AND NAME "No Match"
    AND PARAMETER "found"(Boolean)
END DATA FLOW
```

```
DEFINE PROCESS MODEL FindUserByPhoneNumber
    WITH NAME "Find User By Phone Number"
    AND PROCESS GetPhoneNumber
    AND PROCESS ValidatePhoneNumber

    AND PROCESS PrepareValidationError

    AND PROCESS FindPhonesInDatabase

    AND PROCESS PeekNextUser
    AND PROCESS MatchPhone

    AND PROCESS OutputSearchResult
END PROCESS MODEL
```

Elektroģitāras efektu pārslēgšanas sistēmas kods

```
typedef struct {
    unsigned char current_bank;
    unsigned char current_preset;
    unsigned char mode;
    unsigned char current_loop[8];
} state_t;

typedef struct {
    unsigned char up_pressed;
    unsigned char down_pressed;
    unsigned char edit_pressed;
    unsigned char preset_pressed[8];
} input_t;

typedef struct {
    state_t state;
    input_t input;
} read_input_result_t;

typedef struct {
    unsigned char direction;
    state_t state;
    unsigned char preset;
} decode_command_result_t;

typedef struct {
    unsigned char loop;
    state_t state;
} decode_edit_command_result_t;

read_input_result_t read_input(state_t state) {
    read_input_result_t result;
    result.state = state;
    result.input.up_pressed = PORTAbits.RA0;
    result.input.down_pressed = PORTAbits.RA1;
    result.input.edit_pressed = PORTCbits.RC2;
    result.input.preset_pressed[0] = PORTAbits.RA2;
    result.input.preset_pressed[1] = PORTAbits.RA3;
    result.input.preset_pressed[2] = PORTAbits.RA4;
    result.input.preset_pressed[3] = PORTAbits.RA5;
    result.input.preset_pressed[4] = PORTAbits.RA7;
    result.input.preset_pressed[5] = PORTAbits.RA6;
    result.input.preset_pressed[6] = PORTCbits.RC0;
    result.input.preset_pressed[7] = PORTCbits.RC1;

    return result;
}

state_t toggle_loop(unsigned char loop, state_t state) {
    state.current_loop[loop] = !state.current_loop[loop];
    return state;
}

decode_command_result_t decode_command(state_t state, input_t input) {
    decode_command_result_t result;
    result.state = state;
    result.direction = 0;
    result.preset = 8;
    if (input.up_pressed) {
```

```
        result.direction = 1;
    } else if (input.down_pressed) {
        result.direction = -1;
    }

    for (unsigned char i = 0; i < 8; i++) {
        if (input.preset_pressed[i]) {
            result.preset = i;
            break;
        }
    }

    return result;
}

state_t finish_editing(state_t state) {
    state.mode = 0;
    return state;
}

state_t update_state(state_t state) {
    if (state.mode == 0) {
        eeprom_write(0, state.current_bank);
        eeprom_write(1, state.current_preset);
    } else {
        unsigned int addr = 2 + state.current_bank * sizeof(unsigned char[8])
                + state.current_preset * sizeof(unsigned char);
        unsigned char val = (state.current_loop[0] << 7)
                | (state.current_loop[1] << 6)
                | (state.current_loop[2] << 5)
                | (state.current_loop[3] << 4)
                | (state.current_loop[4] << 3)
                | (state.current_loop[5] << 2)
                | (state.current_loop[6] << 1)
                | state.current_loop[7];
        eeprom_write(addr, val);
    }

    return state;
}

decode_edit_command_result_t decode_edit_command(state_t state, input_t input) {
    decode_edit_command_result_t result;
    result.state = state;
    result.loop = 8;

    for (unsigned char i = 0; i < 8; i++) {
        if (input.preset_pressed[i]) {
            result.loop = i;
            break;
        }
    }

    return result;
}


state_t update_current_bank(unsigned char direction, state_t state) {
    state.current_bank = state.current_bank = (state.current_bank + direction) &
127;
    return state;
}

state_t switch_loops(state_t state) {
    PORTDbits.RD0 = state.current_loop[0];
    PORTDbits.RD1 = state.current_loop[1];
    PORTDbits.RD2 = state.current_loop[2];
    PORTDbits.RD3 = state.current_loop[3];
```

```
    PORTDbits.RD4 = state.current_loop[4];
    PORTDbits.RD5 = state.current_loop[5];
    PORTDbits.RD6 = state.current_loop[6];
    PORTDbits.RD7 = state.current_loop[7];
    return state;
}

state_t update_current_preset(state_t state, unsigned char preset) {
    state.current_preset = preset;
    return state;
}

void stop(void) {
}

state_t initialize(void) {
    state_t result;

    result.current_bank = eeprom_read(0);
    result.current_preset = eeprom_read(1);
    unsigned int addr = 2 + result.current_bank * sizeof(unsigned char[8])
                + result.current_preset * sizeof(unsigned char);
    unsigned char val = eeprom_read(addr);

    result.current_loop[0] = (val >> 7) & 1;
    result.current_loop[1] = (val >> 6) & 1;
    result.current_loop[2] = (val >> 5) & 1;
    result.current_loop[3] = (val >> 4) & 1;
    result.current_loop[4] = (val >> 3) & 1;
    result.current_loop[5] = (val >> 2) & 1;
    result.current_loop[6] = (val >> 1) & 1;
    result.current_loop[7] = val & 1;

    return result;
}

state_t start_editing(state_t state) {
    state.mode = 1;
    return state;
}

void main(void) {
    state_t state;
    input_t input;
    unsigned char loop;
    unsigned char direction;
    unsigned char preset;
    unsigned char should_loop;
    read_input_result_t read_input_result;
    unsigned char not_editing;
    decode_command_result_t decode_command_result;
    unsigned char mode_change;
    unsigned char preset_change;
    unsigned char bank_change;
    unsigned char loop_change;
    unsigned char editing;
    decode_edit_command_result_t decode_edit_command_result;

    state = initialize();

    for (should_loop = 1; should_loop; state = switch_loops(state)) {
        read_input_result = read_input(state);
        state = read_input_result.state;
        input = read_input_result.input;

        not_editing = state.mode == 0;
        if (not_editing) {
            decode_command_result = decode_command(state, input);
```

```
            direction = decode_command_result.direction;
            state = decode_command_result.state;
            preset = decode_command_result.preset;

            mode_change = input.edit_pressed;
            if (mode_change) {
                state = start_editing(state);
            }

            preset_change = preset < 8;
            if (preset_change) {
                state = update_current_preset(state, preset>);
            }

            bank_change = direction != 0;
            if (bank_change) {
                state = update_current_bank(direction, state);
            }
        }

        editing = state.mode == 1;
        if (editing) {
            decode_edit_command_result = decode_edit_command(state, input);
            loop = decode_edit_command_result.loop;
            state = decode_edit_command_result.state;

            mode_change = input.edit_pressed;
            if (mode_change) {
                state = finish_editing(state);
            }

            loop_change = loop < 8;
            if (loop_change) {
                state = toggle_loop(loop, state);
            }
        }

        state = update_state(state);
    }

    stop();
}
```