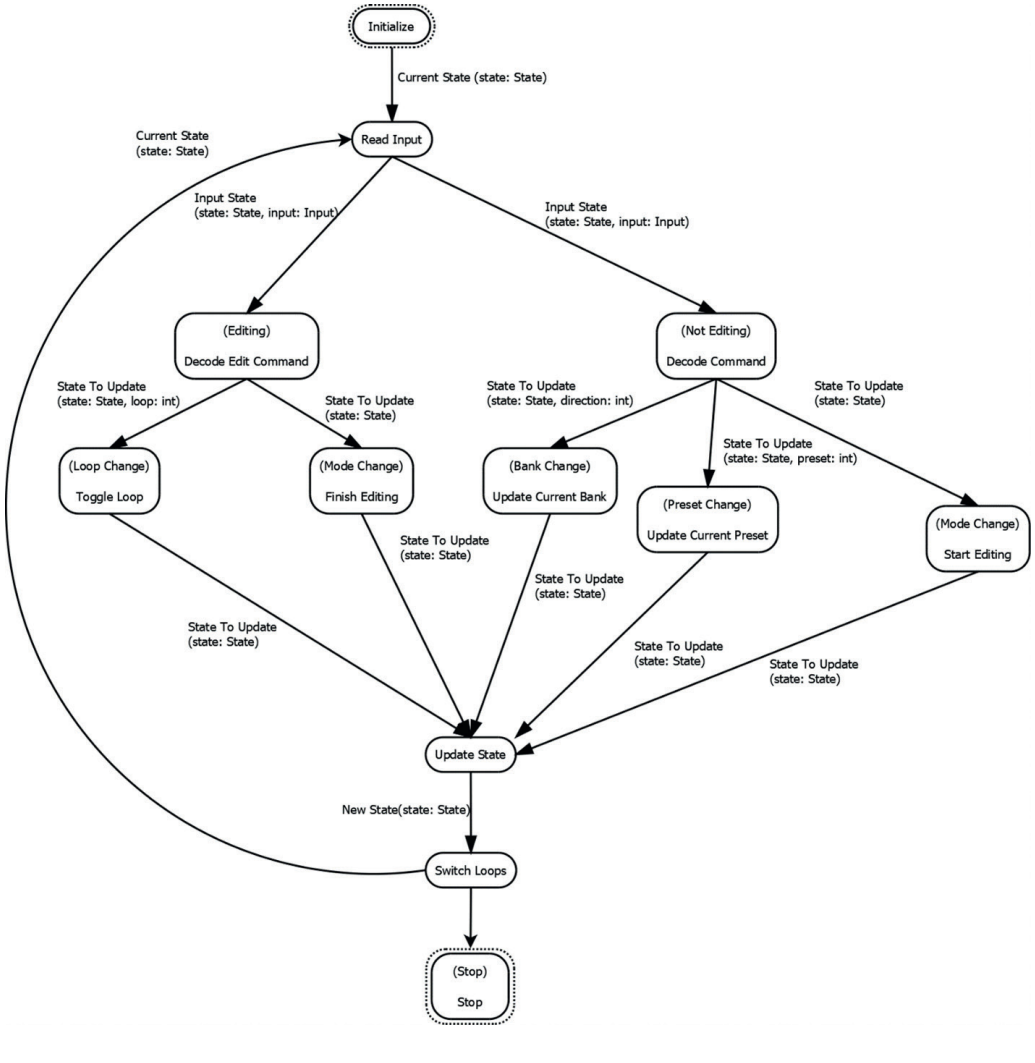
RĪGAS TEHNISKĀ UNIVERSITĀTE

Konstantīns Gusarovs

METODES IZSTRĀDE KODA ĢENERĒŠANAI NO DIVPUSLOŽU MODEĻA

Promocijas darba kopsavilkums




```c
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
    read_input_result_t result;
    result.state = state;
    result.input.up_pressed = PORTAbits
    result.input.down_pressed = PORTAbi
    result.input.edit_pressed = PORTCbi
    result.input.preset_pressed[0] = PO
    result.input.preset_pressed[1] = PO
    result.input.preset_pressed[2] = PO
```

RTU Izdevniecība
Rīga 2021

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 darba kopsavilkums

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

RTU Izdevniecība
Rīga 2021

Gusarovs, K. Metodes izstrāde koda ģenerēšanai no divpusložu modeļa. Promocijas darba kopsavilkums. Rīga: RTU Izdevniecība, 2021. 40 lpp.

Iespiests saskaņā ar promocijas padomes "RTU P-07" 2020. gada 4. novembra lēmumu, protokols Nr. 20-7.

https://doi.org/10.7250/9789934225772
ISBN 978-9934-22-576-5 (print)
ISBN 978-9934-22-577-2 (pdf)

PROMOCIJAS DARBS IZVIRZĪTS ZINĀTNES DOKTORA GRĀDA IEGŪŠANAI RĪGAS TEHNISKAJĀ UNIVERSITĀTĒ

Promocijas darbs zinātnes doktora (*Ph. D.*) grāda iegūšanai tiek publiski aizstāvēts 2021. gada 1. martā plkst. 14.30 tiešsaistē. Pievienoties var, izmantojot saiti: https://rtucloud1.zoom.us/j/91849258957.

OFICIĀLIE RECENZENTI

Profesors *Dr. habil. sc. ing.* Jānis Grundspeņķis,
Rīgas Tehniskā universitāte, Latvija

Profesors *Dr. sc. ing.* Artis Teilāns,
Rēzeknes Tehnoloģiju akadēmija, Latvija

Docents *Ph. D.* Dušans Savičs (*Dušan Savić*),
Belgradas Universitāte, Serbija

APSTIPRINĀJUMS

Apstiprinu, ka esmu izstrādājis šo promocijas darbu, kas iesniegts izskatīšanai Rīgas Tehniskajā universitātē zinātnes doktora (*Ph. D.*) grāda iegūšanai. Promocijas darbs zinātniskā grāda iegūšanai nav iesniegts nevienā citā universitātē.

Konstantīns Gusarovs ……. (paraksts)

Datums: ……

Promocijas darbs ir uzrakstīts latviešu valodā, tajā ir ievads, septiņas nodaļas, secinājumi, literatūras saraksts, 125 attēli, 11 tabulu, pieci pielikumi, kopā 208 lappuses, ieskaitot pielikumus. Literatūras sarakstā ir 122 nosaukumi.

SATURS

IEVADS

Attīstoties programmatūras inženierijai, tās uzdevumi ir paplašinājušies, un tā šobrīd ietver gan izstrādi, gan biznesa procesu analīzi, gan arī projektu pārvaldību. Darba [1] autori izsaka domu, ka mūsdienu programmatūras inženierija ir spējās (angļu val. – *Agile*) metodoloģijas un modeļvadāmas arhitektūras (angļu val. – *Model-Driven Architecture*; *MDA*) attīstības rezultāts. Viena no jaunākajām programmatūras inženierijas paradigmām ir modeļvadāmā programmatūras izstrāde (angļu val. – *Model-Driven Software Development*; *MDSD*) [17], [74]. Atšķirībā no tā sauktās "modeļos bāzētās" izstrādes, kas paredz modeļu izmantošanu dažādos izstrādes cikla posmos, *MDSD* paredz stingri formalizētu modeļu un to apstrādes algoritmu izmantošanu.

Eksistē divas galvenās modeļvadāmas pieejas lietotāju grupas [74] – 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 (programmatūras kods, dokumentācija un citi) automātiskai ģenerēšanai. No tā izriet, ka problēmsfēras modelim ir jābūt saprotamam visām iesaistītajām pusēm, kā arī piemērotam automātiskai koda ģenerēšanai no tā.

Industrijā modelēšanas nolūkiem parasti tiek izmantota [28] vienotā modelēšanas valoda (angļu val. – *Unified Modelling Language*; *UML*) [89]. Lai gan *UML* izmantošana piedāvā plašas koda ģenerēšanas iespējas no tās standarta modeļiem (piemēram, [5], [6], [17], [78], [82], [90] un citi), to var izskaidrot ar *UML* ciešo saistību ar programmatūras kodu – lielākā daļa *UML* diagrammu apraksta jau gatavus risinājumus, nevis sākotnējo problēmsfēru. *OMG* (angļu val. – *Object Management Group*) mājaslapā [89] *UML* tiek definēta kā "grafiskā valoda objektos bāzēto sistēmu vizualizācijai, specificēšanai, konstruēšanai un dokumentēšanai". Tas nozīme to, ka *UML* nav domāta sākotnējo prasību modelēšanai, bet ir domāta gatavo arhitektūras risinājumu aprakstīšanai.

Vairāki autori, piemēram [7], [73], [84], izvirza līdzīgas idejas, uzsverot, ka pašreizējās modelēšanas valodas (tai skaitā *UML*) ir saprotamas modeļvadāmas inženierijas ekspertiem, savukārt nav saprotamas problēmsfēras ekspertiem un biznesa analītiķiem, un izsakot domu, ka ***UML* modeļi ir "pārāk līdzīgi programmatūras kodam, lai tos varētu izmantot *MDSD* kontekstā"**. Tiek uzsvērta nepieciešamība pēc modelēšanas valodas, ko var saprast visas izstrādē iesaistītās puses. Šajā lomā tiek definēti vairāki modelēšanas artefakti:

- lietošanas gadījumu saraksts;
- biznesa procesu modeļi, kas šos lietošanas gadījumus apraksta (piemēram, *BPMN*, angļu val. – *Business Process Model and Notation* [9] notācijā);
- konceptu modeļi, kas satur informāciju par problēmsfēras domēna objektiem.

Tātad var redzēt, ka ir iespējams veikt programmatūras koda ģenerēšanu no modeļa, tomēr šobrīd izmantojamie modeļi ir vāji piemēroti ieviešanai to sarežģītības dēļ. Līdz ar to tiek izvirzīta hipotēze par to, ka modelēšanas valoda varētu ietvert divus artefaktus (līdzīgi kā [7]):

- biznesa procesa modeli – piemēram, *BPMN* [9], datu plūsmu diagrammas (angļu val. – *Data Flow Diagram*; *DFD*) [24], [81] vai arī citas notācijas;

- domēna jeb konceptuālo modeli, kas, piemēram, var būt veidots *ER*-diagrammas (angļu val. – *Entity-Relationship*) [16] veidā.

Pēc autora domām, lietošanas gadījumu modelis var tikt pilnīgi aizstāts ar biznesa procesu modeli – definējot katrām lietošanas gadījumam attiecīgos biznesa procesus, tiek veikta arī lietošanas gadījumu aprakstīšana. Šādai notācijai būtu jāatbalsta *MDSD* ieviešanu industrijā.

Hipotēzes par piemēroto modeli pārbaude

Lai pārbaudītu šo hipotēzi, tika veiktas divas aptaujas. Pirmās aptaujas mērķauditorija bija programmatūras izstrādātāji un arhitekti. Šiem respondentiem tika uzdots viens jautājums par to, kuras no *UML* diagrammām tie ir spējīgi definēt. Kopā tika saņemtas 227 atbildes, pēc kurām var secināt, kā lielākā respondentu daļa ir spējīga definēt klašu, secību, stāvokļu, aktivitāšu un lietošanas gadījumu diagrammas. To varētu izskaidrot ar to faktu, ka tieši šīs diagrammas (izņemot lietošanas gadījumu) palīdz aprakstīt lietojuma arhitektūru un procesus, kas notiek sistēmā.

Otrās aptaujas mērķauditorija bija biznesa analītiķi. 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. Apkopojot tās, var redzēt, ka **lielākā daļa biznesa analītiķu saprot, kas ir *ER*-diagrammas un biznesa procesu diagrammas, spēj tas aprakstīt un izmantot.**

Aptauju rezultāti apstiprina pieņēmumu par to, ka tieši biznesa procesu un *ER*-modeļu izmantošana, visticamāk, atvieglotu *MDSD* ieviešanu industrijā. Līdz ar to tiek uzskatīts, ka koda ģenerēšanai ir nepieciešams izmantot modelēšanas valodu, kas atbalsta šos divus modeļus.

Par tādu modelis tika izvēlēts divpusložu modelis [53], [63]. Vairāki pētījumi, kas tika veikti līdz šim brīdim un ir veltīti divpusložu modeļa izmantošanai *UML* diagrammu ģenerēšanai, parāda, ka tas satur pietiekami daudz informācijas gan statisko, gan arī dinamisko sistēmas analīzes artefaktu definēšanai [53], [54]. Pēc transformācijām iegūtais rezultāts ir *UML* diagrammu komplekss, kas savukārt ļauj veikt koda ģenerēšanu [70]. Līdz ar to tiek izvirzīta hipotēze par to, kā, **ja ir iespējamas transformācijas** "**divpusložu modelis → *UML***" **un** "***UML* → kods**"**, tad ir jābūt iespējamai arī transformācijai** "**divpusložu modelis → kods**". Šo apgalvojumu atbalsta arī pētījumi [32], [33].

Šāda transformācija 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 to, iesaistītajām pusēm piedāvājot pilnu risinājumu.

Promocijas darba mērķis un uzdevumi

Promocijas darbam tiek definēts mērķis izstrādāt programmatūras koda ģenerēšanas algoritmu no divpusložu modeļa, tā validācijas metodi, kā arī praktiski pārbaudīt izstrādātā algoritma darbību. Šī mērķa sasniegšanai ir nepieciešams izpildīt vairākus uzdevumus.

1. Novērtēt divpusložu modeļa piemērotību programmatūras koda ģenerēšanai un, ja nepieciešams, papildināt to.
2. Izvēlēties mērķa programmēšanas valodu programmatūras koda ģenerēšanai, pamatojot izvēli.
3. Izstrādāt algoritmu(s) divpusložu modeļa transformēšanai programmatūras kodā.
4. Definēt validācijas metodoloģiju transformāciju pareizību pārbaudei.
5. Pārbaudīt izstrādāto algoritmu ar praktiskā uzdevuma palīdzību, izmantojot to programmatūras sistēmas izstrādei.

Darba zinātniskais jaunieguvums un praktiskā nozīmība

Darba zinātniskie jaunieguvumi

1. Veikta divpusložu modeļa analīze, identificētas tā priekšrocības un ierobežojumi programmatūras koda ģenerēšanai. Pēc analīzes tika piedāvāti vairāki modeļa uzlabojumi.
2. Izstrādāti jauni transformācijas likumi divpusložu modelim, kas ļauj no tā iegūt programmatūras kodu. Darbā apskatīta *Java* [40] koda ģenerēšana, tomēr piedāvātos likumus var lietot jebkurai koda ģenerēšanai jebkurā programmēšanas valodā.
3. Lai panāktu transformācijas likumu neatkarību no mērķa programmēšanas valodas, izstrādāts starpmodelis koda reprezentācijai, ko ir iespējams izmantot programmatūras koda ģenerēšanai dažādu paradigmu valodās.
4. Izstrādāts klašu attiecību noteikšanas algoritms, ko ir iespējams izmantot arī ārpus modeļvadāmas programmatūras izstrādes. Iespējamās plietošanas jomas var būt koda pārrakstīšana (angļu val. – *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 iesaistītajām pusēm, un tas atvieglo savstarpējo komunikāciju. Izstrādātie transformācijas likumi nav atkarīgi no mērķa programmēšanas valodas un var tikt izmantoti ne tikai ar objektorientētām programmēšanas valodām, kas arī praktiski pieradīts.

Darba aprobācija

Promocijas darba rezultāti atspoguļoti deviņās 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). (Ieguldījums ~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 ~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. (Ieguldījums ~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. (Ieguldījums ~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 ~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 ~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 ~ 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 ~20 %).

Promocijas darba galvenie rezultāti prezentēti septiņās 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, kas saistītas ar problēmsfēru

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 ~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. (Ieguldījums ~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. (Ieguldījums ~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 ~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. (Ieguldījums ~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 ~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 ~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 ~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 ~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 ~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. (Ieguldījums ~25 %). †

* indeksēts SCOPUS, † indeksēts Web of Science.

Darba struktūra

Promocijas darbā ir ievads, septiņas nodaļas, secinājumi, bibliogrāfija un pieci pielikumi.

Darba ievadā pamatota tēmas aktualitāte, definēti darba mērķi un uzdevumi, kas nepieciešami šo mērķu sasniegšanai.

Promocijas darba 1. nodaļā aprakstīts divpusložu modelis, kā arī sniegts īss esošo transformācijas likumu apraksts, kas ļauj iegūt no modeļa dažāda veida artefaktus.

Promocijas darba 2. nodaļā analizētas divpusložu modeļa notācijas, ņemot vērā esošo transformācijas likumu ierobežojumus koda ģenerēšanas kontekstā, kā arī to risināšanas iespējas.

Promocijas darba 3. nodaļa veltīta domēna specifiskās valodas, kas ir paredzēta divpusložu modeļa definēšanai, izstrādei. Lēmums par tās izstrādi pieņemts saistībā ar nepieciešamību pēc vienkārši modificējamas divpusložu modeļa notācijas.

Promocijas darba 4. nodaļā apskatīts uzlabotais klašu attiecību noteikšanas algoritms, kas ir domāts klašu kopas apstrādei. Nepieciešamība pēc šāda algoritma ir viens no definētajiem divpusložu modeļa transformācijas likumu ierobežojumiem.

Promocijas darba 5. nodaļa veltīta transformācijas likumiem. Vispirms tiek izvēlēta mērķa programmēšanas valoda, kā arī aprakstīts starpmodelis, kas tiek izmantots transformāciju laikā. Tālāk tiek apskatītas transformācijas, kas ļauj divpusložu modeli pārveidot par šādu starpmodeli.

Darba 6. nodaļā aprakstīta *Java* koda iegūšana no iegūtā starpmodeļa, izmantojot speciāli izstrādātu piemēra modeli, kas satur speciālus testēšanas gadījumus. Šajā nodaļā apskatīta arī transformācijas rezultātu validācija, kā arī dots īss saistīto pētījumu apskats.

Darbā 7. nodaļā aprakstīts izstrādātās transformācijas likumu praktiskais lietojums. Sniegts sistēmas apraksts, izveidots tās divpusložu modelis, aprakstītas nepieciešamās koda ģenerēšanas algoritma modifikācijas, kā arī analizēti iegūtie rezultāti.

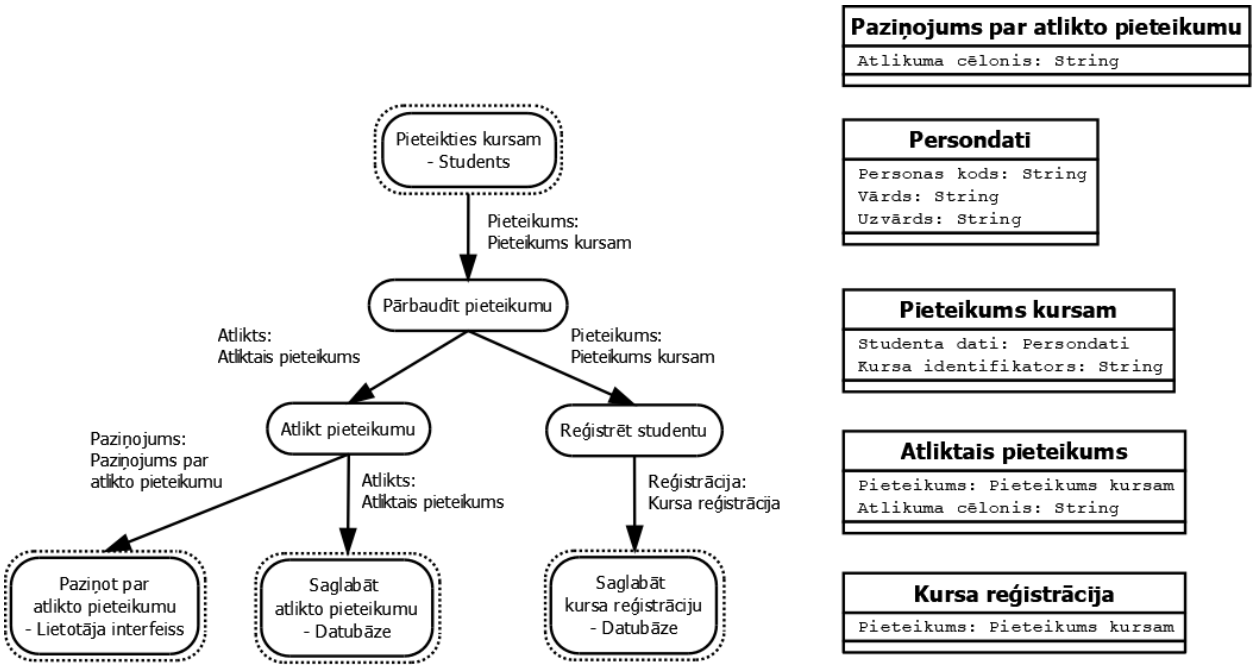
Noslēgumā apkopoti promocijas darba rezultāti.

Promocijas darbam ir pieci pielikumi. 1. pielikumā apskatāmi programmatūras izstrādātāju un arhitektu aptaujas rezultāti, 2. pielikumā – biznesa analītiķu aptaujas rezultāti, 3. pielikumā ir divpusložu modeli aprakstošas *DSL* valodas notācijas apraksts, 4. pielikumā – piemēra divpusložu modelis, kas tiek izmantots algoritma darbības pārbaudei, savukārt 5. pielikumā publicēts mikrokontroliera programmatūras kods, kas iegūts, darbā praktiski lietojot piedāvātos transformāciju likumus.

1. DIVPUSLOŽU MODELIS UN AKTUĀLĀS TRANSFORMĀCIJAS METODES

Divpusložu modeļa nosaukums izvēlēts, ņemot vērā kognitīvās psiholoģijas uzskatus [1], kas cilvēka smadzenes iedala divās puslodēs, kur viena atbild par loģiku, otra – par konceptiem. Ir nepieciešama abu smadzeņu pusložu koordinēta darbība, lai nodrošinātu cilvēka normālu funkcionēšanu. Līdzīgs princips tiek izmantots arī divpusložu modelī. Tas sastāv no divām diagrammām: biznesa procesu diagrammas, kas tiek attēlota modeļa kreisajā daļā, un konceptu diagrammas, kas apraksta datus, kas tiek izmantoti modelī.

Abas diagrammas ir savstarpēji saistītas. Tas tiek nodrošināts ar 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 [57]. Papildus ir iespējams definēt procesu izpildītājus, kas var būt sistēmas lietotāji, citas sistēmas vai arī abstrakti jēdzieni – piemēram, datubāze.



1.1. att. Divpusložu modeļa piemērs.

1.1. attēlā redzams divpusložu modeļa piemērs. Attēlā atspoguļots process, kā students veic pieteikšanos kursam. Sākumā tiek pārbaudīts, vai saņemtais pieteikums ir pareizs – gadījumā, ja tas tā nav, tiek saņemts 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 šajā kursā būs pieejamas papildu vietas. Par to arī tiek paziņots studentam. Tātad – students tiek reģistrēts kursam, un informācija par studentu tiek saglabāta reģistrēto studentu datubāzē.

Promocijas darba izstrādes laikā eksistē vairākas divpusložu modeļa transformācijas metodes, kas var tikt izmantotas dažāda veida artefaktu iegūšanai no tā.

- Darbs [67] piedāvā metodi *UML* [89] 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.
- Darbā [61] 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 papildu transformācijas soli, kas ir paredzēts papildu informācijas iegūšanai – pusautomātisko iespējamo parēju matricu veidošanu.
- Darbs [64] 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 [62] tā autori pirmoreiz piedāvā alternatīvo pieeju divpusložu modeļa transformēšanai. Biznesa procesu modelis tiek salīdzināts ar galīgo automātu [46], [79], [95], un transformāciju pamatā ir fakts, ka jebkuru galīgo automātu ir iespējams pārveidot par regulāro izteiksmi. Šo pārveidošanu rezultātā tiek iegūta *UML* secību diagramma, kas ietver aktierus, objektus, ziņojumus, kā arī divu veidu mijiedarbības fragmentus – cikla un alternatīvo izpildi.
- Darbs [32] ir iepriekšējās transformācijas uzlabojums, kas veltīts programmatūras koda iegūšanai no divpusložu modeļa bez *UML* diagrammu starpniecības. Tomēr darba autori atzīst, ka piedāvātajai metodei ir vairāki ierobežojumi.
- Darbs [33] ir promocijas darba pamats. Tajā tiek attīstītas idejas, kas aprakstītas darbos [32] un [62] – biznesa procesu modeļa salīdzināšana ar galīgo automātu un tā apstrāde, izmantojot dažādus algoritmus.

2. ESOŠIE DIVPUSLOŽU MODEĻA LIETOŠANAS IEROBEŽOJUMI KODA ĢENERĒŠANAS UZDEVUMĀ

Promocijas darba izstrādes sākuma brīdī transformācijas likumi, kas paredzēti divpusložu modeļa apstrādei, ļauj iegūt galvenokārt statisku informāciju – klašu sarakstu, to atribūtus un izpildāmās metodes, kā arīar informāciju par attiecībām starp šīm klasēm [65]. Lai gan ir piedāvāta arī metode secību diagrammu ģenerēšanai [64], šī metode nav pilnīga, jo autori atzīmē faktu, ka nav līdz galam atrisināts jautājums par mijiedarbības fragmentu ģenerēšanu no procesu diagrammas. Līdz ar to ir iespējams secināt, ka esošie divpusložu modeļa transformācijas likumi ir ierobežoti ar statiskas informācijas ģenerēšanu, kas nozīmē to, ka daļa informācijas, kas atrodas avota modelī, netiek izmantotā pilnā apjomā.

Promocijas darba uzdevums ir koda ģenerēšana no divpusložu modeļa, tāpēc ir nepieciešams detalizēti apskatīt mērķa modeli, lai saprastu, kāda papildu informācija, salīdzinot ar avota modeli, tajā tiek ietverta 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.

Programmatūras kods [37] 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 izpildāmās darbības, to operandus un iegūstamos rezultātus. Instrukciju vai operatoru secība definē vienu vai vairākus algoritmus, kas ļauj programmatūras kodam sasniegt nepieciešamos rezultātus.

Esošie divpusložu modeļa transformācijas likumi ļauj pārveidot sistēmas konceptuālo modeli par klašu kopu, kas pašreiz ir pietiekami deklarāciju definēšanai. Savukārt instrukciju jeb operatoru secības ģenerēšana patlaban tiek veikta tikai *UML* secību diagrammas formā [64]. Šīs informācijas ģenerēšanai tiek izmantota sākotnējā informācija no biznesa procesu modeļa. Vienkāršu secīgo instrukciju (bez zarošanas) ģenerēšana var tikt pilnīgi atbalstīta ar esošo divpusložu modeļa notāciju.

Turpinot esošo transformācijas metožu analīzi, var redzēt, ka metodes ģenerē tā saukto 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 darīt 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, parādās divas galvenās datu definēšanas pieejas [12], [85] – anēmiskais datu modelis [20], [22] un bagāto datu modelis. Bagātā datu modeļa pamatideja ir datu un loģikas apvienošana, savukārt anēmiskais datu modelis ir paredzēts koncepciju atdalīšanai – dati un to apstrādes loģika ir divas dažādas programmatūras sastāvdaļas, un tās ir jāatdala vienu no otras. Šāda pieeja arī atvieglo *MVC* [88] (angļu val. – *Model-View-Controller*) arhitektūras veidošanu.

Ņemot vērā gan vairāku autoru [20], [22], [48] izteiktas domas, gan arī savu pieredzi ar industrijā izmantojamām tehnoloģijām (piemēram, *Hibernate* [34], *AngularJS* [2] un citas),

promocijas darba autors uzskata, ka ir nepieciešams atbalstīt arī anēmisko datu modeli. Par to liecina arī darbs [85]. Var teikt, ka dati jeb domēna objekti reālas dzīves gadījumos gandrīz nekad nesatur programmatūras loģiku – tie var būt paredzēti 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 starp 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* [34], izmanto tieši anēmisko datu modeli.

Līdz ar to tiek uzskatīts, ka anēmiskais datu modelis nav antiparaugs, un tā izmantošana ir pilnīgi pieļaujama arī modeļvadāmā programmatūras izstrādē.

Var redzēt, ka gan divpusložu modeļa esošai notācijai, gan arī transformācijas likumiem piemīt vairāki ierobežojumi, kas var apgrūtināt vai padarīt par neiespējamu koda ģenerēšanas uzdevumu. 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 no tām	Notācija	Katrai izejošajai datu plūsmai pievienot informāciju par to, vai tā vienmēr tiks radīta procesa rezultātā. Vai apvienot visas procesa producētās datu plūsmas vienā objektā, kas saturēs atbilstošos atribūtus
Dati, ko spēj pārnest datu plūsmas, ir ierobežoti ar vienu konceptu, kas ir piesaistīts datu plūsmai	Notācija	Piedavāt iespējas piesaistīt datu plūsmai primitīvos datu tipus, masīvus/kolekcijas, kā arī nepiesaistīt vispār
Pēc klašu kopas ģenerēšanas tā netiek analizēta, rezultātā ir iespējama ģenerētā programmatūras koda atkārtojumi, kā arī var nebūt noteiktas visu klašu attiecības	Transformācijas	Definēt algoritmu vai algoritmus, kas var tikt izmantoti klašu kopas apstrādei papildu attiecību noteikšanai
Divpusložu modeļa transformācijas likumi neatbalsta anēmiskā datu modeļa ģenerēšanu [59]	Transformācijas	Atbalstīt šāda datu modeļa ģenerēšanas iespēju
Secību diagrammu ģenerēšanas algoritms atbalsta mijiedarbības fragmentu veidošanu, bet tas nespēj definēt mijiedarbības fragmentu tipus un nosacījumus to izpildei	Transformācijas	Papildināt gan modeļa notāciju, gan arī transformācijas likumus notācijas papildinājuma atbalstam

3. DOMĒNA SPECIFISKĀ VALODA DIVPUSLOŽU MODEĻA DEFINĒŠANAI

Lai būtu iespējams izvairīties no ierobežojumiem divpusložu modeļa notācijā, ir nepieciešams to modificēt. Iepriekšējie pētījumi, kas bija veltīti divpusložu modeļa un transformāciju izmantošanai, tika veidoti ar attiecīgiem programmatūras rīkiem, kas atbalsta divpusložu modeļa grafisko notāciju – *BrainTool* [60] un *BrainTool 2.0* [65]. Lai gan šie rīki piedāvā uzskatāmu divpusložu modeļa vizualizāciju, grafiskais rīks var prasīt salīdzinoši daudz papildu darba jauna elementa pievienošanai [57], [68]. Līdz ar to darbā tiek piedāvāts izmantot domēna specifisko valodu (angļu val. – *Domain Specific Language*; *DSL*) divpusložu modeļa notācijas uzlabošanai [30].

Salīdzinot ar universālajām programmēšanas valodām (angļu val. – *General Purpose Language*; *GPL*), domēna specifiskās valodas ir paredzētas konkrētas problēmas risināšanai vai arī konkrēta biznesa domēna informācijas precīzākai definēšanai [8], [50]. Šādas valodas universalitāte tiek samazināta, palielinot ekspresivitāti konkrētajā jomā.

Domēna specifiskās valodas veidošanai ir nepieciešams definēt problēmas domēna galvenos elementus un izveidot to reprezentācijas, izmantojot izvelēto sintaksi. *DSL* ir iespējams aprakstīt ar dažāda veida notāciju palīdzību, piemēram, ar *XML* shēmas [92] vai paplašinātas Bekusa-Naura formas (angļu val. – *Extended Backus-Naur Form*; *EBNF*) palīdzību [27].

Modeļvadāmās izstrādes gadījumā par *DSL* bieži uzskata grafisko modeli [8], [10], [43], [50], [91]. Visticamāk, to var izskaidrot ar lielu teorētisko darbu skaitu šajā jomā. Tikai neliela daļa piedāvāto metožu ir atbalstīta ar rīkiem, kas ļauj definēt un transformēt attiecīgos modeļus – šāda rīka izstrāde prasa daudz pūļu [60], [65] un parasti ierobežo eksperimentēšanas iespējas [55], [57], [60], [68].

3.1. tabula

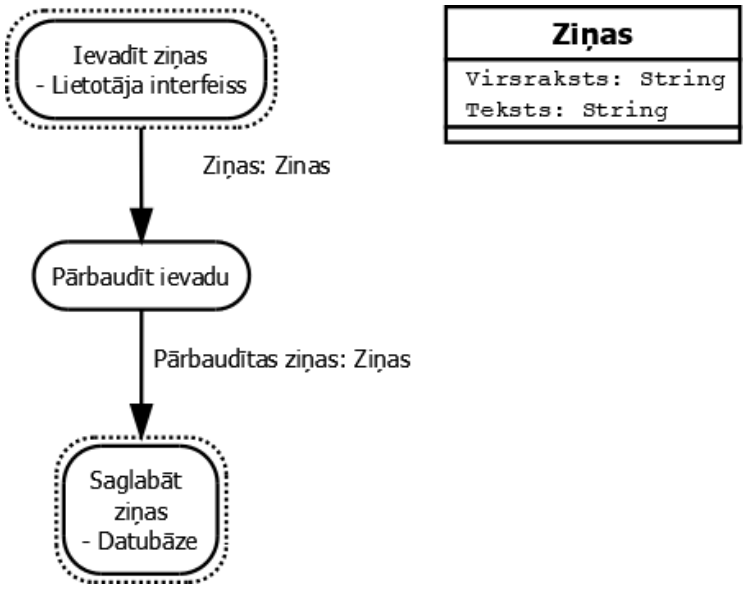
Divpusložu modeļa notācijas ierobežojumu risināšana ar *DSL* palīdzību

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 no tām	*DSL* nerisina šo ierobežojumu – šo uzdevumu var atrisināt transformāciju laikā. Transformāciju rezultātā var eksistēt iespēja iegūt programmatūras kodu dažādās programmēšanas valodās, vairākas no kuram (piemēram, *Python* [94]) atbalsta vairāku objektu atgriešanu no funkcijām
Dati, ko spēj pārnest datu plūsmas, ir ierobežoti ar vienu konceptu, kas ir piesaistīts datu plūsmai	*DSL* maina divpusložu modeļa notāciju, ļaujot datu plūsmai pārnest primitīvus objektus, to masīvus vai kolekcijas, kā arī vispār nepārnest datus

Izstrādātas valodas sintakse ir aprakstīta ar paplašināto Bekusa-Naura formu palīdzību, līdz ar to ir iespējams definēt tās parsētāju ar speciālā rīka (piemēram, [3], [86] palīdzību). Promocijas darbā parsētāja realizācijai ir izvēlēts *ANTLR* [3] rīks, kas ģenerē kodu *Java* valodā [40]. Izstrādātā domēna specifiska valoda ļauj ne tikai definēt divpusložu modeli, bet

arī bagātina tā notāciju, pievienojot vairākus jaunus elementus un uzlabojot iepriekšējā nodaļā definētos ierobežojumus [30]. Informācija par to sniegta 3.1. tabulā.

3.1. un 3.2. attēlā ir parādīts vienkāršs divpusložu modelis, kas tiek definēts attiecīgi ar grafiskas un *DSL* notācijas palīdzību. Var redzēt, ka definīcija, kas izmanto izstrādāto valodu, sniedz papildu informāciju, kas var tikt izmantota transformāciju laikā.



3.1. att. Vienkārša divpusložu modeļa grafiskā notācija.

```
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
```

3.2. att. Vienkārša divpusložu modeļa *DSL* notācija.

4. UZLABOTAIS KLAŠU ATTIECĪBU NOTEIKŠANAS ALGORITMS

Kā parādīts [54], [56], [58], [66], [68], [69] 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, ko sniedz procesa modelis. Lai gan šī pieeja ir pareiza, jo analizējot, kādas klases komunicē savā starpā, ir iespējams noteikt tādas klašu attiecības kā atkarība vai asociācija [89], citu attiecību – agregāciju, mantošanu un realizāciju – noteikšana var prasīt arī ģenerētās klašu kopas analīzi. Piemēram, transformāciju rezultātā var tikt ģenerētas vairākas klases ar līdzīgiem atribūtiem, kas būtu jāapvieno vienā klašu hierarhijā [49]. Līdzīgi, agregāciju noteikšanai neizmanto informāciju, ko sniedz konceptuālais modelis. Klašu attiecību noteikšanas algoritmam tika definētas vairākas prasības.

1. Algoritmam ir jāapstrādā klašu kopa, kas var būt prezentēta vairākos veidos – gan programmatūras koda klašu kopas veidā, gan arī *UML* klašu kopas veidā vai arī citādi.
2. Algoritmam ir jāņem vērā ne tikai informācija par to, kuras klases komunicē savā starpā, bet arī klašu struktūra – to atribūti un metodes.
3. Lai būtu iespējams noteikt, kuras klases komunicē savā starpā, algoritma ieejas datiem ir jāsatur arī papildu informācija par to, kuras klases komunicē savā starpā. Bez šādas informācijas attiecību noteikšana var būt nepilnīga.
4. Jaunģenerēto klašu un interfeisu nosaukumiem ir jābūt nevis automātiski ģenerētiem, bet gan cilvēka noteiktiem.

Piedāvātais algoritms sastāv no četriem galvenajiem soļiem, pirmajā no kuriem tiek identificētas realizācijas un mantošanas, otrajā – agregācijas, trešajā – atkarības, 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.

Realizāciju un mantošanu noteikšanas gaitā ir nepieciešama cilvēka līdzdalība, kas nozīmē to, ka šis algoritma solis tiek veikts pusautomātiskā veidā. Sākumā tiek analizētas ieejas klases un konstruēta kopīgo elementu tabula. Šī tabula satur informāciju par klašu metodēm un atribūtiem un ļauj veikt klašu struktūras salīdzināšanu un analīzi. Pēc kopīgās elementu tabulas izveidošanas tiek veikts mantošanas attiecību definēšanas cikls, kura izpildes laikā no kopīgo elementu tabulas tiek izvēlētas klases – A un B – ar vislielāko kopīgo elementu skaitu. Tad analītiķim tiek piedāvāta izvēle, kas ietver četras darbības:

1) `A` klasi pārveidot par bāzes klasi, `B` klasi – par atvasināto;
2) `B` klasi pārveidot par bāzes klasi, `A` klasi – par atvasināto;
3) izveidot jaunu bāzes klasi – `S`, `A` un `B` klases pārveidot par `S` klases atvasinātajām klasēm;
4) nedarīt neko.

Atkarībā no izvēlētās darbības tiek modificēta definēto klašu attiecību kopa un ieejas klašu kopa.

Līdzīgā veidā notiek arī realizāciju definēšana. Tiek definēta jaucējtabula (*hash table*), kur par atslēgu kalpo metode, par vērtību – klašu, kas satur šo metodi, kopa. Analizējot šo tabulu, algoritms piedāvā analītiķim definēt interfeisu klašu kopai. Gadījumā, ja interfeiss tiek definēts, algoritms pārbauda, vai jau eksistē interfeiss, kas satur visas attiecīgās 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.

Nākamais ar algoritma palīdzību nosakāmais klašu attiecību veids ir agregācija. Analizējot klašu kopu, tiek pārbaudīts, vai noteiktā klase satur atribūtu veidā citas klases no ieejas kopas. Pie tam, ir iespējama situācija, kad A klase, kas ir klase B bāzes , kā atribūtu satur B klasi. Ja ir konstatēts šāds gadījums, agregācijas attiecība definēta netiek, jo jau eksistē mantošanas (vai arī realizācijas) attiecība. Arī ir iespējams, ka B klase, kas manto no A klases, kā atribūtu satur A klasi. Arī šajā gadījumā agregācija netiek definēta.

Lai noteiktu atkarību, tiek izmantota gan informācija, ko satur klašu metožu definīcijas, gan arī papildu informācija par klašu savstarpējo komunikāciju. Tātad šis algoritma solis ir iedalīts divos apakšsoļos – vispirms tiek veikta klašu metožu parametru un atgriežamo vērtību analīze, pēc tam – analizēta papildu informācija par klašu savstarpējo komunikāciju. Gadījumos, ja ir iespējams definēt atkarību, tiek pārbaudīts, vai starp divām klasēm ir iepriekš definēta attiecība jebkurā virzienā – algoritms uzskata, ka mantošanas, realizācijas un agregācijas attiecības ir "svarīgākas" par atkarību.

Asociāciju noteikšanas laikā tiek analizētas jau definētās atkarības starp klasēm. Gadījumā, ja starp divām klasēm ir vairākas atkarības, tās tiek aizvietotas ar asociāciju.

Darbā aprakstīts algoritma darbības piemērs, kur no klašu kopas, kas nesatur nevienu attiecību, tiek iegūta pilnīgi saistīta klašu kopa. Piedāvāto klašu attiecību noteikšanas algoritmu var izmantot ne tikai divpusložu modeļa transformācijai, bet arī citiem nolūkiem, piemēram, koda pārveidošanas (angļu val. – *refactoring*) [23] nolūkiem. Šo algoritmu ir iespējams izmantot arī citām modeļvadāmām metodēm.

5. KODA ĢENERĒŠANA NO DIVPUSLOŽU MODEĻA

5.1. Mērķa programmēšanas valodas izvēle

Mērķa programmēšanas valodas noteikšanai tiek izmantota gan informācija par to popularitāti [87], gan prasības, ka valodai ir jābūt stingri tipizētai un universālai (neuniversālas, bet bieži izmantojamas valodas piemērs ir *SQL* [38]). Vēlams ir arī vairāku platformu atbalsts un atmiņas pārvaldība ar drazu savācēja izmantošanu (koda ģenerēšanas atvieglošanai). Tāpēc par mērķa valodu izvēlēta programmēšanas valoda *Java* [40].

5.2. Koda ģenerēšanas stratēģijas definēšana

Promocijas darbā piedāvāts divpusložu modeļa transformācijas procesu iedalīt vairākos soļos.

1. Vispirms nepieciešams definēt datu struktūras, kas tiks izveidotas transformāciju rezultātā.
2. Pēc tam nepieciešams apstrādāt informāciju, ko sniedz biznesa procesu diagrammas, iegūstot ģenerējamo metožu (vai funkciju) definīcijas.
3. Izmantojot iepriekš izveidotās definīcijas, ir nepieciešams definēt tā saukto darbplūsmu, kas apraksta attiecīgo metožu (kas atbilst avota modeļa procesiem) izpildes secību un nosacījumus.
4. Veikt programmatūras koda ģenerēšanu.

Šāda procesa atbalstam tiek izmantots starpmodelis [31]. Pēc 1.–3. soļa izpildes tiek iegūta programmatūras koda reprezentācija, kas var tikt izmatota tā ģenerēšanai izvēlētajā mērķa valodā. Izmantojot šādu pieeju, jaunas mērķa valodas pievienošana prasīs tikai pēdējā soļa modifikāciju. Biznesa procesu modelis tiek pārveidots par darbplūsmu, kas tiek definēta, izmantojot asamblerim [77] līdzīgu valodu, kas ir īsi aprakstīta 5.1. tabulā. 5.1. attēlā redzams *Java* koda piemērs, 5.2. attēlā – tā paša koda fragments, kas ir definēts, izmantojot darbā [31] piedāvāto metodi.

5.1. tabula

Starpmodeļa instrukcijas

Instrukcija	Apraksts	Piemēri
`Label<Name>`	Iezīmes ievietošana	`Label<1>`
`Var<Id, Name, Type>`	Mainīgā definēšana	`Var<1, S, String>`
`Invoke<Method, Inputs, Output?>`	Metodes vai procesa izpilde	`Invoke<fn, [a, b, c], d>` `Invoke<doNothing, []>`
`GetField<Object, Field, Target>`	Objekta atribūta ievietošana mainīgajā	`GetField<Obj, left, x>`
`PutField<Object, Field, Source>`	Mainīgā ievietošana objekta atribūtā	`PutField<Obj, left, x>`
`Jump<Label>`	Nenosacīta zarošana	`Jump<Label1>`
`Check<Var, Guard>`	Nosacījuma pārbaude	`Check<cond, "X > 0">`
`JumpIf<Var, Label>` `JumpIfNot<Var, Label>`	Nosacīta zarošana	`JumpIf< cond, Label1>` `JumpIfNot<cond, Label2>`
`Return<Var?>`	Metodes vai procesa rezultāta atgriešana	`Return` `Return<x>`

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

5.1. att. *Java* valodas koda piemērs.

```
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], ∅>
Label<L2>
Return<∅>
```

5.2. att. Programmatūras koda reprezentācija.

5.3. Biznesa procesu diagrammas minimizēšana

Lai būtu iespējams no biznesa procesu diagrammas, kas pēc savas būtības ir orientēts multigrafs ar cilpām [19], pāriet uz instrukciju secību, ir nepieciešams veikt tās apstrādi, pārveidojot grafa struktūru. Darbā [62] šīm nolūkam piedāvāts biznesa procesu modeļa diagrammas grafu apskatīt kā galīgo automātu [46], [79], [95]. Tas ļauj izmantot esošos algoritmus, kas ir paredzēti galīgo automātu apstrādei, kā arī biznesa procesu diagrammas apstrādei.

Katrs biznesa process no biznesa procesu diagrammas tiek pārveidots par tā saukto metodes signatūru, kas redzama 5.3. attēlā un ietver attiecīgā procesa nosaukumu, informāciju par ieejošām un izejošam datu plūsmām, kā arī procesa izpildes nosacījumu.

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

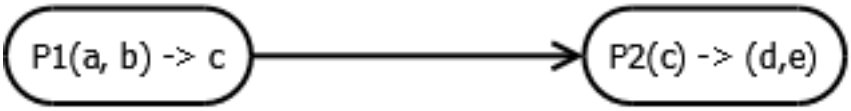
5.3. att. Metodes signatūras definīcija ar izpildes nosacījumu.

Pēc metožu signatūru definēšanas soļa ir iespējams veikt pirmo biznesa procesu diagrammas pārveidošanu, rezultātā iegūstot grafu, kas tiek nosaukts par procesu izsaukumu grafu. Šāds grafs ir līdzīgs funkcionālās struktūras modelim, kas ir aprakstīts darbā [26] un gandrīz vai pilnīgi sakrīt ar sākotnējo procesu diagrammu ar diviem izņēmumiem:

1) grafa loki vairs neatbilst datu plūsmām un tiek pārveidoti par savienojošiem elementiem, kuru vienīgais uzdevums ir definēt visas iespējamās procesu izsaukumu secības; biznesa procesi savukārt tiek aizvietoti ar attiecīgajām metožu signatūrām;
2) grafam tiek pievienotas sākuma un beigu virsotnes, kas nosaka sākotnējo un terminālo sistēmas stāvokli; visi ārējie procesi tiek savienoti ar šīm virsotnēm.

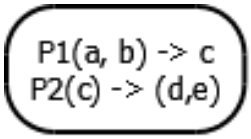
5.4. attēlā parādīts procesu izsaukumu grafa fragments, kas satur informāciju par diviem procesu izsaukumiem, kas seko viens otram:

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.

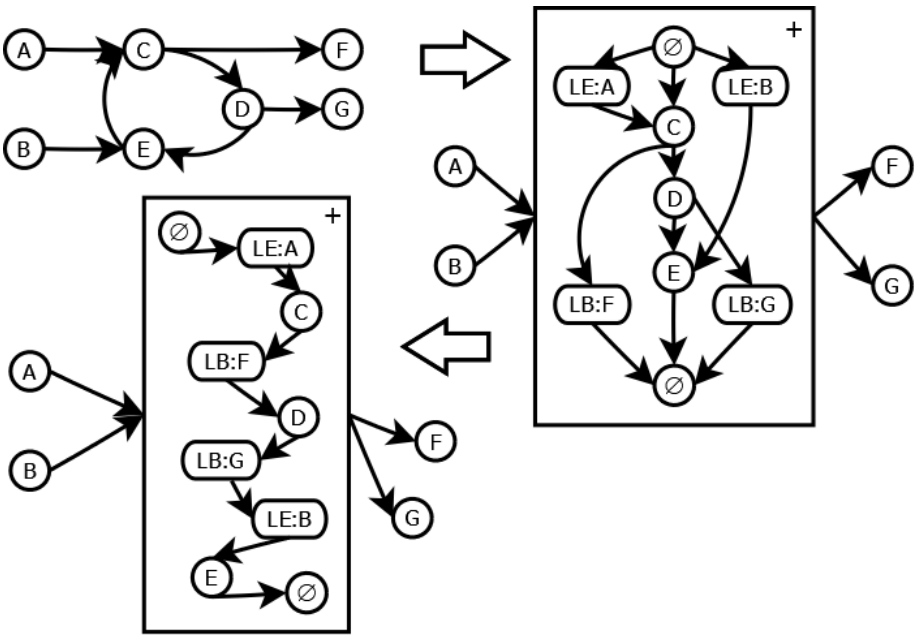


5.4. att. Procesu izsaukumu grafa fragments.

Šie divi procesi ir savienoti ar vienu loku, kas nozīmē, ka starp tiem 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 redzama 5.5. attēlā.



5.5. att. Virsotņu apvienošanas rezultāts.



5.6. att. Ciklu aizvietošana.

Var redzēt, ka virsotņu apvienošanas rezultātā tiek izveidota jauna virsotne, kas satur informāciju par diviem procesiem un to izpildes secību. Šāda apvienošanas operācija ļauj gan minimizēt virsotņu skaitu grafā, gan arī iekļaut vienā virsotnē informāciju par vairākiem procesiem.

Darbā identificētas arī citas iespējas apvienot informāciju par vairākiem procesiem. Lietojot tās, grafa transformācijas turpinās līdz stāvoklim, kad tas saturēs tikai un vienīgi vienu secības tipa virsotni, kas savukārt satur visu informāciju par grafa sākuma stāvokli. Šīs

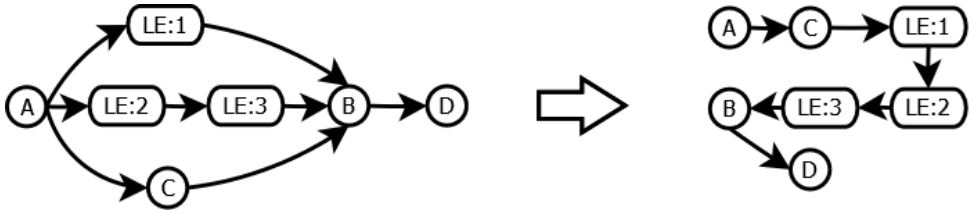
virsotnes saturs ir lineārā struktūra, kas atbilst mērķa modelim – programmatūras kodam. Tātad ir nepieciešams definēt vairākus grafa apstrādes algoritmus, kas, mainot tā struktūru, spēj saglabāt visu informāciju, kas šajā grafā ir attēlota. Šādus apstrādes algoritmus var iedalīt divās grupās.

Pirmā grafa minimizēšanas algoritmu grupa ir paredzēta ciklu apstrādei. To izpildes process ir shematiski parādīts 5.6. attēlā. Vispirms visi cikli tiek identificēti ar Tarjana [83] un Džonsona algoritmu [42] palīdzību. Pēc tam katrs cikls tiek aizvietots ar jauno procesu izsaukumu grafu, kas satur aizvietojamas virsotnes. Tiek definētas arī papildu virsotnes:

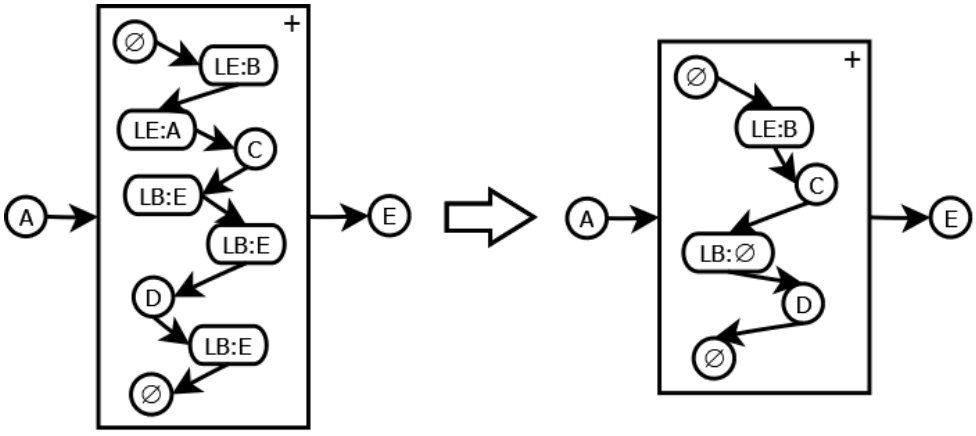
- `LE: from` – punkts, no kura var sākties noteikta cikla izpilde;
- `LB: to` – punkts, kurā noteikta cikla izpilde var tikt pārtraukta.

Pēc ciklu aizvietošanas ir iespējams apstrādāt katra cikla apakšgrafu, un iegūtos rezultātus ievietot sākotnējā. Apstrādājot iegūtos apakšgrafus, ir jāņem vērā arī papildu noteikumi.

1. Ja starp divām grafa virsotnēm `A` un `B` eksistē vairāki ceļi, daži no kuriem satur tikai un vienīgi cikla ieejas vai izejas punktus, ir iespējams šos ceļus apvienot vienā, izveidojot jauno lineāro apakšgrafu, kas satur secīgi savienotas dotās virsotnes, ievietojot šo apakšgrafu pirms attiecīgās virsotnes `B`. Situācija atspoguļota 5.7. attēlā.
2. Analizējot virsotnes, kas atrodas pirms un pēc cikla, var pārkārtot un, iespējams, iznīcināt liekos ieejas un izejas punktus. Arī gadījumos, ja vairāki vienādi ieejas vai izejas punkti seko viens otram, tie tiek aizvietoti ar vienu. Situācija atspoguļota 5.8. attēlā.



5.7. att. Ciklu ieejas un izejas punktu alternatīvu apstrāde.

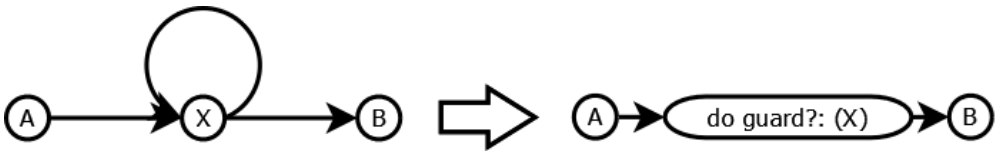
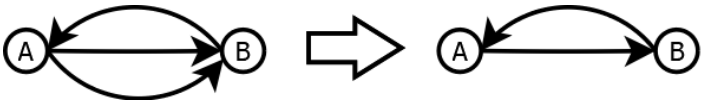
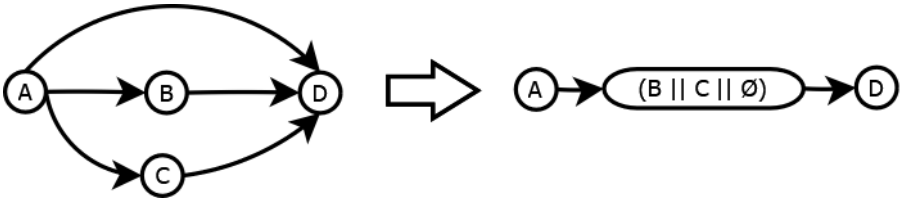
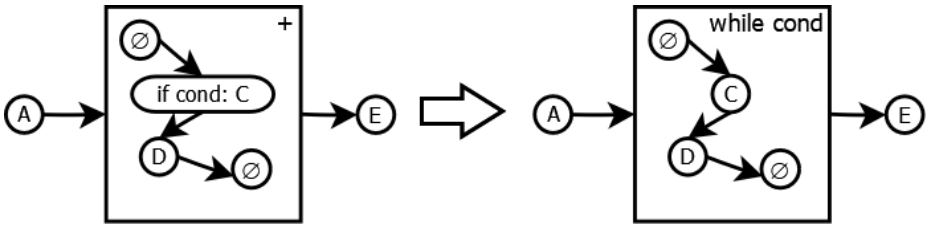


5.8. att. Ciklu ieejas un izejas punktu apstrāde.

Pārējie apstrādes algoritmi, kas tika definēti promocijas darba izstrādes laikā, redzami 5.2. tabulā.

5.2. tabula

Procesu izsaukumu grafa transformācijas

Nosaukums	Transformācija
Cilpu aizvietošana	
Loku dublikātu likvidēšana	
Secīgu virsotņu apvienošana	
Disjunkciju definēšana	
Cikla nosacījumu definēšana	
Secību nosacījumu definēšana	

Lai pārbaudītu definēto algoritmu darbību, tika veikti 10 000 eksperimenti ar nejauši ģenerētiem grafiem, kas ietver dažādu virsotņu daudzumu – 10–100. Grafu ģenerēšanas laikā tika definēti vairāki nosacījumi:

- grafam jāsatur vismaz viena cilpa;
- grafam jāsatur vismaz viens cikls;
- vismaz 10 % grafa virsotņu jābūt savienotām ar vairākām citām virsotnēm – gan caur ieejošiem, gan caur izejošiem lokiem;
- grafam jāsatur vismaz viena sākuma un vismaz viena beigu virsotne; turklāt katrai beigu virsotnei jābūt sasniedzamai no vismaz vienas sākumu virsotnes; starp jebkuru sākuma un beigu virsotni jābūt vismaz vienam ceļam;
- grafam jābūt vāji saistītam.

Eksperimentu veikšanas laikā neizdevās ģenerēt tādus grafus, ko nevar minimizēt (visi grafi bija minimizējami). Līdz ar to tiek pieņemts, ka definētie algoritmi ļauj minimizēt procesu izsaukumu grafu.

5.4. Minimizēta procesu izsaukumu grafa apstrāde

```
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)
```

5.9. att. Instrukciju ģenerēšana.

Procesu izsaukumu grafa minimizēšanas rezultātā 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. Turklāt 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.

Pēc procesa izpildes var iegūt viens vai vairākus rezultātus – tas nozīmē to, ka ģenerētajai metodei būtu jāatgriež neviens vai vairāki koncepti vai primitīvie datu tipi. Ir programmēšanas valodas, kas ļauj to darīt – vai nu tiešā veidā (piemēram, *Python* [94], *Ruby*

[76]), vai arī izmantojot metodes/funkcijas parametrus (piemēram, *C* [75], *C++* [80], *C#* [11]). Citas programmēšanas valodas bez speciālu klašu vai paņēmienu izmantošanas to izdarīt neļauj. Lai panāktu transformācijas likumu maksimālu universālumu, tiek piedāvāts izveidot tā sauktās rezultātu klases, kas apvieno attiecīgā biznesa procesa izpildes rezultātus.

Lai no iegūtā rezultāta būtu iespējams iegūt instrukciju secību, vispirms ir nepieciešams:

- definēt mainīgos, kas tiks izmantoti instrukciju izpildei;
- definēt iezīmes, kas nosaka zarošanās punktus;
- definēt iespējamās zarošanās;
- definēt rezultātu klases.

Izpildot šos papildu soļus, ir iespējams veikt instrukciju ģenerēšanu, kas ir realizēts, apstrādājot minimizēta grafa vienīgās virsotnes saturu. Algoritms, kas veic instrukciju secības ģenerēšanu, redzams 5.9. attēlā.

6. KODA ĢENERĒŠANAS ALGORITMA NOVĒRTĒŠANA

6.1. Algoritma darbības rezultātu validācija

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

Transformācijas validācijas grafa izveidošanai ir nepieciešams veikt instrukciju secības analīzi. Veicot šo analīzi, vispirms tiek izveidotas visas nepieciešamās grafa virsotnes, kas ir vai nu procesu izsaukumu instrukcijas, vai arī iezīmes. Kad visas šī grafa virsotnes ir definētas, nepieciešams definēt tā lokus. Šeit loki ir ne kas cits kā iespējamās kontroles nodošana – ja B procesa izsaukums seko A procesa izsaukumam, tad grafam tiek pievienots loks A → B. Līdzīgi tiek apstrādātas arī zarošanās instrukcijas – ja ir iespējama zarošanās uz iezīmi X, tiek pievienots attiecīgais loks. Rezultātā tiek iegūts grafs, kas satur informāciju par visiem iespējamajiem ceļiem jeb procesu izsaukumu secībām, kas ir iespējamas, veicot attiecīgā 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 jāņem vērā fakts, ka validācijas grafs satur papildu virsotnes, kas atbilst iezīmēm. Līdz ar to nevar veikt salīdzināšanu, izmantojot primitīvu pieeju – pārbaudīt, vai katram procesu izsaukumu grafa lokam atbilst validācijas grafa loks. Vispirms ir nepieciešams pārbaudīt, vai starp divām virsotnēm, ko 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, ja šajā 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 [21], *IDDFS* (angļu val. – *Iterative Deepening Depth-First Search* – iteratīvi padziļinošo pārmeklēšanu dziļumā) [45] algoritmu vai arī citus.

Kad ir noteikts, ka starp divām validācijas grafa virsotnēm eksistē ceļš, ir nepieciešams analizēt, vai šis ceļš satur tikai šīs virsotnes un, iespējams, vairākas iezīmes. Lai atrisināto šo uzdevumu, var izmantot atgriezmeklēšanas (angļu val. – *backtracking*) [44] paņēmienu, kas ļauj pārbaudīt, vai ir iespējams no gala virsotnes nonākt līdz sākotnējai, apmeklējot tikai iezīmju virsotnes.

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, pēc transformācijas tiek saglabāta. Šāds algoritms veic procesa izsaukumu loku analīzi. Katrs loks tiek analizēts šādi:

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

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

6.2. *Java* koda iegūšana no instrukciju secības

Pēc instrukciju secības ģenerēšanas to var izmantot programmatūras koda iegūšanai. Promocijas darba sadaļā tiek aprakstīta viena no *Java* [40] valodas koda iegūšanas iespējām.

Programmēšanas valoda *Java* tiek kompilēta baitu kodā. Tātad – ir iespējama vismaz viena transformācija – aizvietojot katru ģenerēto 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 [15], savukārt *Intel* procesoru asamblera valoda [36] satur ap 2000, to ir iespējams dekompilēt, iegūstot sākotnējo programmatūras kodu.

Dekompilācija ir viena no reversās inženierijas tehnoloģijām, kas paredzēta sākotnējā programmatūras koda iegūšanai no kompilētiem artefaktiem [18]. 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 likumīgas lietošanas piemēri.

- Tā var būt nepieciešama defektu labošanai programmatūrā, kas tika izstrādāta pirms kāda laika, un tās pirmkods vairs nav pieejams.
- Nepieciešamība pēc reversās inženierijas var rasties arī tad, ja jāiegūst informācija 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.

Ir arī citi piemēri, taču promocijas darbā dekompilācijas iespējas tiek izmantotas programmatūras koda iegūšanai no ģenerētās instrukciju secības.

Pirms dekompilācijas veikšanas ir nepieciešams izveidot bināro artefaktu (jeb kompilēto *Java* valodas klasi), kas atbilst ģenerētajai instrukciju secībai. Izveidotā instrukciju kopa ir balstīta uz *JVM* baitu kodu [31], tāpēc ir iespējams izveidot vienkāršus likumus instrukciju pārveidošanai baitu koda elementos. Faktiski katra no piedāvātajām instrukcijām atbilst baitu koda elementu secībai. Šī informācija apkopota 6.1. tabulā. Tiek apskatītas tikai tās instrukcijas, kas tiek izmantotas baitu koda ģenerēšanas nolūkiem.

6.1. tabula

Instrukciju kartēšana uz *JVM* baitu kodu

Instrukcija	***JVM* baitu kods**
`Label<Name>`	*JVM* gadījumā iezīmēm atbilst nobīdes no metožu sākumiem [14].
`Var<Id, Name, Type>`	*JVM* baitu koda instrukcijas neeksistē, tiek izmantotas speciālas tabulas [14].
`GetField<Object, Field, Target>`	`ALOAD Object` `GETFIELD` *`Field`* `ASTORE` *`Target`*
`Invoke<Method, Inputs, Output?>`	`ALOAD 0` ∀ *`v`* ∈ *`Inputs`*`: ALOAD` *`v`* `INVOKESPECIAL` *`Method`* … Ja metode atgriež vienu vai vairākas vērtības: `ASTORE` *`Output`* Gadījumā, ja tiek atgriezts objekts, ir nepieciešams veikt tā atribūtu saglabāšanu lokālajos mainīgajos: ∀ *`f`* ∈ *`Output`*`.fields:` `ALOAD` *`Output`* `GETFIELD` *`f`* `ASTORE` *`var`*
`Jump<Label>`	`GOTO` *`Label`*

`Check<Var, Guard>`	Metožu izpildes nosacījumi tiek definēti rakstiski, tāpēc ir nepieciešams definēt komentāru, kas varētu palīdzēt pareiza nosacījuma definēšanai turpmāk. *JVM* baitu kods komentārus nesatur, līdz ar to tiek piedāvāts šo informāciju saglabāt, turpmākai apstrādei izmantojot `Boolean` klases metodi `valueOf()`:
	`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, nepieciešams to ielādēt:
	`ALOAD` *`Var`*
	un atgriezt:
	`ARETURN`
	Citādi iziet no metodes:
	`RETURN`

Lai būtu iespējams instrukciju secību pārveidot par baitu koda elementiem, promocijas darbā tiek izmantota bibliotēka *ASM* [4].

Pēc baitu koda ģenerēšanas ir iespējams veikt tā dekompilāciju, lai iegūtu *Java* programmatūras kodu. Kā jau minēts, dekompilācijas veikšanai ir nepieciešams izmantot speciālas programmas, kas tiek sauktas par dekompilatoriem. *Java* valodas gadījumā eksistē vairākas šādas programmas, un pētījuma [29] gaitā tika salīdzināti četri dekompilatori – *JAD* [41], *CFR* [13], *Procyon* [51] un *FernFlower* [25]. Rezultātā tika izvēlēts *Procyon* dekompilators, kas arī tiek izmantots *Java* koda iegūšanai.

Pēc dekompilācijas veikšanas ir iespējams apvienot tās rezultātus ar procesu izsaukumu rezultātu klasēm, kā arī klasēm, kas tiek iegūtas no konceptiem. Rezultātā tiek iegūta klašu kopa, kas satur nepieciešamās domēna klases, kā arī biznesa loģikas realizācijas klasi. Tagad ir iespējams veikt šīs kopas apstrādi ar uzlaboto klašu attiecību noteikšanas algoritmu.

Iegūtais programmatūras kods atbilst anēmiskā datu modeļa definētajiem principiem – dati jeb domēna objekti ir atdalīti no to apstrādes loģikas [20], [22], [48]. Balstoties uz savu pieredzi programmatūras izstrādē industrijas kontekstā, darba autors uzskata, ka šādā veidā ģenerētais 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, lietojot pieeju no [60] vai [65]) un iegūstot "bagāto" datu modeli.

6.3. Saistīto pētījumu apskats

Ir veikta arī saistīto pētījumu analīze, kuras laikā apskatīti darbi, kas par avota modeli izmanto stāvokļu diagrammas. Šāda izvēle var tikt pamatota gan ar to, ka izstrādātais algoritms strādā ar līdzīgo datu struktūru, gan arī ar to, ka *UML* klašu, secību un aktivitāšu diagrammas var tikt pārveidotas par programmatūras kodu, izmantojot vienkāršus likumus, jo tas apraksta gatavos risinājumus. Darba izstrādes laikā apskatīti vairāki pētījumi [6], [52], [82], [90]. Analizējot tos, var redzēt, ka piedāvāāas metodes ir paredzētas korekta izpildāmā koda definēšanai. Kā papildu mērķi to autori definē pēc apjoma mazāka koda ar lielāku veiktspēju iegūšanu. Tas tiek panākts, izmantojot dažādas stāvokļu mašīnas reprezentācijas, lietojot vairākus pavedienus paralēlai apstrādei vai minimizējot izpildāmā koda apjomu. Lai gan tas ļauj sasniegt mērķus, ko ir definējuši šo pētījumu autori, tas izraisa grūti lasāma un uzturama koda iegūšanu. Līdz ar to tiek uzskatīts, ka šādas metodes nav īsti piemērotas modeļvadāmai izstrādei.

Papildus tika apskatīti arī pētījumi, kas nav cieši saistīti ar koda ģenerēšanu, bet ir veltīti staprmodeļu definēšanai, kas var tikt izmantoti programmatūras koda ģenerēšanai. Analizējot pētījumus [47], [71], [93], var redzēt, ka tajos kā mērķa valodas parasti tiek izmantotas objektorientētas valodas. Savukārt promocijas darbā piedāvātais starpmodelis nav atkarīgs no paradigmas, un, kā paradīts nākamajā nodaļā, var tikt izmantots arī procedurāla koda ģenerēšanai.

6.4. Koda ģenerēšanas algoritma novērtēšanas rezultāts

Kā parādīts, 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 piedāvā arī transformācijas likumu darbības rezultātu validācijas algoritmu, kas ir balstīts uz iegūtā starpmodeļa salīdzināšanu ar avota modeli.

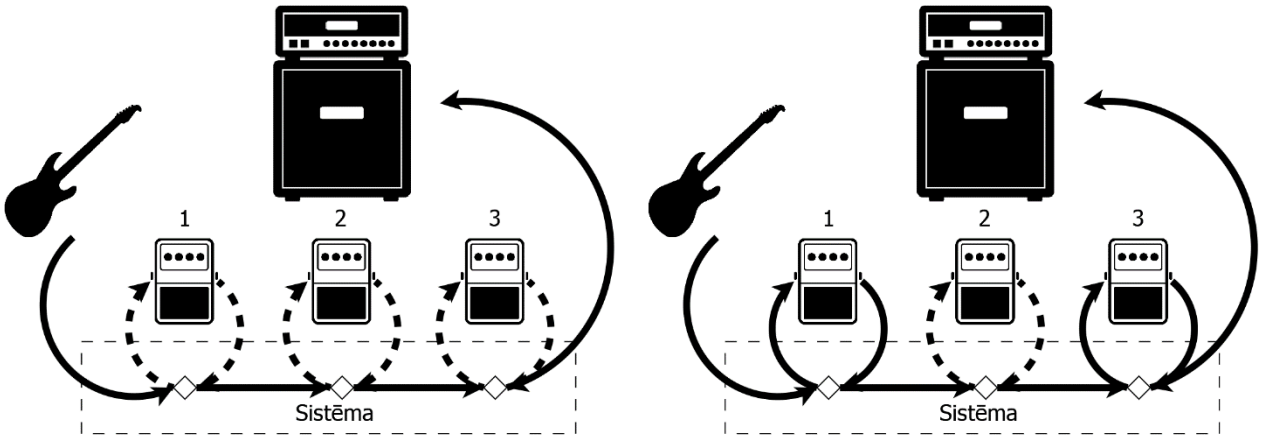
Programmatūras koda iegūšanai izmantots dekompilācijas paņēmiens [18]. Lai būtu iespējams to veikt, vispirms starpmodeļa instrukcijas tiek pārveidotas par *JVM* instrukcijām [15], no kurām pēc tam tiek iegūts *Java* kods. Par transformācijas likumu pareizību liecina arī fakts, ka ir iespējama iegūtā baitu koda korekta dekompilācija.

Promocijas darba autors apskata piemēru, kas tika izveidots tā, lai tas saturētu pēc iespējas vairāk īpaši apstrādājamu gadījumu – tie ir cikli ar vairākiem ieejas un izejas punktiem; zarošanās; datu plūsmas, kas nepārnes nekādu informāciju. Rezultātā no šāda biznesa procesu modeļa ir iegūts korekts *Java* kods, ko turpmāk var modificēt.

Šajā nodaļā apskatīti arī saistītie darbi un definēti galvenie saistīto pētījumu ierobežojumi – to autori galvenokārt izmanto *UML* [89] 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 LIETOJUMS

Promocijas darba izstrādes laikā tā autoram tika piedāvāts izstrādāt programmu *PIC18F45Q10* [72] mikrokontrolierim. Kopā ar pasūtītāju tika nolemts šīm nolūkam izmantot šajā darbā aprakstīto pieeju un vispirms definēt divpusložu modeli, no kura tiks ģenerēts programmatūras kods. Elektroģitāras efekti ir ierīces, kas ļauj mainīt elektroģitāras skaņu [35], piemēram, pievienot atbalsi vai citā veidā mainīt audiosignālu.



7.1. att. Elektroģitāras efektu pārslēgšanas sistēmas shēma.

Pārslēgšanas sistēma ļauj veidot tā sauktās efektu cilpas, kas ļauj veikt dažādu efektu savstarpējo komutāciju, tos pieslēdzot vai izslēdzot no signāla pārraides ceļa. Signāla pārraides ceļš sākas ar elektroģitāru un beidzas ar ģitāras pastiprinātāju. Tajā var ievietot vairākus efektus. 7.1. attēlā redzamas situācijas, kad visi efekti ir izslēgti (kreisajā pusē) vai arī kad 1. un 3. efekts ir ieslēgts (labajā pusē). Protams, ir iespējams veidot arī citus signāla pārraides ceļus. Mikrokontrolieris veic komutācijas ceļu pārslēgšanu, ieslēdzot dažādus no saglabātiem priekšiestatījumiem.

Mikrokontroliera programma tiek definēta programmēšanas valodā *C* [39], [75], tāpēc bija nepieciešams veikt izmaiņas koda ģenerēšanas algoritmā. Tomēr šādas izmaiņas bija minimālas, un kopumā tas liecina, ka izstrādāto metodi var izmantot ne tikai objektorientēta koda ģenerēšanai.

Līdz ar to piedāvātā metode ir aprobēta arī praktiski, un darba autora pieredze darbā ar sistēmas pasūtītāju liecina, ka to ir iespējams izmantot arī citos lielākos projektos.

NOBEIGUMS

Promocijas darbs veltīts programatūras koda ģenerēšanai no avota modeļa *MDSD* kontekstā. Koda ģenerēšana dažādiem nolūkiem (piemēram, integrētās izstrādes vidēs) tiek izmantota no pagājušā gadsimta 80. gadiem, tomēr modeļvadāmas programmatūras izstrādes gadījumā tā nav palīgrīks, bet ir metodoloģijas pamats. Ja ir iespējams iegūt programmatūras kodu no avota modeļa, kas ir saprotams gan programmatūras izstrādātājiem, gan arī problēmsfēras ekspertiem, ir iespējams runāt par pilnu *MDSD* atbalstu. Promocijas darbā analizētas avota modeļa notācijas, pamatota notācijas izvēle, kas jau ir lietota modeļu transformācijās *MDSD* kontekstā dažādos abstrakcijas līmeņos, un šajā pētījumā šī notācija ir modificēta koda ģenerēšanas uzdevuma realizācijai, tādējādi panākot *MDSD* atbalstu pārējo līmeņu ķēdē. Promocijas darba izstrādes laikā visi definētie uzdevumi pilnībā izpildīti.

1. Promocijas darba izstrādes laikā veikta divpusložu modeļa priekšrocību un ierobežojumu analīze koda ģenerēšanas kontekstā. Identificēti vairāki ierobežojumi, ko bija nepieciešams uzlabot, lai padarītu koda ģenerēšanas uzdevumu iespējamu. Visiem atrastajiem ierobežojumiem tika piedāvāti risinājumi, kas arī kļuva par pamatu turpmākajam darbam.
2. Promocijas darba autors ir izvēlējies mērķa programmēšanas valodu, pamatojoties uz *TIOBE* reitinga datiem par programmēšanas valodu popularitāti. Rezultātā tika izvēlēta programmēšanas valoda *Java*.
3. Izstrādāti transformācijas likumi, kas no divpusložu modeļa ļauj iegūt programmatūras kodu. Šie likumi ar starpmodeļa palīdzību ir izmantojami ne tikai objektorientētas programmatūras veidošanai, tas pārbaudīts arī praktiski.
4. Izstrādāta validācijas metodoloģija transformāciju pareizību pārbaudei, kas balstīta uz avota modeļa un iegūto rezultātu savstarpējo salīdzināšanu.
5. Izmantojot izstrādāto algoritmu, 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. Šajā promocijas darbā apskatīta *Java* koda ģenerēšana, savukārt pats algoritms definēts vispārīgā veidā un ļauj iegūt kodu vairākās programmēšanas valodās, kas var būt vai nebūt objektorientētas. Lai to panāktu, ir definēts starpmodelis, kas balstīts uz speciālo instrukciju kopas izmantošanu.

Iegūts arī **papildu rezultāts**, kas nebija definēts sākotnējā uzdevumā, taču bez tā algoritma izstrāde nebūtu iespējama – tā ir divpusložu modeļa modernizācija. Lai būtu iespējams atrisināt ierobežojumus divpusložu modeļa notācijā, ir nepieciešams šo notāciju papildināt.

Kopumā promocijas **darba rezultāti** ir šādi:

1) veikta divpusložu modeļa uzlabošana un papildināšana;
2) definēti divpusložu modeļa transformācijas likumi, kas tiek piedāvāti pseidokoda veidā;
3) definēts starpmodelis, ko var izmantot programmatūras koda ģenerēšanai ne tikai no divpusložu modeļa;

4) piedāvāts klašu attiecību noteikšanas algoritms, kas var tikt izmantots kopā ar citām metodēm, kas var būt arī nesaistītas ar modeļvadāmo programmatūras izstrādi (piemēram, koda pārrakstīšana (angļu val. – *refactoring*);
5) pārbaudītas izstrādātās metodes praktiskās lietošanas iespējas, izmantojot to programmatūras sistēmas izveidošanai.

Balstoties uz promocijas darbā veiktajiem pētījumiem un iegūtajiem rezultātiem, ir definēti **vairāki secinājumi**.

1. Lai gan *MDSD* jomā artefaktu definēšanai parasti tiek izmantota *UML* valoda, 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 iesaistītajām pusēm (tas tika pārbaudīts, pārbaudot piedāvāto risinājumu praksē – sistēmas pasūtītājs bija spējīgs ar minimālo palīdzību izveidot sistēmas modeli). Tas nozīmē, ka to ir iespējams izmantot kā avota modeli sistēmu definēšanai.
3. Divpusložu modeli var izmantot 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. Ir nepieciešams atzīmēt arī 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 atbalstošā rīka izstrādi.

Tātad, modificējot esošo divpusložu modeļa notāciju un adaptējot to transformācijām programmatūras kodā, promocijas darbā izstrādāto koda ģenerēšanas algoritmu pēc tā implementācijas attiecīgajā modeļvadāmās pieejas atbalsta rīkā var ieviest industrijā, jo algoritms dot iespēju no problēmsfēras ekspertiem saprotamā modeļa iegūt programmatūras kodu. Pētījuma gaitā definētie spriedumi un iegūtie rezultāti var būt interesanti arī plaša loka speciālistiem gan industrijā, gan turpinot zinātniskos pētījumus modeļvadāmās programmatūras jomā.

IZMANTOTĀ LITERATŪRA

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. gada 9. decembrī]. Pieejams: https://docs.angularjs.org/guide/concepts.
4. *ANTLR*. [skatīts 2019. gada 9. decembrī]. Pieejams: http://www.antlr.org/.
5. *ASM*. [skatīts 2019. gada 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. gada 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. gada 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. gada 12. decembrī]. Pieejams: http://www.benf.org/other/cfr/.
15. *Chapter 4. The class File Format*. [skatīts 2019. gada 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. gada 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. Daniel, F., Matera, M. *Mashups: Concepts, Models and Architectures*. Berlin, Germany: Springer-Verlag, 2014. 319 p. ISBN 978-3-642-55048-5.
19. Eilam, E. *Reversing: Secrets of Reverse Engineering*. 1st edition. USA: Wiley, 2005 – 624 p. ISBN 978-0764574818.
20. Epp, S. S. *Discrete Mathematics with Applications*. 4th Edition. USA: Cengage Learning, 2010. 984 p. ISBN 978-0495391326.
21. Evans, E. *Domain-driven design: tackling complexity in the heart of software*. USA: Addison-Wesley Professional, 2003. 560 p. ISBN 978-0321125217.
22. 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.
23. Fowler, M. *Anaemic Domain Model*. [skatīts 2019. gada 13. decembrī]. Pieejams: http://www.martinfowler.com/bliki/AnemicDomainModel.html.
24. Fowler, M. *Refactoring: Improving the Design of Existing Code*. 2nd Edition. USA: Addison-Wesley Professional, 2018. 486 p. ISBN 978-0201485677.
25. Gane, C., Sarson, T. *Structured systems analysis: tools and techniques*. USA: Prentice-Hall, 1979. 241 p. ISBN 978-0138545475.
26. *GitHub – fesh0r/fernflower: Unofficial mirror of FernFlower Java decompiler*. [skatīts 2019. gada 13. decembrī]. Pieejams: https://github.com/fesh0r/fernflower.
27. 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.
28. 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.
29. 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.
30. 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.
31. 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.
32. 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.
33. 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.

34. 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.
35. *Hibernate. Everything data. – Hibernate*. [skatīts 2019. gada 13. decembrī]. Pieejams: http://hibernate.org/.
36. Hunter, D. *Guitar Effects Pedals – the Practical Handbook*. Canada: Backbeat Books, 2004. 192 p. ISBN 978-0879308063.
37. *Intel® 64 and IA-32 Architectures Software Developer Manuals | Intel® Software*. [skatīts 2019. gada 13. decembrī]. Pieejams: https://software.intel.com/en-us/articles/intel-sdm.
38. *ISO/IEC 2382:2015(en), Information technology – Vocabulary*. [skatīts 2019. gada 13. decembrī]. Pieejams: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en.
39. *ISO - ISO/IEC 9075-1:2016 - Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. [skatīts 2019. gada 13. decembrī]. Pieejams: https://www.iso.org/standard/63555.html.
40. *ISO - ISO/IEC 9899:2011 – Information technology – Programming languages – C*. [skatīts 2019. gada 13. decembrī]. Pieejams: https://www.iso.org/standard/57853.html.
41. *Java | Oracle*. [skatīts 2019. gada 13. decembrī]. Pieejams: https://www.java.com/en/.
42. *Java Decompiler*. [skatīts 2019. gada 13. decembrī]. Pieejams: http://java-decompiler.github.io/.
43. 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.
44. 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.
45. Knuth, D. E. *The Art of Computer Programming*. 1st Edition. USA: Addison-Wesley Professional, 2003. 9998 p. ISBN 978-0321751041.
46. 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.
47. Koshy, T. *Discrete Mathematics With Applications*. 1st Edition. USA: Academic Press, 2003. 1042 p. ISBN 978-0124211803.
48. 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.
49. Luís Marques. *A defense of so-called anemic domain models. Slides of D-Lang-Silicon-Valley Meetup @ January 28, 2016*. [skatīts 2019. gada 24. decembrī]. Pieejams: http://files.meetup.com/18234529/luis_marques_anemic_domain_models.pdf.
50. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st Edition. USA: Prentice Hall, 2008. 464 p. ISBN 978-0132350884.
51. 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.
52. *mstrobel / procyon – Bitbucket*. [skatīts 2019. gada 24. decembrī]. Pieejams: https://bitbucket.org/mstrobel/procyon.

53. Niaz, I.A., Jiro, T. Mapping UML statecharts to Java code. In: *IASTED Conf. on Software Engineering*, 2004.
54. 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.
55. 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.
56. 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.
57. 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.
58. 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.
59. 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.
60. 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).
61. 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.
62. 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.
63. 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.
64. 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.

65. 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.
66. 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.
67. 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.
68. 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.
69. 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.
70. 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.
71. 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.
72. 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.
73. *Overview: PIC18F45Q10 - 8-bit Microcontrollers.* [skatīts 2020. gada 25. martā]. Pieejams: https://www.microchip.com/wwwproducts/en/PIC18F45Q10.
74. 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.
75. Perisic, B. Model Driven Software Development – State of the Art and Perspectives. In: *Proceedings of INFOTEH 2014*, Volume 13. pp. 1237-1248.
76. Ritchie, D. M., Kernighan, B. W. *The C Programming Language*. Second Edition. USA: Prentice Hall, 1988. 272 p. ISBN 978-0131103627.
77. *Ruby Programming Language*. [skatīts 2019. gada 26. decembrī]. Pieejams: https://www.ruby-lang.org/en/.
78. Salomon, D. *Assemblers and Loaders*. USA: Prentice Hall, 1993. 308 p. ISBN 978-0130525642.

79. 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.
80. Skiena, S. S. *The Algorithm Design Manual*. 2nd Edition. Germany: Springer, 2008. 748 p. ISBN 978-1849967204.
81. *Standard C++*. [skatīts 2019. gada 26. decembrī]. Pieejams: https://isocpp.org/.
82. 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.
83. 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.
84. 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.
85. 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.
86. *The Anaemic Domain Model is no Anti-Pattern, it's a SOLID design | SAPM: Course Blog*. [skatīts 2019. gada 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/.
87. *The LEX & YACC Page*. [skatīts 2019. gada 26. decembrī]. Pieejams: http://dinosaur.compilertools.net/.
88. *TIOBE Index | TIOBE – The Software Quality Company*. [skatīts 2018. gada 1. decembrī]. Pieejams: https://www.tiobe.com/tiobe-index/.
89. *Trygve/MVC*. [skatīts 2019. gada 26. decembrī]. Pieejams: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.
90. *Unified Modeling Language*. [skatīts 2019. gada 26. decembrī]. Pieejams: https://www.omg.org/spec/UML/.
91. 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.
92. 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.
93. *W3C XML Schema*. [skatīts 2019. gada 26. decembrī]. Pieejams: http://www.w3.org/XML/Schema.
94. 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.
95. *Welcome to Python.org*. [skatīts 2019. gada 26. decembrī]. Pieejams: https://www.python.org/.
96. Wright, D. R., *Finite State Machines (CSC215 Class Notes)*. [skatīts 2017. gada 20. septembrī]. Pieejams: http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf.