LATVIJAS UNIVERSITĀTE
Datorikas fakultāte
Matemātikas un informātikas institūts

EDGARS RENCIS

UZ MODEĻU TRANSFORMĀCIJĀM BALSTĪTU RĪKU BŪVES METOŽU IZSTRĀDE UN REALIZĀCIJA

Promocijas darbs

Nozare: datorzinātnes
Apakšnozare: programmēšanas valodas un sistēmas

Zinātniskais vadītājs:
profesors, Dr. Habil. Sc. Comp.
JĀNIS BĀRZDIŅŠ

Rīga 2011

Šis darbs izstrādāts ar Eiropas Sociālā fonda atbalstu projektā «Atbalsts doktora studijām Latvijas Universitātē».

Saturs

Attēli

Tabulas

1. Ievads

Arvien lielāka uzmanība programmatūras sistēmu būves lauciņā mūsdienās tiek pievērsta iespējām automatizēt atsevišķus sistēmu būves procesa posmus. Programmēšanas darba automatizāzija uzskatāma par vienu no būtiskākajiem priekšnoteikumiem ātras, nedārgas un salīdzinoši viegli uzturamas sistēmas izstrādē. Šajā darbā lielākā uzmanība pievērsta konkrētas programmatūras sistēmu tipa – grafisku problēmorientētu rīku – izstrādei.

Rodoties vēlmei pēc jauna problēmorientēta grafiska rīka, mūsdienās vairs netiek nopietni apsvērta iespēja radīt to no nulles, bet tā vietā darba veikšanai tiek izvēlēta kāda no daudzajām pasaulē eksistējošajām šādu rīku būves platformām. Kā populārākie šādu platformu piemēri minami kompānijas MetaCase izstrādātais MetaEdit+ [1], Microsoft izstrādātais DSL Tools [2] un Eclipse radītais GMF [3]. Tāpat vērts pieminēt arī Latvijas Universitātes Matemātikas un informātikas institūtā izstrādāto grafisku problēmorientētu rīku būves platformu METAclipse [4]. Katrā no šīm platformām atrodami savi plusi un savi mīnusi, bet katram konkrētam uzdevumam – konkrēta grafiska rīka izstrādei – parasti iespējams atrast platformu, kas tam der vislabāk.

Strādājot pie platformas GRAF, kuras izstrāde arī sākusies (un vēl joprojām turpinās) Latvijas Universitātes Matemātikas un informātikas institūtā, galvenais uzsvars vienmēr ticis likts uz tās lietojamību un piedāvāto iespēju klāstu. Platforma ir balstīta uz metamodeļiem un veidota akadēmiska tipa, kas ļāvis tajā ietvert un izmēģināt dažādu arī no zinātniskā aspekta interesantu funkcionalitāti. Arī šai platformai, tāpat kā visām pārējām pasaulē eksistējošajām, ir savas priekšrocības un savi trūkumi, turklāt starp tās priekšrocībām noteikti iespējams atrast tāda veida funkcionalitāti, kādu nepiedāvā pat neviena no pasaulē labi zināmajām rūpnieciskajām platformām. Šajā darbā lielos vilcienos aprakstīta pati platforma, tās pamata sastāvdaļas un darbības principi, kā arī dots dziļāks ieskats vairākās konkrētās platformas uzlabošanas iespējās.

Darbs sadalīts astoņās nodaļās. Pēc ievada otrā nodaļa ievada lasītāju modelēšanas un metamodelēšanas pasaulē, kas ir šāda veida rīku būves platformu pamatā. Šajā nodaļā iztirzāti tālāk izmantotie pamatjēdzieni – modelis, metamodelis, uz modeļiem balstīta arhitektūra, modeļu un metamodeļu transformāciju valodas, problēmorientētas valodas un citi. Šī nodaļa uzskatāma par absolūto priekšzināšanu minimumu, ar kuru lasītājam būtu jābūt pazīstamam, pirms pievērsties tālākajām šī darba nodaļām.

Trešajā nodaļā īsumā iztirzāta platforma GRAF. Dots neliels ieskats platformas attīstības vēsturē, ļaujot izsekot līdzi attīstības tendencēm, tādējādi labāk izprotot šī brīža risinājuma pamatotību. Tāpat nodaļā aprakstīti arī GRAF darbības pamata principi, atklājot metamodeli, ar kura palīdzību tiešā veidā tiek definēti problēmorientētas valodas realizējošie rīki. Tālāk šajā nodaļā dots vispārīgs uz transformācijām balstītas arhitektūras apraksts. Šī arhitektūra ir platformas GRAF pamatā un tā detalizētāk tiks aprakstīta Sergeja Kozloviča doktora disertācijā. Nodaļa tiek pabeigta ar svarīgāko platformas izaicinājumu definēšanu, kuru apraksti un risinājuma varianti ir šī darba pamata sastāvdaļas un kas sīkāk iztirzāti tālākajās darba nodaļās.

Ceturtā nodaļa veltīta modeļu un metamodeļu transformāciju valodu saimei Lx. Aprakstīta bāzes transformāciju valoda L0, uz kuras pamata tālāk, izmantojot sāknēšanas algoritmu, tiek būvētas nākamās šīs saimes valodas – L0’, L1, L2 un L3. Aprakstīta visu valodu tekstuālā sintakse, doti to metamodeļi, paskaidrojot valodas konstrukciju semantiku, kā arī pievērsta uzmanība dažām to realizācijas īpatnībām. Īpaši izdalīta valoda L1, kas satur sevī vienu no modeļu transformāciju valodu pamata konstrukcijām – šablonu atbilstības definēšanas funkcionalitāti – un veikts šīs valodas salīdzinājums ar pirmās pakāpes predikātu valodu.

Piektajā nodaļa aprakstīts mehānisms, ar kura palīdzību uz kādu no platformas sastāvā esošajiem metamodeļiem iespējams palūkoties no dažādiem aspektiem. Šeit tiek definēts metamodeļa skata jēdziens un aprakstītas veicamās darbības, kas jāveic, lai konkrētam metamodelim uzbūvētu konkrētu skatu. Nodaļa pabeigta ar skatu mehānisma iespējamo lietojumu aprakstu, kā arī piedāvāts ieskats citur pasaulē veiktajos pētījumos līdzīgā sakarā.

Sestajā nodaļā piedāvāts risinājums, kādā veidā iespējams sadzīvot ar dažāda veida izmaiņām platformā vai ar to nodefinētajos problēmorientētajos rīkos. Nodaļa sākas ar precīzu izskaidrojumu, par kāda veida izmaiņām iespējams runāt, un turpinās ar šīs problēmas risinājuma detalizētu izklāstu. Nodaļas beigās īsumā aprakstīti arī citi šī eksporta/importa mehānisma izmantošanas piemēri bez tiem, kas jau kalpojuši par motivāciju mehānisma radīšanai.

Septītajā nodaļā īsumā ieskicēti tālākie pētījumu virzieni rīku būves platformas attīstīšanas jomā, bet astotā nodaļa noslēdz promocijas darbu – tajā aprakstīti galvenie pētījuma laikā radušies secinājumi.

2. Pamatjēdzieni

Mūsdienās, apgrozoties programmētāju, sistēmu būvētāju un analītiķu aprindās, arvien biežāk nākas dzirdēt runājam par modeļiem un metamodeļiem, par to izmantošanu sistēmu būvē, par to lielo lomu sistēmu projektēšanā un tamlīdzīgi. Šajā nodaļā autors devis vispārēju pārskatu pār šo datorzinātnes nozari, nodrošinot lasītāju ar pamatzināšanām, kas nepieciešamas labākai pārējo nodaļu izpratnei.

2.1. Modelis un metamodelis

Kas tad īsti ir šis pamatu pamata termins – modelis? Visvispārīgākā modeļa definīcija ir šāda – modelis ir jebkas, kas tiek vai var tikt lietots kādā nolūkā kā cita vietā [5]. Ar modeli datorzinātņu sfērā parasti saprot teorētisku konstrukciju, kas attēlo kaut kādu pasaules fragmentu. Modelis, vienkārši runājot, ir kādu savstarpēji saistītu objektu kopa, kas attēlo kādu jēdzienu. Modeļi parasti tiek būvēti nolūkā labāk izprast kādu specifisku sfēru, problēmu apgabalu vai reālās pasaules fragmentu. Fiziski modelis parasti ir grafisks attēls, kurā pēc noteiktiem likumiem izveidoti un izvietoti dažādi grafiskie elementi. Taču, skatoties plašāk, modeļa grafiskā reprezentācija nav obligāta prasība modeļu veidošanā.

Kā jau augstāk minēts, modeļi tiek veidoti stingrā saskaņā ar kaut kādiem modelēšanas likumiem. Kas tad šos likumus nosaka? Vispārīgi runājot, var teikt, ka katrs modelis tiek veidots saskaņā ar noteikumiem, kādus nosaka kāds cits augstāka līmeņa modelis – metamodelis. Par metamodeli var domāt kā par vispārīgu shēmu, kurai var pakļauties vairāki modeļi. Piemēram, viens metamodelis var tikt izmantots uzņēmuma iekšējās struktūras attēlošanā, cits metamodelis – raķešu būves mašinērijā, vēl cits – informatīvās sistēmas aprakstā. Katrā no šiem gadījumiem metamodelis pārstāv vairākus iespējamos modeļus un nosaka likumus, pēc kuriem vadoties iespējams veidot jaunus modeļus, kas atbilstu dotajam metamodelim jeb būtu šī metamodeļa instances. Piemēram, gadījumā ar uzņēmuma iekšējās struktūras attēlojošu metamodeli katrs uzņēmums var veidot savu modeli, vadoties pēc šī viena kopējā metamodeļa. Dažādiem uzņēmumiem šie modeļi iznāks dažādi, bet tos vienos kopējs metamodelis un kopēji zināmi likumi.

Skatoties vēl plašākā mērogā, jāsecina, ka katrā modelēšanas situācijā vienmēr ir iespējami trīs abstrakcijas līmeņi [6] – modeļa instances, pats modelis un tā metamodelis. Patiešām – domājot par kādu modeli, visbiežāk galvenā interese ir gan par to, ko šis modelis sevī varēs saturēt (instanču līmenis), gan arī par to, kādā veidā un pēc kādiem principiem šis modelis ir

veidots (metamodeļa līmenis). Līdz ar to šāds triju pakāpju risinājums ir visai dabisks un intuitīvi viegli saprotams. Literatūrā pieņemts šos trīs abstrakcijas līmeņus apzīmēt ar M0 (modeļa instances), M1 (modelis) un M2 (metamodelis) [6].

Taču reizēm modelētāji runā par vēl vienu pakāpi augstāku abstrakcijas līmeni – M3 jeb meta-metamodeli. Kā jau noprotams, meta-metamodelis nosaka likumus, pēc kuriem veidojami tam atbilstošie metamodeļi jeb šī meta-metamodeļa instances. Meta-metamodeļi parasti pakļaujas paši savai semantikai, tas ir, tie var tikt definēti, izmantojot to pašu jēdzienus. Eksistē, piemēram, tādi meta-metamodeļi kā MOF (*Meta-Object Facility*, [7]) un Ecore [8]. Galvenais princips, kas jāievēro attiecībā uz šiem dažādajiem abstrakcijas līmeņiem, ir fakts, ka par modelēšanas pamatjēdzieniem tiek uzskatīti jēdzieni „Klase" un „Objekts", kā arī spēja no katras konkrētas instances iegūt tās meta-objektu, kas atrodas nākamajā abstrakcijas līmenī. Līdz ar to, balstoties uz šādu pieņēmumu, iespējams veiksmīgi tikt galā ar patvaļīgu skaitu meta-līmeņiem.

2.2. Uz modeļiem balstīta arhitektūra

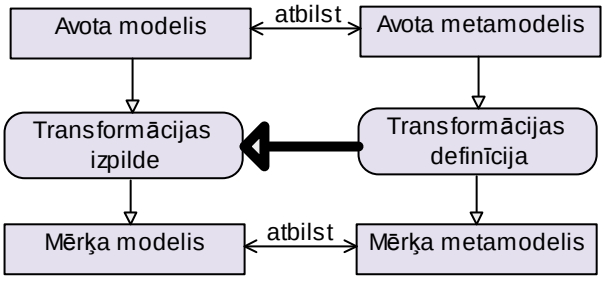
Uz modeļiem balstīta arhitektūra (MDA – *Model Driven Architecture* [9]) ir lietojumprogrammu veidošanas un specifikāciju rakstīšanas veids, kas mūsdienās strauji attīstās. Tas balstās uz lietojumprogrammas vai specifikācijas funkcionalitātes modeli, kas nav atkarīgs no platformas (PIM – *Platform Independent Model*). Pilna MDA specifikācija sastāv no viena no platformas neatkarīga bāzes modeļa un no viena vai vairākiem no platformas atkarīgiem modeļiem (PSM – *Platform Specific Model*), kā arī no saskarņu definīciju kopas, kur katra saskarne apraksta veidu, kā bāzes modelis ir realizēts katrā no piedāvātajām platformām. Savukārt, pilna MDA lietojumprogramma sastāv no viena no platformas neatkarīga modeļa un viena vai vairākiem no platformas atkarīgiem modeļiem un pilnām realizācijām katrā no platformām, kuras lietojumprogrammu izstrādātājs ir izvēlējies atbalstīt.

Šī pieeja izmantot sistēmu arhitektūrā meta-modelēšanas primitīvus ir radusies salīdzinoši nesen – 1996. gadā OMG (*Object Management Group* [10]) pievienoja savai pārraudzībai modelēšanu un 1997. gadā pieņēma vienotu modelēšanas valodu – UML (Unified Modeling Language [11]). Šo laiku var uzskatīt par MDA rašanās sākumu. Kopš šī brīža MDA ir strauji attīstījusies un arī vēl šodien tās attīstības tempi nav mazinājušies.

2.3. Modeļu transformāciju valodas

Metamodeļu transformācija ir process, kas ir pašā pamatā visai modelēšanas zinātnei. Šis process nozīmē kāda modeļa (vai metamodeļa) pārveidošanu par kādu citu modeli (vai metamodeli). Izplatītākais gadījums, ar ko visbiežāk darbojas modelētāji, ir tieši modeļu transformācijas, kas nozīmē paņemt kāda fiksēta metamodeļa modeli un pārveidot to par kāda cita (vai arī tā paša) fiksēta metamodeļa modeli. Taču arī pašu metamodeļu transformēšana nav nekas neparasts mūsdienu modelēšanā.

Modeļu transformāciju valoda ir programmēšanas valoda, kas paredzēta modeļus transformējošu procesu aprakstīšanai. Programma, kas rakstīta kādā no modeļu transformāciju valodām, tipiskākajā gadījumā kā ieejas datus saņem kāda metamodeļa modeli, bet kā izejas datus atgriež kāda cita (vai tā paša) metamodeļa modeli (skatīt attēlu 2.1.). Precīzāk – transformācija tiek definēta kādā no modeļu transformāciju valodām, balstoties uz noteiktu avota un mērķa metamodeli. Savukārt, transformācijas izpilde ir process, kurš kā ieejas datus saņem modeli, kas atbilst avota metamodelim, un transformācijas definīciju, bet kā izejas datus izdod modeli, kas atbilst mērķa metamodelim un ir iegūts, pārveidojot avota modeli atbilstoši saņemtajai transformācijas definīcijai. Tā kā šajā tipiskākajā gadījumā darbs notiek nevis ar metamodeļiem, bet tikai ar to modeļiem, tad šādas valodas dēvējamas par modeļu transformāciju valodām. Turpretī, ja valoda piedāvā līdzekļus ne tikai modeļu, bet arī metamodeļu transformēšanai, tad šāda valoda var tikt pieskaitīta arī metamodeļu transformāciju valodu saimei.



2.1. attēls. **Modeļu transformācijas shēma**

Pasaulē šobrīd eksistē vairākas modeļu transformāciju valodas. Līdz ar vēlmi definēt modeļu transformācijas radās arī nepieciešamība pēc speciālām šim nolūkam paredzētām programmēšanas valodām – transformāciju valodām – jo neviena no esošajām programmēšanas vai modelēšanas valodām nebija tik ļoti piemērota šai vajadzībai, cik nepieciešams. Līdz ar to OMG 2002. gada aprīlī izsludināja konkursu uz šādas valodas standartu – QVT (Queries/Views/Transformations [12]). Šobrīd ir izstrādāta viena vadošā

transformāciju valoda – MOF QVT [13], kā arī vesela virkne citu modeļu transformāciju valodu – Tefkat [14], Viatra [15], GReAT [16], ATL [17], AGG[18], Fujaba[19], UMLX [20], MOLA [21] un citas.

Modeļu transformāciju valodas var būt gan tekstuālas, gan arī grafiskas. Jebkurā gadījumā arī pati transformāciju programma, kas rakstīta kādā no transformāciju valodām, ir uztverama kā modelis, kas atbilst dotās transformāciju valodas metamodelim. No augstāk minētajām transformāciju valodām tekstuālās valodas ir Tefkat, ATL un TRL, savukārt, valodas Viatra, GReAT, AGG, Fujaba, UMLX un valoda MOLA ir grafiskas. Jāatzīmē gan, ka valodai Viatra eksistē arī diezgan laba tekstuālā forma.

Šajā darbā īpaši izceļama būtu modeļu transformāciju valoda MOLA. Tā ir grafiska transformāciju valoda, kas balstīta uz tradicionāliem modelēšanas zinātnes jēdzieniem, tādiem kā šablonu atpazīšana un likumi, kas nosaka to, kā atpazītā šablona elementi būtu jātransformē [21]. Transformāciju valoda izstrādāta Latvijas Universitātes Matemātikas un informātikas Institūtā 2004. gadā. Reizē ar modeļu transformāciju valodas MOLA izstrādi Latvijas Universitātes Matemātikas un informātikas Institūtā ir izstrādāta arī bāzes transformāciju valoda L0 [22].

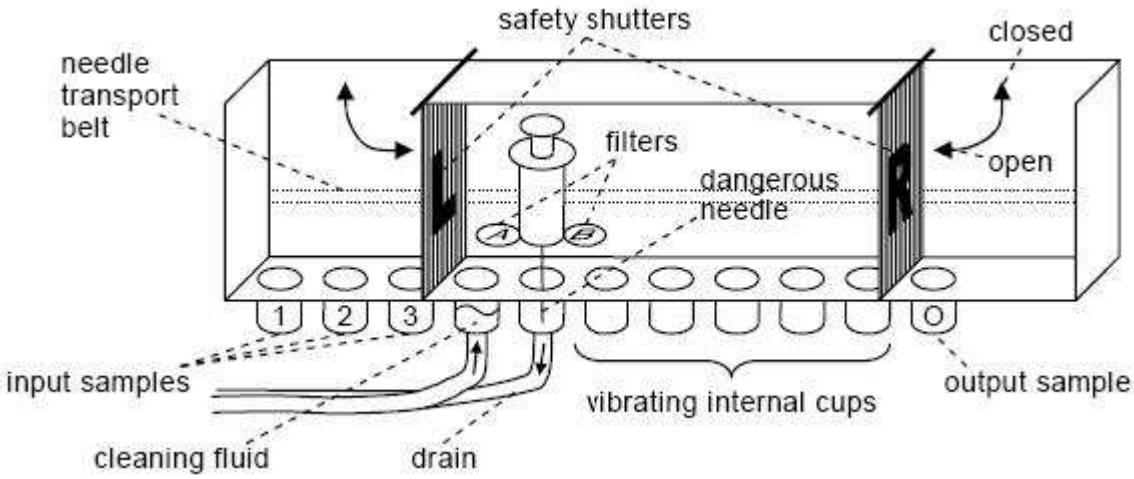
2.4. Problēmorientētas valodas

Ar problēmorientētām valodām tiek saprastas tādas programmēšanas vai specifikāciju valodas, kas paredzētas darbam konkrētā problēmu apgabalā jeb domēnā un piedāvā apstrādes līdzekļus un konstrukcijas šī problēmu apgabala terminos. Pati ideja par speciāla mērķa programmēšanas valodām nav jauna, bet problēmorientētas valodas termins plašāk ieviesies tikai līdz ar problēmorientētās modelēšanas aizsākumiem pagājušā gadsimta deviņdesmito gadu vidū. Problēmorientētas valodas var būt ļoti dažādas atkarībā no problēmu apgabala, ko tās modelē. Tās var būt gan tekstuālas, gan grafiskas. Kā problēmorientētu valodu pretstats minamas vispārējās programmēšanas valodas (tādas kā C++, Java, C# un citas), vai, ja runājam tieši par problēmorientēto modelēšanu – vispārējās modelēšanas valodas (tādas kā UML [11]).

Problēmorientētas valodas ieviešanas un lietošanas mērķis ir atvieglot programmēšanas darbu attiecīgās jomas (problēmu apgabala) ekspertiem, kas ļoti labi pārzina savu terminoloģiju un kam tādējādi būtu jāiegulda mazāks darba apjoms programmēšanas valodas apguvei. Maksa par šādu ērtību iegūšanu var slēpties pašas problēmorientētās valodas izstrādē un realizācijā, tāpēc lielu uzmanību nepieciešams pievērst šīs izstrādes vienkāršošanai, it īpaši, ja plānots

izstrādāt vairākas valodas – katru savam problēmu apgabalam. Šajā sakarā pasaulē attīstījušās programmēšanas sistēmas, sauktas par rīku būves platformām, kas piedāvā ērtu veidu, kā specificēt problēmorientētas valodas. Tāpat šajās platformās iespējams izstrādāt arī šo valodu realizāciju, kas praktiski nozīmē izstrādāt šāda pat veida – problēmorientētu – rīku, kurš piedāvā iespējas darboties ar konkrēto valodu. Šie rīki bieži vien piedāvā iespēju veikt pilnu programmatūras attīstības ciklu, sākot no prasību specifikācijas, kas turpinās ar to realizāciju modeļu līmenī un noslēdzas ar koda ģenerēšanu, iegūstot pilnībā strādājošu lietotni. Viena šāda grafisku problēmorientētu rīku būves platforma – Latvijas Universitātes Matemātikas un informātikas institūtā izstrādātā platforma GRAF – ņemta par pamatu šajā darbā aplūkotajām rīku būves problēmām. Tā dziļāk iztirzāta 3. nodaļā.

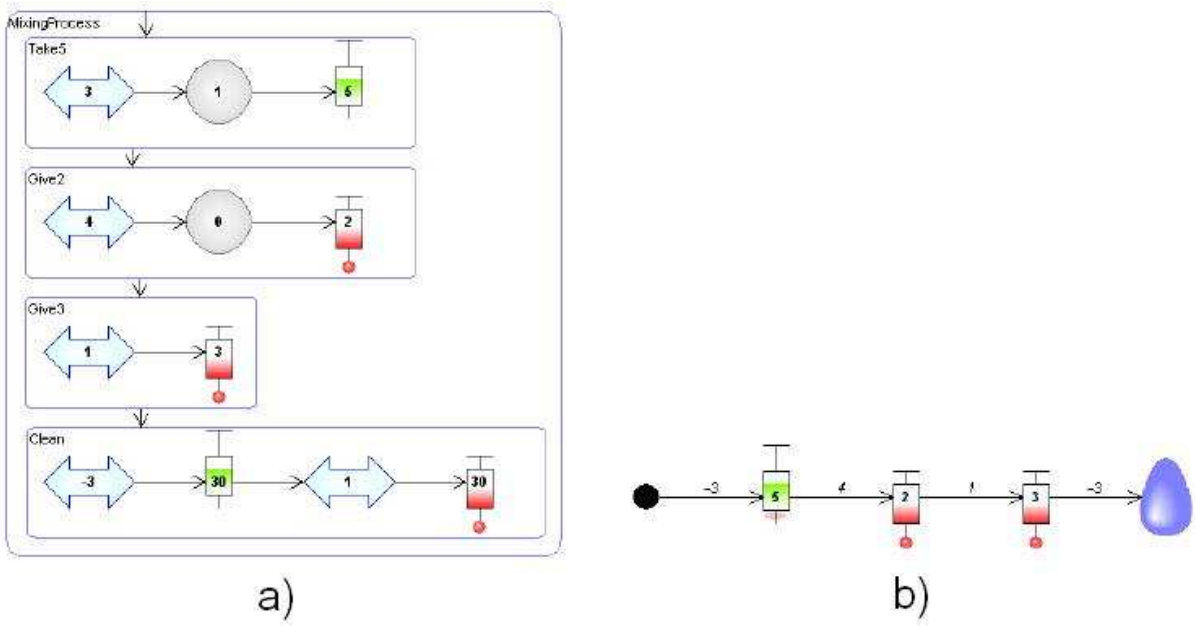
Kā raksturīgs problēmorientētas valodas piemērs minama medicīniskā jaukšanas iekārta [23], kas redzama attēlā 2.2.



2.2. attēls. **Medicīniskā jaukšanas iekārta**

Šī iekārta paredzēta dažādu šķidru medikamentu automātiskai kombinēšanai, sajaucot tos pareizās proporcijās. Katrai no elementārajām darbībām, ko šāda iekārta saprot, šajā valodā eksistē noteikta komanda – pārvietot adatu par noteiktu kolbu skaitu uz vienu vai otru pusi, iesūkt adatā noteiktu daudzumu šķidruma, izpūst no adatas noteiktu daudzumu šķidruma, izmantot kādu no filtriem, utt. –, līdz ar ko valoda uzskatāma par īstu problēmorientētas valodas piemēru.

Lai realizētu šo valodu un automātiski noģenerētu augstākminēto komandu virkni – noteiktas šīs valodas programmas kodu –, iespējams izstrādāt problēmorientētu rīku. Attēlā 2.3. redzams konkrētas šīs valodas programmas piemērs [23], kas izveidots rīkā, kas izstrādāts, izmantojot kompānijas MetaCase rīku būves platformu MetaEdit+ [1].



2.3. attēls. **Medicīniskās jaukšanas iekārtas operējošās valodas piemērs – a) zemāka līmeņa valodas specifikācija; b) augstāka līmeņa valodas specifikācija**

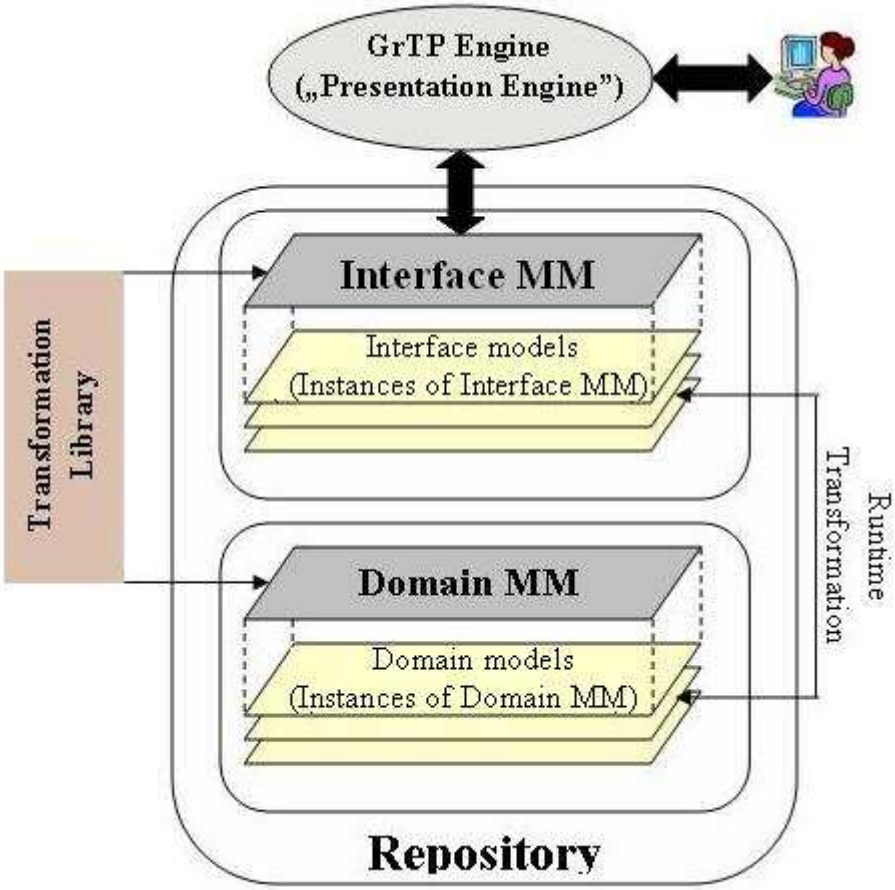
Attēlā 2.3. faktiski redzama viena un tā pati valodas programma, kas pierakstīta divās iespējamās sintaksēs. Attēlā 2.3.a katrai no primitīvajām komandām izveidots speciāls grafiskais simbols, un šos simbolus atļauts grupēt loģiskās grupās, kas semantiski nozīmē kādu noteiktu darbību. Attēlā 2.3.b jau redzama augstāka līmeņa valodas specifikācijas piemērs, kur šī pati programma attēlota īsākā grafiskā sintaksē. Šeit ieviesti grafiski simboli jau loģiskām darbībām, kas apvieno sevī vairākas elementārās operācijas.

Augstākminētais piemērs demonstrē iespēju, ka, eksistējot šāda veida grafisku problēmorientētu rīku būves platformai, ir pietiekami ērti iespējams veidot jaunas problēmorientētas valodas un tās pielāgot katram konkrētajam gadījumam. Nākamajās nodaļās tiks aplūkotas dažāda veida problēmas, ar kurām nākas saskarties rīku būves platformu veidošanas procesā, kā arī piedāvāti tām risinājumu varianti.

3. Grafisku problēmorientētu rīku izstrādes platforma GRAF

3.1. Vēsture un pamatprincipi

GRAF ir uz metamodeļiem balstīta modeļu transformāciju vadīta grafisku rīku būves platforma, kuras realizēšana Latvijas Universitātes Matemātikas un informātikas institūtā uzsākta 2006. gada rudenī. Autors bijis izstrādātāju komandas sastāvā jau no paša sākuma, līdz ar ko piedalījies visos izstrādes posmos.

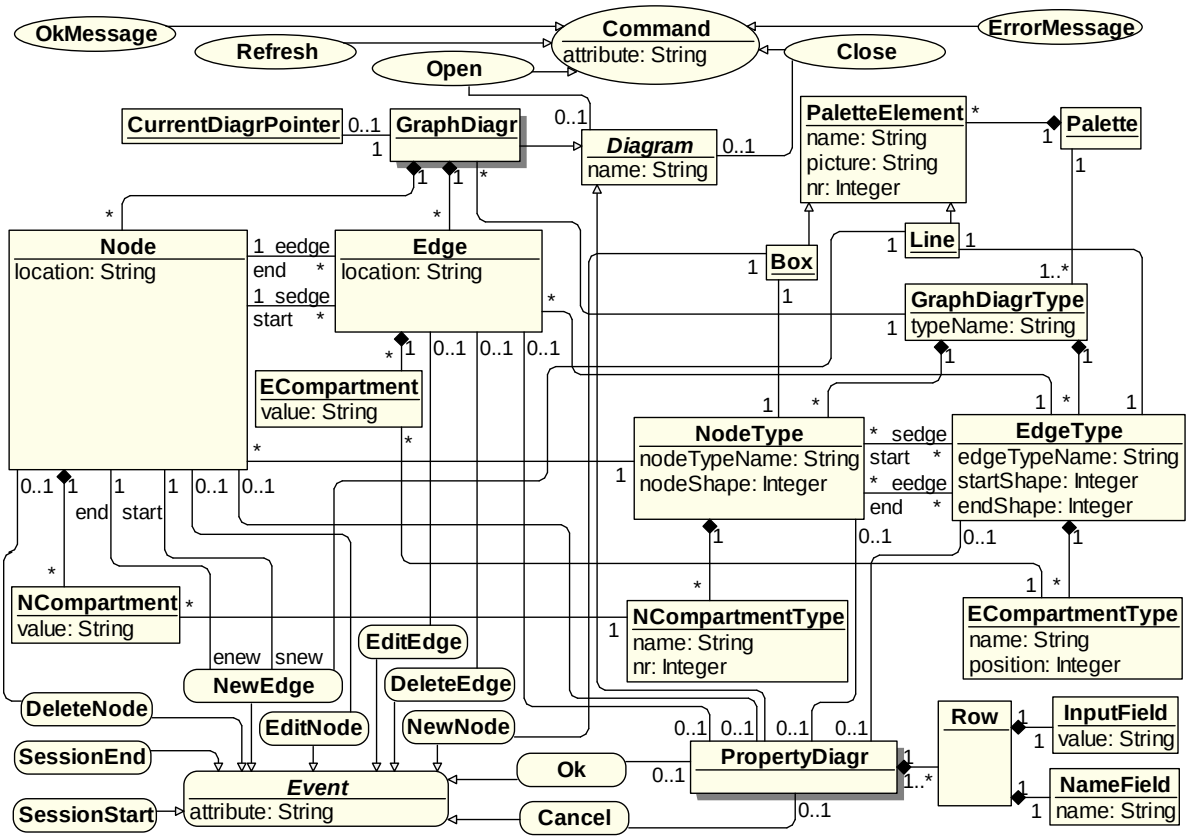


3.1. attēls. **GrTP pirmā versija**

Izstrādes sākuma fāzē platforma balstījusies uz divām pamata komponentēm – grafiska prezentācijas dziņa, kas iepriekš izstrādāts rīka GRADE izstrādes ietvaros [24] un pielāgots jaunajai platformai, un no modeļu repozitorija (dzinis balstās uz attīstītiem grafu izvietošanas algoritmiem, kas ļauj efektīvi tikt galā ar diagrammu vizualizēšanas uzdevumiem pat ļoti lielu diagrammu gadījumā [25, 26]). Repozitorijā esošie metamodeļi – domēna un prezentācijas metamodelis – definēja platformas attiecīgi abstrakto un konkrēto sintaksi, bet modeļi, kurus šie metamodeļi saturēja, definēja konkrētus ar platformu izstrādātus rīkus (skatīt attēlu 3.1.,

kurā redzama rīka, kas toreiz saucās GrTP (***Gr**aphical **T**ool-building **P**latform*), pirmā versija). Savukārt, prezentācijas dziņa uzdevums bija attēlot šos rīkus vizuāli, kā arī nodrošināt lietotāja darbību pārķeršanu. Rīku semantika tika nodrošināta ar modeļu transformāciju palīdzību. Pārķerot kādu lietotāja darbību, prezentācijas dzinis varēja nodot vadību modeļu transformāciju bibliotēkai, kura atkarībā no notikušā notikuma (lietotāja darbības, kuru kodētā veidā repozitorijā ierakstījis dzinis) sazarojās, izsaucot vienu vai otru modeļu transformāciju šī notikuma loģiskajai apstrādei. Transformācija, beidzot darbu, varēja nodot komandu atpakaļ prezentācijas dzinim, ierakstot to kodētā veidā repozitorijā (piemēram, komanda „Pārzīmēt ekrānu atbilstoši modeļa izmaiņai repozitorijā"). Tādējādi, visa sasaiste starp dzini un transformāciju bibliotēku notika tikai caur repozitoriju, nodrošinot sistēmas viendabīgumu un noslēgtību. Šajā posmā autors aktīvi piedalījies modeļu transformāciju izstrādē, testējot un pilnveidojot platformu ar pirmo nosaukumu GrTP, kā arī mēģinot uzbūvēt pirmos reāli strādājošu rīku prototipus – UML klašu un aktivitāšu diagrammas. Rezultāti nopublicēti rakstā „GrTP: Transformation Based Graphical Tool Building Platform" [27].

Jau pašā pirmajā platformas GrTP versijā eksistēja jēdziens par tipu metamodeli – metamodeli, ar kura palīdzību iespējams ievērojami atvieglot modeļu transformāciju rakstītāju darbu katra konkrēta rīka izstrādē ar platformu. Galvenā tipu (jeb oficiālā definīcijā – rīku definēšanas) metamodeļa ideja radās no novērojuma, ka, rakstot modeļu transformācijas dažādiem rīkiem, daudz lietas ļoti bieži atkārtojas. Nepārprotami radās vēlme atdalīt šīs kopīgās lietas atsevišķi un katru reizi no nulles rakstīt tikai atšķirīgo. Tā arī ir rīku definēšanas metamodeļa pamata būtība – sadalīt rīka grafiskos pamatelementus pa tipiem un katru tipu raksturojošās iezīmes realizēt tikai vienreiz. Iedziļinoties nedaudz sīkāk – blakus prezentācijas metamodelim repozitorijā tika izveidots, tā saucamais, tipu metamodelis, kura instances pēc būtības bija katrs konkrētais grafiskais rīks (skatīt attēlu 3.2.). Tipu metamodelī tika nodefinētas tādas virsotņu (jeb kastu) tipu īpašības kā vārds, vizuālais izskats, īpašību redaktora logs, tādas šķautņu (jeb līniju) tipu īpašības kā līnijas sākuma, beigu un vidus izskats, un tādas iezīmju (jeb teksta lauku) tipu īpašības kā pozīcija un formatējums. Līdz ar šādu tipu eksistenci, modeļu transformāciju rakstītājam nu pietika uzrakstīt katra notikuma apstrādājošo transformāciju vienreiz priekš katra elementa tipa.



3.2. attēls. **Vienkāršots rīku definēšanas metamodelis**

Izmantojot rīka pirmo versiju, promocijas darba autors piedalījies AGTIVE konferences ietvaros notiekošā rīku konkursā [28], kura rezultāti atspoguļoti kopīgā publikācijā [29].

Tomēr izrādījās, ka šis transformāciju rakstītāja darbs vienalga vēl joprojām ir ārkārtīgi liels, mehānisks un vienveidīgs, jo liela daļa darbību, kas jāietver loģiskajā transformācijā, ir kopīgas ne tikai elementiem viena tipa, bet pat vesela notikuma ietvaros. Piemēram, notikums „Jaunas kastes veidošana" vispirms ietver sevī virkni darbību, kas kopīgas it visiem elementiem (pārbaude, vai veidošana ir atļauta, jaunas kastes izveide repozitorijā, jaunās kastes pievienošana aktīvajai diagrammai, atbilstošā īpašību redaktora loga atvēršana utt.), tikai pēc kurām tālāk seko tipa specifiskās darbības. Tādējādi neizbēgami radās vēlēšanās pirms iekavām iznest arī šīs katram lietotāja radītam notikumam specifiskās lietas.

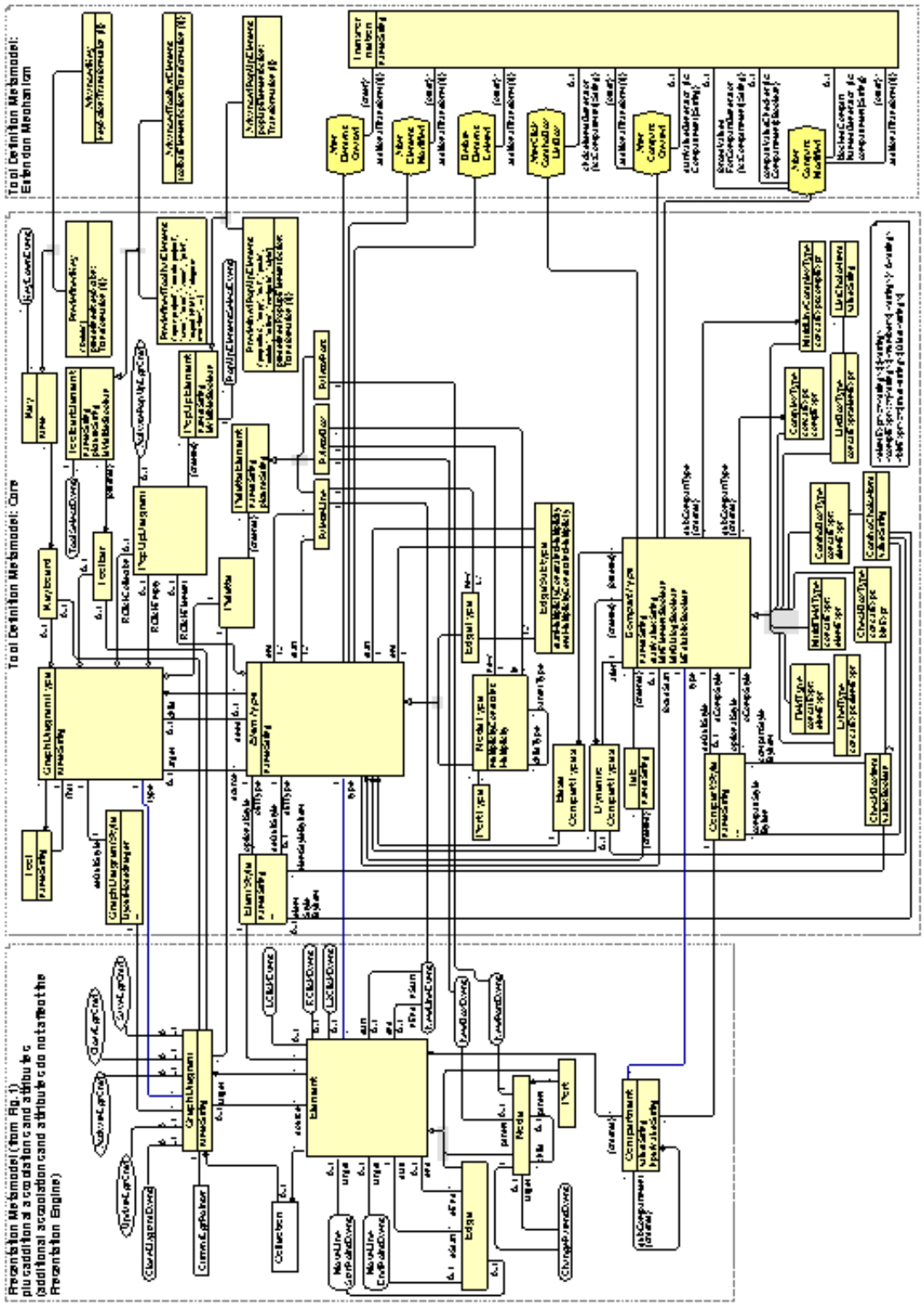
Risinājums šeit tika realizēts, ieviešot, tā saucamo, universālo lietotāja radīto notikumu apstrādājošo modeļu transformāciju, kas vienā noteiktā (universālā) veidā spētu tikt galā ar visiem iepriekšdefinētajiem notikumu veidiem. Katra notikuma kopīgās lietas tiktu realizētas šajā transformācijā, kura notikuma ievaros atšķirīgos datus ņemtu no rīku definēšanas metamodeļa izveidotās instances (kas glabā katram elementa tipam kopīgās lietas). Rezultātā vienīgais, ko būtu nepieciešams izdarīt jauna vienkārša grafiskā rīka izveidei, būtu izveidot korektu rīku definēšanas metamodeļa instanci, kurā pateikts, kādiem jāizskatās elementiem,

kādus teksta laukus tiem jāsatur, kādiem jābūt īpašību redaktoriem un citas tamlīdzīgas lietas. Rīka veidošana tādējādi būtu ne vairs tik daudz tādu modeļu transformāciju rakstīšana, kas, reaģējot uz notikumiem, veic kaut kādas izmaiņas repozitorijā, bet gan tikai tādu transformāciju izveide, kas, sākot darbu, izveido repozitorijā korektu rīku definēšanas metamodeļa instanci. Līdz ar to arī pats rīku definēšanas metamodelis ar laiku kļuva arvien lielāks, nodrošinot arvien jaunas iespējas, ko par jaunveidojamo rīku iespējams pateikt.

Taču, lai vai cik liels un spēcīgs neizaugtu rīku definēšanas metamodelis, vienmēr varēs vēlēties izveidot tādu rīku, kuru šajā metamodelī ielikt nevarēs. Tāpēc pilnībā no iespējas modificēt rīka uzvedību ar specifisku modeļu transformāciju palīdzību atteikties nedrīkstēja. Tas tika ņemts vērā, realizējot tā saucamo paplašināšanas mehānismu – universālajā notikumus apstrādājošajā transformācijā tika paredzētas vietas, no kurām būtu iespējams izsaukt lietotāja (rīka veidotāja) paša rakstītas modeļu transformācijas, kurās viņš tad varētu pierakstīt klāt to, ko nebūtu iespējams priekšdefnēti uzrakstīt rīku definējošā modeļa veidojošajā transformācijā. Tehniski šis paplašināšanas mehānisms tika realizēts ļoti vienkārši – rīku definēšanas metamodelī katram no elementu, diagrammu, iezīmju utt. tipiem tika pievienoti vairāki atribūti, katrā no kuriem varētu pierakstīt vienas transformācijas vārdu, kuru universālajai transformācijai izsaukt noteiktā situācijā. Piemēram, klasei „ElemType” (kas raksturo elementa tipu) tika pievienots atribūts „AfterElementCreated”, kura vērtība būtu tās specifiskās modeļu transformācijas vārds, kuru universālajai transformācijai nepieciešams izsaukt brīdī, kad tā radījusi repozitorijā jaunu elementu ar šo tipu (skatīt attēlu 3.3., kurā gan minētais atribūts un tam līdzīgie iznesti ārpusē kā atsevišķas klases, nemainot būtību). Tādējādi, lietotājs pats var izvēlēties, kuras specifiskās transformācijas tam nepieciešamas un realizēt tās pilnībā lokāli, daudz nedomājot par kopējo notikuma apstrādāšanu. Lai arī šī shēma – universālā transformācija kopā ar rīku definēšanas metamodeli ar paplašināšanas mehānismu – ir ļoti spēcīgs instruments, kas ļauj ļoti viegli veidot jaunus grafiskus rīkus, tomēr dēļ šī metamodeļa apmēriem un sarežģītības ideju ilgi nebija iespējams izskaidrot tik elegantā veidā, lai to pieņemtu publicēšanai. Galu galā rīku definēšanas metamodelis tomēr ir publicēts [30].

Vēlāk radās doma par to, ka bez prezentācijas dziņa sistēmā varētu būt vēl arī citi dziņi – dialogu logu dzinis, datubāžu dzinis, teksta redaktora dzinis utt. –, kas katrs varētu veikt šādu divpusēju sadarbību ar modeļu transformācijām, kā aprakstīts augstāk. Tādējādi sistēmas centrā būtu transformācijas, bet dziņi piedāvātu sistēmai dažādas saskarnes, kur katra saskarne sastāvētu no paša dziņa un tā metamodeļa, ko šis dzinis realizē. Tā dzima doma par transformāciju vadītu arhitektūru, kas nedaudz sīkāk aprakstīta nodaļā 3.2. Šobrīd platforma

GRAF [31] sastāv no principiem, kas veido transformāciju vadīto arhitektūru [32] un kas apaudzēti ar servisiem, kurus piedāvā universālā transformācija kopā ar rīku definēšanas metamodeli ar paplašināšanas mehānismu, kā arī ar rīku konfiguratoru, kas lietotājam ļauj vieglākā – grafiskā – veidā (nekā, rakstot modeļu transformācijas) izveidot rīku definēšanas metamodeļa instances [33].

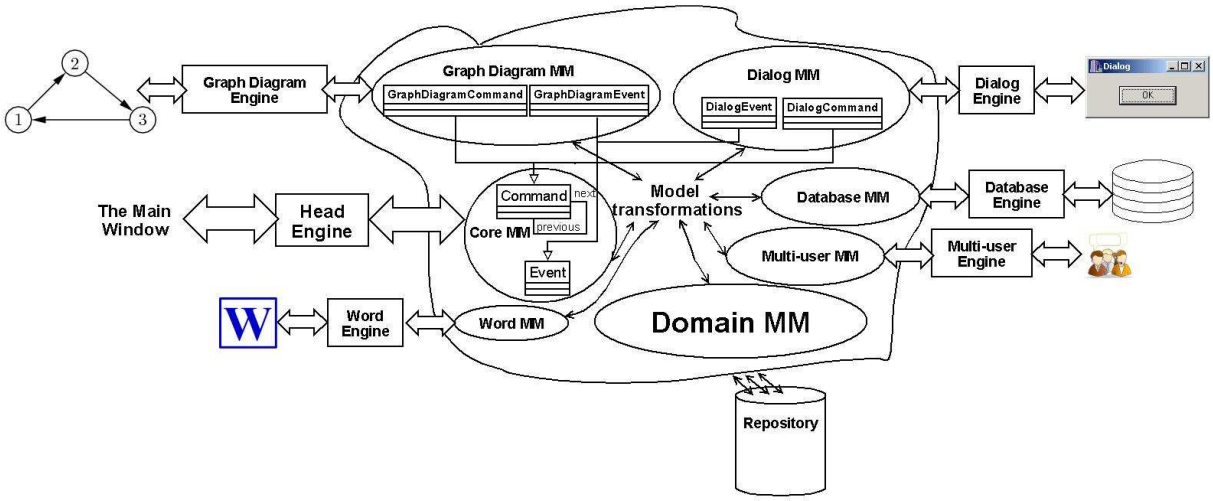


3.3. *attēls.* **Rīku definēšanas metamodelis**

3.2. Ieskats transformāciju vadītā arhitektūrā

Transformāciju vadīta arhitektūra (*The Transformation-Driven Architecture*, turpmāk tekstā – TDA) apvieno sevī principus un idejas jauna veida sistēmu un rīku būves pieejai. Arhitektūras definīcija izstrādāta Latvijas Universitātes Matemātikas un informātikas institūtā 2008. gada otrajā pusē, un tās izstrādes sākuma posmā daļēji piedalījies arī šī darba autors. Arhitektūras pamata idejas nopublicētas rakstā „The Transformation-Driven Architecture" [32].

TDA pamata struktūra attēlota attēlā 3.4. Centrā esošais mākonis reprezentē repozitorija metamodeli, kas loģiskajā līmenī sastāv no vairākām daļām (vairākiem metamodeļiem, starp kuru klasēm arī var pastāvēt asociācijas). Katrs no šiem metamodeļiem tiek saukts par saskarnes metamodeli un kopā ar atbilstošo saskarnes dzini veido vienu noslēgtu saskarni. 4. attēlā, piemēram, redzamas tādas saskarnes kā grafveida diagrammu saskarne, dialogu saskarne, datubāzu saskarne, vairāk-lietotāju saskarne un *Word* saskarne. Tāpat redzama arī tā sauktā galvenā saskarne, kas sastāv no sistēmas kodola metamodeļa un galvas dziņa, kas nodrošina sistēmas galvenā loga funkcionēšanu un pārējo sistēmas komponenšu pārraudzību. Katrs saskarnes dzinis spēj „saprast" un interpretēt savu metamodeli, nodrošinot vēlamo darbību (realizējot noteiktu saskarni). Darbības loģika tiek ielikta centrā esošajās modeļu transformācijās, kas spēj piekļūt visiem metamodeļiem un veidot un koriģēt to instances korektā veidā.



3.4. attēls. **Transformāciju vadīta arhitektūra**

Sadarbība starp abām galvenajām arhitektūras komponentēm – transformācijām un dziņiem – notiek caur repozitoriju. Precīzāk runājot – sistēmas kodola metamodelī ir divas speciālas klases – *Event* un *Command* – kas paredzētas informācijas nodošanai starp abām minētajām komponentēm attiecīgi katrā virzienā. Pārķerot kādu noteiktu lietotāja radītu darbību,

saskarnes dzinis izveido attiecīgo darbību reprezentējošās notikumu klases instanci (un, ja vajag, piesaista to ar saitēm pie citām metamodeļa klašu instancēm atkarībā no konteksta). Visas notikumu klases, kas atrodas dažādu saskarņu metamodeļos, ir apakšklases sistēmas kodola metamodeļa klasei *Event*, kurai vienlaicīgi atļauts saturēt tikai vienu instanci. Pēc instances izveidošanas tiek izsaukta transformācija šī notikuma apstrādei. Transformācija pēc sava darba beigām var nodot kādu ziņu arī atpakaļ dzinim, izveidojot instanci kādai klases *Command* apakšklasei.

Jāpiebilst, ka 4. Attēlā redzams tikai piemērs, no kādām komponentēm varētu sastāvēt sistēma, kas balstīta uz transformāciju vadītu arhitektūru. Reāli obligātās sastāvdaļas ir tikai repozitorijs un sistēmas kodola saskarne, bet pārējās saskarnes un transformācijas var tikt pievienotas sistēmai dinamiskā veidā.

TDA pirmajā versijā tika izvirzīti šādi pamata principi:

1) dati tiek glabāti modeļu repozitorijā ar fiksētu API (piemēram, EMF [8], JGraLab [34], Sesame [35], MiiRep [36] vai JR [37]);
2) repozitorija API jābūt izmantojamam no vienas vai vairākām augsta līmeņa programmēšanas valodām (piemēram, C++ vai Java), kurās tiktu realizēti dziņi;
3) modeļu transformācijas drīkstētu rakstīt jebkurā transformāciju valodā, taču transformāciju kompilatoram vai interpretatoram būtu jāizmanto tas pats repozitorija API, ko izmanto dziņi;
4) kad tiek izsaukta transformācija, tās uzvedība ir atkarīga tikai no datiem, kas atrodas repozitorijā;
5) repozitorija metamodelis izpildes laikā nemainās, visas izmaiņas notiek tikai instanču līmenī;
6) vienlaicīgi tiek izmantots tikai viens repozitorijs, sadarbība starp vairākiem repozitorijiem vai citām sadalītām datu glabātuvēm netiek aplūkota;
7) tikai viens modulis (transformācija vai dzinis) var vienlaicīgi piekļūt repozitorijam, paralēlā izpilde netiek aplūkota.

Atbilstoši šiem principiem tika realizēta pirmā transformāciju vadītas arhitektūras versija. Pašā TDA realizācijā šī darba autors nav piedalījies, taču ir piedalījies vienas tās komponentes – UNDO/REDO mehānisma – realizācijā. Šis mehānisms ir realizēts universālā veidā un integrēts iekš TDA [38, 39].

Vēlākās TDA realizācijās sākotnējie principi tika pārskatīti un laboti/pilnveidoti/papildināti, tādējādi iegūstot arvien plašākas grafisku rīku būves iespējas.

Šī darba mērķis nav gūt sīkāku ieskatu TDA ietvaros realizētajos dziņos. Detalizētāka informācija par galvenajiem dziņiem pieejama rakstos „A Graph Diagram Engine for the Transformation-Driven Architecture" [40, 41] un „A Dialog Engine Metamodel for the Transformation-Driven Architecture" [42]. Vēl sīkāk par vienu no TDA lietojumiem iespējams izlasīt rakstā „Domain Specific Languages for Business Process Management: a Case Study" [43].

3.3. GRAF izaicinājumi

Šajā apakšnodaļā tiks aplūkoti rīku būves platformas GRAF izaicinājumi, kuru risināšana sastāda lielāko daļu disertācijas. Piedāvātie risinājumi sīkāk tiks aplūkoti atlikušajās darba nodaļās.

Veicot iepriekšējās nodaļās izklāstītā kopsavilkumu, jāpatur prātā dažas galvenās atziņas par rīku būves platformu GRAF:

- platformas loģika ir ietverta metamodeļos un modeļos;
- platformas sastāvā eksistē daži dziņi;
- platformas sastāvā eksistē dažas modeļu transformācijas;
- platformai iespējams ērti pievienot savas modeļu transformācijas;
- izmantojot konfiguratoru, iespējams viegli izveidot rīku definējošā metamodeļa instances.

Iedziļinoties šajā kopsavilkumā, iespējams noformulēt galveno secinājumu par rīku būves platformu šajā stadijā:

Izmantojot rīku būves platformu GRAF, ar vienkāršiem līdzekļiem iespējams no nulles ātri un ērti izveidot jaunu vienkāršu grafisku rīku

Šis secinājums pirmajā brīdī izskatās ļoti labi, bet, kā zināms, ikvienu lietu iespējams uzlabot. Šī darba atlikušās daļas mērķis ir noskaidrot līdzšinējās GRAF nepilnības augstākminētajā formulējumā un piedāvāt risinājumus to novēršanai. Pašas nepilnības tiks formulētas šajā apakšnodaļā, bet to risinājumi – tālākajās darba nodaļās.

Ja iedziļināmies augstākminētajā secinājumā, pirmā lieta, kur būtu iespējams veikt kādus uzlabojumus, ir vārds „vienkāršu". Patiešām – ar esošajiem līdzekļiem ātri un ērti iespējams izveidot vienīgi salīdzinoši vienkāršus rīkus. Protams, rīka sarežģītība ir pakļauta lielā mērā

subjektīvam vērtējumam, taču iemesli, kādēļ viegli iespējams veidot tikai vienkāršus (subjektīvā izpratnē) rīkus, ir vairāk vai mazāk objektīvi.

Pirmkārt – lai jaunveidojamo rīku „apaudzētu" ar sarežģītākām iespējām nekā ir ietvertas konfiguratorā, nepieciešams rakstīt savas modeļu transformācijas. Kā jau minēts iepriekš, visa rīka programmēšana rīka pirmssākumos notikusi, izmantojot bāzes modeļu transformāciju valodu L0, kas tomēr ir vairāk piemērota citiem mērķiem, nevis būšanai par mērķa valodu ierindas programmētājiem [22]. Kā šīs problēmas risinājums darbā tiek piedāvāta augstāka līmeņa modeļu transformāciju valodu izstrāde un realizācija, līdz ar ko gala lietotājiem (modeļu transformāciju rakstītājiem) tiktu atvieglots transformāciju rakstīšanas process. Darba gaitā izstrādātas tā saucamās Lx saimes modeļu transformāciju valodas L0', L1, L2 un L3, kā ar to kompilatori uz valodu L0. Apvienojumā ar Agra Šostaka izstrādāto ļoti augsta līmeņa valodas MOLA kompilatoru uz valodu L3 [44] augstākminētā problēma tiek vēl ievērojamāk mīkstināta, atļaujot programmētājiem veidot arī grafiskas modeļu transformācijas valodā MOLA. Sīkāk par Lx saimes valodām un to realizāciju izklāstīts 4. nodaļā.

Otra problēma, kas traucē veidot sarežģītākus rīkus, slēpjas faktā, ka visa rīku būves platformas GRAF loģiskā darbība ietverta metamodeļos (specifikācijas daļa) un to modeļos (izpildāmā un daļēji arī specifikācijas daļa). Līdz ar to ikvienam, kas vēlas platformu papildināt ar savām transformācijām, nepieciešams līdz sīkākajām detaļām izprast attiecīgos metamodeļus, apzināties saistību starp tiem, zināt, kāda veida modeļi ir korekti un kāda veida nav utt. Korektas pietiekami sarežģītas modeļu transformācijas uzrakstīšana vispārīgā gadījumā var kļūt par pietiekami izaicinošu uzdevumu. Tai pat laikā der apzināties, ka ne visas lietas, kas ietvertas metamodeļos, būs nepieciešamas ikdienas programmētājam. Pat vēl vairāk – pieredze liecina, ka lielais vairums modeļu transformāciju var tikt uzrakstītas, ņemot vērā (apzinoties) tikai mazu daļu no metamodeļu saimniecības. Grūtākais uzdevums šajā sakarā ir saprast, kas konkrētajai izstrādāmajai modeļu transformācijai no šīs metamodeļu saimniecības ir vajadzīgs un kas nav. Lai atvieglotu šo uzdevumu, darbā tiek piedāvāts metamodeļa skata jēdziens. Ar skatu uz metamodeli tiek saprasts kāds cits metamodelis, kas pēc būtības ir vienkāršāks par doto metamodeli un kurā ietverti tikai konkrētam uzdevumam nepieciešamie artefakti. Izmantojot šāda veida skatus, modeļu transformāciju rakstītājam programmēšanas darbs tiktu ievērojami atvieglots. Sīkāk par metamodeļa skata jēdzienu un tā realizāciju izklāstīts 5. nodaļā.

Atrisinot abas augstākminētās problēmas, vienlaicīgi tiek arī paplašinātas GRAF iespējas – tagad ar vienkāršiem līdzekļiem iespējams no nulles ātri un ērti izveidot *patvaļīgas*

sarežģītības grafiskus rīkus. Šis formulējums jau ir krietni labāks par iepriekšējo, kas ļāva veidot tikai kaut kādā ziņā *vienkāršus* rīkus. Taču arī šeit vēl ir vieta uzlabojumiem.

Otra lieta, kuru šajā darbā tiek piedāvāts uzlabot, saistās ar vārdu „izveidot". Rīkus gan ir iespējams izveidot, bet nekas netiek teikts par to, kas ar šiem rīkiem notiek pēc tam – vai tos ērti iespējams arī uzturēt. Tā kā ar platformu GRAF iespējams veidot grafiskas problēmorientētas valodas, tad par uzturēšanas jautājumiem šeit būtu krietni jāpiedomā. Kas notiek ar izstrādātajām valodām, ja rīku būves platformā notiek kādas izmaiņas/uzlabojumi? Kas notiek ar šajās valodās izstrādātajiem modeļiem? Kas notiek ar šiem modeļiem, ja mainās pašas valodas specifikācija? Vai vecie modeļi būs savietojami ar jaunajām versijām? Šādi un līdzīgi jautājumi kļūst ļoti svarīgi, izstrādājot valodas un to modeļus platformā, kas vēl joprojām tiek attīstīta. Protams, versiju pārvaldība šāda veida sistēmās ir ārkārtīgi plaša tēma, un tās izpētei varētu tikt veltīta atsevišķa doktora disertācija, tāpēc šajā darbā tiks mēģināts aplūkot tikai daļu no šīs tēmas, kas būtiska tieši šajā kontekstā. Problēmas apakšgadījums, kas svarīgs rīku būves procesā un tiks aplūkots šajā darbā, ir šāds – kā, veicot izmaiņas kādā no izstrādāto problēmorientēto valodu specifikācijām vai pašas platformas specifikācijā, nodrošināt veco (ar iepriekšējo specifikācijas versiju izstrādāto) valodu un modeļu veiksmīgu darbošanos specifikāciju saderības ietvaros. Tā kā ar specifikāciju šeit jebkurā (gan platformas, gan arī problēmorientētas valodas) gadījumā tiek saprasts metamodelis, tad aplūkotais problēmas apakšgadījums arī tiek saistīts ar metamodeļa versiju maiņu. Kā risinājums tiek piedāvāta iespēja eksportēt ar iepriekšējo metamodeļa versiju izstrādāto valodu vai modeli un iespēju robežās importēt to jaunajā versijā. Lai arī pēc būtības abi gadījumi ir vienādi, tomēr tehniskā reprezentācija tiem ir dažāda – ja platformas specifikācija patiešām ir *īsts* metamodelis, tad problēmorientētas valodas specifikācija par metamodeli uzskatāma tikai loģiskā līmenī, bet reāli sistēmā tā tiek attēlota kā rīku definēšanas metamodeļa instanču kopa (skatīt nodaļu 3.1.). Tas rada virkni problēmu, ar kurām jātiek galā, lai konkrētu valodas modeļu eksports un imports būtu iespējami. Sīkāk par eksporta un importa funkcionalitāti izklāstīts 6. nodaļā.

Kā jau minēts iepriekš, platforma GRAF atrodas pastāvīgas attīstības stadijā, un darbs pie tās turpinās nepārtraukti. Jaunākās idejas saistībā ar platformas attīstību saistāmas ar jauna veida rīku būves iespēju paplašināšanas mehānismu, kas zināmā mērā atgādina UML stereotipu mehānismu, tikai tas papildināts ar vairākām izteiksmīgām iespējām. Tās gan šajā darbā netiks sīkāk aprakstītas, tomēr ar tām iespējams iepazīties ar šī darba autora līdzdalību tapušās publikācijās [45] un [46].

4. Metamodeļu transformāciju valodu saime Lx

Kā jau minēts iepriekš nodaļā 3.3, nepieciešamība pēc augstāka līmeņa modeļu transformāciju valodas rīka loģiku definējošo transformāciju izstrādē pamatojama ar vēlmi salīdzinoši vienkāršākā veidā veidot arī sarežģītākus rīkus, kuru radīšana neaprobežotos vien ar to definēšošas instances radīšanu rīku definēšanas metamodelī, bet kuru darbībai būtu nepieciešams rakstīt vēl papildus sīktransformācijas. Rīku būves platformas GRAF pirmsākumos šim nolūkam tika izmantota modeļu transformāciju valoda L0, kas vairāk līdzinās asemblera tipa valodai modeļu transformāciju valodu pasaulē – tā satur tikai ļoti vienkāršas komandas nelielā skaitā un ir vairāk piemērota kalpošanai par mērķa valodu citu valodu kompilācijas procesā, nekā par reāli izmantojamu darba valodu programmu veidošanā.

Valodā L0 trūkst dažas ļoti būtiskas valodas konstrukcijas, kuru eksistence spētu ievērojami atvieglot programmētāja darbu. Divas no šāda veida konstrukcijām ir tradicionālajās programmēšanas valodās ierastās zarošanās un cikla konstrukcijas, bet trešā – tieši modeļu transformāciju valodām raksturīgā meklēšanas iespēja pēc atbilstības kādam noteiktam šablonam (*pattern matching*). Tāpat valodā L0 nav nodrošināts atbalsts patvaļīgu aritmētisku izteiksmju lietošanai, kas atsevišķās situācijās var radīt lielas neērtības.

Šajā nodaļā aplūkota metamodeļu transformāciju valodu saime Lx, kuru autors izstrādājis un realizējis darba ietvaros. Jāpiezīmē, ka, lai arī oficiāli visas šai saimē ietilpstošās valodas saucamas par metamodeļu transformāciju valodām, jo ar to palīdzību var tikt transformēti ne tikai modeļi, bet arī metmamodeļi, tomēr darbā īsrakstīšanas nolūkos tās reizēm tiks sauktas vienkārši par modeļu transformāciju valodām. Tas pamatojams arī ar to, ka metamodeļu transformācijas iespēja, lai arī klāt esoša, pēc būtības nav aplūkota un citās saimes valodās, salīdzinot ar L0, nav uzlabota vai jebkādā citā veidā mainīta.

Faktiski, ņemot vērā visus augstākminētos valodas L0 trūkumus, pietiktu ar vienas jaunas valodas izstrādi, kurā šie trūkumi tiktu novērsti un kurai tad tiktu uzbūvēts kompilators uz valodu L0. Šāds arī bija šīs darba daļas galējais mērķis, kā rezultātā tika radīta valoda L3. Taču, lai padarītu kompilatora izstrādes procesu vienkāršāku un caurspīdīgāku, pa ceļam tika izveidotas vēl vairākas starpvalodas, katrā no kurām tika ieviesta kāda jauna valodas konstrukcija vai atbalsts iespējai, kas iepriekšējā valodā vēl nebija. Tā vispirms tika izstrādāta valoda L1, kurā tika ieviests šablonu atbilstības mehānisms, kāda nebija valodā L0. Pēc valodas L1 kompilatora uz valodu L0 izstrādes tika veidota valoda L2, kuras kompilatoru uz L1 jau varēja rakstīt valodā L1. Valodā L2 nāca klāt speciāla veida cikla konstrukcija.

Visbeidzot valodas L3, kurā nāca klāt zarošanās konstrukcija, kompilatoru uz valodu L2, jau bija iespēja veidot valodā L2. Rezultātā jāsecina, ka valodu saimes Lx izstrāde notikusi pēc sāknēšanas (*bootstrapping* [47]) metodes – katras nākamās valodas realizācija balstīta uz iepriekšējo izstrādāto valodu. Pēc visu minēto valodas konstrukciju ieviešanas augstākajās Lx valodās tika nolemts šajās valodās ieviest arī atbalstu patvaļīga garuma aritmētiskām izteiksmēm. Tā kā šāda veida atbalsts būtu noderīgs visās no valodām L1, L2 un L3, tad augstākminētajā valodu Lx ķēdē starp valodām L0 un L1 tika izveidota starpvaloda L0', kurā realizētas aritmētiskās izteiksmes.

Nodaļā 4.1. īsumā aplūkota modeļu transformāciju valoda L0 un tajā esošās komandas. Tālāk aplūkotas pārējās Lx saimes valodas, piedāvājot to metamodeļus un komandu pieraksta sintaksi tekstuālā formā. Īpaši aplūkota valoda L1 un vērtēta tās izteiksmīguma spēja. Nodaļa pabeigta ar dažu tehnisku realizācijas īpatnību aprakstu, kā arī ar valodu Lx (precīzāk – tieši augstākās šīs valodu saimes valodas L3) izmantošanas piemēriem.

4.1. Modeļu transformāciju valoda L0

L0 ir tekstuāla modeļu transformāciju valoda. Šajā valodā ir pieejamas vienkāršas komandas darbam ar patvaļīgi fiksēta metamodeļa instancēm, piemēram, jaunas instances radīšanas komanda, instances dzēšanas komanda, instances atribūtu nolasīšanas un uzstādīšanas komandas, saišu starp instancēm radīšana un dzēšana, instanču meklēšana, saišu pārbaude, kā arī komandas darbam ar vadības nodošanu, piemēram, daudzās programmēšanas valodās tradicionālās „goto" un „label" komandas, kā arī, tā sauktie, „else" zari vairākām modeļa instanču apstrādes komandām. Pastāvīgo datu glabāšanai valodā L0 iespējams izmantot vienu no vairākiem repozitorijiem, uz kuriem var tikt kompilētas valodas L0 programmas – kādu no Latvijas Universitātes Matemātikas un informātikas Institūtā izstrādātajiem pamatatmiņā esošajiem (*in-memory*) repozitorijiem MII_REP [36] vai JR [37], vai arī uz ECLIPSE platformas [48] balstīto repozitoriju EMF [8].

Šobrīd transformāciju valodai L0 pieejami efektīvi kompilatori uz valodām C++ un Java, tas ir, programmu, kas uzrakstīta valodā L0, ar šī kompilatora palīdzību iespējams pārvērst par C++ vai Java kodu, kuru tālāk iespējams nokompilēt uz MS Windows vidē izpildāmu datni (konkrētāk – uz datni ar paplašinājumu „.dll"). Pēc tam šādu izpildāmu datni iespējams izpildīt kā dotās programmas rezultātu, ņemot par pamatu kādu lietotāja definētu metamodeli.

Sīkāks transformāciju valodas L0 apraksts pieejams Sergeja Rikačova doktora disertācijā [49], bet, lai padarītu šo darbu lasāmu neatkarīgi no augstāk minētā darba, nākamajos šīs

apakšnodaļas punktos īsumā izskaidrotas valodas L0 komandas tradicionālā formā, kā arī dots šīs valodas metamodelis.

4.1.1. Transformāciju valodas L0 komandas

Bāzes modeļu transformāciju valoda L0 ir no procedūrām un funkcijām sastāvoša valoda, kas sastāv no šādām komandām (kas var atrasties procedūras vai funkcijas ķermenī) [22]:

1) **`call`** `<subProgName>` **`(<`**`actualParamList>`**`)`** – izsauc procedūru ar dotajiem parametriem.
2) **`return`** – atgriež vadību izsaucošajai procedūrai/funkcijai.
3) **`return`** `<identifier>` – atgriež `<identifier>` vērtību izsaucošajai procedūrai/funkcijai.
4) **`first`** `<pointerName>` **`:`** `<className>` [**`else`** `<labelName>`] – pozicionē `<pointerName>` uz patvaļīgu klases `<className>` objektu. Ja šai klasei nepieder neviens objekts, vadība tiek nodota iezīmei `<labelName>`.
5) **first** <pointerName1> **:** <className> **from** <pointerName2> **by** <roleName> [**else** <labelName>] – pozicionē <pointerName1> uz tādu patvaļīgu klases <className> objektu, kas no <pointerName2> ir sasniedzams ar saites <roleName> palīdzību. Ja šāds objekts neeksistē, vadība tiek nodota iezīmei <labelName>. Pēc šīs komandas izpildes <pointerName1> vērtību kopa tiek ierobežota ar tām klases <className> instancēm, kuras no <pointerName2> ir sasniedzamas ar lomas <roleName> palīdzību.
6) **`next`** `<pointerName>` [**`else`** `<labelName>`] – pozicionē `<pointerName>` uz nākamo objektu, kas apmierina nosacījumus, kas izvirzīti attiecīgajā „first” komandā (iepriekšējā „first” komandā, kurai ir norādīts šāds pat `<pointerName>`) un kas vēl ar „first” vai „next” komandu nav apmeklēts. Ja šāds objekts neeksistē, vadība tiek nodota iezīmei `<labelName>`.
7) **goto** <labelName> – nodod vadību iezīmei <labelName>.
8) **label** <labelName> – definē iezīmi <labelName>.
9) **addObj** <pointerName> **:** <className> – izveido jaunu klases <className> objektu.
10) **addLink** <pointerName1> **.** <roleName> **.** <pointerName2> – izveido jaunu saiti starp objektiem <pointerName1> un <pointerName2> ar lomas vārdu <roleName> objekta <pointerName2> galā.
11) **deleteObj** <pointerName> – izdzēš objektu, uz kuru norāda <pointerName>.

12) **deleteLink** <pointerName1> **.** <roleName> **.** <pointerName2> – izdzēš saiti starp objektiem <pointerName1> un <pointerName2> ar lomas vārdu <roleName> objekta <pointerName2> galā.
13) **setPointer** <pointerName1> = <pointerName2> – pozicionē <pointerName1> uz objektu, uz kuru ir pozicionēts <pointerName2>.
14) **setPointerF** <pointerName> **=** <funcName> **(**<actualParamList>**)** – pozicionē <pointerName> uz objektu, ko atgriež funkcija <funcName> ar dotajiem parametriem.
15) **`setVar <varName> = <binExpr>`** – uzstāda mainīgā `<varName>` vērtību vienādu ar bināras izteiksmes `<binExpr>` vērtību.
16) **`setVarF <varName> = <funcName> (<actualParamList>)`** – uzstāda mainīgā `<varName>` vērtību vienādu ar funkcijas `<funcName>` ar dotajiem parametriem atgriezto vērtību.
17) **`setAttr <pointerName> . <attrName> = <binExpr>`** – uzstāda objekta, uz kuru ir pozicionēts `<pointerName>`, atribūta `<attrName>` vērtību vienādu ar bināras izteiksmes `<binExpr>` vērtību.
18) **`type <pointerName> == <className> [ else <labelName> ]`** – ja `<pointerName>` ir pozicionēts uz klases `<className>` objektu, vadība tiek nodota nākamajai komandai, citādi vadība tiek nodota iezīmei `<labelName>`. Vienādības vietā drīkst būt arī nevienādība („!=”), kas nozīmē pretēju darbību.
19) **`var <varName> == <binExpr> [ else <labelName> ]`** – ja mainīgā vērtības salīdzinājums ar bināras izteiksmes vērtību ir spēkā, vadība tiek nodota nākamajai komandai, citādi vadība tiek nodota iezīmei `<labelName>`. Vienādības vietā drīkst būt arī cita veida salīdzināšanas operatori („<”, „<=”, „>”, „>=” vai „!=”).
20) **`pointer <pointerName1> == <pointerName2> [ else <labelName> ]`** – ja `<pointerName1>` un `<pointerName2>` ir pozicionēti uz vienu un to pašu objektu, vadība tiek nodota nākamajai komandai, citādi vadība tiek nodota iezīmei `<labelName>`. Vienādības vietā drīkst būt arī nevienādība („!=”), kas nozīmē pretēju darbību.
21) **`attr <pointerName> . <attrName> == <binExpr> [ else <labelName> ]`** – ja objekta, uz kuru pozicionēts `<pointerName>`, atribūta `<attrName>` vērtības salīdzinājums ar binārās izteiksmes vērtību ir spēkā, vadība tiek nodota nākamajai komandai, citādi vadība tiek nodota iezīmei `<labelName>`.

Vienādības vietā drīkst būt arī cita veida salīdzināšanas operatori („<”, „<=”, „>”, „>=” vai „!=”).

22) **link** <pointerName1> **.** <roleName> **.** <pointerName2> [**else** <labelName>] – ja starp objektiem <pointerName1> un <pointerName2> ir saite ar lomas vārdu <roleName> objekta <pointerName2> galā, tad vadība tiek nodota nākamajai komandai, citādi vadība tiek nodota iezīmei <labelName>.
23) **nolink** <pointerName1> **.** <roleName> **.** <pointerName2> [**else** <labelName>] – ja starp objektiem <pointerName1> un <pointerName2> nav saites ar lomas vārdu <roleName> objekta <pointerName2> galā, tad vadība tiek nodota nākamajai komandai, citādi vadība tiek nodota iezīmei <labelName>.
24) **DEBUG_ON –** ieslēdz atkļūdošanas režīmu.
25) **DEBUG_OFF** – izslēdz atkļūdošanas režīmu.

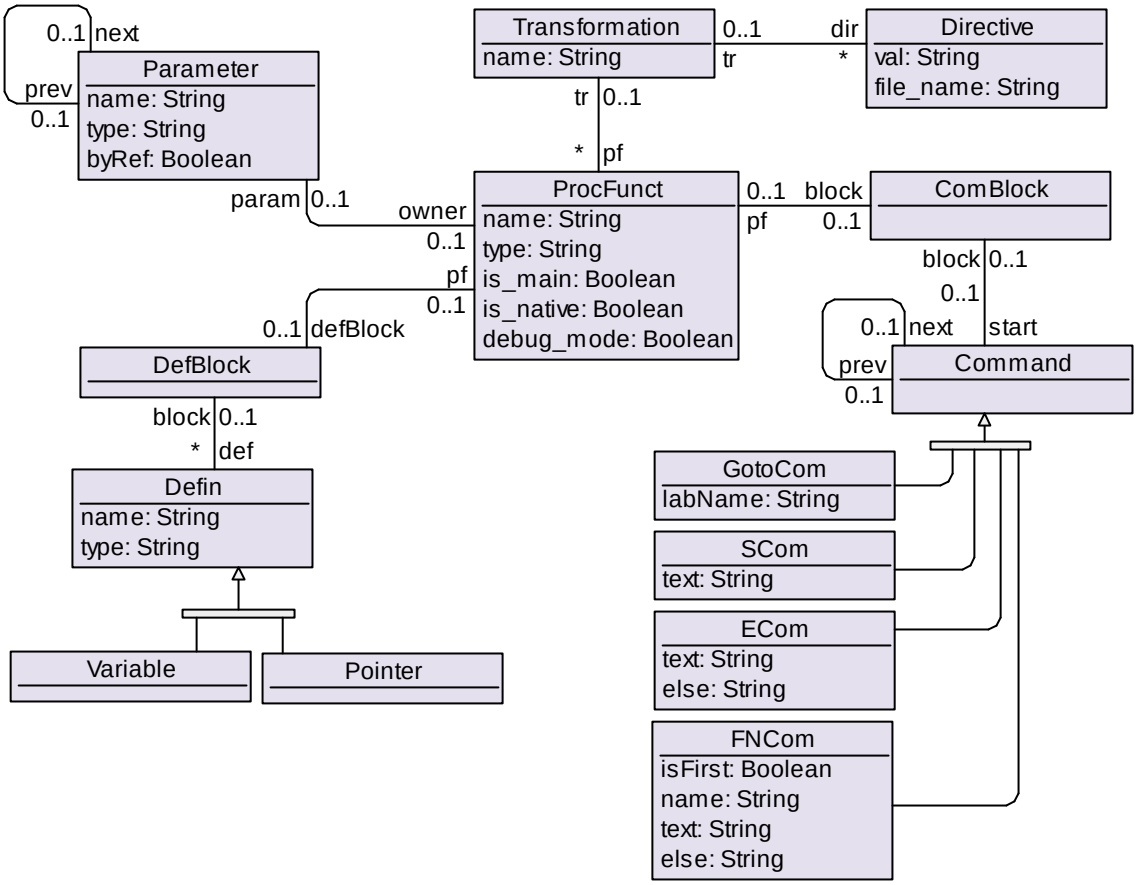
Transformāciju valoda L0 ir stingri tipizēta valoda, tātad tajā nepieciešams katru izmantoto mainīgo deklarēt. Mainīgo deklarēšana katrā procedūrā un funkcijā notiek no procedūras/funkcijas ķermeņa atdalītā atsevišķā mainīgo deklarācijas blokā atbilstoši šādai sintaksei:

1) **var** <varName> **:** <typeName> – primitīvā datu tipa (*Integer*, *Real*, *String* vai *Boolean*) mainīgā deklarēšana;
2) **pointer** <pointerName> **:** <className> – norādes uz klases <className> objektu deklarēšana.

Valodā L0 (tāpat kā jebkurā citā modeļu transformāciju valodā) rakstītu programmu iespējams izpildīt, ņemot par pamatu modeli, kurš ir šīs programmas definīcijā izmantotā avota metamodeļa instance. Tātad, lai vispār varētu sākt rakstīt L0 programmu, vispirms nepieciešams nodefinēt metamodeli, kurš tiks ņemts par pamatu, tas ir, nepieciešams, lai šis metamodelis atrastos tajā repozitorijā, ar kuru strādās transformāciju valoda L0. Kādā veidā tad šādu metamodeli iespējams repozitorijā izveidot? Viens no variantiem ir izmantot kādus esošus rīkus, kas piedāvā iespēju uzzīmēt metamodeļa elementus un ielikt šo metamodeli repozitorijā, taču šajā gadījumā papildus valodas L0 kompilatoram nepieciešams arī šis rīks – metamodeļa veidotājs. Tāpēc arī pašā valodā L0 pieejamas komandas, ar kuru palīdzību iespējams izveidot un noglabāt repozitorijā nepieciešamo metamodeli. Taču, tā kā šīs L0 komandas izmantojamas atsevišķi no standarta L0 komandām, kas paredzētas modeļu transformēšanas procesu apstrādei, tad šajā darbā tās netiks aplūkotas.

4.1.2. Transformāciju valodas L0 metamodelis

Tā kā tālākajās apakšnodaļās aprakstīto transformāciju valodu metamodeļi tiks balstīti uz valodas L0 metamodeli, tad šajā brīdī nepieciešams nedaudz vairāk izskaidrot valodas L0 metamodeli un valodā L0 rakstītas programmas vietu tajā (skatīt attēlu 4.1.).



4.1. attēls. **Transformāciju valodas L0 metamodelis**

Par valodas L0 metamodeļa galveno klasi uzskatāma klase „Transformation”, kuras instances ir L0 programmas jeb transformācijas. Transformāciju raksturo tās vārds (atribūts „name” ar tipu *String*) un tai var būt piekārtotas direktīvas (klases „Directive” instances), kuras tiek raksturotas ar direktīvas nosaukumu (atribūts „val” ar tipu *String*) un datnes vārdu (atribūts „file_name” ar tipu *String*) atbilstoši valodas L0 semantikai. Tā kā tam ar šo darbu tieša sakara nav, tad tas, kas ir direktīva, šeit sīkāk izklāstīts netiks. Transformācijas īpašību sastāvēt no procedūrām un funkcijām atspoguļo saite no klases „Transformation” uz klasi „ProcFunct”, kas satur procedūru un funkciju instances, katra no kurām tiek raksturota ar tās vārdu (atribūts „name” ar tipu *String*), tipu (atribūts „type” ar tipu *String* – tikai funkcijām), kā arī ar trim *Boolean* tipa atribūtiem – „is_main” (norāda, vai šī procedūra/funkcija ir galvenā dotajā transformācijā), „is_native” (norāda, vai šī procedūra/funkcija ir realizēta pašā transformācijā vai ārpus tās) un „debug_mode” (norāda, vai šai procedūrai/funkcijai ir uzlikts

ārējs atkļūdošanas režīms). Savukārt, procedūrai/funkcijai piederošie parametri tiek definēti kā klases „Parameter" instances – uz pirmo parametru dotajā procedūrā/funkcijā norāda saite „param", bet uz katru nākamo parametru no iepriekšējā parametra ir saite „next". Parametru raksturo tā vārds (atribūts „name" ar tipu *String*), tā tips (atribūts „type" ar tipu *String*), kā arī fakts, vai šis parametrs attiecīgajā procedūrā/funkcijā tiek padots pēc vērtības vai pēc norādes (atribūts „byRef" ar tipu *Boolean*).

Katrai procedūrai/funkcijai ir savs mainīgo deklarācijas bloks (klases „DefBlock" instance), kas satur tajā deklarētos primitīvo datu tipu mainīgos (klases „Variable" instances) un klašu norādes (klases „Pointer" instances), kas tiek identificēti ar vārdu (atribūts „name" ar tipu *String*) un tipu (atribūts „type" ar tipu *String*).

Bez mainīgo deklarācijas bloka katrai procedūrai/funkcijai ir arī savs ķermenis (klases „ComBlock" instance), kas satur visas dotās procedūras/funkcijas komandas (klases „Command" instances). Komandu secība tiek panākta ar asociācijas „start" no klases „ComBlock" uz klasi „Command" un saites „next" starp klases „Command" instancēm palīdzību. Valodā L0 iespējamas četru tipu komandas:

1) klases „GotoCom" instances ir valodas L0 vadības nodošanas komandas („goto"), un tās raksturo tām piekārtotās iezīmes vārds (atribūts „labName" ar tipu *String*);
2) klases „FNCom" instances ir abas valodas L0 objektu meklēšanas komandas („first" un „next"), un tās raksturo objekta vārds (atribūts „name" ar tipu *String*), komandas teksts (atribūts „text" ar tipu *String*), „else" zaram piekārtotās iezīmes vārds (atribūts „else" ar tipu *String*), kā arī tās tips, kas norāda, vai šī ir „first" vai „next" komanda (atribūts „isFirst" ar tipu *Boolean*); ar komandas tekstu šeit saprot komandas tekstu, sākot ar klases vārdu un beidzot ar „from ... by ..." daļu, ja tāda eksistē, bet neiekļaujot komandu noslēdzošo semikolu;
3) klases „ECom" instances ir valodas L0 pārbaudes tipa komandas – „type", „var", „pointer", „attr", „link" un „noLink" – , un tās raksturo komandas teksts (atribūts „text" ar tipu *String*) un „else" zaram piekārtotās iezīmes vārds (atribūts „else" ar tipu *String*); ar komandas tekstu šeit saprot visu komandas tekstu līdz atslēgas vārdam „else" (ja tāds eksistē), ieskaitot komandas nosaukumu un pārējos komandas parametrus, bet neieskaitot komandu noslēdzošo semikolu;
4) klases „SCom" instances ir pārējās valodas L0 komandas – „call", „return", „label", „addObj", „addLink", „deleteObj", „deleteLink", „setPointer", „setPointerF", „setVar", „setVarF", „setAttr", „DEBUG_ON" un „DEBUG_OFF" –, un tās raksturo komandas teksts (atribūts „text" ar tipu *String*). Šajā gadījumā ar komandas tekstu

saprot pilnu komandas tekstu, ieskaitot komandas nosaukumu un pārējos komandas parametrus, bet neieskaitot komandu noslēdzošo semikolu.

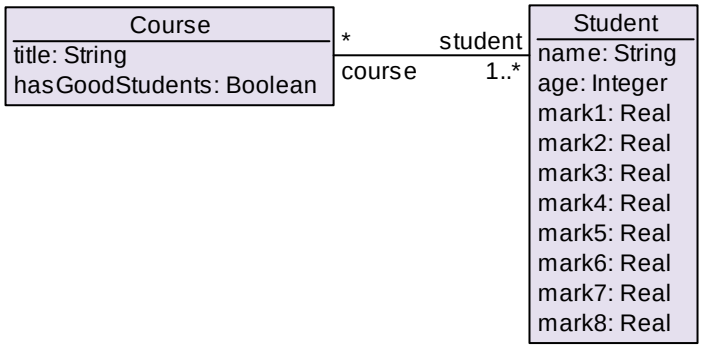
Tātad visas valodas L0 komandas tiek sadalītas pa četru veidu komandām – vadības nodošanas komanda, objektu meklēšanas komandas, pārbaudes komandas un pārējās komandas. Pie pirmajām divām grupām pieder tikai attiecīgi viena un divas valodas L0 komandas, bet attiecībā uz pēdējām divām grupām varētu veikt arī vēl tālāku sadalījumu apakštipos, izdalot savu klasi katrai no L0 komandām. Taču tas netiek darīts, pamatojoties ar to, ka šāda sadalījuma ieviešana nenestu nekāda veida reālu labumu. Proti, valodas L0 kompilācijā nekādas darbības ar L0 metamodeli darītas netiek, tiek analizēta vienīgi L0 programmas tekstuālā forma, līdz ar ko vienīgā L0 metamodeļa loma ir šī metamodeļa izmantošana tālāko Lx saimes valodu kompilācijā, kas aprakstīta šīs nodaļas nākamajās apakšnodaļās. Bet šajā kompilācijā nav vajadzības atpazīt katru konkrēto valodas L0 komandu, pietiek atpazīt vienīgi katru no šīm četrām augstāk minētajām komandu klasēm. Tādējādi transformāciju valodas L0 metamodelis netiek lieki sarežģīts.

4.1.3. Modeļu transformācijas piemērs

Aplūkosim samērā vienkāršu modeļu transformācijas piemēru, kas tiks izmantots arī tālākajās šīs nodaļas apakšnodaļās. Piemērs sastāvēs no teksta, kurā būs parastā valodā izskaidrots uzdevums, no metamodeļa, kas šajā uzdevumā jāņem par pamatu, kā arī no valodā L0 rakstītas programmas, kas šo uzdevumu paveic.

Pieņemsim, ka dots metamodelis, kas sastāv no divām klasēm – „Student” un „Course” (skatīt attēlu 4.2.). Klases „Student” objekti ir datorzinātņu maģistrantūras studenti, un tie tiek raksturoti ar vārdu (atribūts „name” ar tipu *String*), vecumu (atribūts „age” ar tipu *Integer*) un vidējām atzīmēm bakalauru studiju laikā katrā no astoņiem semestriem (atribūti „mark1” līdz „mark8” ar tipu *Real*). Vienkāršības labad tiek pieņemts, ka visi datorzinātņu maģistrantūras studenti pirms tam ir mācījušies astoņus semestrus garā bakalauru studiju programmā. Klases „Course” objekti ir datorzinātņu maģistrantūras studiju kursi, un tie tiek raksturoti ar nosaukumu (atribūts „title” ar tipu *String*) un faktu par to, vai tiesa, ka vidējā bakalauru studiju atzīme pa visiem pilngadīgajiem studentiem, kas apgūst šo kursu, ir vismaz 8 balles (atribūts „hasGoodStudents” ar tipu *Boolean*). Vienkāršības labad tiek pieņemts, ka atribūts „title” dotās klases ietvaros ir unikāls, tas ir, neeksistē divi kursi ar vienādu nosaukumu. Starp abām klasēm eksistē asociācija, kas norāda, kuri studenti kādus kursus ir izvēlējušies apgūt. Jāatzīst, ka šāds metamodelis varbūt nav labākais risinājums dotajam problēmu apgabalam, ja

skatās tieši no modelēšanas viedokļa, taču ar šāda piemēra palīdzību turpmāk ļoti ērti būs iespējams demonstrēt transformāciju valodu Lx iespējas.



4.2. attēls. **Kompleksajā piemērā izmantotais metamodelis**

Uzdevums, kas jāveic, ņemot vērā šo metamodeli, ir šāds – ierakstīt pareizu atribūta „hasGoodStudents" vērtību kursam ar nosaukumu „Operating Systems". Tātad tas nozīmē izrēķināt visu pilngadīgo studentu, kas apgūst šo kursu, vidējo atzīmi bakalauru studiju laikā un salīdzināt iegūto rezultātu ar skaitli 8 – ja iegūtais rezultāts ir zemāks par 8, tad piešķirt minētajam atribūtam vērtību *false*, pretējā gadījumā piešķirt tam vērtību *true*.

Tālāk šajā apakšnodaļā dots transformāciju valodā L0 rakstītas programmas piemērs, kas atrisina augstāk minēto uzdevumu.

```
transformation example;
   main procedure main();
      pointer c:Course;
      pointer s:Student;
      var x:Real;
      var avg:Real;
      var count:Integer;
   begin;
      first c:Course else endOfProg;
      label startFinding;
      attr c.title==”Operating Systems” else getNextCourse;
      goto courseFound;
      label getNextCourse;
      next c else endOfProg;
      goto startFinding;
      label courseFound;
      setVar x=0;
      setVar count=0;
```

```
        first s:Student from c by student else noMoreStudents;
        label startCounting;
        attr s.age>=18 else getNextStudent;
        setVar count=count+1;
        setVar avg=s.mark1;
        setVar avg=avg+s.mark2;
        setVar avg=avg+s.mark3;
        setVar avg=avg+s.mark4;
        setVar avg=avg+s.mark5;
        setVar avg=avg+s.mark6;
        setVar avg=avg+s.mark7;
        setVar avg=avg+s.mark8;
        setVar avg=avg/8;
        setVar x=x+avg;
        label getNextStudent;
        next s else noMoreStudents;
        goto startCounting;
        label noMoreStudents;
        var count>0 else writeGood;
        setVar x=x/count;
        var x<8 else writeGood;
        setAttr c.hasGoodStudents=false;
        goto endOfProg;
        label writeGood;
        setAttr c.hasGoodStudents=true;
        label endOfProg;
    end;
endTransformation;
```

4.2. Modeļu transformāciju valodas Lx

Ar transformāciju valodām Lx jeb, tā saukto, Lx valodu saimi šajā darbā tiek saprasta transformāciju valoda L0, kuras vispārējs apraksts dots iepriekšējā nodaļā, un tai radniecīgās transformāciju valodas L0', L1, L2 un L3. Katra nākamā šīs saimes valoda būvēta uz iepriekšējās valodas bāzes, pievienojot kaut kādas papildus valodas konstrukcijas. Šajā nodaļā autors iepazīstina lasītāju ar transformāciju valodai L0 sekojošām transformāciju valodām L0', L1, L2 un L3, kuru izstrādē autors ir piedalījies, un dod detalizētu šo valodu sintakses un semantikas aprakstu.

4.2.1. Valodu vispārēji apraksti

L0' vispārējs apraksts

Modeļu transformāciju valoda L0' (lasīt – „*L0 prim*") ir veidota, ņemot par pamatu transformāciju valodu L0 un papildinot to ar attīstītākām aritmētisko izteiksmju veidošanas iespējām.

Aritmētiskās izteiksmes šajās transformāciju valodās lietojamas gadījumos, kad nepieciešams vai nu piešķirt vērtību mainīgajam vai kādas klases objekta atribūtam, vai arī salīdzināt kāda mainīgā vai atribūta vērtību ar kādas izteiksmes vērtību. Valodā L0 ir atļauts veidot vienīgi unāras un bināras aritmētiskās izteiksmes, tas ir, tādas izteiksmes, kas sastāv no viena vai diviem argumentiem un no ne vairāk kā viena aritmētiskā operatora (saskaitīšanas, atņemšanas, reizināšanas vai dalīšanas). Pie tam ir jāšķiro gadījumi, kad drīkst lietot bināras aritmētiskas izteiksmes (to drīkst darīt, piemēram, divu veselu skaitļu konstanšu saskaitīšanā) un kad pat bināru aritmētisku izteiksmju lietošana nav atļauta (nekāda veida bināra izteiksme nav atļauta, piemēram, ja viens no operandiem ir funkcijas izsaukums). Līdz ar to gadījumos, kad pēc būtības nepieciešams lietot garākas aritmētiskās izteiksmes, tās jāsadala vairākās binārās izteiksmēs, nepieciešamības gadījumā lietojot palīg-mainīgos, un pēc tam jāizpilda viena binārā izteiksme pēc otras. Tas var būt ļoti apgrūtinoši.

Valodā L0' atļauts lietot patvaļīga garuma aritmētiskas izteiksmes. Īsumā tas nozīmē, ka var izmantot visus četrus aritmētiskos operatorus un tradicionālās iekavas („(" un „)") un ar to palīdzību veidot aritmētiskas izteiksmes patvaļīgā garumā. Kā operandi šajās izteiksmēs var tikt lietoti mainīgie, konstantes, objektu atribūti, kā arī funkcijas, ievērojot datu tipu saderību.

Detalizētāka operatoru pielietošana konkrētiem datu tipiem un atbilstošie rezultāta datu tipi attēloti taulbā 4.1. tabulā. Valodas L0' aritmētiskās izteiksmēs drīkst katram no četriem aritmētiskajiem operatoriem pielietot to un tikai to datu tipu operandus, kādi norādīti šajā tabulā. Tātad *String* tipa objektiem (mainīgajiem, konstantēm, atribūtiem un funkcijām) drīkst pielietot vienīgi saskaitīšanas operatoru, kas semantiski nozīmē simbolu virkņu sakabināšanu, savukārt, *Integer* un *Real* tipu objektiem bez saskaitīšanas iespējama arī atņemšana, reizināšana un dalīšana.

4.1. tabula. **Aritmētisko operatoru pielietojums datu tipiem**

Operators	Kreisās puses operands	Labās puses operands	Rezultāts
+	*Integer*	*Integer*	*Integer*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*
	String	*String*	*String*
-	*Integer*	*Integer*	*Integer*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*
*	*Integer*	*Integer*	*Integer*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*
/	*Integer*	*Integer*	*Real*
	Integer	*Real*	*Real*
	Real	*Integer*	*Real*
	Real	*Real*	*Real*

Valodā L0’ tiek ņemtas vērā tradicionālās operatoru izpildes prioritātes. Tās ir šādas (secībā no augstākās uz zemāko):

1) funkciju izsaukums;
2) iekavas;
3) reizināšana un dalīšana;
4) saskaitīšana un atņemšana.

Tātad, kā redzams, prioritātes šeit ir analoģiskas izteiksmju izpildes prioritātēm aritmētikā, taču bez tām ir nākusi klāt vēl viena prioritāte – funkcijas izsaukumi. Tas nepieciešams tādēļ, ka valodā L0 funkciju izsaukumiem nedrīkst pielietot nekādu aritmētisko operatoru, tādēļ nepieciešams vispirms izsaukt funkciju un tās atgriezto vērtību piešķirt kādam atbilstošā tipa mainīgajam, ko pēc tam var normālā veidā izmantot tālāk aritmētiskajā izteiksmē, ievērojot normālās aritmētiskās operatoru izpildes prioritātes.

L1 vispārējs apraksts

Transformāciju valoda L1 veidota, balstoties uz valodu L0’ un paplašinot to ar šablonu atpazīšanas (*pattern matching*) iespēju. Tātad valodā L1 atļautas visas tās pašas konstrukcijas, kas valodā L0’, kā arī vēl papildus pievienota iespēja meklēt instances metamodelī pēc dažāda veida šabloniem. Kas tad īsti šeit tiek saprasts ar šablonu? Meklēt kādas klases instances pēc šablona valodā L1 nozīmē iespēju norādīt dotajai klasei kādus raksturlielumus, kuriem būtu jāizpildās katram konkrētam objektam, lai tas piederētu kādam šablonam. Atšķirībā no valodas L0’ (un līdz ar to arī valodas L0), kur nebija iespējama nekāda veida klases objektu meklēšana, izņemot visu kādas klases objektu apstaigāšanu pēc kārtas nedefinētā secībā, transformāciju valodā L1 iespējams meklēt objektus, kuriem piemīt kaut kādas īpašības, tas ir, meklēt tos pēc kaut kādiem lietotāja definētiem kritērijiem. Turklāt, dažādos veidos kombinējot šablonus, kas attiecas uz kādas atsevišķas klases objektiem, iespējams izveidot jau sarežģītākus šablonus, kas attiecināmi uz kaut kādu instanču diagrammas fragmentu, kurā bez klašu objektiem un to atribūtiem varētu būt iekļautas arī saites starp šiem objektiem.

L2 vispārējs apraksts

Transformāciju valoda L2 veidota uz valodas L1 bāzes, pieliekot klāt cikla veidošanas iespēju. Valodā L2 iespējams veidot ciklu vai nu pa visiem kādas klases objektiem, vai arī pa visiem kādas klases objektiem, kas atbilst kādiem lietotāja definētiem kritērijiem, kas definēti, izmantojot šablonu definēšanas iespēju.

Valodā L1 jebkāda veida ciklus bija iespējams veidot vienīgi, izmantojot vadības nodošanas komandas „goto” un iezīmes, kas definētas ar komandas „label” palīdzību. Tas, protams, sarežģītākos gadījumos varēja radīt diezgan nepārskatāmu kodu, it sevišķi, ja bija nepieciešamība veidot iekļautos ciklus. Tāpat tas radīja arī zināmas programmēšanas grūtības. Transformāciju valodā L2 ir atļauts izmantot, tā saukto, „foreach” ciklu, kas nozīmē, ka, izmantojot iteratoru, iespējams pēc kārtas pārlasīt visas instances no kaut kādas instanču kopas. Instanču kopa, kuru iespējams pārlasīt, var būt divu veidu:

1) kādas noteiktas klases visas instances;
2) tās kādas noteiktas klases instances, kas atbilst dotam (lietotāja definētam) šablonam.

L3 vispārējs apraksts

Transformāciju valoda L3 būvēta uz transformāciju valodas L2 bāzes, pieliekot klāt zarošanās konstrukciju. Valodā L2 nebija iespējama nekāda veida zarošanās atkarībā no kādiem lietotāja

definētiem nosacījumiem, izņemot valodā L0 iebūvētās vadības nodošanas komandas, kas plašiem un ērtiem pielietojumiem bija stipri par trūcīgām. Valodā L0 (un līdz ar to arī valodā L2) vadību varēja nodot vienīgi vai nu ar komandas „goto" palīdzību, vai arī, izmantojot „else" zarus dažādām L0 komandām. Valodā L3 zarošanās atbalstam ir pievienota standarta „if-then-else" konstrukcija, kas ir sastopama praktiski visās mūsdienu programmēšanas valodās.

Valodā L3 ieviestā zarošanās konstrukcija pēc savas semantikas ir analoģiska jebkuras programmēšanas valodas „if-then-else" konstrukcijas semantikai, tas ir, šeit iespējams norādīt kaut kādus nosacījumus, atkarībā no kuru patiesuma vērtības vadība tiek nodota vai nu „then" zaram, vai „else" zaram. Taču, atšķirībā no tradicionālajām programmēšanas valodām, kurās kā zarošanās nosacījums var būt vienīgi ļoti ierobežotas valodas konstrukcijas (visbiežāk – loģiskas izteiksmes), valodā L3 šajā ziņā atļauta daudz plašāka rīcības brīvība (skatīt sīkāku aprakstu nākamajā punktā).

4.2.2. Valodu metamodeļi

Katras nākamās valodu saimes Lx valodas metamodelis veidots uz iepriekšējās valodas metamodeļa bāzes, papildinot to ar jaunām klasēm un/vai asociācijām. Attēlā 4.3. redzams valodas L3 metamodelis, kur treknrakstā attēlotas klases un asociācijas, kas nākušas klāt valodās L0', L1, L2 un L3, salīdzinot ar valodu L0.

Valoda L0' papildinājusi L0 metamodeli ar klasi „Expression", kuras instances ir katra konkrētā aritmētiskā izteiksme (precīzāk – katras izteiksmes parādīšanās L0' programmā, tas ir, ja L0' programmā vairākās vietās parādās pēc izskata vienādas aritmētiskās izteiksmes, tām tomēr metamodelī atbilst dažādas klases „Expression" instances), kā arī ar klasi EElem un tās apakšklasēm, kuru instances kodē dažādus aritmētiskās izteiksmes primitīvus – mainīgos (primitīvo datu tipu vai norādes uz objektiem), atribūtus, funkciju izsaukumus ar parametru sarakstu, konstantes, aritmētiskos operatorus (+, -, *, /) un apaļās iekavas. Tā kā aritmētiskās izteiksmes valodā L0' var tikt lietotas vienīgi, lai vai nu piešķirtu vērtību kādam mainīgajam vai atribūtam, vai arī lai salīdzinātu kāda mainīgā vai atribūta vērtību ar doto izteiksmi, tad L0' metamodelī no klases „Expression" eksistē asociācijas uz klasēm „SCom" un „ECom", kas satur sevī komandas, kas atbildīgas par attiecīgi vērtību piešķiršanu un vērtību salīdzināšanu. Līdz ar to katra aritmētiskā izteiksme tiek piesaistīta ne vairāk kā vienai komandai.

Lai arī šablonu atpazīšanas iespēja ir viens no fundamentālākajiem modelēšanas jēdzieniem un tieši tas ir nācis klāt transformāciju valodā L1, salīdzinot ar transformāciju valodu L0', tomēr valodas L1 metamodelī vienīgās izmaiņas, salīdzinot to ar valodas L0' metamodeli, ir tādas, ka ir pievienota viena jauna asociācija starp klasēm „FNCom" un „ComBlock" (skatīt attēlu 4.3.).

Tātad katrai objektu meklēšanas komandai („first" un „next"), kas metamodelī tiek attēlota kā klases „FNCom" instance, valodā L1 iespējams norādīt tā saucamo „suchthat" bloku, kurā, savukārt, ar valodas L1 līdzekļiem iespējams nodefinēt šablonu, pēc kura meklēt kārtējo objektu. L0 metamodelī klases „ComBlock" instances pārstāv bloku, kurš satur valodas L0 komandas. Analoģiski – valodā L1 šīs klases instances pārstāv bloku, kurš satur valodas L1 komandas, savukārt, viena noteikta tipa komandas (klases „FNCom" instances) var saturēt vēl vienu (savu iekšējo) bloku, kas atkal var saturēt patvaļīgas L1 komandas. Tādējādi ar „suchthat" blokiem iespējams veidot šablonus vienu iekš otra patvaļīgā dziļumā, kā arī, protams, tos kombinēt horizontālā hierarhijā, līdz ar to no vienkāršiem šabloniem veidojot arvien sarežģītākus.

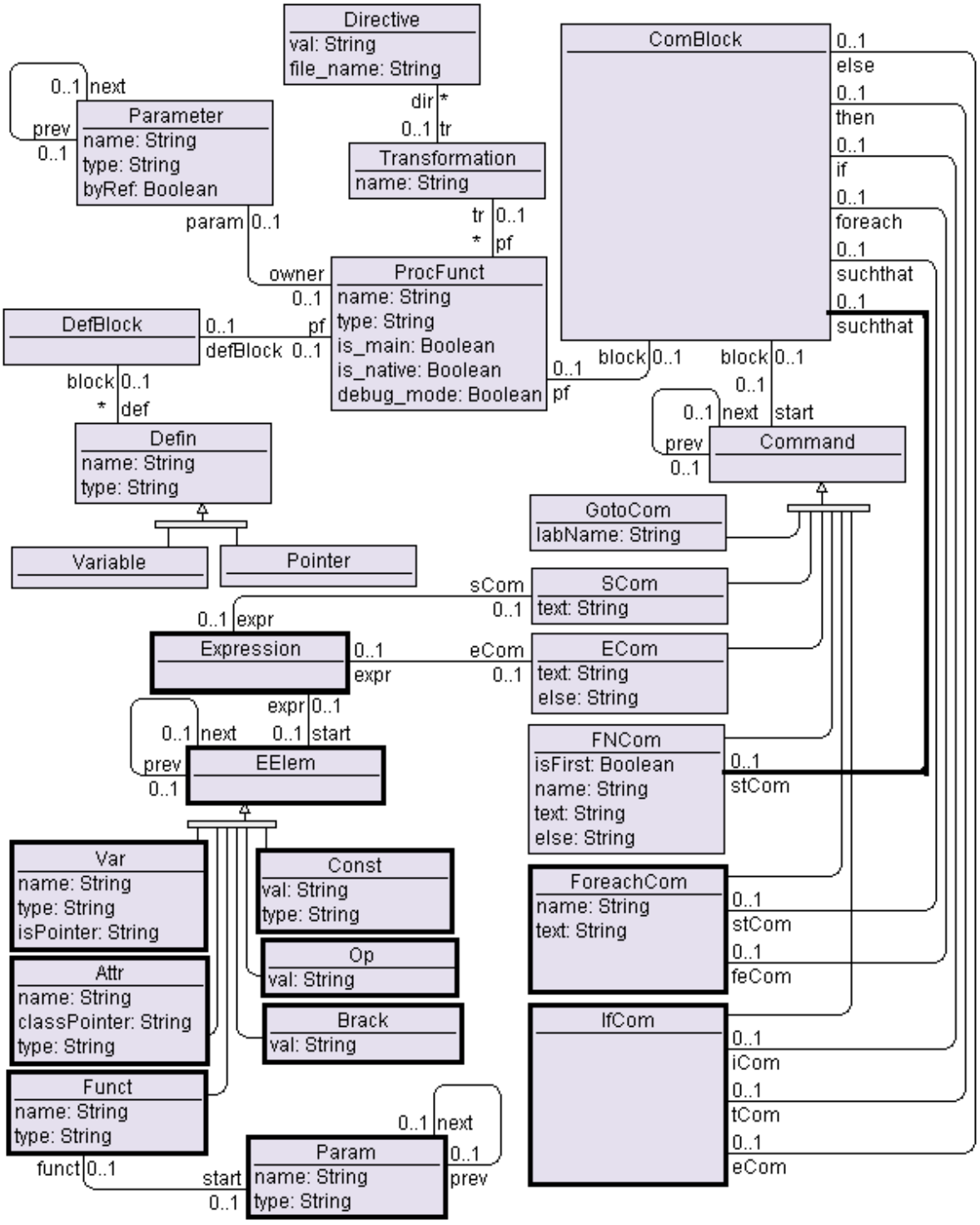
Transformāciju valodas L2 metamodelis, salīdzinot ar valodas L1 metamodeli, ir papildināts ar vienu jaunu klasi – „ForeachCom" –, kuras instances atbilst šai jaunajai cikla komandai (skatīt attēlu 4.3.). No šīs klases ir asociācija „foreach" uz klasi „ComBlock", kas instanču līmenī nozīmē saiti uz to bloku, kurā aprakstīts cikla ķermenis, tas ir, uz bloku, kurš satur tās valodas L2 komandas, kuras jāizpilda katram ciklā pārlasāmajam dotās klases objektam. Tāpat katrai cikla komandai iespējams arī norādīt šablonu, pēc kura atlasīt instances. Tas darāms, izmantojot asociāciju „suchthat" no klases „ForeachCom" uz klasi „ComBlock". Šis „suchthat" bloks semantiski nozīmē to pašu, ko „suchthat" bloks objektu meklēšanas komandām valodā L1. Klases „ForeachCom" objekti tiek raksturoti ar vārdu (atribūts „name" ar tipu *String*) un pārējo komandas tekstu (atribūts „text" ar tipu *String*), kas ietver sevī klases vārdu un „from... by..." daļu, ja tāda eksistē.

Tā kā klases „ForeachCom" katras instances „foreach" blokā iespējams veidot patvaļīgas valodas L2 komandas, ieskaitot arī cikla veidošanas komandu, tad skaidrs, ka valoda L2 piedāvā iespēju veidot arī iekļautos ciklus patvaļīgā dziļumā.

Valodas L3 metamodelis, salīdzinot ar valodas L2 metamodeli, ir papildināts ar vienu jaunu klasi – „IfCom", kuras instances ir zarošanās komandas valodā L3 rakstītā programmā (skatīt attēlu 4.3.). Katrai šādai zarošanās komandai iespējams norādīt trīs komandu blokus – komandu bloku, kurš satur komandas, kas atbilst komandas nosacījumam (asociācija „if"

starp klasēm „IfCom” un „ComBlock”), komandu bloku, kas satur komandas, kas izpildāmas gadījumā, ja lietotāja definētie nosacījumi atzīti par patiesiem (asociācija „then” starp klasēm „IfCom” un „ComBlock”), un komandu bloku, kas satur komandas, kas izpildāmas gadījumā, ja lietotāja definētie nosacījumi atzīti par aplamiem (asociācija „else” starp klasēm „IfCom” un „ComBlock”).

Tātad patiesa un aplama nosacījuma gadījumā iespējams norādīt komandu bloku, kurš satur tieši tās valodas L3 komandas, kas izpildāmas katrā no šiem gadījumiem. Bet ko tad īsti valodā L3 saprot ar zarošanās nosacījumu? Tā kā šis nosacījums arī ir komandu bloks, kas var sastāvēt no valodas L3 komandām, tad šāda bloka patiesuma vērtība tiek definēta tieši tādā pat veidā, kā tā tika definēta „suchthat” blokam transformāciju valodas L1 gadījumā, proti, nosacījumu bloks tiek uztverts kā *begin-end* izteiksme (skatīt nodaļu 4.3.1.), kas tātad tiek atzīta par patiesu tad un tikai tad, ja, sākot izpildīt tās saturošās komandas, sākot ar pirmo, iespējams veiksmīgi nonākt līdz pēdējai dotās izteiksmes komandai un veiksmīgi to izpildīt; pretējā gadījumā bloka patiesuma vērtība tiek pieņemta par aplamu. Un tā kā katrā no šiem blokiem atļauts ieviest patvaļīgas valodas L3 komandas, tad zarošanās komanda valodā L3 ir veiksmīgi integrējama visu pārējo L3 komandu starpā un savietojama ar tām, radot iespēju veidot iekļautas komandas.



4.3. attēls. **Transformāciju valodas L3 metamodelis ar treknrakstā iezīmētiem elementiem, kas nākuši klāt, salīdzinot ar bāzes valodu L0**

4.2.3. Valodu jaunu konstrukciju tekstuālā sintakse

Aritmētisko izteiksmju pieraksts

Aritmētisko izteiksmju pierakstā lietojama infiksā notācija, tas ir, aritmētiskais operators katrā gadījumā pierakstāms pa vidu starp abiem tā operandiem. Ja kāds no abiem operandiem ir

negatīvs, tas obligāti jāliek iekavās, izņemot gadījumus, kad šis operands ir pats pirmais operands vai nu visā aritmētiskajā izteiksmē vai arī tādā izteiksmes daļā, kura ielikta iekavās (tas ir, tieši pirms kuras atrodas iekava „(” un tieši pēc kuras atrodas iekava „)”). Tā, piemēram, pieraksti „-1+2”, „a+(-5-b)” un „a+(-3)” uzskatāmi par korektiem, savukārt, pieraksts „a+-3” uzskatāms par nekorektu.

Konstanšu pieraksts

Pierakstot konstantes aritmētiskajā izteiksmē, jāņem vērā datu tipa, kuram šīs konstantes pieder, īpatnības. *Integer* tipa konstante ir jebkura ciparu virkne, kā arī ciparu virkne, kurai priekšā pierakstīta „-” zīme. *Real* tipa konstante ir jebkura ciparu virkne, kurā tieši vienu reizi sastopams simbols „.”, kā arī jebkura šāda virkne, kurai priekšā pierakstīta „-” zīme. Savukārt, *String* tipa konstante ir jebkura simbolu virkne, kas ielikta pēdiņās. Tā, piemēram, pieraksts 17 tiek uzskatīts par *Integer* tipa konstanti, pieraksts 17.0 – par *Real* tipa konstanti, savukārt, pieraksts „17” – par *String* tipa konstanti.

Datu tipi

Ikvienam aritmētiskās izteiksmes operandam ir svarīgi zināt tā datu tipu, lai būtu iespējams izveidot korektu aritmētisko izteiksmi un lai varētu korekti noteikt rezultātā iegūtās vērtības datu tipu. Konstanšu gadījumā datu tipi tiek iegūti pēc augstāk minētā apraksta, tātad tie ir vienmēr viennozīmīgi nosakāmi. Mainīgo un funkciju gadījumā datu tips iegūstams no šī mainīgā vai funkcijas deklarācijas vietas, jo valodā L0 (un līdz ar to arī valodā L0’) katrs mainīgais un katra funkcija pirms izmantošanas ir noteikti jādeklarē, nosakot tā(-s) vārdu un datu tipu, kuram šis mainīgais vai funkcija pieder. Līdz ar to arī mainīgo un funkciju gadījumā datu tips ir viennozīmīgi iegūstams.

Bet ko darīt atribūtu gadījumā? Uz atribūtiem neattiecas nekādas prasības par to deklarēšanu pirms izmantošanas, kā arī pēc atribūta pieraksta nav iespējams noteikt tā datu tipu. Tādēļ atribūtu izmantošanas gadījumā tā datu tips jānorāda pašā aritmētiskajā izteiksmē, rakstot uzreiz aiz atribūta vārda kolu un aiz tā – šī atribūta datu tipu, piemēram, „person1.age:Integer”.

Formāla aritmētiskas izteiksmes definīcija

Augstāk tika neformāli aprakstīts, kādai jāizskatās aritmētiskai izteiksmei valodā L0’. Tagad to var nodefinēt arī formālāk.

Aritmētiska izteiksme ir:

1) *String* tipa konstante;
2) *Integer* vai *Real* tipa pozitīva konstante;
3) (-C), kur C – *Integer* vai *Real* tipa pozitītva konstante;
4) *Integer*, *Real* vai *String* tipa mainīgais;
5) *Integer*, *Real* vai *String* tipa atribūts, kas pierakstīts augstāk minētajā veidā – `<pointerName>.<attributeName>:<typeName>;`
6) *Integer*, *Real* vai *String* tipa funkcijas izsaukums;
7) (E), kur E – aritmētiska izteiksme;
8) E+F, kur E un F – aritmētiskas izteiksmes ar saderīgiem tipiem;
9) E-F, kur E un F – aritmētiskas izteiksmes ar saderīgiem tipiem;
10) E*F, kur E un F – aritmētiskas izteiksmes ar saderīgiem tipiem;
11) E/F, kur E un F – aritmētiskas izteiksmes ar saderīgiem tipiem.

Šablonu definēšanas pieraksts

Šablonu atbilstības meklēšanas bloku abām objektu meklēšanas komandām (komandām „first” un „next”) pieraksta, lietojot aiz šīs komandas, bet pirms „else” zara, ja tāds eksistē, atslēgas vārdu „suchthat” un tālāk rakstot šī šablonu veidošanas bloka komandas starp atslēgas vārdiem „begin” un „end”:

```
   first <pointerName1> : <className> [ from <pointerName2> by
<roleName> ] [ suchthat
   begin
      <L1Commands>
   end ];
   next <pointerName> [ suchthat
   begin
      <L1Commands>
   end ];
```

Praktiskos lietojumos šo bloku var izmantot, norādot, piemēram, atribūtu vērtības, kādām meklētajam objektam būtu jāpiemīt, vai arī saites ar kādām dotajam objektam būtu jābūt saistītam ar kādiem citiem objektiem, taču principā šeit netiek uzlikti nekādi ierobežojumi – atļauts lietot jebkādas valodas L1 komandas. Komandas beigās obligāti rakstāms semikols.

Cikla komandas pieraksts

Cila komanda pierakstāma, lietojot komandas nosaukumu „foreach” un norādot cikla mainīgo un klasi, pa kuru notiks ciklošanās, kā arī vajadzības gadījumā norādot objektu vērtību apgabala ierobežojumus analoģiskā veidā, kā tas tiek darīts komandā „first” valodā L0:

```
   foreach <pointerName1> : <className> [ from <pointerName2> by
<roleName> ] [ suchthat
   begin
      <L2Commands>
   end ]
   do
   begin
      <L2Commands>
   end;
```

Tālāk iespējams norādīt „suchthat” bloku tādā pat veidā kā objektu meklēšanas komandu gadījumā – rakstot atslēgas vārdu „suchthat” un tālāk norādot šablonu definēšanas bloka komandas, rakstot tās starp atslēgas vārdiem „begin” un „end”. Šis bloks cikla komandai nav obligāts – ja tas netiks pierakstīts, ciklošanās notiks pa visām dotās klases instancēm. Savukārt, pēc tam obligāti jānorāda bloks, kas satur komandas, kuras izpildāmas katram pārlasāmajam dotās klases objektam. Tas darāms, rakstot atslēgas vārdu „do” un tālāk norādot „foreach” bloka komandas, rakstot tās starp atslēgas vārdiem „begin” un „end”. Komandas beigās obligāti liekams semikols.

Zarošanās komandas pieraksts

Zarošanās komanda pierakstāma, lietojot atslēgas vārdus „if”, „then” un „else”, un aiz katra no tiem pierakstot attiecīgā komandu bloka komandas, ietverot tās atslēgas vārdos „begin” un „end”:

```
   if
   begin
      <L3Commands>
   end
   then
   begin
      <L3Commands>
   end
   [ else
```

```
begin
   <L3Commands>
end ];
```

Tāpat kā tradicionālajās programmēšanas valodās, arī transformāciju valodā L3 „else” zars nav obligāts, tāpēc atslēgas vārdu „else” kopā ar atslēgas vārdiem „begin” un „end” un to definējošo bloku drīkst arī nerakstīt. Turpretī, „if” un „then” zari ir obligāti, līdz ar ko obligāti ir arī tiem atbilstošie komandu bloki, kas ietveri starp atslēgas vārdiem „begin” un „end”.

4.2.4. Piemērs

Uzdevumu, kas dots nodaļā „4.1.3. Modeļu transformācijas piemērs”, transformāciju valodā L3 iespējams pierakstīt, piemēram, kā šādu programmu:

```
transformation example;
   main procedure main();
      pointer c:Course;
      pointer s:Student;
      var x:Real;
      var count:Integer;
   begin;
      first c:Course suchthat
      begin
         attr c.title==”Operating Systems”;
      end else endOfProg;
      setVar x=0;
      setVar count=0;
      foreach s:Student from c by student suchthat
      begin
         attr s.age>=18;
      end
      do
      begin
         setVar count=count+1;
         setVar x = x + ( s.mark1:Real + s.mark2:Real +
s.mark3:Real + s.mark4:Real + s.mark5:Real + s.mark6:Real +
s.mark7:Real + s.mark8:Real ) / 8;
      end;
      if
      begin
```

```
            var count>0;
            setVar x=x/count;
            var x<8;
         end
         then
         begin
            setAttr c.hasGoodStudents=false;
         end
         else
         begin
            setAttr c.hasGoodStudents=true;
         end;
         label endOfProg;
      end;
   endTransformation;
```

Pateicoties valodai L0', parādījusies iespēja salikt kopā vairākas aritmētiskās izteiksmes vienā komandā. Pateicoties valodas L1 šablonu bloka iespējai, šeit vairs nav jāmeklē derīgais kurss un derīgie studenti, izmantojot zema līmeņa vadības nodošanas komandas un iezīmes, bet gan iespējams veiksmīgi izmantot *begin-end* izteiksmes (skatīt nodaļu 4.3.1.) objektu meklēšanas komandu šablonu definēšanas blokos. Šajā gadījumā tas ievērojami atvieglo programmas lasāmību.Tāpat abu studentu meklēšanas komandu vietā ieviesta viena cikla komanda, kas vēl vairāk uzlabo programmas lasāmību, kā arī iizmantotas zarošanās komandas piedāvātās iespējas, lai apvienotu viena veida gadījumus vienā zarā.

4.2.5. L3 sintakses formāla definīcija

Šajā punktā dota pilna transformāciju valodas L3 sintakses definīcija atbilstoši Bekusa-Naura (*Backus-Naur*, [50]) normālformai.

```
<L3Program>          ::= <transformation> [ <L3Program> ]
<transformation>     ::= transformation <identifier> ;
<transfPartList> endTransformation;
<identifier>         ::= <letter> [ <string> ]
<specialID>          ::= <specialLetter> [ <string> ]
<string>             ::= <letter> [ <string> ] | <digit> [ <string>
]
<letter>             ::= <specialLetter> | _
```

```
<specialLetter>      ::= a | b | c | d | e | f | g | h | i | j | k
| l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B
| C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S
| T | U | V | W | X | Y | Z
<digit>              ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<transfPartList>     ::= <transfPart> [ <transfPartList> ]
<transfPart>         ::= <nativeProc> | <nativeFunct> | <directive>
| <debug> | <varDeclaration> | <procedure> | <function>
<nativeProc>         ::= native procedure <identifier> ( [
<paramList> ] );
<paramList>          ::= <parameter> [ , <paramList> ]
<parameter>          ::= <identifier> : <primTypeOrMMElem>
<nativeFunct>        ::= native function <identifier> ( [
<paramList> ] ): <primTypeOrMMElem> ;
<directive>          ::= <dirType> ” <fileName> ”;
<dirType>            ::= useMM | include | useLib | useUnit
<fileName>           ::= [ <specialLetter> :\] [ <folderList> ]
<string> [ . <string> ]
<folderList>         ::= <string> \ [ <folderList> ]
<debug>              ::= DEBUG_ON; | DEBUG_OFF;
<varDeclaration>     ::= <primitiveVarDecl> | <pointerDecl>
<primitiveVarDecl>   ::= var <specialID> : <primTypeName> ;
<primTypeName>       ::= Integer | Real | String | Boolean
<pointerDecl>        ::= pointer <identifier> : <metaModelElement>
;
<metaModelElement>   ::= <letter> [ <stringPlus> ]
<stringPlus>         ::= <stringPlusElem> [ <stringPlus> ]
<stringPlusElem>     ::= <letter> | <digit> | # | ::
<procedure>          ::= [ main ] procedure <identifier> ( [
<paramList> ] ); [ <varList> ] begin; [ <L3CommandList> ] end;
<function>           ::= function <identifier> ( [ <paramList> ] ):
<primTypeOrMMElem> ; [ <varList> ] begin; [ <L3CommandList> ] end;
<primTypeOrMMElem>   ::= <primTypeName> | <metaModelElement>
<varList>            ::= <varDeclaration> [ <varList> ]
<L3CommandList>      ::= <L3Command> [ <L3CommandList> ]
<L3Command>          ::= <call> | <return> | <first> | <next> |
<goto> | <label> | <addObj> | <addLink> | <deleteObj> | <deleteLink>
| <setPointer> | <setPointerF> | <setVar> | <setVarF> | <setAttr> |
```

```
<type> | <var> | <pointer> | <attr> | <link> | <noLink> | <debug> |
<foreach> | <if>
<call>                ::= call <identifier> ( [ <paramList> ] );
<return>              ::= return [ <identifier> ] ;
<first>               ::= first <identifier> : <metaModelElement> [
from <identifier> by <metaModelElement> ] [ suchthat begin [
<L3CommandList> ] end ] [ else <specialID> ] ;
<next>                ::= next <identifier> [ suchthat begin [
<L3CommandList> ] end ] [ else <specialID> ] ;
<goto>                ::= goto [ <specialID> ] ;
<label>               ::= label <specialID> ;
<addObj>              ::= addObj <identifier> : <metaModelElement> ;
<addLink>             ::= addLink <identifier> . <metaModelElement>
. <identifier> ;
<deleteObj>           ::= deleteObj <identifier> ;
<deleteLink>          ::= deleteLink <identifier> .
<metaModelElement> . <identifier> ;
<setPointer>          ::= setPointer <identifier> = <identifier> ;
<setPointerF>         ::= setPointerF <identifier> = <identifier> (
[ <paramList> ] );
<setVar>              ::= setVar <specialID> = <expression> ;
<setVarF>             ::= setVarF <specialID> = <identifier> ( [
<paramList> ] );
<setAttr>             ::= setAttr <identifier> . <metaModelElement>
= <expression> ;
<type>                ::= <identifier> <pointerRelOp>
<metaModelElement> [ else <specialID> ] ;
<var>                 ::= <identifier> <relationOperator>
<expression> [ else <specialID> ] ;
<pointer>             ::= <identifier> <pointerRelOp> <identifier> [
else <specialID> ];
<attr>                ::= attr <identifier> . <metaModelElement>
<relationOperator> <expression> [ else <specialID> ];
<link>                ::= link <identifier> . <metaModelElement> .
<identifier> [ else <specialID> ] ;
<noLink>              ::= noLink <identifier> . <metaModelElement> .
<identifier> [ else <specialID> ] ;
```

```
<foreach>            ::= foreach <identifier> : <metaModelElement>
[ from <identifier> by <metaModelElement> ] [ suchthat begin [
<L3CommandList> ] end ] do begin [ <L3CommandList> ] end;
<if>                 ::= if begin [ <L3CommandList> ] end then
begin [ <L3CommandList> ] end [ else begin [ <L3CommandList> ] end ]
;
<expression>         ::= <boolExpr> | <notBoolExpr>
<boolExpr>           ::= true | false
<notBoolExpr>        ::= <exprPart> [ <arithmeticOper>
<notBoolExpr> ]
<exprPart>           ::= <const> | <identifier> | <attribute> |
<functionCall>
<arithmeticOper>     ::= + | - | * | /
<const>              ::= <integerConst> | <realConst> |
<stringConst>
<integerConst>       ::= <positiveNumber> | (– <positiveNumber> )
<positiveNumber>::= <digit> [ <positiveNumber> ]
<realConst>          ::= <positiveNumber> . <positiveNumber> | (–
<positiveNumber> . <positiveNumber> )
<stringConst>        ::= ” <extendedString> ”
<extendedString>     ::= <symbol> [ <extendedString> ]
<symbol>             ::= <letter> | <digit> | <relationOperator> |
~ | ‘ | ! | @ | # | $ | % | ^ | & | ( | ) | { | } | : | < | > | ? |
[ | ] | ; | ‘ | \ | , | . | <space>
<attribute>          ::= <identifier> . <metaModelElement> :
<primTypeName>
<functionCall>       ::= <identifier> ( [ <paramList> ] )
<relationOperator>   ::= <pointerRelOp> | < | > | <= | >=
<pointerRelOp>       ::= == | !=
<space>              ::=
```

Piezīme – neterminālais simbols <space> tiek saprasts kā tukšuma simbols (32. simbols ASCII simbolu tabulā).

4.3. Valodas L1 izteiksmīgums

4.3.1. Šablona definēšanas bloka semantika

Kāda tad ir „suchthat" bloka semantika? Tā kā šablonu definēšanas bloka komandas, kas atrodas starp atslēgas vārdiem *begin* un *end* par katru aplūkojamo objektu spēj viennozīmīgi rast atbildi uz jautājumu par šī objekta atbilstību dotajam šablonam, tad uz šo bloku var raudzīties kā uz jauna veida loģiskā datu tipa izteiksmi, kas turpmāk tiks saukta par *begin-end* izteiksmi [51]. Precīzāk – ar *begin-end* izteiksmi tiek saprasta konstrukcija izskatā:

```
begin
   <L1Commands>
end
```

Kad ieviests šāds apzīmējums, varam definēt „suchthat" bloka (jeb *begin-end* izteiksmes) semantiku – *begin-end* izteiksme pieņem patiesu vērtību tad un tikai tad, ja, ņemot vērā konkrēto aplūkojamo objektu un izpildot pēc kārtas visas dotā bloka komandas, sākot no pirmās, iespējams veiksmīgi sasniegt bloka beigas, tas ir, sekmīgi izpildīt tā pēdējo komandu.

Kā valodā L1 tiek definēta sekmīga kādas komandas izpilde? Lai atbildētu uz šo jautājumu, pietiks aplūkot divu tipu komandas – „goto" komandu un komandas, kurām var tikt piekārtots „else" zars ar iezīmi. Visu pārējo komandu gadījumā nevar būt runa par sekmīgu neizpildīšanos, jo tās vienmēr ir sekmīgi izpildāmas šajā nozīmē. Aplūkosim šo abu augstāk minēto tipu komandas sīkāk:

1) Saskaņā ar valodas L0 sintaksi vadības nodošanas („goto") komandai ir jābūt pierakstītai tieši vienai iezīmei. Saskaņā ar valodas L0 semantiku pēc šīs komandas izpildes vadība tiek nodota tai iezīmes komandai, kas satur tieši tādu pat iezīmi, kādu satur dotā vadības nodošanas komanda. Valodā L1 šī prasība ir mainīta – vadības nodošanas komandai *begin-end* izteiksmes ietvaros var tikt pierakstīta ne vairāk kā viena iezīme. Ja šī komanda nesatur nekādu iezīmi, attiecīgā *begin-end* izteiksme, nonākot līdz šādas komandas izpildei, pieņem aplamu patiesuma vērtību un atlikušās šī bloka komandas (ja tādas eksistē) vairs netiek pildītas. Tātad vadības nodošanas komanda sekmīgi tiek izpildīta tad un tikai tad, ja tai ir pierakstīta tieši viena iezīme (pie kam šai iezīmei noteikti jāeksistē pie kādas attiecīgā bloka „label" komandas).
2) Saskaņā ar valodas L0 sintaksi katrai no otra augstāk minētā tipa komandām drīkst būt ne vairāk kā viens „else" zars. Saskaņā ar valodas L0 semantiku vadība pēc šāda tipa komandas apstrādes tiek nodota nākamajai komandai blokā tad un tikai tad, ja šī

komanda ir saturējusi pārbaudi, kuras vērtība ir patiesa. Ja pārbaudes rezultāts ir izrādījies nepatiess un dotajai komandai ir bijis piekārtots „else” zars, tad vadība tiek nodota tai komandai, kura satur šim zaram piekārtoto iezīmi. Taču, ja negatīvas pārbaudes gadījumā komandai „else” zars neeksistē, tad vadība tiek nodota attiecīgās procedūras vai funkcijas beigām. Valodā L1 negatīvas pārbaudes gadījumā neesošs „else” zars kādai no *begin-end* izteiksmes komandām noved pie tā, ka šī izteiksme pieņem aplamu patiesuma vērtību. Tātad šāda tipa komandas sekmīgi tiek izpildītas tad un tikai tad, ja vai nu pārbaude ir izrādījusies sekmīga, vai arī komandai eksistē „else” zars ar tam piekārtotu iezīmi (pie kam šai iezīmei noteikti jāeksistē pie kādas attiecīgā bloka „label” komandas).

4.3.2. Paplašināto objekta meklēšanas komandu semantika

Šajā vietā nepieciešams nedaudz dziļāk izskaidrot objektu meklēšanas komandu („first” un „next”) semantiku, kas, iespējams, uzreiz var nebūt intuitīvi precīzi skaidra. Valodā L0 šo komandu semantika tiek skaidrota šādā veidā:

1) Sastopot „first” komandu ar piekārtotu norādi `<pointerName>` uz klasi `<className>`, šai norādei tiek izdalīts vērtību apgabals, tas ir, tiek noteiktas tās klases `<className>` instances, uz kurām turpmāk `<pointerName>` varēs norādīt. Ja dotā „first” komanda nav saturējusi „from ... by ...” daļu, tad šis vērtību apgabals ir visas klases `<className>` instances, tas ir, `<pointerName>` teorētiski varētu norādīt uz jebkuru objektu no klases `<className>`. Savukārt, ja dotajai „first” komandai ir bijusi piekārtota „from ... by ...” daļa, tad `<pointerName>` vērtību apgabals tiek ierobežots ar tieši tām klases `<className>` instancēm, kuras ir sasniedzamas no „from ... by ...” daļā norādītā objekta ar tajā norādīto lomas vārdu. Papildus vērtību apgabala noteikšanai pēc šīs komandas izpildes norādei `<pointerName>` tiek piekārtots patvaļīgs objekts no notiktā vērtību apgabala un pēc tam šis objekts no vērtību apgabala tiek izslēgts. Ja, izpildot „first” komandu, tiek konstatēts, ka `<pointerName>` vērtību apgabals ir tukša kopa, vadība tiek nodota iezīmei, kas pierakstīta šīs komandas „else” zaram (ja tāds eksistē).
2) Sastopot „next” komandu ar piekārtotu norādi `<pointerName>` (tā pati norāde, kas iepriekš bijusi piekārtota „first” komandai), no iepriekš izveidotā `<pointerName>` vērtību apgabala norādei `<pointerName>` tiek piekārtots patvaļīgs objekts (ja tāds eksistē), pēc tam uzreiz šo objektu izslēdzot no atbilstošā vērtību apgabala. Ja, izpildot „next” komandu, tiek konstatēts, ka `<pointerName>` vērtību apgabals ir tukša

kopa, vadība tiek nodota iezīmei, kas pierakstīta šīs komandas „else” zaram (ja tāds eksistē).

3) Sastopot „next” komandu ar piekārtotu norādi, kas pirms tam nav tikusi aplūkota nevienā „first” komandā (tātad tai nav noteikts vērtību apgabals), programmas darbība nav definēta.

Transformāciju valodā L1 pilnībā tiek pārņemta objektu meklēšanas komandu semantika no valodas L0, papildinot to ar nosacījumiem, kas atbilst šablonu definēšanas bloka semantikai:

1) Sastopot „first” komandu ar piekārtotu norādi `<pointerName>`, tās vērtību apgabals tiek izveidots tieši tādā pat veidā, kā tas notiek valodas L0 gadījumā. Papildus tam norādei `<pointerName>` tiek piekārtots tāds patvaļīgs objekts no šī vērtību apgabala, kuram atbilstošā šablonu definēšanas bloka *begin-end* izteiksme (ja tāda eksistē) pieņem patiesu vērtību, kā arī šis objekts no vērtību apgabala tiek izslēgts. Ja šāds objekts neeksistē (vai nu vērtību apgabals ir tukša kopa, vai arī nevienam no vērtību apgabala objektiem neizpildās minētais nosacījums), tad vadība tiek nodota iezīmei, kas pierakstīta šīs komandas „else” zaram (ja tāds eksistē).
2) Sastopot „next” komandu ar piekārtotu norādi `<pointerName>`, kurai iepriekš ir izveidots vērtību apgabals (iepriekš izpildīta „first” komanda ar šo norādi), norādei `<pointerName>` tiek piekārtots tāds patvaļīgs objekts no šī vērtību apgabala, kurš apmierina piekārtoto šablonu definēšanas bloka *begin-end* izteiksmi (ja tāda eksistē), kā arī šis objekts no vērtību apgabala tiek izslēgts. Ja šāds objekts neeksistē, vadība tiek nodota iezīmei, kas pierakstīta šīs komandas „else” zaram (ja tāds eksistē).
3) Sastopot „next” komandu ar tādu piekārtotu norādi, kurai iepriekš nav noteikts vērtību apgabals, programmas darbība nav definēta.

4.3.3. Šablonu definēšanas bloka piemēri

Vienkāršs šablons, pēc kura vadoties tiek atrasta pirmā klases „Person” instance, kurai izpildās nosacījums, ka tās vecums ir vienāds ar 24 (šis un nākamie piemēri veidoti, balstoties uz vienkāršu metamodeli, kurš sastāv no vienas klases „Person” ar vienu atribūtu „age” ar tipu „Integer” un vienu asociāciju „father/son”):

```
first p:Person suchthat
begin
   p.age==24;
end;
```

Šajā gadījumā komanda „first” atradīs pirmo tādu *p* no klases „Person”, kuram varēs sekmīgi izpildīt šo vienu komandu – „`p.age==24;`”. Ko šajā gadījumā nozīmē sekmīga izpilde? Tā kā šī ir pārbaudes tipa komanda, tad šādas pārbaudes rezultāts var būt vai nu patiess vai aplams. Saskaņā ar valodas L0 sintaksi un semantiku, patiesuma gadījumā izpilde nonāk pie nākamās komandas, bet aplamības gadījumā notiek pāreja pa „else” zaru, ja tāds ir, vai notiek izeja no bloka, ja tāds nav. Tā kā dotajā gadījumā nav „else” zara, tad šī arī ir vienīgā iespēja, kā augstāk minētā komanda varētu sekmīgi neizpildīties – iespēja, kad salīdzināšanas rezultāts ir aplams. Tātad pēc būtības tas nozīmē tieši to, kas bija vajadzīgs – objekts tiks atzīts par piederīgu dotajam šablonam tad un tikai tad, ja izpildīsies nosacījums, ka atribūta „age” vērtība šim objektam ir vienāda ar 24.

Šablons, pēc kura vadoties, tiek atrasta nākamā šīs pašas klases instance, kurai ir spēkā tas pats nosacījums:

```
next p suchthat
begin
   attr p.age==24;
end
else no_more_persons;
```

Šajā gadījumā instances tiek meklētas no to instanču kopas, kuras vēl ar „first” vai „next” komandām šai norādei *p* nav tikušas apmeklētas. Ja meklēšanas komandai ir pievienots „else” zars kā šajā piemērā, tad tā semantika ir tieši tāda pati, kāda tā ir valodas L0 gadījumā meklēšanas komandu „else” zaram, tas ir, pa šo zaru vadība aiziet tajā gadījumā, ja neizdodas atrast vairs nevienu instanci, kurai izpildītos dotais nosacījums.

Šablons, pēc kura vadoties, tiek atrasta pirmā klases „Person” instance, kurai izpildās nosacījums, ka tā ir 24 gadus veca un tā ir kādas dotas personas, uz kuru norāda agrāk definēta norāde *father*, dēls:

```
first p:Person suchthat
begin
   attr p.age==24;
   link father.son.p;
end
else no_such_persons;
```

Kā redzams, arī šeit nevienai no abām šablona blokā ietvertajām komandām nav norādīts „else” zars, lai panāktu to, ka gadījumā, ja kāds no šiem nosacījumiem neizpildās, notiktu

iziešana no „suchthat” bloka, nesasniedzot tā beigas, tādējādi nosakot, ka dotais objekts ir šablonam neatbilstošs.

Varētu būt uzdevums atrast tās personas, kurām ir kāds 24 gadus vecs dēls. Tādā gadījumā atbilstošā L1 komanda, kas atrod pirmo no šādām personām, varētu izskatīties, piemēram, šādi:

```
first parent:Person suchthat
begin
   first p:Person suchthat
   begin
      link parent.son.p;
      attr p.age==24;
   end;
end
else no_such_persons;
```

Tātad, ja šī komanda izpildās sekmīgi un vadība neaiziet pa „else” zaru, tad pēc šīs komandas izpildes norāde *parent* norādīs uz tādu klases „Person” instanci, kura atbilst augstāk minētajām prasībām (turklāt, norāde *p* norādīs uz tādu klases „Person” instanci, kurai ar atrasto instanci, uz kuru norāda norāde *parent*, ir saite ar doto vārdu). Šajā piemērā iekšējo „first” komandu iespējams lasīt kā „eksistē”, proti, visu šablonu kopā varētu lasīt kā „Atrast pirmo tādu *parent*, kuram eksistē tāds *p*, kurš ir attiecībā *son* ar norādi *parent* un kurš ir 24 gadus vecs”. Līdzīgā veidā iespējams veidot arī vēl dziļākus „suchthat” blokus, kas iekļauti viens otrā, tādējādi veidojot sarežģītākus šablonus lielāku instanču diagrammas daļu atpazīšanai.

4.3.4. L1 salīdzinājums ar predikātu valodu

Cik spēcīgi tad savā izteiksmīgumā īstenībā ir transformāciju valodā L1 piedāvātie šablonu definēšanas bloki? Kādu kategoriju uzdevumus tie spēj atrisināt? Atbildes uz šiem jautājumiem atrodamas šajā punktā.

Šablonu definēšanas bloks (jeb precīzāk – tam piekārtotā *begin-end* izteiksme), vienkārši runājot, spēj sniegt atbildi uz jautājumu par to, vai kāds dots objekts atbilst noteiktajiem kritērijiem vai nē, tātad, kā jau iepriekš minēts, *begin-end* izteiksme par katru aplūkojamās kopas objektu spēj rast tieši vienu konkrētu loģiskā datu tipa atbildi – patiess vai aplams. Palūkojoties uz šablonu bloku šādā veidā, varētu rasties vēlēšanās veikt zināma veida salīdzinājumu ar predikātu loģikas formulām, kas arī ir loģiskā tipa objekti. Ja transformāciju

valoda L1 šobrīd ir pazīstama tikai nelielam cilvēku skaitam, tad predikātu loģika uzskatāma par klasiku un ierindojama starp matemātikas zinātnes pamata disciplīnām, tādēļ šo divu izteiksmju (*begin-end* izteiksmes un predikātu loģikas formulas) salīdzinājums dotu lielāku priekšstatu par transformāciju valodas L1 iespēju klāstu.

Aplūkosim tipizētu predikātu loģikas valodu (*many-sorted first-order logic*) [52]. Saskaņā ar definīciju šādas valodas alfabēts sastāv no šādām simbolu kopām:

1) sanumurējama tipu kopa $S \cup \{bool\}$, kas satur speciālo tipu *bool*, pie kam S nav tukša kopa, un tā nesatur tipu *bool*;
2) loģiskie saikļi: $\wedge$ (konjunkcija), $\vee$ (disjunkcija), $\supset$ (implikācija) un $\equiv$ (ekvivalence), kas visi ir ar rangu ($bool \times bool \rightarrow bool$), $\neg$ (negācija) ar rangu ($bool \rightarrow bool$) un $\perp$ (tukšais jēdziens) ar rangu ($\varnothing \rightarrow bool$);
3) kvantori: $\forall_s$ (universālais kvantors) un $\exists_s$ (eksistences kvantors) katrai kopai $s \in S$;
4) vienādības simbols: $=_s$ ar rangu ($s \times s \rightarrow bool$) katrai kopai $s \in S$;
5) mainīgie: bezgalīga sanumurējama kopa $V_s = \{x_0, x_1, x_2, ...\}$ katrai kopai $s \in S$, kur katrs x_i ir ar rangu ($\varnothing \rightarrow s$);
6) palīg-simboli: „(” un „)”;
7) ne-loģikas simbolu alfabēts L, kas sastāv no:
    a. funkcionālajiem simboliem: sanumurējama kopa $FS = \{f_0, f_1, ...\}$ ar ranga funkciju r: $FS \rightarrow S^+ \times S$ (ar S^+ tiek saprasti visi S vārdi, izņemot tukšo vārdu, tas ir virknes garumā $n>0$, kuru visi elementi pieder kopai S), kas katram funkcionālajam simbolam f piekārto pāri $r(f) = (u,s)$, kur u tiek saukts par funkcionālā simbola f aritāti (*arity*), bet $s \in S$ – par f tipu;
    b. konstantēm: sanumurējama kopa $CS_s = \{c_0, c_1, ...\}$ katrai kopai $s \in S$, kur katrs c_i ir ar rangu ($\varnothing \rightarrow s$);
    c. predikātu simboliem: sanumurējama kopa $PS = \{P_0, P_1, ...\}$ar ranga funkciju r: $PS \rightarrow S^* \times \{bool\}$ (ar S^* tiek saprasti visi S vārdi, ieskaitot arī tukšo vārdu), kas katram predikātu simbolam P piekārto pāri $r(P) = (u, bool)$, kur u tiek saukts par predikātu simbola P aritāti.

Šeit tiek pieņemts, ka kopas V_s, FS, CS_s un PS savstarpēji nešķeļas ne pie kādām $s \in S$ vērtībām.

Ņemot vērā šādu valodas definīciju, predikātu valodas termi un atomārās formulas tiek definētas šādā veidā:

1) katra konstante un katrs mainīgais no kopas s ir kopas s terms;
2) ja t_1, ..., t_n ir termi, kur katrs t_i ir no kopas u_i, bet f ir funkcionālais simbols ar rangu ($<u_1, ..., u_n> \rightarrow s$), tad $f(t_1, ..., t_n)$ ir kopas s terms;
3) katrs predikātu simbols ar aritāti $\varnothing$, kā arī tukšais jēdziens ($\perp$) ir atomāra formula;
4) ja t_1 un t_2 ir kopas s termi, tad $=_s(t_1, t_2)$ ir atomāra formula;
5) ja t_1, ..., t_n ir termi, kur katrs t_i ir no kopas u_i, bet P ir predikāta simbols ar aritāti u_1, ..., u_n, tad $P(t_1, ..., t_n)$ ir atomāra formula.

Kad nodefinēti terma un atomāras formulas jēdzieni, var ķerties klāt svarīgākajam jēdzienam – formulai:

1) katra atomāra formula ir formula;
2) ja A un B ir formulas, tad arī $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $(A \equiv B)$ un $\neg A$ ir formulas;
3) ja x ir kopas s mainīgais, bet A – formula, tad arī $\forall_s x(A)$ un $\exists_s x(A)$ ir formulas.

Tagad aplūkosim pilnas tipizētas predikātu loģikas valodas apakškopu, ko nosauksim par valodu P^-. Definēsim valodas P^- alfabētu šādā veidā:

1) sanumurējama tipu kopa $S \cup \{bool\}$, kas satur speciālo tipu *bool*, pie kam S nav tukša kopa, un tā nesatur tipu *bool*;
2) loģiskie saikļi: $\wedge$ (konjunkcija) un $\vee$ (disjunkcija) ar rangu ($bool \times bool \rightarrow bool$), $\neg$ (negācija) ar rangu ($bool \rightarrow bool$) un $\perp$ (tukšais jēdziens) ar rangu ($\varnothing \rightarrow bool$);
3) kvantori: $\forall_s$ (universālais kvantors) un $\exists_s$ (eksistences kvantors) katrai kopai $s \in S$;
4) vienādības simbols: $=_s$ ar rangu ($s \times s \rightarrow bool$) katrai kopai $s \in S$;
5) mainīgie: bezgalīga sanumurējama kopa $V_s = \{x_0, x_1, x_2, ...\}$ katrai kopai $s \in S$, kur katrs x_i ir ar rangu ($\varnothing \rightarrow s$);
6) palīg-simboli: „(” un „)”;
7) ne-loģikas simbolu alfabēts L, kas sastāv no:
   a. funkcionāliem simboliem: sanumurējama kopa $FS = \{f_0, f_1, ...\}$ ar ranga funkciju r: $FS \rightarrow S \times S$, kas katram funkcionālajam simbolam f piekārto pāri $r(f) = (s,s)$;
   b. konstantēm: sanumurējama kopa $CS_s = \{c_0, c_1, ...\}$ katrai kopai $s \in S$, kur katrs c_i ir ar rangu ($\varnothing \rightarrow s$);
   c. predikātu simboliem: sanumurējama kopa $PS = \{P_0, P_1, ...\}$ar ranga funkciju r: $PS \rightarrow S^2 \times \{bool\}$, kas katram predikātu simbolam P piekārto pāri $r(P) = (<s,s>, bool)$.

Tādā gadījumā termi un atomārās formulas valodā P^- var tikt definētās sādā veidā:

1) katra konstante un katrs mainīgais no kopas s ir kopas s terms;
2) ja t ir terms no kopas u, bet f ir funkcionālais simbols ar rangu (u → s), tad f(t) ir kopas s terms;
3) ⊥ ir atomāra formula;
4) ja t_1 un t_2 ir kopas s termi, tad $=_s(t_1, t_2)$ ir atomāra formula;
5) ja t_1 un t_2 ir termi, kur katrs t_i ir no kopas u_i, bet P ir predikāta simbols ar aritāti $<u_1, u_{2>}$, tad $P(t_1, t_2)$ ir atomāra formula.

Bet formulas valodā P tiek definētās šādi:

1) katra atomāra formula ir formula;
2) ja A un B ir formulas, tad arī $(A \wedge B)$, $(A \vee B)$ un $\neg A$ ir formulas;
3) ja x ir kopas s mainīgais, bet A – formula, tad arī $\forall_s x(A)$ un $\exists_s x(A)$ ir formulas.

Šādas valodas P^- izveide, sašaurinot pilno tipizēto predikātu loģikas valodas definīciju, ļauj sasaistīt predikātu loģikas valodu ar transformāciju valodu L1. Lai arī abu šo valodu definīcijās izmantoti dažādi termini, iespējams pamanīt sakarības starp tiem (skatīt 2.2. tabulu, kurā katram no augstāk minētās valodas P^- alfabēta definīcijas deviņiem punktiem (6 pamata punkti un 3 sepītā punkta apakšpunkti) piekārtots kāds no transformāciju valodas L1 jēdzieniem).

4.2. tabula. **Valodu P- un L1 jēdzienu sasaiste**

P^- jēdziens	**L1 jēdziens**
Tipu kopa S	Kopa C ∪ {*Integer*, *Real*, *String*, *Boolean*}, kur C – visu izmantojamā metamodeļa klašu kopa
Tukšais jēdziens ⊥	*Boolean* tipa vērtība *false*
Pārējie loģiskie saikļi	Tiks interpretēti kontekstā
Eksistences kvantors $\exists_s$, kur s∈C, kur C – visu izmantojamā metamodeļa klašu kopa	Komanda „first”
Universālais kvantors $\forall_s$, kur s∈C, kur C – visu izmantojamā metamodeļa klašu kopa	Tiks interpretēts, pārveidojot šo kvantoru saturošo izteiksmi formā, kas satur eksistencs kvantoru
Vienādības simbols $=_s$, kur s∈{*Integer*, *Real*, *String*, *Boolean*}	Komanda „var”

Vienādības simbols $=_s$, kur $s \in C$, kur C – visu izmantojamā metamodeļa klašu kopa	Komanda „pointer”
Mainīgo kopa Vs, kur $s \in$ {*Integer*, *Real*, *String*, *Boolean*}	Primitīvo datu tipu mainīgie, kas deklarēti ar atslēgas vārdu „var”
Mainīgo kopa Vs, kur $s \in C$, kur C – visu izmantojamā metamodeļa klašu kopa	Norādes uz klašu objektiem, kas deklarētas ar atslēgas vārdu „pointer”
Palīgsimboli „(” un „)”	Simboli „(” un „)”, kā arī tiks interpretēti kontekstā
Viena argumenta funkcionālais simbols	Klases atribūts
Tipu konstantes	Primitīvo datu tipu konstantes
Binări predikātu simboli $P(t_1, t_2)$, kur $t_1, t_2 \in C$, kur C – visu izmantojamā metamodeļa klašu kopa	Komanda „link”

Kad ir skaidri nodefinēts, kas tiek saprasts ar valodas P^- formulu, kā arī noteiktas sakarības starp valodu P^- un L1 terminiem, iespējams noformulēt teorēmu.

Teorēma. Katrai predikātu valodas P^- formulai eksistē tāda transformāciju valodas L1 *begin-end* izteiksme, kura pieņem tieši tādu pat loģisko vērtību, kāda ir dotās formulas loģiskā vērtība.

Pierādījums. Šai teorēmai autors piedāvā konstruktīvu pierādījumu, tas ir, tiks ne tikai pierādīts, ka katrai formulai šāda komandu virkne eksistē, bet arī tiks parādīts, kā tādu uzkonstruēt.

Tātad, lai pierādītu doto teorēmu, nepieciešams katrai no valodas P^- formulu klasēm uzrādīt tādu *begin-end* izteiksmi valodā L1, kas, vadoties pēc augstāk aprakstītās shēmas, kā *begin-end* izteiksmēm tiek piekārtotas patiesuma vērtības, katrā konkrētā gadījumā pieņem tādu pat patiesuma vērtību kā katra konkrētā formula, kas pieder aplūkojamajai formulu klasei. Lai to izdarītu, nepieciešams ieviest divas funkcijas:

1) expr: <P^- formula> → <L1 *begin-end* izteiksme> – viena argumenta funkcija, kas dotajai valodas P^- formulai viennozīmīgi piekārto atbilstošu valodas L1 *begin-end* izteiksmi;
2) insert: <L1 *begin-end* izteiksme> × <*String*> → <L1 *begin-end* izteiksme> – divu argumentu funkcija, kas no pirmajā argumentā dotās valodas L1 *begin-end* izteiksmes iegūst jaunu valodas L1 *begin-end* izteiksmi, pieliekot sākotnējā izteiksmē trūkstošajās

vietās (pie „goto” komandām bez iezīmēm, kā arī pie neeksistējošajiem „else” zariem tām komandām, kurām drīkst būt „else” zari) tādu iezīmes vārdu, kāds padots otrajā parametrā.

Visas valodas P^- formulas klases un to atbilstošās valodas L1 *begin-end* izteiksmes aplūkojamas 2.3. tabulā (koda konstruēšanas procesā ieviestās iezīmes „unicalLabel”, „unicalLabelForA” un „endLabel”, kā arī ieviestās norādes „unicalPtrName1” un „unicalPtrName2” un mainīgie „unicalVarName1” un „unicalVarName2” uzskatāmi par unikāliem visas dotās procedūras vai funkcijas, kuras ķermenī tie atrodas, ietvaros). Papildus ieviestās norādes un mainīgie nodrošina sintaktiski korekta valodas L1 koda veidošanu, jo, piemēram, komandā „var” salīdzināšanas operatora kreisajā pusē valodā L0 (un līdz ar to arī valodā L1) drīkst atrasties vienīgi primitīvā datu tipa mainīgais, taču, kā redzams tabulā, šeit pēc būtības var atrasties patvaļīgs terms. Piešķirot šī terma vērtību primitīvā datu tipa mainīgajam un pēc tam salīdzināšanu veicot, izmantojot šo mainīgo, minēto problēmu iespējams apiet.

4.3. tabula. **L1 koda konstruēšana no predikātu valodas formulām**

F	**expr(F)**
$\bot$	**goto**;
$=_s(t_1,t_2)$, kur s∈{*Integer*, *Real*, *String*, *Boolean*}	**setVar** unicalVarName1=t_1; **setVar** unicalVarName2=t_2; **var** unicalVarName1==unicalVarName2;
$=_s(t_1,t_2)$, kur s∈C, kur C – visu izmantojamā metamodeļa klašu kopa	**setPointer** unicalPtrName1=t_1; **setPointer** unicalPtrName2=t_2; **pointer** unicalPtrName2==unicalPtrName2;
$P(t_1, t_2)$	**setPointer** unicalPtrName1=t_1; **setPointer** unicalPtrName2=t_2; **link** unicalPtrName1.P.unicalPtrName2;
A∧B	expr(A) expr(B)
A∨B	**insert**(expr(A),”unicalLabel”) **goto** endLabel; **label** unicalLabel; expr(B) **label** endLabel;
¬A	**insert**(expr(A),”unicalLabel”) **goto**;

	label unicalLabel;
$\exists_s x(A)$	**first** x:S **suchthat** **begin** expr(A) **end**;
$\forall_s x(A) \equiv \neg\exists_s x(\neg A)$	**first** x:S **suchthat** **begin** insert(expr(A),"unicalLabelForA") **goto**; **label** unicalLabelForA; **end else** unicalLabel; **goto**; **label** unicalLabel;

Jāievēro, ka gadījumā, kad F apzīmē tukšo jēdzienu, atbilstošā *begin-end* izteiksme satur vienu komandu – „goto;". Šāds gadījums valodā L1 ir atļauts un nozīmē tieši to pašu, ko atbilstošā valodas P^- formula – šāda L1 izteiksme ir aplama jebkurā gadījumā, neatkarīgi no nekādiem blakus apstākļiem (jo šī viena vadības nodošanas komanda nekad nevar sekmīgi izpildīties neeksistējošas iezīmes dēļ). Interesanti, ka valodas P^- pēdējās minētās formulu klases gadījumā atbilstošo valodas L1 *begin-end* izteiksmi ērtāk konstruēt, izmantojot formulas universālā kvantora formas aizvietošanu ar tai ekvivalento eksistences kvantora formu un pēc tam pielietojot jau iepriekš aprakstītos konstrukcijas paņēmienus.

Lai labāk izprastu konstrukcijās izmantoto funkciju nozīmi, 2.4. tabulā doti konstrukcijas piemēri katrai no valodas P^- formulu klasēm.

4.4. tabula. **L1 koda konstruēšana no predikātu valodas formulām – piemēri**

F	**expr(F)**
$\perp$	**goto**;
$=_{Integer}(x,17)$	**setVar** unicalVarName1=x; **setVar** unicalVarName2=17; **var** unicalVarName1==unicalVarName2;
$=_{Person}(p,q)$	**setPointer** unicalPtrName1=p; **setPointer** unicalPtrName2=q; **pointer** unicalPtrName1== unicalPtrName2;
father(p,q)	**setPointer** unicalPtrName1=p; **setPointer** unicalPtrName2=q; **link** unicalPtrName1.father.unicalPtrName2;

(father(p,q)∧ $_{Integer}$(age(p),18))	**setPointer** unicalPtrName1=p; **setPointer** unicalPtrName2=q; **link** unicalPtrName1.father.unicalPtrName2; **setPointer** unicalPtrName3=p; **attr** unicalPtrName3.age==18;
((father(p,q) ∧ $=_{Integer}$ (age(p), 18)) ∨ $=_{Integer}$ (age(q),18))	**setPointer** unicalPtrName1=p; **setPointer** unicalPtrName2=q; **link** unicalPtrName1.father.unicalPtrName2 **else** unicalLabel; **setPointer** unicalPtrName3=p; **attr** unicalPtrName3.age==18 **else** unicalLabel; **goto** endLabel; **label** unicalLabel; **setPointer** unicalPtrName4=q; **attr** unicalPtrName4.age==18; **label** endLabel;
¬$=_{Integer}$(age(p),18)	**setPointer** unicalPtrName1=p; **attr** unicalPtrName1.age==18 **else** unicalLabel; **goto**; **label** unicalLabel;

$\exists_{Person}$p ($=_{Integer}$(age(p),18))	**first** p:Person **suchthat**
	begin
	  setPointer unicalPtrName1=p;
	  attr unicalPtrName1.age==18;
	end;

$\forall_{Person}$p ($=_{Integer}$(age(p),18)) ≡ ¬$\exists_{Person}$p (¬$=_{Integer}$(age(p),18))	**first** p:Person **suchthat**
	begin
	  setPointer unicalPtrName1=p;
	  attr unicalPtrName1.age==18 **else** unicalLabelForA;
	  goto;
	  label unicalLabelForA;
	end else unicalLabel;
	goto;
	label unicalLabel;

Lai arī *begin-end* izteiksmju konstrukcija vairumā gadījumu ir induktīva, tomēr viegli redzēt, ka katras valodas P^- formulas klases gadījumā patiešām iespējams uzkonstruēt tādu valodas

L1 *begin-end* izteiksmi, kuras patiesuma vērtība ir vienāda ar dotās formulas patiesuma vērtību. Pierādījuma beigas.

Jāteic, ka patiesībā transformāciju valodā L1 ar *begin-end* izteiksmju palīdzību iespējams izdarīt vēl vairāk kā ar augstāk minētās predikātu loģikas palīdzību. Tas iespējams galvenokārt triju iemeslu dēļ [51]:

1) *begin-end* izteiksmēs iespējams operēt arī ar elementārajiem mainīgajiem;
2) *begin-end* izteiksmēs tiek norādīta komandu izpildes secība šablona piemērošanas laikā (tas ir, secība, kādā tiek apmeklētas klašu instances);
3) pēc šablona atbilstības konstatēšanas visiem tajā iesaistītajiem elementiem tiek piesaistītas vērtības, kas turpmāk var tikt izmantotas.

4.4. Modeļu transformāciju valodu Lx realizācija

Šajā nodaļā aprakstītas galvenās idejas, kā iespējams realizēt transformāciju valodas L0', L1, L2 un L3. Ko tad šajā kontekstā īsti nozīmē realizēt kādu transformāciju valodu? Tas nozīmē panākt to, lai programmu, kas uzrakstīta šajā valodā, būtu iespējams izpildīt. Citiem vārdiem runājot – izveidot līdzekli, ar kura palīdzību no dotajā transformāciju valodā uzrakstītas programmas teksta būtu iespējams iegūt MS Windows vidē izpildāmu datni.

Balstoties uz piedāvātajām realizācijas idejām, autors ir radījis valodu L0', L1, L2 un L3 kompilatorus un šajā nodaļā detalizēti aprakstījis kompilācijas gaitu un galvenās kompilācijas problēmas. Tāpat šajā nodaļā ikvienam lasītājam tiek piedāvāta iespēja pašam izmēģināt nokompilēt patvaļīgu transformāciju valodā L3 rakstītu programmu, lejupielādējot kompilatoru no Lx valodu saimes mājas lapas [53].

4.4.1. Valodu Lx realizācijas pamata ideja

Kā jau minēts nodaļā 4.1., transformāciju valodai L0 jau eksistē efektīva realizācija, proti, kompilators uz programmēšanas valodu C++, no kura jau tālāk ar C++ līdzekļiem iespējams iegūt MS Windows vidē izpildāmu datni. Tā kā transformāciju valodas L0', L1, L2 un L3 ir būvētas, balstoties uz valodu L0, tad visai loģiska šo valodu realizācija būtu realizācija caur transformāciju valodu L0, kura jau ir izpildāma (tādā nozīmē, kā aprakstīts augstāk). Tas arī ir galvenais doto valodu realizācijas princips – uzbūvēt valodā L0 kompilatoru no katras no šīm valodām uz valodu L0 un tālāk paļauties jau uz L0 realizācijas būtību. Taču, iedziļinoties šai problēmā vēl vairāk, jāsecina, ka vēl efektīvāk būtu nevis realizēt katru no valodām L0', L1, L2 un L3 kā kompilatoru no šīs valodas uz valodu L0, bet gan realizēt šo kompilāciju

pakāpeniski, katru no šīm valodām realizējot ar šīs valodas tiešo priekšteci valodu Lx saimē. Tas ir, radīt kompilatoru no valodas L0’ uz valodu L0, pēc tam radīt kompilatoru no valodas L1 uz valodu L0’ (kura šajā brīdī jau ir realizēta, līdz ar ko var uzskatīt, ka tādējādi arī valoda L1 jau kļūst realizējama), pēc tam radīt kompilatoru no valodas L2 uz valodu L1 un visbeidzot izveidot kompilatoru arī no valodas L3 uz valodu L2. Šādai realizācijai ir būtiska priekšrocība – tā ļauj katrā no kompilatoriem strādāt tikai ar tām valodas konstrukcijām, kas šai valodai ir pievienotas attiecībā pret iepriekšējo Lx valodu, visas pārējās konstrukcijas atstājot neskartas, nevis katrā kompilatorā apstrādāt pilnīgi visas izmaiņas, kas šai valodā notikušas, salīdzinot ar valodu L0. Un tā kā valodas Lx patiešām tika speciāli būvētas tādā veidā, ka katra nākamā valoda mantoja pilnīgi visas iepriekšējās valodas konstrukcijas, pievienojot klāt vēl savas specifiskās iezīmes, tad šāda veida doto valodu realizācija ir iespējama.

Šādu algoritmu, kāds aprakstīts iepriekšējā rindkopā, literatūrā pieņemts saukt par sāknēšanas (*bootstrapping*, [47]) algoritmu. Sāknēšanas princips balstīts uz ideju veidot kādas valodas kompilatoru uz citu valodu mērķa valodā. Tātad Lx transformāciju valodu saimes gadījumā tas nozīmētu izveidot valodas L0’ kompilatoru uz valodu L0, kas rakstīts pašā valodā L0, pēc tam izveidot valodas L1 kompilatoru uz valodu L0’, kas rakstīts valodā L0’, pēc tam izveidot valodas L2 kompilatoru uz valodu L1, kas rakstīts valodā L1, un visbeidzot izveidot valodas L3 kompilatoru uz valodu L2, kas rakstīts valodā L2. Tā kā valoda L0 pēc savām konstrukcijām un iespējām ir visu pārējo četru Lx saimes valodu apakškopa, tad, praktiski realizējot četrus augstāk minētos kompilatorus, tos visus pieļaujams rakstīt bāzes valodā L0. Šādi rīkojoties, sāknēšanas algoritma pamata ideja netiek pārkāpta, jo var uzskatīt, ka realizācija patiesībā notikusi katra kompilatora mērķa valodā, tikai kompilatora izveidē nav izmantotas visas valodas piedāvātās konstrukcijas.

Atšķirībā no tradicionālajām kompilatoru būves idejām [54], kurās par pamatu tiek ņemta programmu tekstuālo formu analīze, modeļu transformāciju valodu gadījumā sāknēšanas algoritma ideja ir ļoti parocīga, jo viegli realizējama modeļu kontekstā. Proti, katra programma kādā no modeļu transformāciju valodām uzskatāma par noteiktu modeli dotās transformāciju valodas metamodelī. Ja zināmi gan abu kompilācijas procesā iesaistīto (avota un mērķa) valodu metamodeļi, gan arī konkrētais avota valodas modelis, kas atbilst kompilējamajai programmai, tad kompilācijas uzdevums uztverams kā standarta modeļu transformācijas uzdevums – pārveidot modeli, kas dots vienā metamodelī, par modeli kādā citā metamodelī. Līdz ar to visu četru kompilatoru izveide var tikt reducēta uz modeļu

transformāciju izveidi, ņemot vērā iepriekšējā nodaļā precīzi nodefinētos visu Lx saimes transformāciju valodu metamodeļus.

Runājot par sāknēšanas algoritma realizācijas iespējamību, noteikti būtu jāsaprot arī tas, vai patiešām katru no programmām, ko iespējams uzrakstīt kādā no augstāka līmeņa transformāciju valodām, būs iespējams uzrakstīt arī zemāka līmeņa transformāciju valodā. Teorētiski ir iespējama situācija, kad kādā no šādu transformāciju valodu virknes valodām ir parādījusies konstrukcija, kas ne tikai atvieglo programmētāja darbu, bet arī piedāvā kādu būtiski jaunu iespēju programmu veidošanā, kas līdz šim nebija pieejama. Līdz ar to der pārbaudīt visas četras realizējamās transformāciju valodas L0', L1, L2 un L3, vai tās nepiedāvā šādas būtiski jaunas konstrukcijas.

4.4.2. Transformāciju valodas L0' realizācija

Transformāciju valoda L0', salīdzinot ar valodu L0, ir tikusi papildināta ar garu aritmētisko izteiksmju veidošanas iespēju. Tātad galvenais jautājums šajā kontekstā ir – vai jebkuru aritmētisku izteiksmi, kas pierakstāma valodā L0', iespējams nosimulēt arī valodā L0, tas ir, vai valodā L0 iespējams iegūt tādas aritmētiskas izteiksmes, kas kā viena izteiksme pierakstāma valodā L0', rezultātu? Tā kā valodā L0' aritmētiskajās izteiksmēs atļauts lietot vienīgi unārus un binārus aritmētiskos operatorus (tas ir operatorus, kuri pieprasa vienu vai divus operandus), tad skaidrs, ka katru šādu valodas L0' aritmētisku izteiksmi iespējams sadalīt pa vairākām binārām izteiksmēm, savukārt, gadījumos, kad valodā L0 netiek pieļauta pat binära izteiksme (piemēram, funkciju izsaukumos), šādu binäru izteiksmi var sadalīt divās daļās, atsevišķi aprēķinot katras daļas vērtību, piešķirot šo vērtību kādam mainīgajam un rezultātā veicot sākotnējās binārās izteiksmes operāciju šiem mainīgajiem (valodā L0 ar mainīgajiem binäras izteiksmes ir atļautas vienmēr). Tātad transformāciju valodā L0' esošās aritmētiskās izteiksmes vienmēr ir pierakstāmas arī ar valodas L0 līdzekļiem, līdz ar ko valodai L0' iespējams uzrakstīt kompilatoru uz valodu L0.

4.4.3. Transformāciju valodas L1 realizācija

Transformāciju valoda L1, salīdzinot ar valodu L0', ir tikusi papildināta ar šablonu definēšanas iespēju, tas ir, meklējot kādas klases kārtējo instanci, iespējams norādīt kaut kādus nosacījumus, kuriem šai instancei būtu jāizpildās. Jautājums – vai šāda iespēja realizējama arī valodā L0'? Tātad, ja dotas objektu meklēšanas komandas „first" un „next" bez šablonu definēšanas bloka, iespējams sameklēt kārtējo dotās klases objektu un pēc tam mēģināt iziet cauri visiem nosacījumiem (komandām), kuriem valodas L1 gadījumā būtu

jāatrodas šablonu definēšanas blokā, pie kam tām komandām, kurām trūkst „else” zara (un kurām tāds vispār drīkst būt), iespējams pielikt klāt šādu zaru ar norādi attiecīgā gadījumā iet uz kaut kādu jaunu iezīmi. Līdzīgā veidā šo pašu jauno iezīmi iespējams pielikt arī pie tām „goto” komandām, kurām dotajā šablonu definēšanas blokā nebija pierakstīta neviena iezīme. Tādā gadījumā nonākšana līdz šai jaunajai iezīmei signalizētu par to, ka kāds no nosacījumiem nav izpildījies, līdz ar ko atrastais objekts nav derīgs, tādēļ nepieciešams sameklēt nākamo objektu un atkārtot procedūru. Savukārt, ja tiek iziets cauri visām nosacījumu komandām un līdz augstāk minētajai jaunajai iezīmei aiziets netiek, tad jāsecina, ka atrastais objekts ir izturējis šablona nosacījumus un tādējādi ir atzīstams par derīgu. Vispārīga kompilācijas shēma attēlota 3.1. tabulā.

4.5. tabula. **Šablonu bloka realizācijas princips**

L1	L0’
first <ptrName1>:<className> [**from** <ptrName2> **by** <roleName>] **suchthat** **begin** <command_1>; <command_2>; ... <command_n>; **end** [**else** <labelName>];	**first** <ptrName1>:<className> [**from** <ptrName2> **by** <roleName>] [**else** <labelName>]; **label** ___L_i; <command_1> [**else** ___L_i+1]; <command_2> [**else** ___L_i+1]; ... <command_n> [**else** ___L_i+1]; **goto** ___L_i+2; **label** ___L_i+1; **next** <ptrName1> [**else** <labelName>]; **goto** ___L_i; **label** ___L_i+2;
next <ptrName> **suchthat** **begin** <command_1>; <command_2>; ... <command_n>; **end** [**else** <labelName>];	**next** <ptrName> [**else** <labelName>]; **label** ___L_i; <command_1> [**else** ___L_i+1]; <command_2> [**else** ___L_i+1]; ... <command_n> [**else** ___L_i+1]; **goto** ___L_i+2; **label** ___L_i+1; **next** <ptrName> [**else** <labelName>]; **goto** ___L_i; **label** ___L_i+2;

Šajā shēmā „else” zari ar jaunajām iezīmēm klāt likti, protams, tiek tikai tām šablonu definēšanas bloka komandām, kurām vispār šāds zars ir iespējams un kurām šāds zars jau nav definēts. Ar iezīmēm formātā „___L_i” šeit saprot jaunas iezīmes, kādas nedrīkst būt transformāciju valodā L0’ rakstītā programmā. Savukārt, ja kāda no šablonu definēšanas bloka komandām atkal izrādās objektu meklēšanas komanda ar „suchthat” bloku, to valodā L0’ atkal nepieciešams izvērst, izmantojot valodas L0’ kompilatora līdzekļus, un tā rekursīvi veikt šo izvēršanu „suchthat” blokā patvaļīgā dziļumā.

Tātad pēc šādas analīzes iespējams atbildēt uz augstāk uzdoto jautājumu – transformāciju valodā L1 rakstītu meklēšanas komandu ar šablonu definēšanas bloku iespējams pierakstīt arī valodā L0’ ar šīs valodas piedāvātajām iespējām, līdz ar ko valodai L1 iespējams uzrakstīt kompilatoru uz valodu L0’.

4.4.4. Transformāciju valodas L2 realizācija

Transformāciju valoda L2, salīdzinot ar valodu L1, ir tikusi papildināta ar ciklu veidošanas iespēju. Kā jau noprotams, valodā L1 bija iespējams veidot ciklus, izmantojot komandas „goto” un „label”, kā arī vairāku komandu „else” zarus, taču noteikti nepieciešams pārliecināties par to, vai valodā L1 patiešām iespējams izveidot jebkuru ciklu, kas iespējams valodā L2.

Cikls transformāciju valodā L2 tiek saprasts tādā veidā, ka, pēc kārtas atlasot objektus no kaut kādas dotas objektu kopas, ar šiem objektiem tiek veiktas kaut kādas darbības. Aplūkojamo objektu kopa iegūstama vai nu, ņemot visus kādas klases objektus, vai arī, ņemot tos kādas klases objektus, kas atbilst definētam šablonam. Tātad šāda veida objektu kopu iespējams iegūt arī valodā L1, kurā jau ir piedāvāta šablonu definēšanas iespēja. Valodas L2 cikla ķermenī atļautas patvaļīgas L2 komandas, līdz ar ko valodā L1 šīs komandas var izpildīt uzreiz pēc tam, kad ir atrasts kārtējais apstrādājamais objekts. Pēc šo komandu izpildes tad būtu jāatrod nākamais vajadzīgais objekts un izpilde jāatkārto. Šādi rīkojoties, kamēr vēl ir objekti, ko apstrādāt, iespējams simulēt valodas L2 ciklu. Šīs simulācijas shēma attēlota 3.2. tabulā.

4.6. tabula. **„Foreach" cikla realizācijas princips**

L2	L1

foreach <ptrName> **:** <className>
[**suchthat**
begin
  <command_1>;
  <command_2>;
  ...
  <command_n>;
end]
do
begin
  <do_command_1>;
  <do_command_2>;
  ...
  <do_command_k>;
end;
	first <ptrName> **:** <className>
	[**suchthat**
	begin
	<command_1>;
	<command_2>;
	...
	<command_n>;
	end]
	else __L_i;
	label __L_i+1;
	<do_command_1>;
	<do_command_2>;
	...
	<do_command_k>;
	next <ptrName> [**suchthat**
	begin
	<command_1>;
	<command_2>;
	...
	<command_n>;
	end]
	else __L_i;
	goto __L_i+1;
	label __L_i;

Tātad šajā cikla realizācijā tiek izveidotas divas objektu meklēšanas komandas, vienu reizi tiek atrasts pirmais objekts un katru nākamo reizi tiek meklēti nākamie objekti. Ja kādā no meklēšanām jauns objekts vairs netiek atrasts, tas nozīmē, ka visi objekti jau ir apstrādāti un līdz ar to cikls ir beidzies. Savukārt, ja objekts tiek atrasts, tiek izpildītas cikla ķermenī esošās komandas. Ja valodā L2 rakstītajā programmā būtu kāds cikls bez šablonu definīcijas bloka (kas var būt, jo šis bloks cikla komandai nav obligāts), tad arī attiecīgajā valodas L1 programmā nevienai no objektu meklēšanas komandām netiktu veidots „suchthat" bloks (kas arī ir neobligāts). Ja valodas L2 cikla ķermenī vai „suchthat" blokā būtu vēl kāda cikla komanda, to, protams, arī nāktos analoģiski izvērst ar valodas L1 kompilatora līdzekļiem.

Tātad jebkuru transformāciju valodā L2 pierakstāmu ciklu var pierakstīt arī ar valodas L1 līdzekļiem, līdz ar ko valodai L2 iespējams uzrakstīt kompilatoru uz valodu L1.

4.4.5. Transformāciju valodas L3 realizācija

Transformāciju valoda L3, salīdzinot ar valodu L2, ir tikusi papildināta ar zarošanās iespēju. Valodā L2 zarošanās problēmas tika risinātas līdzīgā veidā, kā valodā L1 tika risinātas cikla problēmas – izmantojot valodas L0 piedāvātos līdzekļus „goto", „label" un vairāku komandu „else" zarus. Tomēr, tāpat kā valodā L2 ciklu realizācijas gadījumā, arī šajā gadījumā jāpārliecinās, vai jebkuru zarošanās komandu, ko iespējams pierakstīt transformāciju valodā L3, iespējams realizēt arī valodā L2.

Zarošanās komanda valodā L3 sākas ar nosacījumu bloka komandām. Valodā L2 arī visas šīs komandas var izpildīt vienu pēc otras un beigās pārliecināties, ka vadība ir veiksmīgi izgājusi tām cauri un tās izpildījusi. Līdzīgi kā valodas L1 realizācijas gadījumā, arī šajā gadījumā tām komandām, kurām atļauti „else" zari, bet kurām tie nav definēti, būtu jādefinē savi „else" zari ar norādi attiecīgajā gadījumā pāriet uz jaunu iezīmi, tādējādi pārtverot gadījumus, kad atbilstošais L3 zarošanās komandas bloks iegūst aplamu patiesuma vērtību. Līdzīgi – „goto" komandām, kurām nav pierakstīta iezīme, jāpieraksta šī jaunā iezīme. Pēc šo komandu izpildes valodā L2 notiktu to komandu izpilde, kas definēta zarošanās komandas „then" blokā, savukārt, ja vadība nokļūtu pie augstāk minētās iezīmes, tad tiktu izpildītas tās komandas, kas definētas atbilstošās zarošanās komandas „else" blokā. Kompilācijas shēma attēlota 3.3. tabulā.

4.7. tabula. **Zarošanās komandas realizācijas princips**

L3	L2
if **begin** <if_command_1>; <if_command_2>; ... <if_command_n>; **end** **then** **begin** <then_command_1>; <then_command_2>;	<if_command_1> [**else** _L_i]; <if_command_2> [**else** _L_i]; ... <if_command_n> [**else** _L_i]; <then_command_1>; <then_command_2>; ... <then_command_k>; **goto** _L_i+1; **label** _L_i; [<else_command_1>;

```
   ...
   <then_command_k>;
end
[else
begin
   <else_command_1>;
   <else_command_2>;
   ...
   <else_command_l>;
end];
```

```
<else_command_2>;
...
<else_command_l>;]
label _L_i+1;
```

Ja kādā no trijiem zarošanās komandas blokiem („if”, „then” vai „else”) tiktu sastapta vēl kāda zarošanās komanda, arī to nāktos realizēt valodā L2 un līdzīgā veidā iet rekursīvi tik dziļi kādā no šiem blokiem, cik nepieciešams, lai visas zarošanās komandas pārveidotu par attiecīgajām valodas L2 komandām. Ja zarošanās komandai nebūtu definēts „else” zars (kas var būt, jo šis zars zarošanās komandai nav obligāts), tad arī valodā L2 netiktu rakstītas šīs komandas.

Tātad jebkuru zarošanās komandu, kas iespējama valodā L3, var realizēt ar valodas L2 līdzekļiem, līdz ar ko valodai L3 iespējams uzrakstīt kompilatoru uz valodu L2.

4.5. Valodu saimes Lx lietojumi

Uz augstākajām metamodeļu transformāciju valodu saimes Lx valodām, kas būvētas uz valodas L0 bāzes, iespējams paskatīties vismaz no diviem dažādiem aspektiem, iegūstot divus valodu saimes lietojumus. No vienas puses šajās valodās iekļauti tie papildus valodas primitīvi, kas bija trūkuši valodā L0, lai to varētu izmantot kā pilnvērtīgu līdzekli modeļu transformēšanas procesā ikdienas vajadzībām. Valoda L3 satur šos primitīvus un tādējādi ir lietojama kā ierindas programmēšanas valoda modeļu transformēšanai, piemēram, izstrādājot transformāciju bibliotēku rīku būves platformai GRAF vai lietojot to kā bāzes valodu sīktransformāciju rakstīšanas procesā.

No otras puses, veidojot augstākās Lx saimes valodas, veiksmīgi ticis ievērots aditivitātes princips, tas ir, katrā nākamajā valodā vien pieliktas klāt kādas jaunas iespējas, taču nav atņemtas vai mainītas vecās. Līdz ar to arī valoda L3 vēl ir pietiekami vienkārša, lai varētu kalpot par bāzes valodu kādas citas – vēl augstākas – valodas realizācijas procesā, izmantojot šo pašu sāknēšanas procesu, kas izmantots valodas L3 realizācijā. Tā kā valodā L3 jau realizēta virkne iespēju, kas visticamāk būtu sastopamas arī vēl augstāka līmeņa valodās, tad

šāda kompilatora uzrakstīšana valodā L3 būtu ievērojami vieglāka nekā, kompilējot augstākā līmeņa valodu uzreiz uz valodu L0. Šis lietojums arī ir eksperimentāli pārbaudīts, realizējot ļoti augsta līmeņa grafisku modeļu transformāciju valodu MOLA [21], izveidojot tās kompilatoru uz valodu L3. Pats kompilators arī rakstīts valodā L3, tādējādi pieturoties pie sāknēšanas algoritma principa. Modeļu transformāciju valoda MOLA tiek plaši lietota dažādu veidu modeļu transformāciju uzdevumos gan Latvijā, gan arī citur pasaulē, tostarp ES 6. ietvara projektā ReDSeeDS [55]. Sīkāk par valodas MOLA realizāciju caur valodu L3 iespējams izlasīt [44].

5. Skati uz metamodeli

Kā jau minēts nodaļā 3.3., otra lieta papildus augstāka līmeņa modeļu transformāciju valodas izstrādei, kas atvieglotu sīktransformāciju rakstīšanas procesu platformas GRAF gala lietotājam (rīka būvētājam), ir iespēja kaut kādā veidā orientēties milzīgajos platformas komponenšu metamodeļos. Tā kā pašreizējā platformas realizācijā iespējams darboties tikai ar vienu repozitoriju, tad neatkarīgi no tā, vai dažādie metamodeļi, kas ietilpst platformas sastāvā, ir vai nav loģiski kaut kādā veidā saistīti, tie tomēr tehniski atrodas visi kopā vienā repozitorijā. Gala lietotājam, kurš vēlas rakstīt transformāciju kaut kādam mazam specifiskam apgabalam, nebūtu jāpiesārņo galva ar visiem šiem liekajiem metamodeļu fragmentiem. Tāpat iespējams, ka gala lietotājs vēlas strādāt ar citiem terminiem, nevis tiem, kas paredzēti reizēm ļoti tehniskajos dziņu vai rīku definēšanas metamodeļos.

Lai nodrošinātu šāda veida metmodeļu pārvaldību, šajā nodaļā ieviests jēdziens par skatu uz metamodeli un demonstrētas iespējas strādāšanai (modeļu pārvaldīšanai) caur to. Kā jau noprotams pēc nosaukuma, skats ir līdzeklis, ar kura palīdzību repozitorijā esošos datus (modeli) iespējams aplūkot citādi, nekā to piedāvā šai pat repozitorijā esošais metamodelis. Ļoti lielam uzdevumu klāstam ar to bieži vien arī pietiek, lai gala lietotājs spētu uzrakstīt savas sīktransformācijas, vadoties pēc datiem, ko viņš redz caur šāda veida skatu. Ideālā gadījumā gan varētu vēlēties caur skatu ne tikai skatīties uz datiem, bet arī tos rediģēt. Šāda vēlme gan daļēji atrodas ārpus tā mehānisma, kurš aprakstīts tālākajās šīs nodaļas apakšnodaļās, lai arī nobeigumā atrodams īss spriedums par to, cik lielā mērā tas tomēr var tikt īstenots, balstoties uz aprakstīto mehānismu.

5.1. Ievads

Viens no galvenajiem relāciju datu bāžu un datu glabāšanas principiem ir saistīts ar datu dublēšanās (*redundancy*) minimizāciju. Katrai informācijas vienībai jātiek glabātai ne vairāk kā vienā vietā, lai, pirmkārt, tiktu taupīta atmiņa, bet, otrkārt (kas daudz būtiskāk) – lai atvieglotu datu integritātes nodrošināšanu. Tomēr dažādās lietotnēs bieži vien ir nepieciešams aplūkot datus no dažādiem skatu punktiem. Šim mērķim relāciju datu bāzēs ir ieviests skatu mehānisms.

Ja izvēlamies datus glabāt nevis relāciju datu bāzē, bet uz metamodeļiem balstītā repozitorijā, galvenie principi paliek nemainīgi. Viena no plašāk zināmajām metamodelēšanas valodām ir MOF (*Meta-Object Facility* [7]). Tā ir ļoti izteiksmīga valoda, lai gan tās eksotiskākās

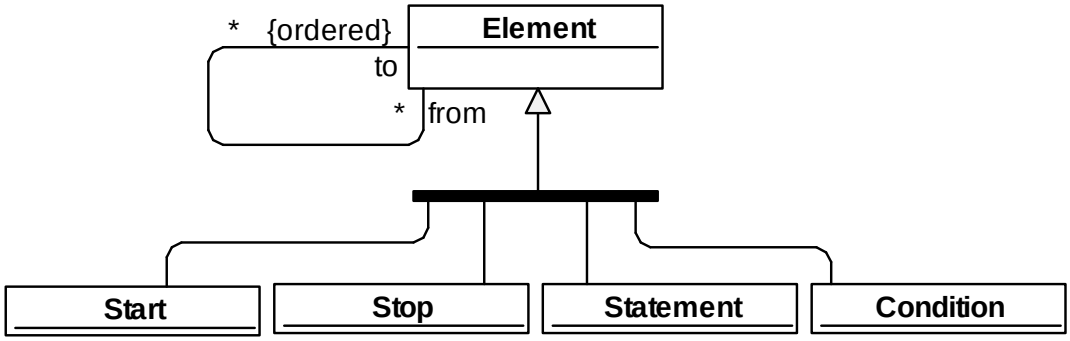
iespējas praksē tiek lietotas diezgan reti. Tāpēc šajā darbā aplūkota tikai būtiskākā un biežāk lietotā MOF daļa – EMOF (*Essential MOF*). Ja EMOF pilda metamodeļa funkciju, tad katrs EMOF modelis var tikt, savukārt, uzskatīts par metamodeli, kuram pašam var tikt veidoti modeļi nākamajā līmenī. Modeļu transformāciju valodas pārsvarā balstās uz diviem metamodelēšanas līmeņiem; šajā gadījumā par metamodeli tiktu uzskatīts EMOF modelis, bet par modeļiem – šī metamodeļa modeļi.

Datu dublēšanās samazināšanas princips tiek uzskatīts par acīmredzamu, būvējot metamodeļus konkrētiem problēmu apgabaliem. Tomēr praktiskos lietojumos parādās vajadzības pēc dažādiem skatiem uz datiem. Labs piemērs šajā sakarā varētu būt grafisku rīku būves lauciņš (piemēram, tādi rīki kā MetaEdit+ [1], Microsoft DSL Tools [2], Eclipse GMF [3], METAclipse [4] vai arī šajā darbā par pamatu ņemtais GRAF [31]). Šajā sfērā bieži vien ierasta prakse ir uzturēt vairākus metamodeļus, katrs no kuriem kodē apmēram vienu un to pašu informāciju dažādiem mērķiem.

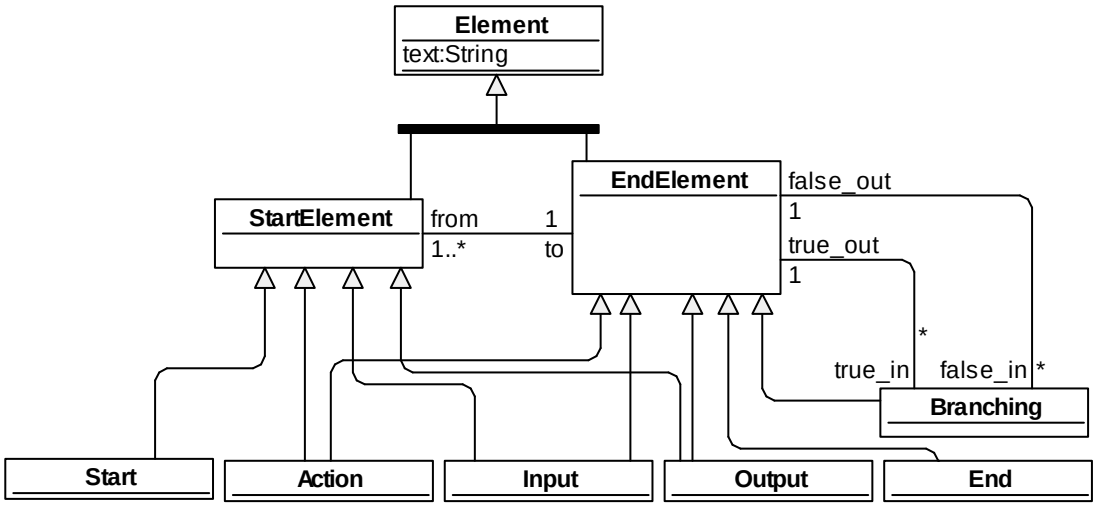
Piemēram, ja vēlamies aplūkot problēmu apgabalu, kas apraksta blokshēmas, mēs varam tam uzzīmēt dažādus metamodeļus. Sākot par šo problēmu domāt pirmo reizi, mēs, iespējams, sākotnēji nonāktu pie apmēram tāda metamodeļa kā redzams attēlā 5.1.a. Domājot tālāk, mēs varam veikt dažādus uzlabojumus un nonākt līdz attēlā 5.1.b redzamajam metamodelim. No otras puses, dažos gadījumos varbūt mums varētu arī pietikt ar metamodeli, kas apraksta visu (attēls 5.1.c). Ir skaidrs, ka katru konkrētu blokshēmu mēs varam nokodēt kā instanci jebkuram no minētajiem metamodeļiem. Citiem vārdiem sakot – mēs varam palūkoties uz jebkuru blokshēmu caur jebkuru no šiem metamodeļiem kā brillēm. Lai atbildētu uz jautājumu, kurš metamodelis mūsu specifiskajām vajadzībām ir piemērotākais, ir nepieciešams saprast konkrētā gadījuma kontekstu.

Ļoti izplatīta prakse grafisku rīku būvē ir uzturēt tā saukto domēna metamodeli, kas apraksta jēdzienu abstrakto sintaksi, un prezentāciju metamodeli, kurš apraksta šo jēdzienu konkrēto sintaksi. Ja gan domēna metamodelis ar tā instancēm, gan arī prezentācijas metamodelis ar tā instancēm tiek fiziski glabāts atmiņā vai kur citur, šos datus nepieciešams savā starpā sinhronizēt. Tas daudzos gadījumos var būt pietiekami sarežģīts uzdevums. Tāpat šajā gadījumā vērojama arī datu dublēšanās samazināšanas principa pārkāpšana. Līdz ar to šis gadījums var kalpot par labu piemēru skatu izmantošanai – prezentācijas metamodelis var kalpot par bāzes metamodeli, bet domēna metamodelis var pildīt skata funkciju (vai arī otrādi – atkarībā no konkrētās situācijas).

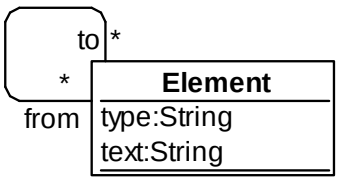
Otrs skatu izmantošanas piemērs varētu būt vienkāršota universitātes ontoloģija. Tajā var gribēt definēt atvasināto klasi „Good Student” no ontoloģijā esošas klases „Student”, nosakot, ka students pieder klasei „Good Student” tad un tikai tad, ja viņš ir savācis noteiktu skaitu kredītpunktus un nokārtojis visus obligātos kursus.



a) ļoti vienkāršs blokshēmu metamodelis



b) sarežģītāks blokshēmu metamodelis



c) vēl cita blokshēmu metamodeļa versija

5.1. attēls. **Blokshēmu metamodelis – dažādas pieejas**

Parasti visa nepieciešamā informācija mums tiek piedāvāta vai nu viena liela vai vairāku mazu metamodeļu formā. Tomēr ne vienmēr ir acīmredzams, kuras no metamodeļa daļām mums ir svarīgas konkrēta uzdevuma veikšanai vai kuras daļas nepieciešams interpretēt citādi, lai tās padarītu lietojamas konkrētiem mērķiem. Mums prātā var būt konkrētā

aplūkojamā priekšmetu apgabala mentāls tēls, kuru mums vajadzētu spēt attēlot uz repozitorijā esošo metamodeli. Tas ne vienmēr ir vienkāršs uzdevums.

Apzinoties šo konktekstu, varam identificēt divus galvenos uzdevumus:

1. izdomāt metodi, kā definēt šādus skatus EMOF modeļiem bez datu dublēšanās;
2. atrast veidu, kā ļaut modeļu transformāciju valodām strādāt ar šādiem skatiem.

Tālāk šajā nodaļā mēģināts rast atbildes uz abiem šiem jautājumiem. Skata definīcija aprakstošā veidā dota nodaļā 5.2. Tā sastāv no divām daļām, katrai no kurām veltīta viena nākamā līmeņa apakšnodaļa. Tā kā izvairīšanās no datu dublēšanās tiek uzskatīta par pamata principu, kas jāņem vērā skatu definēšanā, ne visa informācija tiks reāli glabāta atmiņā. Līdz ar to nepieciešams veikt kādas darbības, lai padarītu šos datus kaut kādā veidā tomēr pieejamus modeļu transformāciju valodām. Šie jautājumi dziļāk aplūkoti nodaļā 5.3. Nodaļā 5.4. pieeja demonstrēta uz konkrēta piemēra, bet nodaļā 5.5. spriests par aplūkotā skatu mehānisma iespējamajiem pielietojumiem grafisku rīku būves lauciņā. Visbeidzot, nodaļā 5.6. aplūkoti pasaulē zināmi pētījumi, kuros risinātas līdzīgas problēmas.

5.2. Skata definēšana

Šajā apakšnodaļā tiek ieviests jēdziens par skata metamodeli. Tāpat ieskicēts arī vispārējs mehānisms skatu būvēšanai uz patvaļīgiem metamodeļiem.

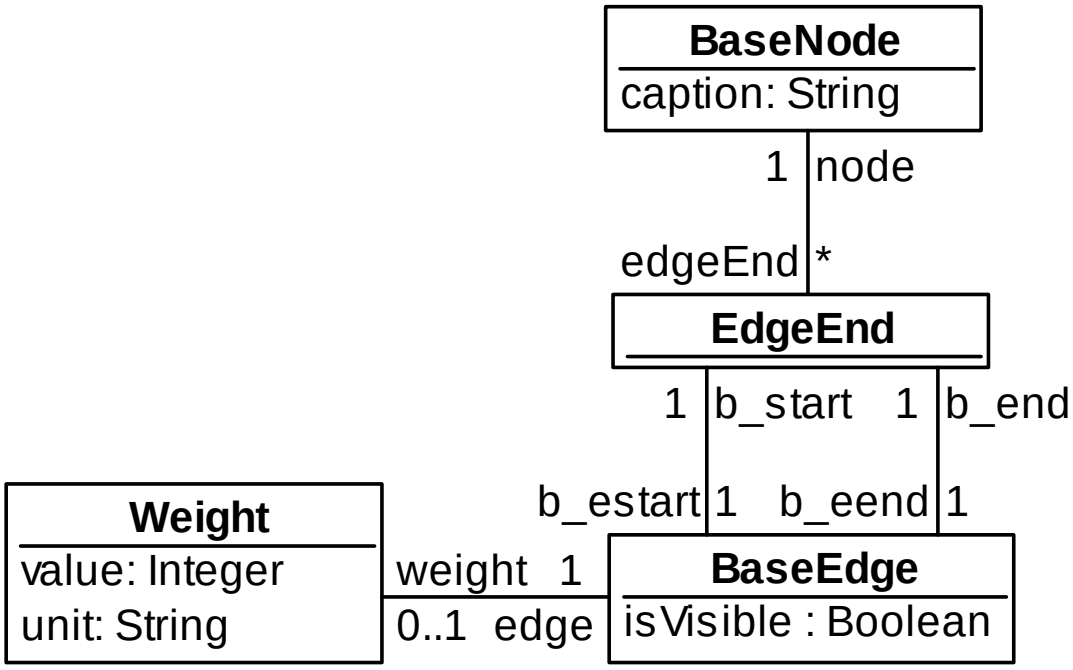
5.2.1. Pirmā skatu definēšanas daļa – attiecība UZ

Pamata ideju skaidrošanai par piemēru tiks ņemts orientēta grafa metamodelis (skatīt attēlu 5.2.). Tas kalpos par bāzes metamodeli, uz kura tiks būvēts skats.

Pieņemsim, ka mēs vēlamies uzbūvēt skatu saskaņā ar šādiem nosacījumiem:

1. mēs vēlamies skatā redzēt visas virsotnes;
2. mēs vēlamies skatā redzēt tikai tās šķautnes, kas skaitās redzamas (tās, kuru atribūta “isVisible” vērtība ir *true*);
3. mēs nevēlamies redzēt klasi “EdgeEnd”, bet tās vietā savienot šķautnes ar virsotnēm „pa tiešo” (izmantojot divas asociācijas – katram šķautnes galam savu);
4. mēs vēlamies šķautnes svaru parādīt kā šķautnes atribūtu „label”, kurā būtu savienota svara vērtība ar mērvienību. Klase „Weight” līdz ar to mums skatā nav vajadzīga;
5. mēs vēlamies pārsaukt dažus jēdzienus citādi:
    a. virsotņu un šķautņu klases sauks attiecīgi “ViewNode” un “ViewEdge”;

b. virsotnes „caption” tiks pārsaukts par „name”.

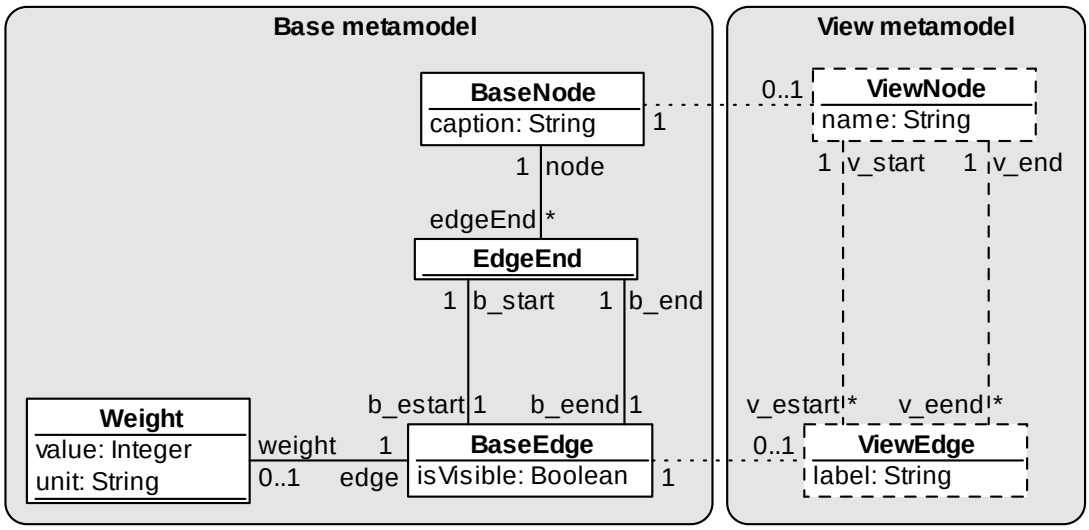


5.2. attēls. **Orientēta grafa metamodelis – bāzes metamodelis**

Kad esam šādā veidā nodefinējuši savas vēlmes, ir iespējams uzkonstruēt metamodeli šim pašam priekšmetu apgabalam (orientētajiem grafiem), kas tās ņem vērā (skatīt attēlā 5.3. redzamo metamodeli, kura klases un asociācijas iezīmētas ar raustītu līniju). Šī skata metamodeļa klases ViewNode un ViewEdge tiek definētas UZ bāzes klasēm BaseNode un BaseEdge (attiecība UZ attēlā 5.3. attēlota kā punktota saite). Tāpat skata metamodelī ieviestas divas jaunas asociācijas un divi jauni atribūti.

Ir būtiski tas, ka katrai skata metamodeļa klasei atbilst tieši viena bāzes metamodeļa klase, kurai tā ir kā *pieder*. Līdz ar to izteiciens, ka „klase A ir definēta UZ klases B” jāsaprot tādā veidā, ka mums ir definēta funkcija UZ no visu skata klašu kopas uz visu bāzes klašu kopu. Šī skata īpašība vēlāk tiks izmantota. Tagad katrs orientētais grafs, kas var tikt nokodēts kā bāzes metamodeļa instance, var arī tikt nokodēts kā skata metamodeļa instance. Tādējādi mēs varam uz katru grafu, kas nokodēts bāzes kodējumā, paskatīties caur skatu, ko piedāvā izmantotais skata metamodeli, kas sastāv tikai no divām klasēm. Jebkuru mūsu rīcībā esošu bāzes metamodeļa instanci mēs varam pārtransformēt par atbilstošu skata metamodeļa instanci, ja vien zinām nosacījumus, pēc kādiem šis skats būvēts (šajā piemērā – tie pieci nosacījumi, kas minēti šīs apakšnodaļas sākumā). Jāpiebilst, ka šī transformācija ne obligāti vienmēr strādā arī

pretējā virzienā (piemēram, atribūtu *label* var neizdoties vienmēr sadalīt pa atribūtiem *value* un *unit* bez papildus zināšanām).



5.3. attēls. **Orientēta grafa metamodelis – skata metamodelis piesaistīts bāzes metamodelim**

5.2.2. Otrā skata definēšanas daļa – metožu kopa

Tā kā izvairīšanās no datu dublēšanās bija deklarēts kā viens no galvenajiem skatu veidošanas iemesliem, tad skaidrs, ka nebūtu pieļaujams repozitorijā glabāt gan bāzes, gan skata metamodeli. Tā kā attēlā 5.3. skata metamodelis bija iezīmēts raustītām līnijām, tad reāli repozitorijā netiks glabāts ne šis metamodelis, ne arī tā instances. Tā kā reāli līdz ar to eksistē tikai bāzes metamodeļa instances, tad jēdziens par skatu (skatu definēšošie nosacījumi, kas minēti nodaļā 5.2.1.) kaut kādā veidā jāpiekārto reālajam bāzes metamodelim, kas atrodas repozitorijā. Galvenais mērķis šeit būtu spēt apstaigāt „virtuālo" skata modeli, neskatoties uz faktu, ka tas neatrodas repozitorijā. Tātad īstenībā apstaigāšana notiktu pa bāzes modeli, bet atstājot uz apstaigātāju tādu iespaidu, it kā tas darbotos ar vēlamo skata modeli.

Pamata ideja šī mērķa īstenošanā ir noteiktas metožu kopas izveide, piekārtojot šīs metodes tām bāzes metamodeļa klasēm, UZ kurām tiek definēta kāda skata metamodeļa klase. Izsaucot šīs metodes konkrētiem objektiem, būtu iespējams iegūt dažāda veida informāciju par skata klasēm/asociācijām/atribūtiem šī objekta kontekstā. Piemēram, apstaigājot attēlā 5.2. redzamā metamodeļa kādas instances grafu ar kādas modeļu transformāciju valodas palīdzību, mēs varam gribēt apstāties pie kāda konkrētas klases (piemēram, BaseNode) objekta p un pajautāt: „Vai es redzētu objektu p kā klases ViewNode instanci, ja skatītos uz šo pašu grafu caur skatu, kas redzams attēlā 5.3.?" Lai atbildētu uz šo jautājumu, ir jāzina nosacījumi, kas definē

klasi ViewNode un jāpiemēro tie klases BaseNode objektiem. Tā kā šie nosacījumi ir zināmi iepriekš (tie izveidoti jau skata metamodeļa definēšanas procesā), tad shēmu atbildei uz šāda tipa jautājumu arī iespējams iepriekš vienu reizi izveidot un tad lietot to katram konkrētās klases objektam p. Atbildes shēma tehniski tiks aprakstīta kā klases BaseNode metode, kas realizē klases ViewNode semantiku. Viens variants, kā varētu izskatīties šādas metodes prototips, ir šāds:

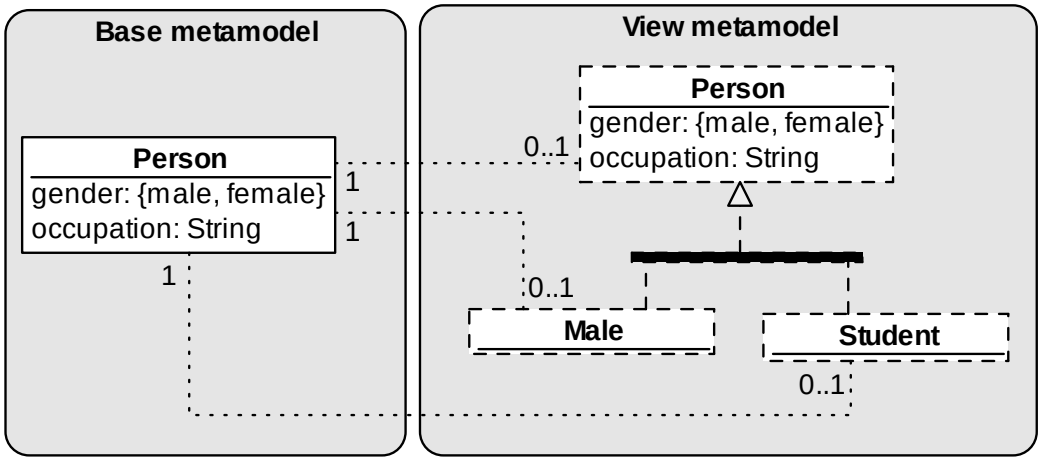
```
isInstanceOfViewClass(): Boolean;
```

Šī metode atgriež patiesu vērtību, ja objektam, kuram tā izsaukta, jāpieder virtuālajai skata klasei, kas definēta UZ dotās bāzes klases, un aplamu vērtību pretējā gadījumā. Tomēr, ja UZ vienas un tās pašas bāzes klases vēlamies definēt vairākas skata klases, metodē ir jāvar specificēt, par kuru skata klasi katrā konkrētajā gadījumā ir runa. Līdz ar to varētu būt vēlēšanās pamainīt metodi, kas atbild uz augstāk minēto jautājumu, tādā veidā, lai tā spētu tikt galā arī ar šo situāciju:

```
getClassName(): String;
```

Šādā veidā iespējams iegūt tās skata klases vārdu, kurai aplūkojamam objektam jāpieder (ja tam nav jāpieder nevienai skata klasei, metode var atgriezt, piemēram, tukšu simbolu virkni). Tomēr, iedziļinoties situācijā vēl vairāk, var rasties jautājums, vai kāds objekts nevarētu vienlaicīgi piederēt vairākām skata klasēm. Var iedomāties situāciju, kurā, piemēram, skatā vēlamies definēt kāda priekšmetu apgabala dziļāku hierarhiju, nekā tā ir definēta bāzē. Šāds piemērs aplūkojams attēlā 5.4. – bāzes metamodelī ir tikai viena klase „Person”, bet skata metamodelī parādās divas jaunas klases „Male” un „Student”. Objektu piederība šīm klasēm var tikt noteikta pēc atribūtu „gender” un „occupation” vērtībām, tādējādi radot iespējamību kādam objektam piederēt arī abām šīm klasēm vienlaicīgi (un faktiski turklāt arī to virsklasei). Var teikt, ka starp bāzes un skata klasēm pastāv 1:n tipa attiecība. Iepriekš minētā metode *getClassName* vairs nespēj tikt galā ar šādu situāciju, tāpēc atkal nepieciešams to pamainīt:

```
isInstanceOfViewClass(viewClassName: String): Boolean;
```



5.4 . attēls. **Viens bāzes objekts var piederēt vairākām skata klasēm vienlaicīgi**

Šādā redakcijā metode *isInstanceOfViewClass* ir pietiekami universāla, lai to varētu izmantot skatu definēšanas mehānismā vispārīgā gadījunā. Katra skata gadījumā šo metodi nepieciešams realizēt katrai bāzes klasei, UZ kuras definēta kāda skata klase. Esot šādai metodei, būs iespējams viegli iegūt atbildes uz visiem tāda tipa jautājumiem, kā minēts iepriekš. Taču, apstaigājot instanču grafu, varētu rasties arī cita veida jautājumiem, kuriem katram arī nepieciešama metode, kas uz to rod atbildi. Piemēram, nepieciešama metode kāda „virtuāla" atribūta vērtības noskaidrošanai, metode, kas atgriež objektu, ar kuru dotais objekts ir savienots caur kādu „virtuālu" saiti utt. Tāpat varētu būt vēlēšanās pēc pilnīgi cita metožu komplekta, kas paredzēts instanču grafa rediģēšanai, bet par to, kā jau minēts iepriekš, šajā darbā netiks īpaši runāts.

Grafa apstaigāšanas vajadzībām nepieciešamais metožu komplekts sastāv no piecām metodēm:

1. `isInstanceOfViewClass (viewClassName: String): Boolean;`

Šī metode atgriež patiesu vērtību tad un tikai tad, ja objektam, kuram tā izsaukta, jāpieder skata metamodeļa klasei *viewClassName*. Piemēram, aplūkotajā orientētu grafu gadījumā šī metode (tāpat kā pārējās tālāk minētās) tiktu definēta klasēm BaseNode un BaseEdge. Klases BaseEdge gadījunā metode atgrieztu patiesu vērtību tad un tikai tad, ja tā tiktu izsaukta ar vienīgā parametra vērtību vienādu ar „ViewEdge" un konkrētā objekta atribūta „isVisible" vērtība būtu patiesa. Klases BaseNode gadījumā metode būtu vēl vienkāršāka – tā atgrieztu patiesu vērtību vienmēr, kad izsaukta ar parametra vērtību vienādu ar „ViewNode" (un aplamu vērtību pārējos gadījumos), jo, veidojot skatu, mēs gribējām tā klasē ViewNode iekļaut pilnīgi visas grafa virsotnes.

2. `getViewAttrValue (viewAttrName: String): String;`

Šī metode atgriež konkrētā objekta skata atribūta *viewAttrName* vērtību. Mūsu grafu piemērā – ja metode izsaukta klases BaseEdge objektam ar parametra vērtību vienādu ar „label”, tad tai jāatgriež šim objektam piesaistītā klases „Weight” objekta atribūtu „value” un „unit” savienojums. Klases BaseNode gadījumā šai metodei jāatgriež bāzes atribūta „caption” vērtība, ja tā izsaukta ar parametru „name”.

3. `getFirstBaseObjectByViewLink(viewRoleName:String):Thing;`

Šī metode atgriež bāzes metamodeļa instanci, kurai jābūt sasniedzamai kā pirmajai skata izpratnē no instances, kurai šī metode izsaukta, ejot pa saiti ar lomas vārdu *viewRoleName* pretējā galā. Šeit un turpmāk ar tipu „Thing” tiks saprasta visu bāzes klašu virsklase, kas šajā gadījumā nozīmē, ka konkrētajai atgrieztā objekta klasei nav jābūt zināmai iepriekš.

4. `getNextBaseObjectByViewLink (prevBaseObject: Thing, viewRoleName: String): Thing;`

Šī metode atgriež nākamo sasniedzamo objektu pēc tiem pašiem nosacījumiem, kas aprakstīti iepriekšējā punktā.

5. `existsViewLinkToBaseObject (viewRoleName: String, baseObject: Thing): Boolean;`

Šī metode atgriež patiesu vērtību tad un tikai tad, ja starp objektu, kuram šī metode izsaukta, un objektu *baseObject* jāeksistē saitei skata izpratnē ar lomas vārdu *viewRoleName* objekta *baseObject* galā.

Ja mūsu rīcībā ir šīs piecas metodes, ir iespējams simulēt skata metamodeļa instanču grafa apstaigāšanu, īstenībā apstaigājot bāzes metamodeļa instanču grafu un vajadzības gadījumā izsaucot kādu no šīm metodēm. Piemēram, ja vēlamies uzsākt apstaigāšanu ar pirmās „virtuālās” klases „ViewNode” instances iegūšanu, varam to izdarīt šādi (rakstīts preidokodā):

```
n = getFirstObject (”BaseNode”)
while (n.isInstanceOfViewClass (”ViewNode”) != true)
   n = getNextObject (”BaseNode”, n)
```

Šeit tiek pieņemts, ka izmantojamajā modeļu transformāciju valodā eksistē komandas pirmā kādas klases objekta iegūšanai un nākamā šīs klases objekta iegūšanai. Lai atšķirtu skatu

definējošo metožu izsaukumus no modeļu transformāciju valodas komandām, šie metožu izsaukumi šeit un turpmāk tiks pasvītroti (šajā piemērā – tikai viens metodes izsaukums).

Varētu rasties jautājums, kā transformācijas rakstītājs var zināt, par kuru bāzes klasi katrā konkrētā gadījumā ir jāuzdot jautājums, tas ir, kuras klases attiecīgā metode jāizsauc. Lai uz to atbildētu, jāatceras svarīgā skata īpašība, kas tika minēta augstāk – katra skata klase tiek definēta UZ tieši vienas bāzes klases. Līdz ar to šīs zināšanas (funkcija ON no skata klašu kopas uz bāzes klašu kopu) uzskatāmas par daļu no skata definīcijas (otra daļa ir augstāk minētās metodes). Tā kā īstās bāzes klases atrašana neprasa cilvēka līdzdalību (tā var tikt atrasta, izmantojot funkciju UZ), tad šis process var tikt automatizēts. Tādējādi programmētājs varētu rakstīt programmu, domājot tikai skata metamodeļa terminos un lietojot pierastās modeļu transformāciju valodas komandas, it kā šis skata metamodelis reāli atrastos repozitorijā:

```
n = getFirstObject (”ViewNode”)
```

Tālāk notiktu automātisks kompilācijas solis, kurā no šīs vienas rindiņas tiktu uzģenerēts iepriekš redzētais kods, kas satur metožu izsaukumus. Līdz ar to programmētāji strādātu ar skata metamodeli kā ar īsto reālās pasaules (repozitorijā esošu) metamodeli, tomēr viņu rakstītās programmas vienalga būtu strādājošas, jo tiktu automātiski nokompilētas uz tādām, kas strādā ar bāzes metamodeli. Nodaļā 5.3. šis kompilācijas process demonstrēts detalizētāk.

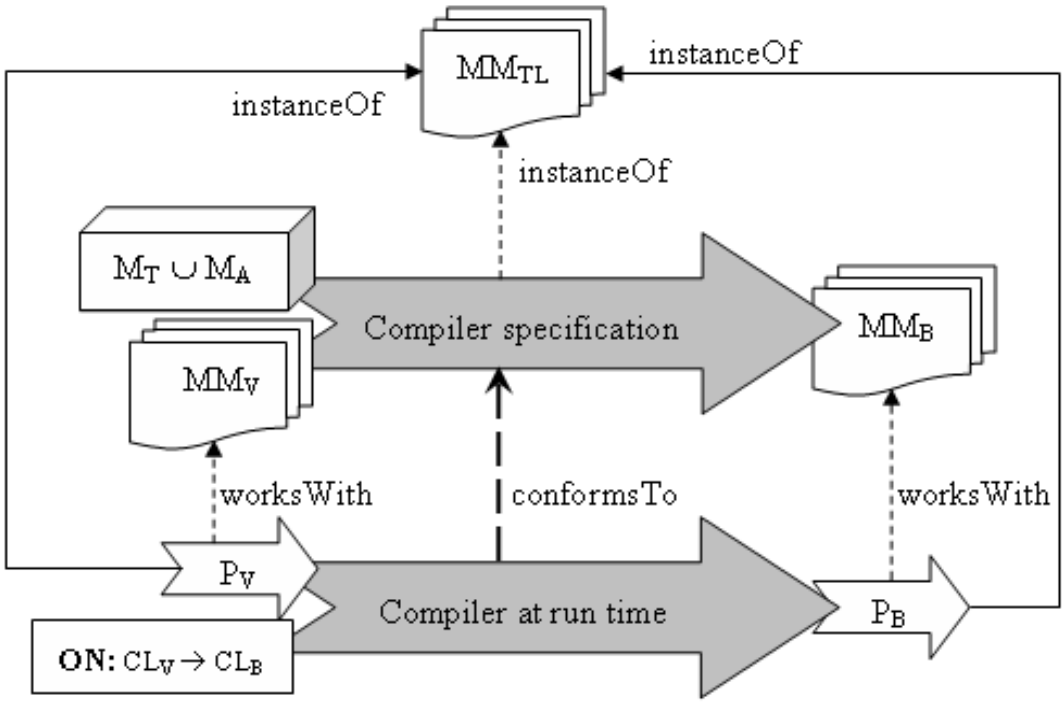
5.3. Ar skatiem strādājošu transformāciju kompilācija

Kā tika minēts iepriekš, skatu definēšanas mehānisma izstrādes mērķis ir ļaut programmētājiem rakstīt savas programmas, turot prātā vienīgi skata metamodli, kas izstrādāts kādām noteiktām vajadzībām. Šādas programmas tālāk var tikt pārkompilētas par programmām, kas strādā ar bāzes metamodeli, izmantojot universālu kompilatoru.

Šī procesa pamata shēma attēlota attēlā 5.5. Kompilators tiek specificēts kā modeļu transformāciju programma, kas spēj transformēt skata metamodeļa (MM_V) modeļus par atbilstošiem modeļiem bāzes metamodelī (MM_B). Papildus skatu metamodelim personai, kas izstrādā šādu kompilatoru, ir vēl jāzin bāzes klašu metodes. Līdz ar to šajā shēmā arī kopas M_T (instanču grafa apstaigāšanas metodes) un M_A (instanču grafa mainīšanas metodes, ja tādas tiktu sacerētas) kalpo par ieejas datiem kompilatora specifikācijā. Jāpiezīmē, ka par šīm metodēm šajā brīdī nepieciešamas zināt tikai to signatūras (un, protams, arī semantiku); kompilatora specifikācijai nav jāzin, kā šīs metodes realizētas. Līdz ar to pēc vienas metodes

specifikācijas tā var tikt realizēta dažādos atšķirīgos veidos bez nepieciešamības rediģēt kompilācijas procesu.

Izpildes laikā kompilators kā ieejas datus saņem programmu P_V, kas strādā ar skata metamodeli, un pārveido to par programmu P_B, kas strādā ar bāzes metamodeli. Kā papildus ieejas dati nepieciešama funkcija UZ, kas izpildes laikā nosaka atbilstību starp skata un bāzes klasēm. Visbeidzot attēla augšējā daļā redzams, ka eksistē modeļu transformāciju valoda (MM_{TL}), kurā pierakstīta kompilatoram ieejam padotā un izejā saņemtā programm. Šajā shēmā pieņemts, ka kompilatora specifikācija rakstīta tajā pašā modeļu transformāciju valodā, kurā abas minētās programmas, bet tā, protams, nepavisam nav obligāta prasība – šim mērķim var kalpot jebkura modeļu transformāciju valoda.



5.5. attēls. **Kompilācijas shēma**

Aplūkosim kādas ar skatu strādājošas programmas piemēru, lai padarītu šo shēmu saprotamāku. Pieņemsim, ka mēs vēlamies atrast visas šķautnes mūsu orientētajā grafā un katrai šķautnei parādīt paziņojuma logu (message box) ar tās virsotnes vārdu, kurā šī šķautne sākas. Programma P_V, kas veic šo uzdevumu, kā arī tās nokompilētais variants – programma P_B, kas strādā ar bāzes metamodeli – redzama tabulā 5.1. Tāpat kā iepriekš arī šeit bāzes klases metodes ir pasvītrotas, lai tās būtu vieglāk atšķiramas no pseidokodā pierakstītās modeļu transformāciju valodas komandām.

Tabulā 5.1. attēlota vien vispārīga shēma, kā tikko ieviestās metodes tiek izmantotas programmā, kas strādā ar bāzes metamodeli. Dažas specifiskas detaļas šajā shēmā nav attēlotas, lai nepadarītu to lieki sarežģītu. Piemēram, kas notiek, ja bāzes klasei „BaseEdge" nav nevienas instances, kas pieder skata klasei „ViewEdge"? Programmā P_V šādā gadījumā izpildīsies trešā rinda, kas liks vadībai tālāk pārlēkt uz iezīmi *done*, tādējādi pabeidzot programmu. Tai pat laikā izskatās, ka programma P_B šajā situācijā varētu iecikloties pirmajā rindā. Tā kā mēs nezinām skatu definējošās metodes "isInstanceOfViewClass" realizāciju klasei "BaseEdge", tad mēs patiesībā nevaram būt droši par to, vai cikls būs mūžīgs vai nē. Tikpat labi šajā metodē varētu būt pieņemts, ka vērtība NULL ir derīgs klases „ViewEdge" instances piemērs, uz kuru šai metodei tātad jāatgriež patiesa vērtība. Ja tas tā nav, tad, iespējams, nepieciešams nedaudz pamainīt uzģenerēto P_B kodu. Skaidrs, ka šādas detaļas ir jāņem vērā, izstrādājot kompilatoru, bet šajā piemērā tās ir apzināti ignorētas labākas uztveramības dēļ.

5.1. tabula. **Ar skatu un ar bāzi strādājošu programmu piermērs**

Skata programma (P_V)	**Bāzes programma (P_B)**

`e = getFirstObject (”ViewEdge”);`	`e = getFirstObject (”BaseEdge”);`
	`while (!e.isInstanceOfViewClass (”ViewEdge”))`
	`e = getNextObject (e, ”BaseEdge”);`

`label start;`	`label start;`
`if (e=NULL) goto done;`	`if (e=NULL) goto done;`
`n = getLinkedObject(e,”v_start”);`	`n=e.getFirstBaseObjectByViewLink (”v_start”);`
`s = n.name;`	`s = n.getViewAttrValue (”name”);`
`showMessage (s);`	`showMessage (s);`

`e = getNextObject (e,”ViewEdge”)`	`e = getNextObject (e, ”BaseEdge”);`
	`while (!e.isInstanceOfViewClass (”ViewEdge”))`
	`e = getNextObject (e, ”BaseEdge”);`

`goto start;`	`goto start;`
`label done;`	`label done;`

Programmu P_B kompilators ir uzģenerējis, zinot tikai metožu signatūras un nozīmi. Metožu realizētāja atbildība iekļauj sevī nodrošinājumu, ka pēc šādas shēmas kopilētās programmas P_V semantika atbilst sākotnējās programmas P_B semantikai. Lai iegūtu vēl detalizētāku ieskatu šajā procesā, aplūkosim kādas metodes rezlizāciju orientētu grafu piemēram. Klases „BaseEdge" metode "isInstanceOfViewClass" varētu tikt realizēta, piemēram, šādi (pseidokodā):

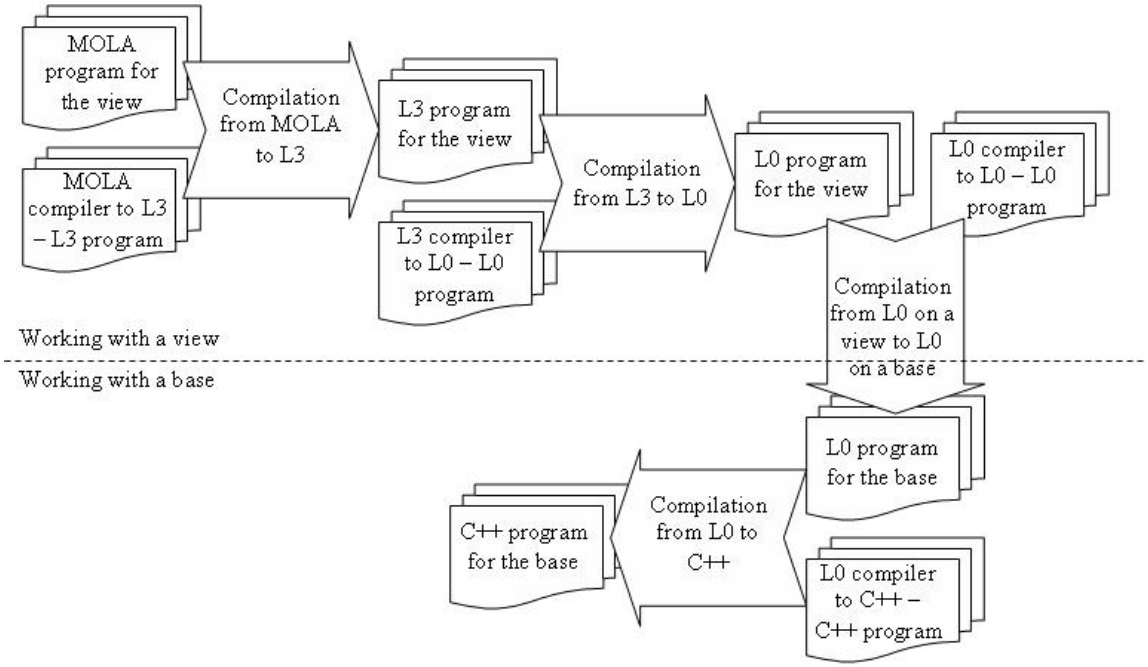
```
   function BaseEdge::isInstanceOfViewClass(viewClassName:String):
Boolean {
      if (viewClassName = ″ViewEdge″ AND this.isVisible = true)
         return true;
      else
         return false;
   }
```

Tā kā šajā gadījumā UZ bāzes klases „BaseEdge" definēta tikai viena skata klases („ViewEdge"), tad šī metode patiesu vērtību var atgriezt tikai tajā situācijā, kad izsaukta ar parametra vērtību vienādu ar „ViewEdge" (un pie tam – tikai tad, ja šī šķautne ir redzama, kā tas tika postulēts skatu definējošajos nosacījumos).

5.4. Pieejas pārbaude

Lai pārliecinātos par metodes darbspēju reālās situācijās, skatu mehānisms ticis realizēts un notestēts dažādiem gadījumiem. Lai demonstrētu pieeju, skatu veidošanai un strādāšanai ar tiem izvēlēta modeļu transformāciju valoda L0, kas aprakstīta šī darba 4. nodaļā. Taču vērts atzīmēt, ka šī pieeja nav piesaistīta nevienai konkrētai modeļu transformāciju valodai, tā kā kompilators varēja tikt rakstīts arī jebkurai citai valodai.

Viens no iemesliem, kādēļ par būvējamā kompilatora mērķa valodu tika izvēlēta valoda L0, ir tas, ka tā kalpo par bāzes valodu virknei augstāka līmeņa modeļu transformāciju valodām – L0', L1, L2, L3 un grafiskajai valodai MOLA. Līdz ar to, ieviešos skata kompilatoru valodā L0, tas automātiski kļūst izmantojams arī visās šajās augstākajās valodās – programmu, kas sakstīta skatam, piemēram, valodā L3, ar šī kompilatora palīdzību iespējams pārtransformēt par programmu, kas strādā ar bāzi.



5.6. attēls. **Kompilācijas shēma no MOLAs līdz C++**

Attēlā 5.6. redzama kopējā kompilācijas shēma pilnajā ceļā no valodas MOLA līdz C++. Kopilators, kas pārveido programmas, kas strādā ar skatu, par programmām, kas strādā ar bāzi, tiek realizēts tikai valodai L0. Tai pat laikā šāda kompilatora eksistence ļauj programmētājam lietot skata apstrādei jebkuru no augstāk minētajām valodām. Sāknēšanas process paveiks pārējo.

Transformāciju programma, kas pārvar robežu starp skatu un bāzi – skata kompilators – arī pati rakstīta valodā L0. Tajā ietvertas instanču grafa apstaigāšanas metodes, līdz ar ko šajā versijā skats ir tikai lasāms. Lai padarītu skatu arī rediģējamu, būtu jādefinē metožu kopa vēlamajām darbībām un pēc tam jāpapildina kompilators, lai tas ņemtu vērā šīs metodes.

Skata definēšanā izmantots nodaļā 5.2 aprakstītais piemērs. Funkcija UZ definēta tādā veidā, kā redzams attēlā 5.3. Minētajam piemēram realizētas piecas skatu definējošās metodes, kas izklāstītas nodaļā 5.2.2 (katrai no divām bāzes klasēm, UZ kurām definēta kāda skata klase). Metodes definētas valodā L0 (valoda piedāvā līdzekļus operāciju realizācijai un piesaistei klasēm).

Visbeidzot vairākas modeļu transformācijas tikušas uzrakstītas, lai notestētu izveidoto skatu. Transformāciju rakstītājam nav bijusi informācija par reāli repozitorijā esošo bāzes metamodeli, un viņš, rakstot transformācijas, izmantojis tikai piedāvāto skata metamodeli. Tests apliecināja, ka instanču grafu apstaigāšanas uzdevumos piedāvātais skatu mehānisms ir

lietojams. Tāpat mehānismā nav būtisku trūkumu, kas varētu liegt izveidot arī kopu ar metodēm, kas paredzētas grafa rediģēšanai, metožu semantiku atstājot metožu realizētāja ziņā.

Šajā nodaļā aprakstītais skatu mehānisms nopublicēts konferences DB&IS 2010 rakstu krājumā [56], kā arī pēc konferences atlasīto rakstu krājumā [57].

5.5. Skatu mehānisma lietojumi

Galvenie šajā darbā iztirzātā skatu mehānisma lietojumi parādās gadījumos, kad repozitorijā atrodas viens ļoti liels metamodelis, pie kura dažādām daļām vēlas vērsties dažādi lietotāji. Kā jau minēts šīs nodaļas ievada daļā, tā ir ļoti tipiska situācija tieši rīku būves kontekstā. Laika gaitā pieaugot dažādo iespēju klāstam, kuras atbalsta rīkus definējošais metamodelis, pats metamodelis var izplesties ļoti liels un kļūt arī tehniski sarežģīts.

Piemēram, ja runājam par transformāciju vadīto arhitektūru [32] GRAF platformas kontekstā, tad tā satur sevī vairākas saskarnes, no kurām katra sastāv no dziņa un no metamodeļa, ko šis dzinis spēj izpildes laikā interpretēt. Tas var izskatīties pēc ļoti skaidra visa lielā metamodeļa sadalījuma mazākajos saskarņu metamodeļos, un tās tā patiešām arī loģiskajā līmenī ir. Taču fiziskajā līmenī visi saskarņu metamodeļi vienalga atrodas visi kopā vienā repozitorijā, un dažādi lietotāji (piemēram, dažādi dziņi) mēģina uz to skatīties, ņemot vērā tikai sev vajadzīgās lietas.

Šis gan ir ļoti vienkāršots skata jēdziena lietojums – padarīt par skata metamodeli tikai kādu bāzes metamodeļa apakšmetamodeli (kā apakšgrafu). Šajā gadījumā bāzes klašu metodes ir ļoti triviālas, jo bāzes un skata klases pēc būtības sakrīt. Vēl jo vairāk – ja skats iznāk kā īsts bāzes apakšgrafs un nav nepieciešams izpildīt nekādus citus ierobežojumus, tad pēc šīm metodēm vispār nav nekādas vajadzības, jo programmas, kas rakstītas darbam ar skata metamodeli automātiski būs spējīgas strādāt arī ar bāzes metamodeli.

Interesantāka situācija parādās gadījumā, ja vēlamies uz vieniem un tiem pašiem loģiskajiem datiem skatīties no dažādu glabāšanas formātu (tas ir, dažādu metamodeļu) viedokļa. Visdabiskākais piemērs šeit būtu labi zināmā „domēna – prezentācijas" paradigma. Dati tiek glabāti domēna metamodelī, kas veido to loģisko struktūru. Tai pat laikā grafiskās attēlošanas dzinis strādā ar citu – prezentācijas – metamodeli, kas sastāv no grafiskajiem primitīviem (piemēram, tādiem kā platformas GRAF sastāvā esošajā grafveida diagrammu vizualizācijas dzinī [40, 41]). Praksē bieži vien abi šie metamodeļi tiek sasaistīti, izmantojot dažāda veida attēlojumus (*mappings*). Neskatoties uz šo attēlojumu veidu (statiski vai dinamiski,

izmantojot modeļu transformācijas), abi metamodeļi ar to instancēm parasti tiek glabāti repozitorijā, tādējādi ieviešot datu dublēšanos. Šajā situācijā noderīgs varētu izrādīties šajā darbā aprakstītais skatu mehānisms. Viens no abiem metamodeļiem šajā gadījumā var tikt izvēlēts par „īsto" (kas kalpos par bāzi), bet otrs veidots kā skats. Rīku definējošā instance līdz ar to tiktu glabāta tikai vienā eksemplārā, bet modeļu transformācijas varētu strādāt tāpat kā līdz šim, uzskatot, ka arī otrs metamodelis ir „īsts".

5.6. Citi pētījumi šajā jomā

Atgriezīsimies pie universitātes ontoloģijas piemēra, kas tika izmantots šīs nodaļas ievadā. Ja mums ir klase „Student", tad varētu būt vēlme no tās atvasināt klasi „Good Student" tādā veidā, ka šai klasei pieder tieši tie studenti, kas savākuši noteiktu daudzumu kredītpunktu un nokārtojuši visus obligātos kursus. Tā kā šis nosacījums satur aritmētiku, tas nevar tikt aprakstīts ar OWL (Web Ontology Language [58]) ierobežojumu (*restriction*) klasēm (ja vien netiek izveidots speciāls veids, kā reprezentēt punktus un atzīmes). Dažus mehānismus atvasinātu klašu definēšanai piedāvā OCL (Object Constraint Language [59]), bet daudzos gadījumos tie ir pārāk ierobežoti (piemēram, atvasinātajai klasei nevar tikt pievienoti jauni atribūti). Citos pētījumos bieži atrodams termins „skats", kas lietots arī šajā darbā (piemēram, tas ir viens no pamata jēdzieniem modeļu transformāciju valodā MOF QVT [13]), taču tas bieži vien tiek darīts pavisam citā nozīmē.

Diezgan līdzīgs skata jēdzienam ir jēdziens par modeļa tipu, kas izmantots Kermeta metamodelēšanas valodā [60]. Piemēram, J. Steel un J. M. Jezquel definē modeļa tipu kā tā metamodeli [61]. Modeļa tipa tālākus pētījumus šie paši autori prezentē vēlāk [62]. Galvenā risināmā problēma šajā sakarā ir šāda – kā dotajam modelim piemeklēt no repozitorija tādu metamodeli, kuram tas atbilst. Viņu vēlme ir tāda pati, kā skatu mehānismam šajā darbā – ļaut modeļu transformācijām strādāt ar doto modeli. Minēti autori risina arī problēmu par drošu metamodeļu aizvietojamību, un tiek piedāvāts algoritms modeļu tipu saderības pārbaudei (tas ir, mūsu rīcībā ir kaut kāds modeļa tips un mēs vēlamies pārbaudīt, vai tas ir derīgs mūsu modelim vai nē).

Šī pati modeļu tipu apakštipu meklēšanas problēma ir tālāk analizēta valodā Maude [63]. Maude ir augsta līmeņa valoda un augstas veiktspējas interpretators un kompilators OBJ algebrisko specifikāciju saimē, kas atbalsta vienādojumu loģiku un sistēmu loģiku specifikāciju pārveidošanu un programmēšanu [63]. Šeit modeļa tipa apakštips (fakts, ka metamodelis M' ir apakštips metamodelim M, jeb M'≤M) tiek definēts ar precīzu loģikas

izteiksmju palīdzību. Metamodeļa apakštips tālāk tiek definēts caur apakštipa jēdzienu dažādu metamodeļa elementu līmenī – pakotņu, klašu, atribūtu un norāžu –, kur tas, savukārt, tiek reducēts uz attiecību $\preceq$ primitīvo datu tipu (Integer, Double, Boolean) ietvaros. Tālāk tiek pieņemts apgalvojums, ka modelim nav tikai viens tips. Lai atrastu kādu dotā modeļa tipu, var sākt ar šī modeļa starta metamodeli (derīgu šī modeļa metamodeli, kas satur minimālu elementu kopu) un apstaigāt dotā repozitoriju metamodeļus, meklējot tādus, kas spēj droši aizvietot esošo metamodeli, vadoties pēc piedāvātā apakštipu meklēšanas algoritma. Maude izmanto KM3 – specializētu tekstuālu metamodeļu specificēšanas valodu [64] –, lai demonstrētu šo pieeju.

Lielākais šīs pieejas trūkums slēpjas tajā, ka esošajam metamodelim ir jābūt ļoti līdzīgam pieprasītajam. Patiesībā, tā kā tam ir jābūt īstam pieprasītā metamodeļa apakštipam, tad tas ir apmēram tas pats ļoti vienkāršais gadījums, kas minēts šī darba nodaļā 5.5. Lielākā problēma, iespējams, ir prasība metamodeļa elementu vārdu vienādībai, tas ir, klasēm (un citiem metamodeļa elementiem) ir jāsaucas pilnīgi vienādi abos metamodeļos, lai viens no tiem varētu tikt atzīts par otra apakštipu. Šāda prasība nav spēkā jēdzienam par skatu, kas aprakstīta šajā darbā, līdz ar to pieļaujot lielāku brīvību skata metamodeļa izveidē.

Modeļa tipa jēdziens tiek izmantots arī valodā MOF QVT. Šeit tas tiek definēts kā metamodelis, atbilstības veids un neobligāta OCL ierobežojumu izteiksmju kopa [13]. Metamodelis, kas deklarēts modeļu transformācijas deklarāciju līmenī, tiek saukts par efektīvo metamodeli, bet modamodelis, kas tiek lietots transformācijas izpildes brīdī – par aktuālo metamodeli. Starp šiem metamodeļiem, protams, nepieciešama zināma atbilstība. Ja modeļa tipa definīcijā minētais atbilstības veids ir *strict*, tad modelim jābūt īstai efektīvā metamodeļa instancei. Savukārt, ja atbilstības tips ir *effective*, tad katrai modeļa līmeņa instancei, kuras tips atrodams modeļa tipa metamodelī, ir jāsatur vismaz tās īpašības (*properties*), kas definētas efektīvā metamodeļa metaklasē, un to tipiem jābūt saderīgiem. Tomēr arī šeit saistība starp metamodeļa elementiem ir balstīta uz vārdu salīdzināšanu, pieļaujot vien speciāla veida pārsaukšanu, izmantojot *alias* birku, lai ieviestu lielāku brīvību. Tiek atzīts, ka modeļu tipu salīdzināšana un klasificēšana nav MOF QVT specifikācijas sastāvdaļa.

Skata metamodeļa jēdziens atrodams citā darbā [65]. Šeit skats tiek definēts kā apraksts, kas attēlo sistēmu noteiktā perspektīvā un tādējādi iekļauj sevī sistēmas elementu (piemēram, moduļu) apakškopas. Šis risinājums atšķiras no šajā darbā piedāvātā ar to, ka šeit tiek aplūkoti vairāki paralēli skatu metamodeļi, kamēr šajā darbā autors aplūko vienu bāzes metamodeli, uz

kura var tikt definēti vairāki skatu metamodeļi. Līdz ar to šeit parādās ļoti nopietni jautājumi par visu paralēlo modeļu atbilstības nodrošināšanu un konsistences pārbaudi starp skatiem. Šajā darbā šie jautājumi atkrīt, jo reāli repozitorijā atrodas tikai viens metamodelis.

Ļoti līdzīga izpratne par skatu atrodama vēk kādā darbā [66]. Šeit arī tiek lietots termins „virtuāls skats", tādējādi apzīmējot nematerializētu skatu, kas tiek definēts skata metamodeļa formā uz kāda reāli eksistējoša metamodeļa. Skats tiek specificēts, izmantojot modificētu trijnieku grafu gramatiku (*triple graph grammars*, TGG) versiju. Skata definīcija sastāv no TGG likumu kopas, kas atļauj patvaļīgas sarežģītības reālā modeļa fragmenta ģenerēšanu brīdī, kad kāds vēlas izveidot kādu elementu skata metamodelī. Lai nodrošinātu saikni starp atribūtu vērtībām skata un bāzes metamodeļu klasēs, netiek izmantota modeļu transformāciju valoda, bet tā vietā tiek lietotas OCL ierobežojumu anotācijas.

5.7. Tālākais darbs skatu mehānisma sakarā

Kā jau minēts, šajā nodaļā aprakstītajā skatu definēšanas mehānisma piemērā galvenā uzmanība tika pievērsta iespējai izmantot skatu uzdevumos, kuros nepieciešama instanču grafa apstaigāšana. Rezultātā tika izstrādātas piecas metodes, kuras nepieciešams realizēt un ņemt vērā skatu kompilatoram šī mērķa realizēšanai. Tāpat tika norādītas darbības, kas būtu jāveic, ja rastos vēlēšanās šo skatu paplašināt ar iespēju instanču grafu arī modificēt. Šis uzskatāms par vienu no tālākajiem veicamajiem darbiem skatu mehānisma sakarā – izstrādāt metožu kopu, ar kuru palīdzību instanču grafā varētu veikt zināmas izmaiņas, kā arī papildināt skatu kompilatoru ar šo metožu ņemšanu vērā.

Otra lieta, kas jāmin šajā sakarā, saistīta ar nodaļā minēto jēdzienu UZ. Pašreizējā variantā UZ ir funkcija, kas nozīmē, ka katrai skata klasei var tikt piekārtota tikai viena bāzes klase. Praksē, iespējams, tomēr varētu būt situācijas, kad mēs gribētu ļaut vienu skata klasi definēt UZ vairākām bāzes klasēm, tādējādi UZ uztverot kā patvaļīgu bināru attēlojumu. Piemēram, pašreizējā variantā nav iespējams attēlā 5.4. redzamos bāzes un skata metamodeļus apmainīt vietām un uz visu bāzes metamodeļa hierarhiju paskatīties tikai caur vienu klasi „Person". Cits piemērs – varētu būt vēlme definēt kādu skata klasi kā vairāku bāzes klašu apvienojumu. Padarot UZ par patvaļīgu bināru attēlojumu, skatu kompilatoru nepieciešams nodrošināt ar kādu papildus informāciju par to, kā katrā gadījumā pareizi atrast īsto bāzes klasi. Šāda iespēja būtu ļoti labs skatu mehānisma papildinājums, kā arī tai nevajadzētu būt pārlieku sarežģītai, jo citas mehānisma daļas (neskaitot kompilatoru) mainīt nav nepieciešams.

Vēl viena apdomāšanas vērta lieta ir atsevišķas augsta līmeņa valodas izstrāde skata definēšanai, kas būtu lietotājam draudzīgāka nekā tā, kas piedāvāta šajā risinājuma variantā, kurā nepieciešams rakstīt metožu realizācijas un atsevišķi definēt funkciju UZ. Šī valoda varētu būt grafiska, piedāvājot specificēt metodes grafiska konfiguratora izskatā (piemēram, līdzīgi kā darbojas konfigurators GRAF platformas ietvaros, kur tā uzdevums ir specificēt katru konkrētu rīku [33]). Tā kā šī valoda ir problēmorientēta (par domēnu var uzskatīt skatu definēšanas mehānisma metamodeli), tad to iespējams izstrādāt un realizēt pašas platformas GRAF ietvaros. Ja tiktu izstrādāta šāda valoda, parādītos vēl viens translācijas solis – skatu, kas definēts šādā valodā, būtu nepieciešams pārtranslēt tajā valodā, kas aprakstīta šajā nodaļā (varam saukt to par bāzes valodu skatu definēšanai) un kas jau ir realizēta.

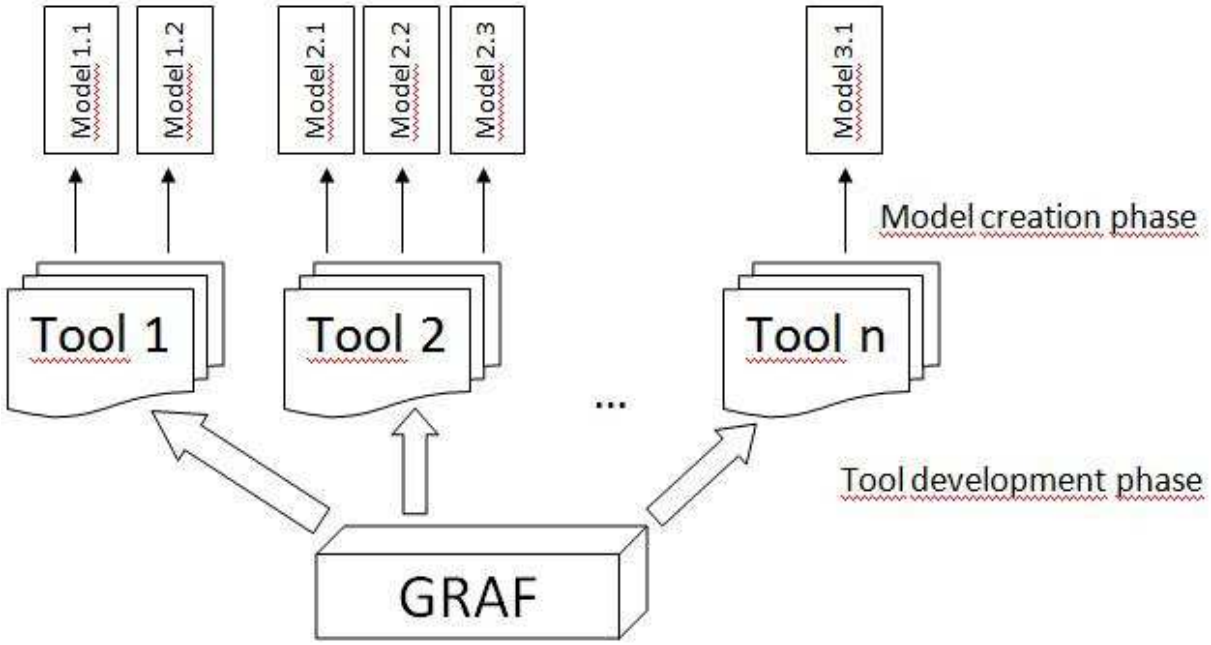
6. Modeļu eksporta un importa mehānisms

Iepriekšējās divās nodaļās iztirzāti rīku būves platformas GRAF servisi, kuru integrēšana platformā atstāj tiešas pozitīvas sekas uz jaunu rīku radīšanas procesu, ļaujot modeļu transformāciju (specifisko transformāciju) rakstītājiem ātrāk un ērtāk izstrādāt transformāciju programmas. Tas bija viens no šī darba mērķiem, kas tika aprakstīts nodaļā 3.3. Otrs lielais mērķis platformas lietojamības uzlabošanas procesā bija atvieglot jau gatava rīka funkcionētspējas saglabāšanu mainīgas apkārtējās vides apstākļos. Šai problēmai risinājums piedāvāts šajā nodaļā. Uzreiz jāsaka, ka šeit nekādā ziņā nav runa par pilnīgu rīka dažādo komponenšu versiju pārvaldību, kas pati par sevi varētu būt pilnīgi atsevišķas disertācijas tēma.

Nodaļas pirmajā apakšnodaļā detalizēti izklāstīta risināmās problēmas būtība, tālāk piedāvāts tās risinājuma variants, ieviešot jēdzienu par modeļa eksportu un importu vēlamajā kontekstā. Modeļa eksports un imports pēc būtības ir krietni plašāks un vispārīgāks risinājums par to, kas nepieciešams tieši GRAF platformas rīku funkcionētspējas nodrošināšanai, tāpēc tas, iespējams, būtu lietojams arī citos kontekstos. Šajā nodaļā gan vispārīgais risinājums iztirzāts, paturot prātā tieši šo vienu konkrēto lietojumu. Tālākās apakšnodaļās risinājums demonstrēts uz konkrētiem piemēriem. Nodaļa pabeigta, īsumā apspriežot citus risinājuma pielietojuma variantus un ieskicējot turpmāk veicamos darbus šajā sakarā.

6.1. Apkārtējās vides mainība

Ar apkārtējo vidi šeit sapratīsim kontekstu, kādā darbojas aplūkojamā sistēmas daļa. Runājot par sistēmas daļām, jāievēro, ka pamatā var izšķirt trīs to līmeņus (skatīt attēlu 6.1.). Visa pamatā atrodas platforma GRAF, kurā iespējams izstrādāt grafiskus problēmorientētus rīkus. Katrs rīks tiek izstrādāts noteiktā platformas versijā un normālā gadījumā pēc tam arī strādā šajā versijā. Tā kā katrs šāds rīks pēc būtības ir problēmorientēta valoda, tad ar tā palīdzību iespējams veidot konkrētus šīs valodas modeļus. Šeit parādās trešais līmenis – katrs modelis ir izstrādāts ar noteiktu tā valodas versiju un ar šo versiju tas turpina „dzīvot”.



6.1. attēls. **Trīs līmeņu sistēmas daļas**

Kā jau minēts augstāk, viss notiek korekti, kamēr vien nemainās kādas versijas. Bet kas notiek, ja tomēr tiek ieviestas kādas izmaiņas? Šeit jāšķiro divi gadījumi atkarībā no tā, ko šīs izmaiņas ietekmē. Pirmā veida izmaiņas ir izmaiņas kādas konkrētas problēmorientētas valodas specifikācijā. Mainoties specifikācijai, dabiska būtu vēlme kaut kādā veidā pielāgot ar veco valodas variantu radītos modeļus tā, lai tie būtu darboties spējīgi arī ar jauno valodas versiju. Šeit, protams, atkal būtu jāpēta dziļāk, kāda veida izmaiņas valoda piedzīvojusi un vai tās maz principā ir ieviešamas vecajos modeļos. Taču tā kā šajā nodaļā nav cerēts nodrošināt pilnīgu versiju pārvaldību, tad mērķi šeit definēsim šādi – pieņemot, ka izmaiņas valodā ir pilnībā vai daļēji ieviešamas vecajos modeļos, nodrošināt šo modeļu integrāciju jaunajā problēmorientētās valodas versijā.

Otrā veida izmaiņas ir izmaiņas pašā apakšējajā līmenī – platformas specifikācijā. Lai arī platforma GRAF pēc būtības jau ir diezgan attīstīta un stabila, tomēr ik pa laikam tajā tiek ieviesti papildinājumi ar mērķi bagātināt platformu ar jaunām iespējām vai novērst kādus trūkumus jau esošajā funkcionalitātē. Līdz ar to iespējams, ka platformas izmaiņas var radīt traucējumus kādu ar veco platformas versiju izstrādāto problēmorientēto rīku (un, iespējams, arī to modeļu) darbībā. Arī šī izmaiņu veida gadījumā mērķis var tikt noformulēts līdzīgā veidā – pieņemot, ka izmaiņas platformā ir pilnībā vai daļēji ieviešamas vecajos ar to izstrādātajos rīkos, nodrošināt šo rīku integrāciju jaunajā platformas versijā.

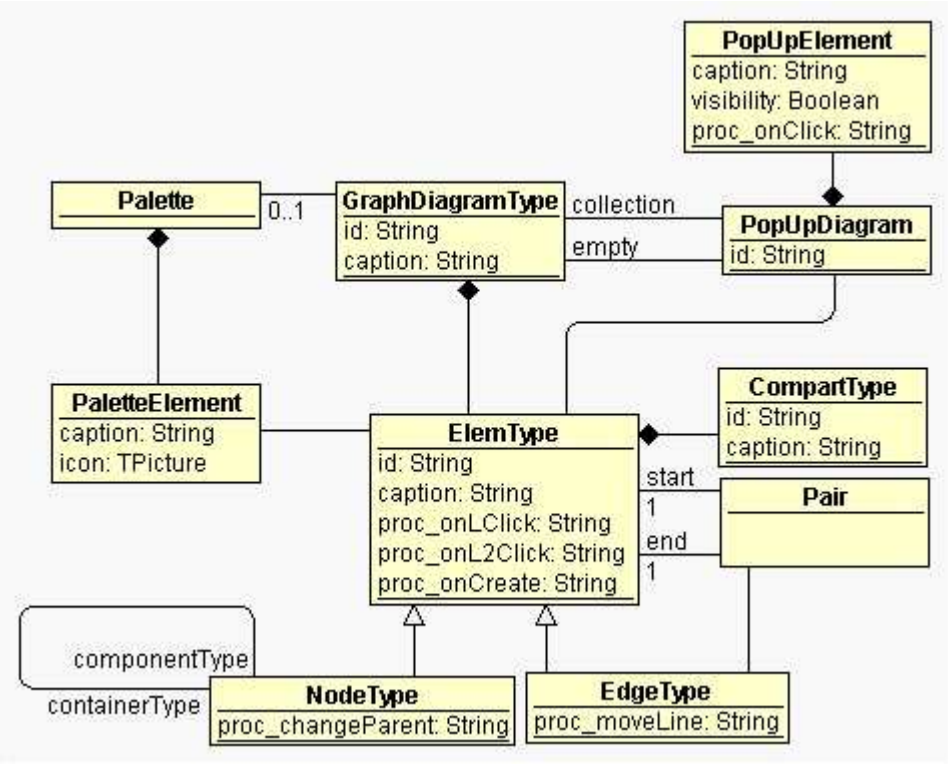
Tālāk šajā apakšnodaļā apskatīti abi augstāk minētie izmaiņu gadījumi atsevišķi, dziļāk un tehniskāk paskaidrojot, kas tad katrā gadījumā var tikt mainīts un ko nepieciešams saglabāt.

Lai arī abi šie gadījumi ir savā starpā diezgan atšķirīgi, tiem tomēr tiks piemeklēts kopīgs universāls risinājuma variants, kas aplūkots tālākajās apakšnodaļās.

6.1.1. Izmaiņas rīka specifikācijā

Divi jēdzieni, kas bieži vien tiek lietoti ļoti saistītā kontekstā, ir problēmorientēta valoda un problēmorientēts rīks. Platforma GRAF paredzēta problēmorientētu rīku būvēšanai, bet, lai uzbūvētu kādu rīku, vispirms nepieciešams izstrādāt valodu, kuru šis rīks realizēs. Var teikt, ka ar problēmorientētas valodas palīdzību tiek uzdota specifikācija, kas nosaka būvējamā rīka funkcionalitāti. Tātad – mainoties valodai, līdzi jāmainās arī ar to saistītajam rīkam.

Valodas uzdošana platformā GRAF notiek, norādot, no kādiem grafiskajiem elementiem tā sastāvēs, kāda būs saistība starp šiem elementiem (kas ko varēs saturēt, kādos elementos varēs sākties un beigties kādas līnijas u.tml.), kādas darbības lietotājs varēs veikt ar šiem elementiem u.tml. Šī informācija par problēmorientēto valodu jeb rīka specifikācija tiek kodēta rīku definēšanas metamodelī, kurš tika nedaudz ieskicēts 3. nodaļā. Attēlā 6.2. precīzāk attēlota būtiskākā rīku definēšanas metamodeļa daļa, kas kodē pamata informāciju par rīku.



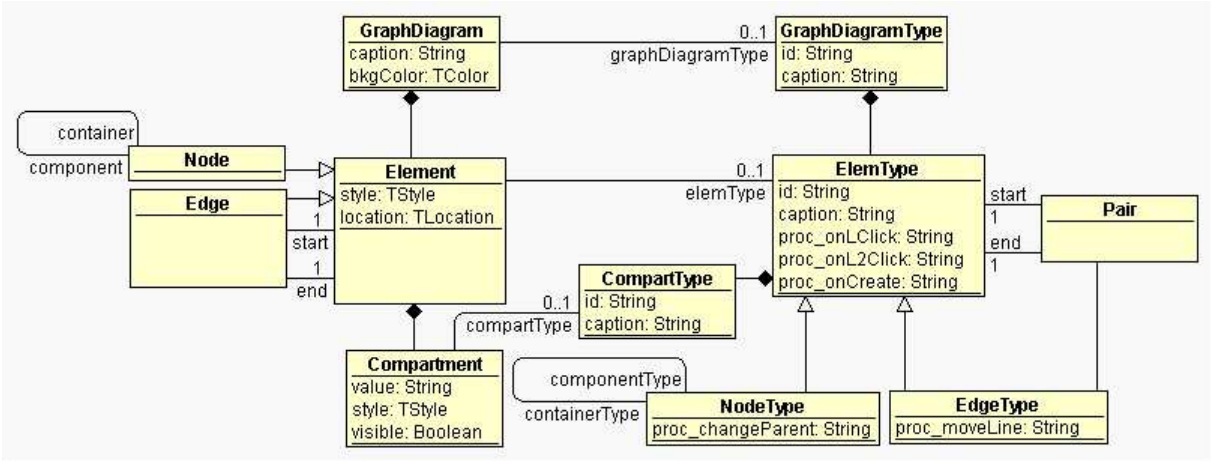
6.2. attēls. **Rīku definēšanas metamodeļa kodols**

Centrālā klase šajā metamodelī ir klase „GraphDiagramType”, kuras instances ir katrs konkrētais grafveida diagrammas tips. Katrs šāds tips var saturēt dažāda veida elementu tipus

(klase „ElemType”), kas var būt vai nu virsotnes („NodeType”) vai šķautnes („EdgeType”). Ar asociāciju palīdzību šeit iespējams arī norādīt saistību starp elementiem – faktu, ka kādai virsotnei iespējams atrasties iekš kādas citas virsotnes („containerType/componentType”), un faktu, ka līnijai jāsākas un jābeidzas noteikta tipa virsotnēs vai šķautnēs (klase „Pair”). Tālāk katrs elementa tips var saturēt virkni nodalījuma tipus (klase „CompartType”), ar nodalījumu saprotot vietu elementā, kurā iespējams ievietot tekstu vai attēlu. Katram elementa tipam iespējams arī piesaistīt paletes elementu (klase „PaletteElement”), ar kuru rīkā šī tipa elementus būs iespējams veidot.

Lai elementu tipiem piekārtotu lietotāja darbības, kādas būs iespējams veikt ar noteiktā tipa elementiem, ieviesta klase „PopUpDiagram”, kas nozīmē izlecošo izvēlni un sastāv no izvēlnes vienumiem – „PopUpElement”. Izlecošā izvēlne var tikt piesaistīta vai nu noteiktam elementa tipam vai arī patiešo grafveida diagrammas tipam – vai nu atzīmētai elementu kopai (caur saiti „collection”) vai arī, izsaucot to tukšā diagrammas vietā (caur saiti „empty”). Katram izlecošās izvēlmes vienumam iespējams pierakstīt tās modeļu transformācijas vārdu, kura tiks izsaukta brīdī, kad lietotājs šo vienumu rīka darbības laikā izvēlēsies, piemēram, noklikšķinot uz tā ar peles kreiso pogu. Tāpat transformāciju vārdus iespējams norādīt arī dažādām cita veida darbībām, kā tas redzams rīku definēšanas metamodeļa klašu atribūtos, kuru nosaukumi sākas ar prefiksu „proc_”. Taču šī darba mērķis nav pilnībā izskaidrot rīku definēšanas metamodeli, tikai iedziļināties tajā tiktāl, cik tas nepieciešams eksporta un importa skaidrojumam, tāpēc šīs transformācijas šeit sīkāk skaidrotas netiks.

Izveidot jaunu rīku nozīmē izveidot šī rīku definēšanas metamodeļa korektu instanci, kuru universālā transformācija rīka darbības laikā spēj interpretēt. Vienā repozitorijā ar rīku definēšanas metamodeli atrodas arī, piemēram, grafu vizualizācijas dziņa metamodelis, kura instances ir konkrētas grafveida diagrammas. Abu minēto metamodeļu klases saistītas ar asociācijām, kuru saites rīka darbības laikā tiek veidotas, rediģētas un dzēstas, ievērojot zināmus ierobežojumus (katram elementam, piemēram, rīka darbības laikā jābūt tieši vienam tipam). Attēlā 6.3. redzama šo metamodeļu saistība.



6.3. attēls. **Grafveida diagrammu vizualizācijas metamodeļa saistība ar rīku definēšanas metamodeli**

Tagad, kad skaidra šo metamodeļu saistība, varam atgriezties pie galvenā jautājuma, uz kuru atbildi jāmēģina rast šajā apakšnodaļā – kas notiek, ja vēlamies ieviest kādas izmaiņas rīka specifikācijā? Rīka specifikācija nokodēta kā rīku definēšanas metamodeļa konkrēta instance, kas, iespējams, izplatīta vairākās kopijās dažādiem lietotājiem, kas, to izmantojot, sazīmējuši ar to grafveida diagrammas, tas ir, saveidojuši grafveida diagrammu vizualizācijas metamodeļa instances un sasaistījuši tās ar minēto rīku definēšanas metamodeļa instanci. Ja valodas izstrādātāji tagad pie sevis veic izmaiņas rīka specifikācijā, nepieciešams šīs izmaiņas ieviest arī visos jau izmantojamajos rīkos.

Lai ieviestu izmaiņas, iespējams apsvērst vismaz divus dažādus risinājuma variantus. Viens variants būtu noformēt šīs izmaiņas kā modeļu transformāciju, kuru nepieciešams izpildīt uz vecās rīka versijas, lai šī versija tiktu „salabota” un kļūtu vienāda ar jauno rīka versiju. Taču šim variantam piemīt daži būtiski trūkumi. Pirmkārt, izstrādātājiem nepieciešams vest „dubultu grāmatvedību” – visas rīka izmaiņas gan ieviest attīstāmajā rīkā normālā veidā (tas ir, atjaunināt rīku definējošo instanci, izmantojot, piemēram, konfiguratora piedāvātās iespējas), gan arī papildus tam ieviest šīs izmaiņas modeļu transformācijas izskatā. Tiesa, šo trūkumu zināmā mērā varētu novērst, automātiski ģenerējot veriju salabojošo transformāciju no rīku definējošās instances izmiņām, kas tiek veiktas ar konfiguratora palīdzību. Otrs trūkums minētajam risinājumam ir tas, ka šāda versiju migrācija iespējama tikai no konkrētas rīka versijas uz kādu citu konkrētu versiju. Ja gala lietotājiem izmantojamās rīka versijas atšķirtos, tad katram no tiem būtu nepieciešams piedāvāt citu versiju salabojošo transformāciju, kas rīku pārceltu uz jaunāko versiju. Nepieciešamība uzturēt dažādas transformācijas dažādām versijām, protams, rada vēl virkni problēmu. Kā jau minēts, versiju pilnīga pārvaldība ir ļoti sarežģīta problēma, un šī darba mērķis nav pārklāt to visu.

Tādēļ šajā darbā tiks aplūkots cits risinājuma variants augstākminētajai problēmai – izeksportēt modeli, kas izveidots ar veco rīka versiju (neatkarīgi no tā, ar cik vecu) un ieimportēt to jaunajā rīka versijā, pēc iespējas mēģinot atjaunot visus datus, kurus vispār iespējams atjaunot.

Šāda veida imports sastāvētu no divām daļām – vispirms jaunajā rīka versijā nepieciešams ieimportēt datus, kas atbilst vecajiem modeļiem (grafveida diagrammu vizualizācijas metamodeļa instances), bet pēc tam – korektā veidā izveidot saites no šiem datiem uz tiem datiem, kas definē jauno rīku un jau iepriekš atradās repozitorijā. Arī eksports līdz ar to sastāv no diviem soļiem – 1) vajadzīgo modeļu izņemšana no repozitorija un saglabāšana kaut kādā formātā; un 2) papildus informācijas saglabāšana par to, kādā veidā vēlāk iespējams atjaunot saites uz īstajiem diagrammu, elementu un nodalījumu tipiem (kā arī citas saites uz rīka definīciju).

6.1.2. Izmaiņas platformā

Otrs izmaiņu veids ir izmaiņas pašā platformā GRAF. Platformā, protams, var notikt dažnedažādāko veidu izmaiņas, kas saistītas ar dažādām platformas konponentēm, bet izmaiņas, kas tiks apskatītas šajā apakšnodaļā, ir tās, kas saistītas ar rīku definēšanas konponentes darbību. Tātad ar izmaiņām platformā šeit sapratīsim tāda veida izmaiņas, kas maina veidu, kā definēt jaunus rīkus. Kā jau minēts iepriekš, nodefinēt rīku nozīmē izveidot korektu rīku definēšanas metamodeļa instanci, un tas tiek darīts, izmantojot konfiguratora komponenti. Tā kā arī konfigurators pats par sevi ir problēmorientēts rīks, kura mērķis ir veidot citus problēmorientētus rīkus, un tas arī veidots platformā GRAF, tad šo izmaiņu veidu zināmā mērā iespējams reducēt uz iepriekš aplūkoto gadījumu, kad izmaiņas notikušas kādas valodas vai rīka specifikācijā. Šajā gadījumā gan esam pacēlušies vienu abstrakcijas līmeni augstāk – izmaiņas notikušas rīkā, kas definē citus rīkus, un gala lietotāji, kuriem šīs izmaiņas nepieciešams ieviest, paši ir rīku definētāji.

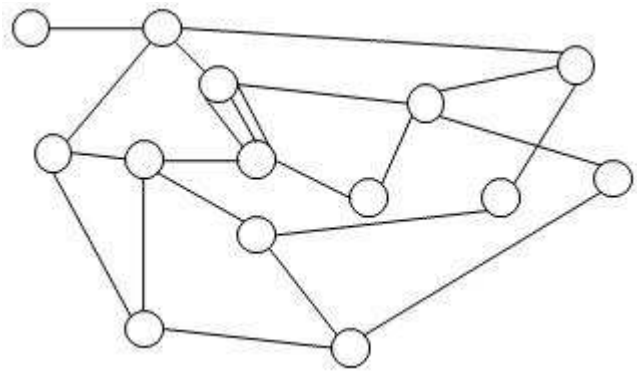
Tātad, ja veidojam salīdzinājumu ar iepriekš aprakstīto izmaiņu veidu, tad šajā gadījumā eksportēts tiktu modelis, kuru veido konfiguratora diagrammas un kas apraksta kādus izstrādātos rīkus. Rīka definēšanas metamodeļa instance, ar kuru nepieciešams atjaunot saites, šajā gadījumā būtu instance, kas definē konfiguratora diagrammas un sastāv no tādiem elementu tipiem kā „nodeType”, „edgeType”, „subType” u.tml. (sīkāk par konfiguratora konponenti skatīt [33]). Protams, tā kā darbojamies vienu līmeni augstāk, tad papildus nepieciešams atrisināt vēl dažas tehniskas lietas, bet pamatā eksports un imports spējīgs

darboties pēc līdzīgiem principiem kā jebkuras citas problēmorientētas valodas/rīka specifikācijas izmaiņu gadījumā

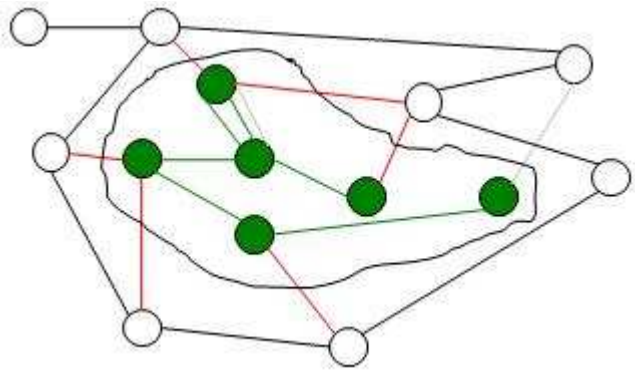
6.2. Risināmās problēmas vispārinājums

Nodaļā 6.1. tika aplūkota situācija, kurā nepieciešams eksportēt daļu no repozitorijā esošajiem datiem un pēc to importa citā repozitorijā atjaunot noteiktas saites uz jau šajā repozitorijā eksistējošiem objektiem. Par piemēru tika ņemts viena konkrēta platformas GRAF saskarne – grafveida diagrammu vizualizācija. Repozitorija daļa, kuru nepieciešams saglabāt, šajā gadījumā bija konkrētās ar veco rīka versiju izveidotās grafveida diagrammas. Lai arī tas netika īpaši uzsvērts, šis uztverams tikai kā piemērs tam, ko varētu vēlēties saglabāt. Transformāciju vadītās arhitektūras ietvaros platformā GRAF eksistē vēl vairākas saskarnes, kuras arī savos metamodeļos glabā noteiktu informāciju par datiem, kas arī var būt ar saitēm savienoti ar rīku definīcijas daļām. Tāpat TDA paredz iespēju lietotājam izstrādāt un pievienot platformai arī savas saskarnes. Lai padarītu eksporta/importa mehānismu universālāku, šajā nodaļā noformulēta vispārīgāka risināmā problēma, kas pilnībā pārklāj iepriekš aprakstītās vēlmes, tai pat laikā paredzot iespēju tās arī paplašināt.

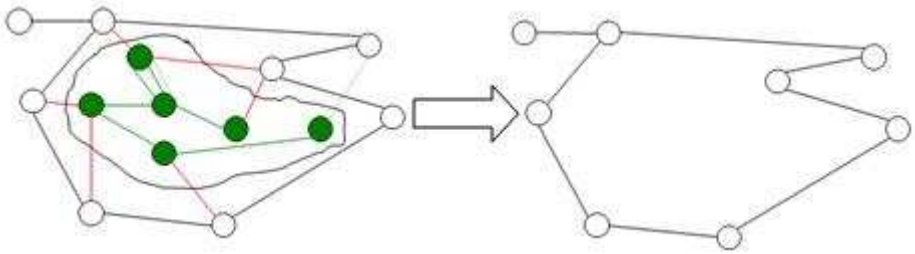
Pamata nostādne, kurai vērsīsim uzmanību, ir noformulēta šajā rindkopā. Repozitorijs sastāv no divu veidu datiem – tādiem, kurus nepieciešams saglabāt, pārejot uz jaunu repozitorija versiju, un tādiem, kurus saglabāt nav nepieciešams. Starp šiem datiem instanču līmenī iespējamas saites. Daļa no šīm saitēm nav būtiska, bet daļu ir vēlme atjaunot jaunajā repozitorijā. Atjaunošana notiek, atrodot jaunajā repozitorijā objektu, kurš kaut kādā ziņā ir *līdzīgs* oriģinālajam objektam, kurš attiecīgās saites galā atradās vecajā repozitorijā. Attēlā 6.4. demonstrēta šāda datu pārnese starp repozitorijiem.



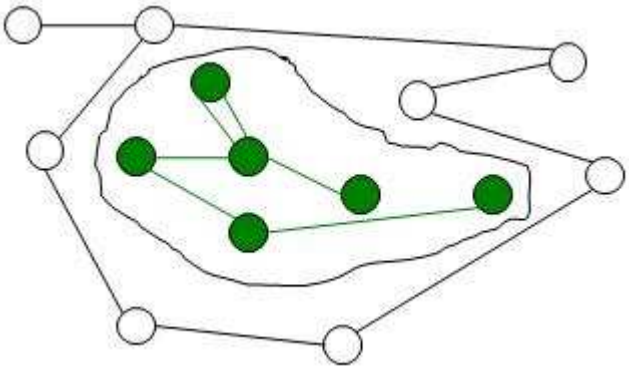
a) instanču grafs vecajā repozitorijā



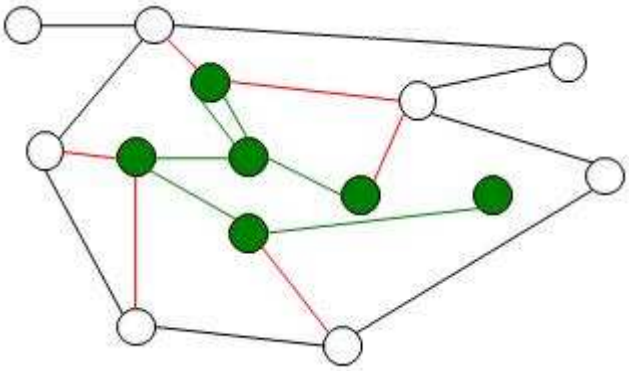
b) divu veidu instances vecajā repozitorijā – zaļās nepieciešams saglabāt; arī divu veidu pārraujamās saites – sarkanās būs nepieciešams atjaunot



c) jaunā repozitorija baltās instances līdzīgas vecā repozitorija baltajām instancēm, bet zaļo daļu no vecā varianta nepieciešams pārcelt uz jauno



d) saglabājamās instances pārceltas uz jauno repozitoriju, vēl atlicis pēdējais darbs – saišu atjaunošana



e) jaunais repozitorijs ar tajā pilnībā integrētu vecā repozitorija fragmentu

6.4. attēls. **Datu pārneses starp repozitorijiem vispārīga shēma**

Attēlā 6.4. redzami instanču grafi vecajā un jaunajā repozitorijā. Sākumā (attēls 6.4.a) eksistē vecais repozitorija instanču grafs. Vecajā repozitorijā loģiskā līmenī mēs spējam sadalīt instances divās daļās (attēls 6.4.b) – tajās, kuras nepieciešams saglabāt (attēlotas zaļas), un tajās, kas ir novecojušas (attēlotas baltas). Starp šo daļu instancēm eksistē saites, kuras eksporta rezultātā tiks pārrautas – daļu no tām būs nepieciešams atjaunot jaunajā repozitorijā uz *līdzīgajiem* objektiem (atjaunojamās saites attēlotas sarkanas, bet tās, kuras nebūs nepieciešams atjaunot – iepelēkotas). Tālāk eksporta gaitā nepieciešams it kā izgriezt saglabāt nepieciešamās instances no vecā repozitorija un ievietot tās jaunajā repozitorijā (attēls 6.4.c), kurā atrodami sākotnējā repozitorija objektiem *līdzīgi* objekti (attēloti balti). Attēlā 6.4.d redzams jaunais repozitorijs, kurā ievietots saglabāt nepieciešamais modelis no vecā repozitorija. Visbeidzot nepieciešams atjaunot vajadzīgās saites, iegūstot pilnībā funkcionējošu rīku kopā ar saglabātajiem tā modeļiem jaunajā variantā (attēls 6.4.e).

Skaidrs, ka šeit aprakstītais repozitorija fragmenta izgriešanas (jeb precīzāk – kopēšanas) un ielīmēšanas citā repozitorijā mehānisms ir vispārīgāks par nodaļā 6.1. aprakstīto gadījumu. Atrisinot šāda veida kopēšanu/ielīmēšanu vispārīgā veidā, gala lietotājs to varētu izmantot pēc saviem ieskatiem līdzīgi kā teksta redaktorā katrs pēc saviem ieskatiem var izmantot teksta kopēšanas/ielīmēšanas funkcionalitāti. Divi labi šī mehānisma izmantošanas piemēri tad būtu iepriekš aprakstītie izmaiņu nodrošināšanas gadījumi.

6.3. Risinājuma pamata idejas

Eksporta/importa mehānisma risinājums balstās uz pamata ideju formālā veidā specificēt noteiktu modeļa fragmentu, kuru pēc tam iespējams apstaigāt, veicot zināmas darbības ar katru no šī fragmenta elementiem. Modeļa fragments tiek specificēts tā metamodeļa līmenī, norādot to metamodeļa *apakšmetamodeli*, kura klasēm un asociācijām atbilstošie objekti un saites mūs interesēs. Kā jau minēts iepriekš, standarta lietojumos mēs vēlēsimies it kā *izgriezt* kādu metamodeļa fragmentu. Ja metamodeli iedomājamies kā parastu grafu ar virsotnēm un šķautnēm, tad šī izgriešana nozīmē dažu šķautņu pārraušanu (likvidēšanu) tādā veidā, lai no sākotnējā grafa (pieņemsim, ka tas bijis sakarīgs) tiktu atdalīts atsevišķs sakarīgs grafs. Pēc tam šo jauniegūto grafu pievienosim citam grafam, vajadzības gadījumā atjaunojot daļu no pārrautajām šķautnēm. Tālāk šajā apakšnodaļā šis neformālais risinājuma apraksts izskaidrots formālā veidā.

Vispirms mēģināsim formālā veidā definēt repozitoriju. Šī apraksts nebūtu jāuzskata par pilnu repozitorija definīciju, repozitorijs šeit tiek definēts tikai tik lielā mērā, cik tas nepieciešams risinājuma izskaidrošanai.

Pieņemsim, ka mums doti divi grafi M (metamodeļa grafs) un D (modeļa jeb datu grafs), kas uzdoti kā virsotņu un šķautņu kopu pāriši, tas ir, $M = (V_M, E_M)$, $D = (V_D, E_D)$, kur V_M un V_D ir virsotņu kopas, pie kam V_M elementi tiek unikāli identificēti ar vārdiem (klašu vārdi metamodelī), bet $E_i = \{(x,a,y) \mid x \in V_i \ \& \ y \in V_i\}$ ir šķautņu kopas, kur katrai orientētajai šķautnei piekārtots vārds (lomu vārdi metamodelī un modelī).

Ieviešam attēlojumu T: $V_D \rightarrow V_M$ ar šādām īpašībām:

- T definēts visā kopā V_D
- Nevienam $x \in V_D$ neatbilst vairāk par vienu $y \in V_M$
- Ja $(x,a,y) \in E_D$, tad $(T(x),a,T(y)) \in E_M$

Pirmās divas īpašības faktiski nozīmē to, ka T ir funkcija, bet trešā īpašība nosaka, ka attēlojums pēc būtības darbojas ne tikai starp grafu virsotnēm, bet arī starp tā šķautnēm – ja datu grafam pieder kāda šķautne ar noteiktu vārdu, tad arī metamodeļa grafā šķautne ar šādu vārdu eksistēs starp datu grafa šķautnes galapunktu attēlojumiem. Kā jau nojaušams, derīgs šāda attēlojuma piemērs būtu, piemēram, tipa attiecība starp objektiem un saitēm modelī un klasēm un asociācijām metamodelī.

Definīcija. Par **repozitoriju** sauc objektu trijnieku (M, D, T), kur M un D ir grafi, bet T – attēlojums ar augstāk minētajām īpašībām.

Definīcija. Par repozitorija (M, D, T) **fragmentu** sauc objektu četrinieku (M', D', T', L), kuram spēkā šādi nosacījumi:

- (M', D', T') ir repozitorijs
- M' ir M apakšgrafs
- D' ir D apakšgrafs
- $T' \subseteq T$
- Eksistē tāda grafa D' virsotne, no kuras sasniedzama jebkura cita grafa D' virsotne
- $L \subseteq \{(x,a,y) \mid x \in V_{D'} \ \& \ y \notin V_{D'} \ \& \ (x,a,y) \in E_D\}$

Šajā repozitorija fragmenta definīcijā pirmais nosacījums apgalvo, ka fragmentā iekļautajai modeļa un metamodeļa daļai jābūt savstarpēji saskaņotai – ja fragmentā iekļaujam, piemēram, kādu objektu, tad tajā jāiekļauj arī šī objekta klase. Nākamie trīs nosacījumi norāda saistību ar sākotnējo repozitoriju, bet ceturtais nosacījums apgalvo, ka mēs interesēsimies tikai par sakarīgiem grafiem. Iepriekš aprakstītajos lietojumos patiesi vienmēr var atrast kādu tā saucamo sākuma elementu, no kura, ejot pa saitēm, iegūt pārējos eksportējamos elementus.

Piemēram, ja nepieciešams eksportēt kādu grafveida diagrammu, tad par sākuma objektu iespējams izvēlēties atbilstošo klases „GraphDiagram" instanci – visus pārējos vajadzīgos objektus varēst atrast no tās.

Pēdējais nosacījums attiecas uz fragmenta izgriešanas procesā pārrautajām šķautnēm. Šeit ņemtas vērā šķautnes metamodeļa līmenī, kas sākas kādā no fragmentā iekļautajām virsotnēm, bet beidzas kādā no sākotnējā repozitorija virsotnēm, kas nav iekļautas fragmentā. Šāda veida šķautnes tātad tiek pārrautas. Taču tā kā atjaunot jaunajā repozitorijā vajadzēs tikai daļu no tām, tad kopa L tiek specificēta kā apakškopa no visām pārrautajām šķautnēm – tajā iekļaujamas tikai tās šķautnes, kuras būs vēlēšanās saglabāt. Jāpievērš uzmanība tam, ka fragmenta pārrāvumi specificējami tikai modeļa līmenī – metamodeļa līmeņa pārrāvumu atjaunošanu, kā vēlāk būs redzams, iespējams automatizēt, daļēji pateicoties tam, ka kopas V_M objekti tiek unikāli identificēti pēc to vārdiem.

Ņemot vērā divas augstākminētās definīcijas, tagad iespējams ļoti īsi formulēt veicamo uzdevumu.

Uzdevums. Doti divi repozitoriji R1 un R2, kā arī repozitorija R1 fragments F. Izveidot jaunu repozitoriju R kā repozitorija R2 apvienojumu ar repozitorija fragmentu F.

Lai šis uzdevums būtu pilnībā saprotams, nepieciešams nodefinēt vēl vienu lietu – repozitorija un tā fragmenta apvienojumu.

Definīcija. Par repozitorija (M1, D1, T1) un repozitorija fragmenta (M2, D2, T2, L) **apvienojumu** sauc repozitoriju (M, D, T), kuram spēkā šādi nosacījumi:

- $V_M = V_{M1} \cup V_{M2}$
- $E_M = E_{M1} \cup E_{M2}$
- $V_D = V_{D1} \cup V_{D2}$
- $T = T1 \cup T2$
- $E_D = E_{D1} \cup E_{D2} \cup E_L$, kur $E_L = \{(x,a,y_D) \mid y_D \in V_D \ \& \ (x,a,y_{D'}) \in L \ \& \ \text{Similar}(y_D,y_{D'})\}$, kur Similar – predikāts ar lietotāja definētu patiesuma vērtību tabulu

Pirmie divi nosacījumi šajā apvienojuma definīcijā nozīmē, ka metamodeļa līmenī repozitorija un fragmenta virsotņu un šķautņu kopas tiek vienkārši apvienotas. Tā kā metamodeļa virsotņu kopā pastāv unikāla elementu identifikācija pēc vārda, tad tajā nevar parādīties divi vienāda vārda elementi – ja apvienojamajās daļās (repozitorijā un fragmentā) ir klases ar vienādiem vārdiem, rezultātā iegūtajā repozitorijā gluži vienkārši šī klase būs tikai vienā eksemplārā.

Līdz ar to nav arī īpaši jārūpējas par šķautņu atjaunošanu metamodeļa līmenī – vecajā repozitorijā pārrautās būtiskās asociācijas jau eksistē arī jaunajā repozitorijā un tā kā jaunas būtiskas klases ar fragmentu apvienošanas laikā nevar rasties, tad arī asociācijas nav nepieciešams radīt no jauna. Tiesa, šis modelis nerisina situāciju ar gadījumu, ja kādas klases vecajā repozitorijā tā jaunajā variantā tiek atzītas par nevajadzīgām – tās vienalga tiks pārnestas uz jauno repozitoriju, lai arī netiks pēc būtības izmantotas. Ja kādu iemeslu dēļ patiešām nepieciešams no tām atbrīvoties, tas darāms kā papildus darbs pēc eksporta un importa veikšanas.

Datu līmenī vienkāršais gadījums attiecas vairs tikai uz virsotnēm, kas tiek apvienotas ierastajā veidā, kā arī uz attēlojumu T, kas arī tiek definēts kā parasts kopu apvienojums. Interesantāk ir ar jaunās šķautņu kopas radīšanu, kas aprakstīta apvienojuma definīcijas pēdējajā nosacījumā. Kopējā šķautņu kopā E_D tiek vispirms ietveras visas tās šķautnes, kas piederējušas apvienojamajam repozitorijam (E_{D1}) vai apvienojamajam repozitorija fragmentam (E_{D2}). Visbeidzot nepieciešams radīt no jauna arī šķautnes, kas savienos kādas repozitorija daļas ar kādām fragmenta daļām (E_L). Šīs šķautnes tiek izrēķinātas no tām, kas padotas līdzi fragmentam kopas L izskatā. Kopa L saturēja informāciju par pārrautajām šķautnēm, daļu no kurām nepieciešams atjaunot. Katras šķautnes beigu virsotne ($y_{D'}$) tika atstāta ārpus fragmenta, un nu tai nepieciešams atrast aizvietotāju jaunajā repozitorijā (y_D). Lai to izdarītu, tiek ieviests jauns divvietīgs predikāts *Similar*, ar kura palīdzību lietotājam kaut kādā veidā būtu iespējams specificēt virsotņu līdzību un kurš tad tiktu izmantots līdzīgo virsotņu atrašanā šķautņu atjaunošanas procesa ietvaros.

Šajā brīdī ir noformulētas pamata idejas eksporta/importa mehānisma nodrošināšanā un precīzāk noformulēts veicamais uzdevums. Taču palikuši neatbildēti vairāki jautājumi par to, kā tehniski realizēt šī procesa soļus. Tas izklāstīts nodaļā 6.4.

6.4. Risinājuma tehniskās detaļas

Šajā apakšnodaļā aprakstītas idejas, kā realizēt būtiskākos nodaļā 6.3. aprakstītā eksporta/importa mehānisma soļus, izklāstot arī to dziļākās tehniskās detaļas. Būtiskākie vēl līdz šim neatbildētie jautājumi šajā sakarā ir trīs:

- Kā specificēt konkrētu repozitorija fragmentu?
- Kādā formā eksportēt specificēto repozitorija fragmentu?
- Kādā formā uzdot predikāta *Similar* patiesuma vērtību tabulu?

6.4.1. Repozitorija fragmenta specificēšana

Specificējot konkrētu repozitorija fragmentu, nepieciešams norādīt divas lietas – metamodeļa grafa apakšgrafu, kas iekļaujams fragmentā, kā arī sākuma instanci modelī, no kuras pēc tam būs iespējams sasniegt jebkuru citu fragmenta modeļa instanci. Tā kā šobrīd interesējamies vien par sakarīgiem grafiem, tad galvenā ideja metamodeļa apakšgrafa norādīšanā izskaidrojama, ieviešot *robežšķautnes* jēdzienu.

Definīcija. Par **robežšķautni** sauc tādu kopas E_M šķautni, kura sākas fragmentā iekļautā virsotnē un beidzas fragmentā neiekļautā virsotnē.

Tā kā runa šeit ir tikai par metamodeļa līmeņa šķautnēm, varam robežšķautni vieglākas uztveramības dēļ saukt arī par *robežasociāciju*. Specificējot repozitorija fragmentu, no tā metamodeļa grafa tiek izgriezts apakšgrafs, pārgriežot dažas šādas robežasociācijas, kurām atbilstošās saites vēlāk būs nepieciešams atjaunot. Nu, kad mūsu rīcībā ir robežšķautnes jēdziens, varam reducēt repozitorija metamodeļa grafa apakšgrafa specificēšanas uzdevumu uz robežšķautņu kopas specificēšanu. Ja pārējās repozitorija metamodeļa grafa šķautnes, kas nav robežšķautnes nosaucam par *normālām* šķautnēm, tad šī reducēšana var tikt noformulētā šādā veidā – repozitorija fragmentam pieder tās un tikai tās repozitorija metamodeļa grafa virsotnes, kas sasniedzamas no virsotnes $T(x_S)$, ejot tikai pa normālajām šķautnēm, kā arī tieši tās normālās šķautnes, kuru abi galapunkti pieder fragmentam (x_S – iepriekš minētā modeļa sākuma instance, no kuras sasniedzama jebkura cita modeļa instance, bet T – nodaļā 6.3. iztirzātais attēlojums).

Skaidrs, ka sākot metamodeļa grafa apakšgrafa veidošanu, iekļaujot tajā tikai vienu virsotni $T(x_S)$ un katrā solī papildinot šo apakšgrafu ar virsotnēm, kas no jau iekļautajām virsotnēm sasniedzamas ejot pa kādu no normālajām šķautnēm (un iekļaujot arī pašu šo šķautni), agri vai vēlu pienāks solis, kad neviena jauna virsotne apakšgrafā vairs iekļaut netiks. Šajā brīdī vajadzīgais metamodeļa grafa apakšgrafs būs izrēķināts. Šādai apakšgrafa specificēšanai būtiski, ka fragmenta metamodeļa daļa no pārējās repozitorija daļas tiek atdalīts ar robežšķautnēm, bet tas, protams, nenozīmē, ka visas šīs pārrautās šķautnes (precīzāk – tām atbilstošās modeļa līmeņa šķautnes) vēlāk būs nepieciešams atjaunot. Tāpēc skaidrības labad nepieciešams robežķautnes sadalīt divu veidu šķautnēs – *būtiskajās* šķautnēs, kuras būs nepieciešams atjaunot, un *ignorējamajās* šķautnēs, kuras atjaunot nebūs vajadzīgs.

Kad specificēts fragmentā iekļaujamais metamodeļa grafa apakšgrafs, varam pievērsties fragmentā iekļaujamā modeļa grafa apakšgrafa specificēšanai. Šeit gan situācija ir jau

vienkāršāka – ja iepriekš specificēts metamodeļa grafa apakšgrafs un tāpat zināma arī modeļa sākuma instance, tad modeļa grafa apakšgrafs izrēķināms pēc līdzīga rekursīva algoritma, kā tika rēķināts metamodeļa grafa apakšgrafs. Darbs tiek sākts ar sākuma instanci x_S un katrā nākamajā solī šim apakšgrafam tiek pievienotas tās repozitorija modeļa līmeņa instances, kuras no iepriekšējā solī pievienotajām instancēm sasniedzamas pa šķautni (saiti), kurai attēlojuma T kontekstā definētā atbilstošā metamodeļa grafa šķautne (asociācija) iekļauta fragmentā.

Tātad visas repozitorija metamodeļa grafa šķautnes sadalās trijās daļās – būtiskajās šķautnēs, ignorējamās šķautnēs un normālajās šķautnēs. Tehniski iespējami dažādi veidi, kā specificēt konkrētu metamodeļa grafa apakšgrafu, bet jebkurā gadījumā nepieciešams norādīt divas no šīm trim šķautņu kopām. Reālajos lietojumos parasti visērtāk norādīt būtisko šķautņu kopu, kuras vēlāk nepieciešams atjaunot, kā arī normālo šķautņu kopu, pa kurām nepieciešams plesties plašumā fragmenta specifikācijas procesā. Šim risinājuma variantam ir vismaz divas priekšrocības, salīdzinājumā ar citiem variantiem – 1) normālās šķautnes parasti ir pietiekami maz, lai tās būtu ērtāk uzskaitīt nekā ignorējamās šķautnes, kas var vest uz jebkuru citu repozitorija daļu, un 2) starp ignorējamajām šķautnēm var būt dažādu veidu sastatnes, kuras saviem lietojumiem ieviesuši modeļu transformāciju rakstītāji, līdz ar ko ne vienmēr tās iespējams pārskaitīt. Neskatoties uz šīm priekšrocībām, aprakstītais mehānisms neizslēdz iespēju norādīt arī ignorējamās šķautnes (papildus vēl kādai šķautņu kopai), ja tas kādā lietojumā izrādās ērtāk.

Tehniski katra no šķautņu kopām uzdodama kā TRemarkableString tipa objekts. Tips TRemarkableString ir speciālā formātā kodēta simbolu virkne, un tā gramatika uzdota nodaļā 6.4.4.

Lai vēl dziļāk izprastu dažādos šķautņu veidus un to izmantojamību eksporta procesā, nepieciešams formālāk definēt veidu, kā šis process darbojas. Tātad algoritms, vadoties pēc kura tiek izvērtēti norādītie šķautņu veidi, ir šāds – eksports sākas ar norādīto objektu, iekļaujot to rezultātā, un tālāk iet pa visām no tā izejošajām saitēm; katra asociācija A, kuras saite tiek aplūkota, tiek izvērtēta šādā secībā:

1. Ja A atzīmēta kā *normāla* asociācija, eksports iet pa šo saiti, pievienojot rezultātam objektu šīs saites beigu galā; pēc tam pāriet pie nākamās saites;
2. Ja A atzīmēta kā *būtiska* asociācija, eksports tālāk pa šo saiti neiet, bet tai izveido piesaistīto objektu (skatīt nodaļu 6.4.3.), ko iekļauj rezultātā; pēc tam pāriet pie nākamās saites;

3. Ja A atzīmēta kā *ignorējama* asociācija, eksports pāriet pie nākamās saites;
4. Ja A nav atzīmēta nevienā asociāciju grupā, tā tiek uzskatīta par *normālu* asociāciju, tātad eksports iet pa šo saiti, pievienojot rezultātam objektu šīs saites beigu galā; pēc tam pāriet pie nākamās saites.

Tātad faktiski nav aizliegts vienu asociāciju iekļaut arī vairākās grupās – tā tiks izvērtēta tikai pēc tiem kritērijiem, kas tiks sasniegti pirmie, ņemot vērā augstāk aprakstīto algoritmu. Kāpēc varētu rasties vēlēšanās kādu asociāciju iekļaut vairākās grupās – par to stāstīts nodaļā 6.4.4., kur izklāstīti šo grupu uzdošanas veidi, kā arī doti vairāki piemēri konkrētiem gadījumiem.

6.4.2. Eksporta formāts

Tā kā platforma GRAF nepiedāvā iebūvētus līdzekļus modeļu vai metamodeļu eksportam kādā vispārzināmā formātā, tad, būvējot šajā nodaļā aprakstīto eksporta/importa mehānismu, bija nepieciešams izdomāt savu formātu, kādā veikt eksportu. Viens būtisks faktors formāta izvēlei bija fakts, ka platformā izmantotais repozitorijs MII_REP neatbalsta vairāku repozitoriju vienlaicīgu atrašanos operatīvajā atmiņā, līdz ar ko nav iespējams kopēšanas/ielīmēšanas tipa risinājums, kad kopējamais fragments vienā repozitorijā tiek organizēts „uz vietas", kaut kā atzīmējot eksportējamos vienumus, un pēc tam pārnests uz otru repozitoriju, kamēr iepriekšējais vēl ir atvērts.

Tātad eksports veicams, iesaistot papildus starpdatni, kurā vispirms saglabāt informāciju no viena repozitorija, kamēr tas ir atvērts, un pēc tam, atverot otru repozitoriju, ielasīt tajā datnē saglabāto informāciju. Tā kā modeļu transformāciju rakstīšanā platformā GRAF pamatā tika izmantota valodu saime Lx, tad šīs saimes bāzes valoda L0 arī tika izvēlēta par kodējumu, kādā glabāt eksporta rezultātu. Pamata ideja eksporta mehānismam tātad ir šāda – eksporta laikā izveidot tādu L0 programmu, kuru izpildot jaunajā repozitorijā, notiktu korekta šī repozitorija un vajadzīgā repozitorija fragmenta apvienošana. Eksporta rezultātā izveidotajai L0 programmai tātad būtu jāsatur gan eksportējamais repozitorija fragments, gan arī kaut kāda veida papildus informācija par to, kā veikt minēto apvienošanu. Protams, šai datnei nav jāsatur pats apvienošanas algoritms, jo tas ir viens visiem eksporta/importa gadījumiem un to nav nepieciešams dublēt katram eksportam no jauna – to satur programma, kas izsauks šo uzģenerēto L0 programmu importa laikā.

Kā aprakstīts šī darba 4. nodaļā, L0 ir vienkārša modeļu transformāciju valoda, kurai eksistē kompilators, piemēram, uz valodu C++. Fakts, ka L0 ir kompilējama valoda, īsti neder tāda veida lietojumiem, kādus paredz šeit aprakstītais eksporta/importa mahānisms. Nav

pieļaujama situācija, kurā gala lietotājs pēc eksporta veikšanas būtu spiests uzģenerēto datni ar L0 programmu vēl kompilēt, pirms to būtu iespējams izmantot importa procesā. Tādēļ eksporta/importa sekmīgai nodrošināšanai bija nepieciešams radīt L0 translatoru – programmu, kas L0 programmas kodu spēj interpretēt programmas izpildes laikā. Tiesa, nebija nepieciešams interpretēt valodu L0 visā pilnībā, bet gan aplūkot tikai tās komandas, kuras principā var tikt uzģenerētas eksporta procesa rezultātā. Tā kā eksports ģenerē L0 programmu, kas pēc tam repozitoriju papildinās ar jaunu fragmentu, tad šādā programmā var parādīties pamatā tikai tās L0 programmas, kas veido modeli vai metamodeli. Šādas komandas ir pavisam astoņas:

- *addClass* – jaunas klases izveide
- *addAttr* – jauna atribūta pievienošana klasei
- *addAssoc* – jaunas parastas asociācijas starp klasēm izveide
- *addCompos* – jaunas kompozīcijas tipa asociācijas starp klasēm izveide
- *addRel* – jaunas vispārināšanas tipa attiecības starp klasēm ieviešana
- *addObj* – jauna objekta izveide
- *setAttr* – vērtības piešķiršana atribūtam
- *addLink* – jaunas saites starp objektiem izveide

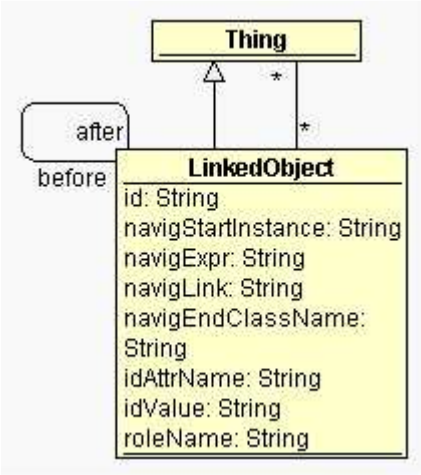
Papildus šīm komandām tehniskiem nolūkiem eksporta rezultātā iespējams vēl parādīties komandai *setPointerF*, ar kuras palīdzību kādai norādei piešķirt vērtību. Tātad bija nepieciešams uzbūvēt interpretatoru, kurš spētu izpildes laikā izpildīt šīs augstākminētās L0 komandas, kā arī interpretēt pārējo tehnisko L0 programmas koda daļu (piemēram, apsaimniekot deklarācijas blokā norādīto norāžu saimniecību).

Interpreators ticis uzrakstīts valodā C++, bet tā izsaukumi kopā ar pārējo importa saimniecības nodrošinājumu (saišu atjaunošanu) rakstīti valodā L0.

6.4.3. Līdzīgo objektu meklēšana repozitorijā

Galvenā ideja pārrauto saišu atjaunošanā ir ieviest repozitorijā jaunu klasi, kuras objekti tiktu veidoti katram šādam pārrāvumam un kuros noteiktā veidā tiktu nokodēts algoritms sākotnējam objektam līdzīgā objekta atrašanai jaunajā repozitorijā. Katrs šāds objekts tad tiktu piesaistīts pie tā fragmenta objekta, kurā sākusies pārrautā saite, un eksportēts uz L0 programmas datni kopā ar fragmentu. Vēlāk, importa laikā, visi šāda veida objekti tiktu interpretēti un katram no tiem atrasts vajadzīgais repozitorija objekts, kurš tad tiktu piesaistīts ieimportētajam objektam.

Attēlā 6.5. redzams repozitorija metamodeļa fragments, kas satur minēto jaunieviesto klasi – „LinkedObject". Ar klasi „Thing" šeit jāsaprot visu fragmentā iekļaujamo eksportējamo klašu virsklase. Tā var būt vai nu statiska repozitorijā nepārtraukti eksistējoša visu klašu virsklase vai arī tā var tikt izveidota tikai eksporta vajadzībām un padarīt par virsklasi tikai reāli eksportējamajām klasēm. Pašreizējā lietojumā šī klase repozitorijā atrodas pastāvīgi un ir padarīta par virsklasi tām repozitorija klasēm, kuras varētu tikt eksportētas iepriekš minētajos lietojumos – rīka vai platformas izmaiņu nodrošināšanai.



6.5. attēls. **Klase „LinkedObject", kuras instances var tikt piesaistītas jebkuram eksportējamajam objektam**

Kā redzams, klase „LinkedObject" satur vairākus atribūtus, kuros tiek nokodēta informācija priekš vajadzīgā objekta repozitorijā meklēšanas apgoritma (zemāk atrodamā aprakstā minētā datu tipa TNavigationString tehniskais kodējums atrodams nodaļā 6.4.4.):

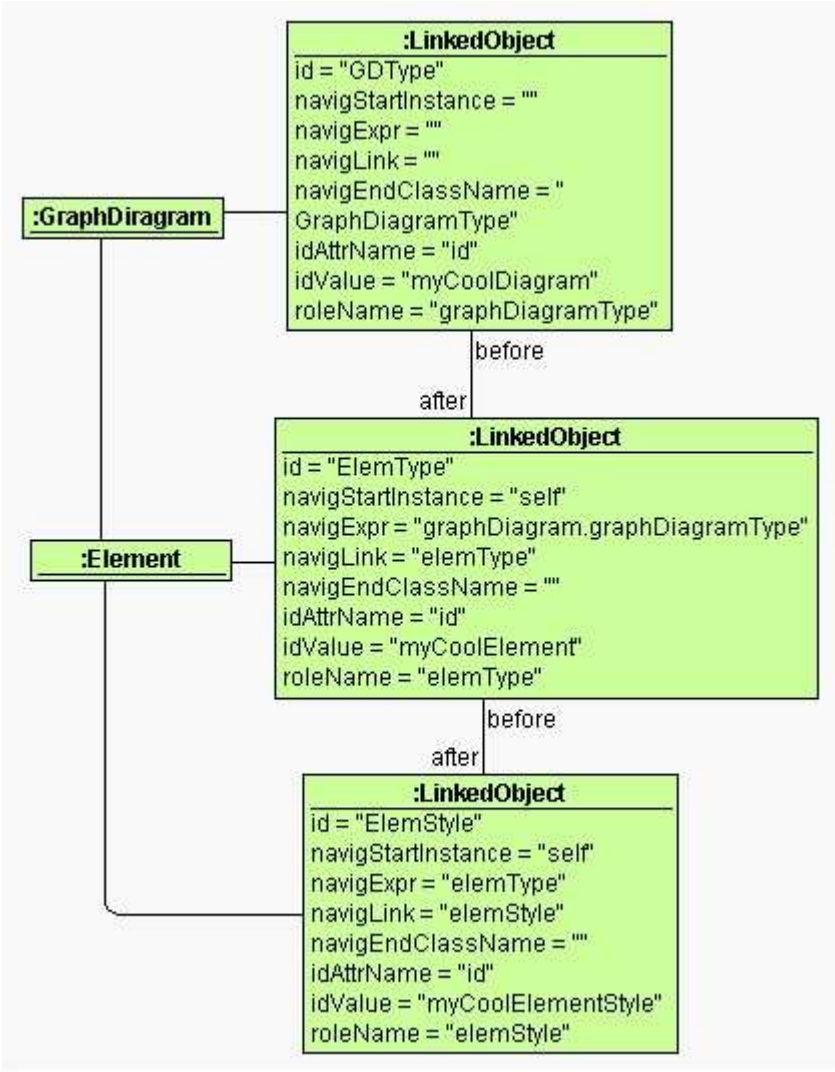
- id – unikāls identifikators, pēc kura šīs klases objektu var būt eksporta laikā vajadzīgs atrast.
- navigStartInstance – TNavigationString tipa simbolu virkne, ar kuras palīdzību tiek norādīts, kā atrast repozitorija instanci, ar kuru sākt vajadzīgā objekta meklēšanas procesu. Patiesībā vajadzīgā objekta meklēšana var būt divējāda – atkarīga no konteksta (noteikta sākuma objekta) vai neatkarīga no tā. Šo navigācijas starta instanci nepieciešams norādīt, ja vēlamies izmantot pirmā veida meklēšanu. Ja navigStartInstance vērtība vienāda ar tukšu simbolu virkni, meklēšana notiks pēc otrā veida parauga (šādā gadījumā nav vairs arī nozīmes atribūtiem navigExpr un navigLink, bet varam pievērsties tālākajiem atribūtiem aiz tiem).
- navigExpr – TNavigationString tipa simbolu virkne, ar kuras palīdzību tiek uzdots ceļš, kā, sākot no starta objekta un ejot pa saitēm, iespējams nokļūt līdz kopai, no

kuras vajadzīgais objekts sasniedzams pa vienu saiti. Ja navigExpr vērtība vienāda ar tukšu simbolu virkni, tad meklējamais objekts ir pats starta objekts.

- navigLink – lomas vārds saitei, pa kuru ejot no kopas, kas iegūta, izpildot iepriekš aprakstīto navigācijas izteiksmi, iespējams iegūt tādu objektu kopu, kas jau satur meklējamo objektu.
- navigEndClassName – ja meklēšana nav atkarīga no konteksta (noteikta sākuma objekta), tas nozīmē, ka nepieciešams atrast konkrētas klases konkrētu objektu. Šajā atribūtā šādā gadījumā tiek norādīts vajadzīgās klases vārds. Ja meklēšana notiek pirmajā veidā – ar navigācijas izteiksmju palīdzību – šī atribūta vērtībai nav nozīmes.
- idAttrName – pieņemot, ka meklēšana nonākusi līdz noteiktai objektu kopai (vai nu, ejot pa navigācijas izteiksmē norādīto ceļu pirmajā gadījumā, vai arī aplūkojot noteiktas klases visus elementus otrajā gadījumā), šī atribūta vērtība ir tā atribūta vārds, pēc kura vēlamies veikt meklēšanu šajā objektu kopā.
- idValue – pieņemot, ka meklēšana nonākusi līdz noteiktai objektu kopai, šajā atribūtā norādām meklējamā objekta dotā atribūta (tā, kura vārds norādīts atribūtā idAttrName) vērtību.
- roleName – lomas vārds pārrautās saites otrā galā (galā pie no jauna atrastā objekta), ar kādu šī saite jāatjauno

Papildus minētajiem atribūtiem, klases „LinkedObject” instances savā starpā var būt saistītas arī ar saiti „before/after”, kas norāda, ka kāds no šiem objektiem interpretējams tikai pēc tam, kad jau interpretēts kāds cits objekts (ja pirmais objekts, piemēram, savā navigācijas izteiksmē atsaucas uz saiti, kura vēl tikai tiks atjaunota otrā objekta interpretācijas laikā).

Attēlā 6.6. redzams klases „LinkedObject” objektu izmantošanas piemērs. Šeit ir vēlēšanās eksportēt kādu konkrētu modeli (grafveida diagrammu) uz jaunu problēmorientētās valodas un rīka versiju. Jaunā versija nosaka klašu „GraphDiagramType”, „ElemType” un „ElemStyle” instances, bet modelis sastāv no klašu „GraphDiagram” un „Element” instancēm, kuras vēlamies eksportēt.



6.6. attēls. **„LinkedObject" lietojuma piemērs**

Pieņemsim, ka mūsu eksportējamā grafveida diagramma sastāv tikai no viena elementa bez nodalījumiem. Tādā gadījumā modeļa eksporta rezultātā tiks pārrautas trīs saites (patiesībā jau vairākas, bet aplūkosim tikai šīs trīs) – „graphDiagram/graphDiagramType" starp grafveida diagrammu un tās tipu, „element/elemType" starp elementu un tā tipu un „element/elemStyle" starp elementu un tā stilu, kas arī var tikt noteikts problēmorientētās valodas specifikācijā.

Katrai no pārrautajām saitēm tiek izveidots viens klases „LinkedObject" objekts, kā redzams attēlā 6.6. Šiem objektiem šajā gadījumā ir arī stingri noteikta secība – vispirms jāinterpretē grafveida diagrammas tipa saiti atjaunojošais objekts, tad elementa tipu atjaunojošais objekts un visbeidzot elementa stilu atjaunojošais objekts.

Lai atjaunotu grafveida diagrammas tipu, tiek izmantota otrā veida meklēšana, kas nav atkarīga no konteksta. Tiek pieņemts, ka problēmorientētās valodas specifikācijas izmaiņas nav ietekmējušas galvenos tipus, stilus un citus aspektus identificējošos atribūtus. Līdz ar to, lai jaunajā repozitorijā atrastu pareizo diagrammas tipu, tas tiek meklēts kā klases

„GraphDiagramType” instance (jo navigStartInstance = „” un navigEndClassName = „GraphDiagramType”) ar atribūtu „id” (jo idAttrName = „id”) vienādu ar „myCoolDiagram” (jo idValue = „myCoolDiagram”). Šādā veidā tiek atrasts īstais diagrammas tips un piesaistīts pie apstrādājamās diagrammas (tās, pie kuras piesaistīts apstrādājamais „LinkedObject” objekts) ar saiti, kuras lomas vārds tipa galā vienāds ar „graphDiagramType” (jo roleName = „graphDiagramType”).

Lai atjaunotu elementa tipu, tiek izmantota pirmā veida meklēšana, jo šoreiz nepietiek meklēt kādu noteiktu klases „ElemType” instanci pēc kāda atribūta vērtības, bet nepieciešams ņemt vērā arī diagrammas tipu, kuras ietvaros šī meklēšana jāveic (dažādi diagrammu tipi varētu saturēt elementu tipus ar vienādām identificējošo atribūtu vērtībām). Tātad šajā gadījumā meklēšana tiek sākta no paša apstrādājamā elementa (jo navigStartInstance = „self”) un tālāk tiek mēģināts atrast grafveida diagrammu, kurā šis elements atrodas un no tās – tās tipu, kurš šajā brīdī jau ir atrasts un piesaistīts, jo augstākminētais diagrammas tipu atjaunojošais „LinkedObject” objekts ir izpildījies vispirms. Šāda veida navigācijas ceļu nosaka atribūts navigExpr = „graphDiagram.graphDiagramType”. Šajā brīdī esam ieguvuši īsto diagrammas tipu, kuras kontekstā jāmeklē vajadzīgais elementa tips un meklējamā kopa iegūstama, ejot no šī diagrammas tipa pa saiti „elemType” (jo navigLink = „elemType”). Tālāk jau darbošanās notiek līdzīgi tai, kas aprakstīta iepriekšējā rindkopā – atrastajā kopā tiek meklēts objekts, kuram atribūta „id” (jo idAttrValue = „id”) vērtība vienāda ar „myCoolElement” (jo idValue = „myCoolElement”). Saite tiek atjaunota ar lomas vārdu „elemType” (jo roleName = „elemType”) elementa tipa galā.

Visbeidzot arī saite no elementa uz tā stilu notiek pēc pirmā veida meklēšanas parauga, sākot meklēšanu no paša elementa un ejot pa saiti „elemType” (kura tikko kā atjaunota), tādējādi atrodot elementa tipu, kura kontekstā tālāk caur saiti „elemStyle” iegūstama kopa, kurā īstais stils tiek atrasts pēc atribūta „id” vērtības „myCoolElementStyle”.

Šajā piemērā skaidri iezīmējas vairāki šabloni, kas nosaka vajadzīgā objekta meklēšanu repozitorijā atkarībā no konteksta vai bez tā. Skaidrs, ka visiem elementiem, piemēram, tipa meklēšanas ceļš vienmēr būs viens un tas pats – ejot caur šim elementam piesaistīto diagrammu, tad caur tās tipu un tad pārmeklējot visus šim diagrammas tipam piesaistītos elementu tipus. Līdz ar to īstenībā nav nepieciešams veidot jaunu klases „LinkedObject” instanci katrai atjaunojamajai saitei, bet pietiek izveidot pa vienam katrai līdzīgo saišu grupai. Piemēram, elementa tipu atjaunojošais objekts būtu tikai viens vienīgs, bet piesaistīts visiem elementiem, kuru tipus nepieciešams atjaunot. Rezultātā fragmentā iekļaujamo un papildus

eksportējamo objektu kopu var vērtēt kā konstantu (pie tam nelielu) lielumu – tas nav atkarīgs no eksportējamā fragmenta apjoma.

Visbeidzot jāatbild uz jautājumu, kā ērtā veidā iespējams specificēt klases „LinkedObject" instances eksporta veikšanas laikā. Viens variants ir darīt to manuāli, izveidojot un piekārtojot vajadzīgajiem eksportējamajiem objektiem minētās instances. Taču tā kā šis process ir samērā vienveidīgs un kopīgs visam mehānismam, tad tas ir padarīts universāls. Šādus objektus nepieciešams veidot katrai no būtiskajām šķautnēm, tāpēc jau šo šķautņu uzdošanā (tipa TRemarkableString simbolu virknē) paredzēta vieta objekta piesaistīšanai. Tehniski tas izdarāms, norādot tādas modeļu transformācijas vārdu, kura atgriež tā klases „LinkedObject" objekta atribūta „id" vērtību, kurš piekārtojams aplūkojamās būtiskās asociācijas sākuma gala objektiem. Šai transformācijai tiek padots sākuma gala objekts, kuram būs nepieciešams atrast beigu objektu, un tā, kad izsaukta, tad var vai nu izveidot jaunu klases „LinkedObject" instanci un atgriezt tās atribūta „id" vērtību, vai arī atrast jau iepriekš izveidotu instanci un atgriezt tās vērtību. Transformācijas vārda norādīšana TRemarkableString objektā aprakstīta nodaļā 6.4.4. Jāpiezīmē, ka biežākajiem eksporta/importa mehānisma lietojumiem izstrādāta arī bibliotēka ar funkcijām, kas veido klases „LinkedObject" instances visiem nepieciešamajiem gadījumiem.

6.4.4. Šķautņu uzdošanas un navigācijas izteiksmju formāti

Visu triju veidu šķautņu – būtisko, normālo un ignorējamo – uzdošanai lietojams datu tips TRemarkableString, kas pēc būtības ir speciālā formātā uzdota simbolu virkne. Šeit šī simbolu virkne precizēta ar bekonteksta gramatikas palīdzību, izmantojot Bekusa-Naura normālformu:

```
<TRemarkableString> ::= „[<list>]"
<list> ::= <item>[;<list>]
<item> ::= <className>[.<assocName>][-[<dllName>#]<procName>]
```

Šajā gramatikā izmantotie neaprakstītie papildus simboli:

- <className> – repozitorijā eksistējošas klases vārds
- <assocName> – repozitorijā eksistējošas no attiecīgās klases izejošas asociācijas vārds (precīzāk – šīs asociācijas lomas vārds pretējā galā)
- <dllName> – eksistējošas .dll datnes vārds, kurā atrodas procedūra <procName> (ja <dllName> tiek izlaists, tad <procName> tiek meklēts platformas GRAF noklusētajā bibliotēkā „main.dll")

- <procName> – norādītajā <dllName> eksistējošas funkcijas ar vienu parametru – norādi uz repozitorija objektu – vārds (šī funkcija norādāma tikai būtiskajām asociācijām un atgriež pie aplūkojamās asociācijas sākuma gala objektiem pievienojamā klases „LinkedObject” instances atribūta „id” vērtību)

TRemarkableString simbolu virkne tātad sastāv no vairāku asociāciju saraksta, kur katru asociāciju iespējams specificēt kā klases vārdu kopā ar lomas vārdu no šīs klases izejošas asociācijas pretējā galā. Ja šis lomas vārds (simbols <assocName>) netiek norādīts, ar to tiek saprasts apakšsaraksts ar visām no aplūkojamās klases izejošajām asociācijām. Šāda noruna daudzos gadījumos ievērojami atvieglo asociāciju virkņu veidošanu, ļaujot izvairīties no nepieciešamības pārskaitīt pilnīgi visas no kādas klases izejošās asociācijas.

Aplūkosim dažus korektus TRemarkableString virkņu piemērus:

- „” – tukša virkne, kas nesatur nevienu asociāciju.
- „GraphDiagram.element” – virkne, kas satur vienu asociāciju – no klases „GraphDiagram” izejošo asociāciju ar lomas vārdu „element” pretējā galā.
- „GraphDiagram.graphDiagramType-getLOForGDType” – būtisko asociāciju virknes piemērs, kas satur vienu asociāciju (no grafveida diagrammas uz tās tipu) un kurai norādīta funkcija, kas atgriež visām grafveida diagrammām piekārtojamā klases „LinkedObject” atribūta „id” vērtību. Šajā gadījumā funkcija „getLOForGDType” tiks meklēta noklusētajā funkciju bibliotēkā „main.dll”.
- „GraphDiagram.graphDiagramType-getLOForGDType;Element.elemType-myCoolDll.dll#getLOForElemType” – virkne, kas sastāv no divām asociācijām, no kurām pirmā ir tieši tāda pati, kā iepriekšējā pirmērā, bet otrai norādīto piesaistāmā objekta identifikatora atgriešanas funkciju „getLOForElemType” norādīts meklēt bibliotēkā „myCoolDll.dll”.
- „GraphDiagramType” – virkne ar visām tām asociācijām, kas iziet no minētās klases (iepriekš nefiksētā skaitā).
- „GraphDiagram.element;Element.compartment;Compartment.subCompartment” – virkne ar vairākām uzskaitītajām asociācijām, kas kopumā definē grafveida diagrammas kokveida struktūru, iekļaujot tajā diagrammas elementus, to nodalījumus un apakšnodalījumus.

Kā jau minēts iepriekš, vienas un tās pašas asociācijas iekļaušana dažādu asociāciju grupu sarakstos nav aizliegta. Tas var izrādīties noderīgi situācijās, kad dažas no kādas klases izejošās asociācijas vēlamies, piemēram, pasludināt par būtiskajām asociācijām, bet par visām

pārējām pateikt, ka tās ir ignorējamas. Aplūkosim konkrētu piemēru, kur tas ir ērti – vienas grafveida diagrammas eksportēšana. Viens no variantiem, kā specificēt eksportējamo repozitorija fragmentu, ir, norādot būtiskās, ignorējamās un normālās asociācijas šādā veidā (uzskatāmības dēļ šeit izlaista funkciju specifikācija būtisko asociāciju gadījumā, jo tā šajā piemērā netiek aplūkota):

```
essential = "GraphDiagram.graphDiagramType;Element.elemType;
Compartment.compartType;Element.elemStyle; Compartment.compartStyle"
normal = "GraphDiagram.element;Element.compartment;
Compartment.subCompartment;Element.eStart;Element.eEnd"
ignorable = "GraphDiagram;Element;Node;Edge;Compartment"
```

Šeit būtiskās asociācijas ir tās, kas tiks pārrautas un kuras nepieciešams atjaunot – diagrammas, elementa un nodalījuma tipus piesaistošās asociācijas, kā arī elementa un nodalījuma stilu piesaistošās asociācijas. Normālās asociācijas, pa kurām eksports virzīsies, ir tās, kas definē diagrammas kokveida struktūru, bet visas pārējās asociācijas ir vēlēšanās pasludināt par ignorējamām. Tas ērti izdarāms, norādot tikai fragmentā iekļauto klašu vārdus – šādā veidā ignorētas tiks visas no šīm klasēm izejošās asociācijas, kuras nebūs iekļautas nevienā citā grupā. Redzams, ka, piemēram, asociācija „graphDiagramType”, kas iziet no klases „GraphDiagramType” iekļauta divās grupās – gan pie būtiskajām, gan pie ignorējamajām asociācijām. Šajā gadījumā saskaņā ar nodaļā 6.4.1. aprakstīto eksporta algoritmu šī asociācija vispirms tiks izvērtēta kā būtiskā asociācija un līdz ar to algoritms nemaz nenonāks pie tās kā ignorējamās asociācijas izvērtēšanas. Tātad šādi rīkoties ir korekti.

Otrs simbolu virkņu formāts, kas tika aplūkots iepriekšējās apakšnodaļās, bija TNavigationString, kas paredzēts navigācijas izteiksmju kodēšanai klases „LinkedObject” objektu atribūtos. Šī tipa virknes pēc būtības ir ļoti vienkāršas, jo satur tikai asociāciju lomu vārdu sarakstu, kas atdalīts ar punktiem:

```
<TNavigationString>   ::= „[<list>]”
<list>                ::= <assocName>[.<assocName>]
```

6.5. Eksporta/importa mehānisma lietojumi

Divi no eksporta/importa mehānisma lietojumiem jau tika detalizēti iztirzāti šīs nodaļas sākumā un kalpoja par motivāciju mehānisma radīšanā. Tā bija nepieciešamība tikt galā ar noteikta veida izmaiņām, kas saistītas ar rīku būves platformu GRAF.

Pirmkārt, izmaiņas vai notikt kādā no problēmorientētajām valodām, ko platforma atbalsta, un līdz ar to arī ar attiecīgo problēmorientēto rīku, kas šo valodu realizē. Mainoties šādas valodas specifikācijai, rodas nepieciešamība šim jaunajam specifikācijas variantam piesaistīt ar specifikācijas iepriekšējo variantu radītos konkrētos modeļus, lai tie būtu darboties spējīgi arī turpmāk. Izmantojot šajā nodaļā aprakstīto eksporta/importa mehānismu, iespējams vecos modeļus izeksportēt no vecās specifikācijas rīka un ieimportēt jaunās specifikācijas rīkā, pēc iespējas saglabājot tos datus, kurus iespējams saglabāt attiecībā pret specifikāciju maiņu.

Otrkārt, izmaiņas var notikt pašā platformā, precīzāk sakot – problēmorientētajā valodā, kas izstrādāta priekš citu problēmorientētu valodu un rīku izstrādes. Mainoties šīs augstākā līmenī esošās valodas specifikācijai, parasti līdzi mainās arī rīks, kas šo valodu realizē un ar kura palīdzību izstrādāti citi rīki. Eksporta/importa mehānisms paredz iespēju eksportēt veselus izstrādātos rīkus (gan šo rīku specifikāciju, gan arī pašu rīku diagrammas, kas radītas ar augstākā līmeņa rīku izstrādes rīka jeb konfiguratora palīdzību) no vecās platformas versijas un ieimportēt tos jaunajā versijā.

Tā kā mehānismā nav iebūvēti nekādi priekšdefinētie gadījumi, kādus modeļu fragmentus vai kādas diagrammas iespējams eksportēt, tad, korekti norādot eksportējamā fragmenta robežas, mehānismu iespējams izmantot arī patvaļīgu repozitorija datu eksportēšanai no viena repozitorija un importēšanai citā. Tas lietojams, piemēram, gadījumos, kad pie viena un tā paša projekta vienlaikus strādā vairāki cilvēki, katrs pie sava datora, kas ik pa laikam vēlas savā starpā apmainīties ar savu saražoto informāciju (piemēram, savām diagrammām). Eksporta/importa mehānisms šādam mērķim der lieliski. Tāpat, izmantojot šo mehānismu, viens lietotājs var apmainīties ar datiem starp dažādiem repozitorijiem uz sava datora.

Vēl kāds interesants eksporta/importa mehānisma lietojums atrodams daudzlietotāju režīma nodrošināšanas kontekstā. Platformā GRAF eksistē daudzlietotāju saskarne, kas realizē daudzlietotāju režīmu. Tā pamatā ir ideja par vienu centrālo serveri, uz kura glabājas centrālais projekta repozitorijs, no kura vairāki lietotāji vajadzības gadījumā var pie sevis „atvilkt” noteiktas šī repozitorija daļas, kas pēc būtības atbilst veselām grafveida diagrammām. Pēc pastrādāšanas ar šīm diagrammām tās iespējams „nolikt” atpakaļ uz servera. Pamata funkcionalitāte šai diagrammu sūtīšanai uz un no servera realizēta, izmantojot eksporta/importa mehānismu – katra diagramma vai diagrammu kopa tiek izeksportēta no klienta repozitorija (gadījumā, ja klients vēlas nosūtīt savu diagrammu uz servera) un saglabāta atsevišķā teksta datnē kā programmas teksts. Šāda veida datnes tad tiek kopētas uz serveri un atpakaļ. Var teikt, ka principā servera galā viss projekts tiek glabāts kā šādu datņu

kolekcija, tādējādi padarot daudzlietotāju režīmu vieglāk saprotamu. Protams, daudzlietotāju režīmam nepieciešams nodrošināt arī vēl citas iespējas, piemēram, diagrammu bloķēšanu, lai nerastos kolīzijas gadījumos, kad vairāki lietotāji vienlaicīgi vēlas rediģēt vienu un to pašu diagrammu.

6.6. Nākotnes plāni

Pie nākotnes plāniem minamas vairākas idejas, kā turpināt eksporta/importa mehānisma uzlabošanu dažādos aspektos.

Pirmais aspekts saistāms ar repozitorija fragmenta specificēšanu. Kā jau izklāstīts iepriekš, repozitorija fragments tiek specificēts, norādot tā sākuma objektu modelī un specificējot metamodeļa grafa sakarīgu apakšgrafu, kura robežās no starta objekta plesties plašumā pa asociācijām. Šajā sakarā iespējams padomāt par diviem procesa pilnveidošanas soļiem.

Pirmkārt, metamodeļa grafa apakšgrafa uzdošana šobrīd notiek, norādot kādas no trim asociāciju grupām – būtiskās, normālās un ignorējamās –, kur katra grupa tiek uzdota kā tipa TRemarkableString objekts. No gala lietotāja viedokļa ērtāk būtu šīs asociāciju grupas uzdot nevis tekstuālā, bet kādā intuitīvākā grafiskā formātā. Iespējams izstrādāt problēmorientētu grafisku valodu tieši metamodeļa grafa apakšgrafa uzdošanai, ar kuras palīdzību ērtā veidā ātri un nekļūdīgi būtu iespējams norādīt vēlamo fragmentu.

Otrkārt, šī brīža risinājumā būtiska repozitorija fragmenta īpašība ir tā, ka uzdotais metamodeļa grafa (un līdz ar to arī modeļa grafa) apakšgrafs ir sakarīgs. Ja vienā eksportēšanas reizē (uz vienu kopīgu eksporta datni) nepieciešams eksportēt vairākas savstarpēji nesaistītas modeļa daļas, to izdarīt nav tik viegli (nepieciešams veikt zināmu manuālu darbu, lai kopā sasaistītu vairākus eksporta rezultātus). Šobrīd gan nav zināms neviens konkrēts lietojums, kur šāda vēlme varētu parādīties, bet ir vērts padomāt par to, vai eksportu nav iespējams vispārināt uz patvaļīgākiem repozitorija fragmentiem, ne tikai sakarīgiem.

Vēl ir kāda apdomāšanas vērta lieta, kas gan tieši nesaistās ar pašu eksporta/importa mehānismu, bet tikai ar vienu tā posmu – specificēto repozitorija fragmentu. Iespējams, ka šādi norādītam repozitorija fragmentam var atrast vēl citus pielietojumus bez tā, ka tas var tikt eksportēts uz teksta datni. Piemēram, šādā veidā specificētu fragmentu var vēlēties no repozitorija izdzēst. Cits piemērs – ja specificējam kādu fragmentu, kas atbilst kādam grafiskam attēlojumam, var būt vēlēšanās kaut kādā veidā mainīt tā attēlošanas veidu –

iezīmēt to citā krāsā vai mainīt visiem fragmentā esošajiem elementiem formu. Tikpat labi – specificēto fragmentu var gribēt aplūkot kā HTML vai SVG formātā attēlotu datni. Vispārīgā gadījumā – ja repozitorija fragments ir specificēts, tam varētu tikt piekārtota noteikta ar to veicamo darbību virkne. Visi šie jautājumi noteikti nākotnē ir apdomāšanas vērti.

7. Platformas tālāka attīstīšana

Promocijas darbā aprakstītie pētījumi nav pabeigti, bet atrodas pastāvīgā procesā – platformas uzlabošana, attīstīšana ir nebeidzams darbs. Promocijas darba autors iesaistījies arī tālākos pētījumos par platformas nākotnes attīstības virzieniem. Divi no šiem tālākās pētniecības virzieniem ir UML stereotipu piedāvāto iespēju paplašināšana un izmantošana problēmorientētu valodu un rīku būvē, kā arī platformas balstīšana uz metamodeļa specializācijas jēdziena.

UML stereotipu paplašināšana un izmantošana problēmorientētu valodu un rīku būvē aprakstīta autora publikācijā [45]. Pamata princips šeit ir ļaut gala lietotājam veidot savas valodas un rīkus, kas tās realizē, balstoties uz kādu vispārīgu grafveida diagrammu redaktoru un pielāgojot izmantojamos valodas primitīvus savām individuālajām vajadzībām. Publikācijā aprakstītais valodas papildināšanas mehānisms ļauj viegli gan papildināt/izmainīt jau esošas valodas, gan arī it kā veidot jaunas valodas no nulles (reāli valoda tiks veidota uz grafveida diagrammas aprakstošās valodas bāzes, bet gala lietotājam tiks radīts iespaids, it kā viņš valodu veidotu no nulles).

Izmantojot šādu paplašināta stereotipa jēdzienu, iespējams gan izmainīt jau esošu valodas sastāvdaļu (elementu – kastu un līniju –, nodalījumu, paletes elementu, izvēlņu utt.) definīcijas, gan arī veidot jaunas sastāvdaļas. Tāpat iespējams specificēt arī dažāda veida atkarības starp elementiem, kā arī piesaistīt tiem vizuālos attēlošanas veidus. Publikācijā nedaudz ieskicēta arī paplašināto stereotipu mehānisma saistība ar šajā darbā aprakstīto skatu mehānismu.

Otrs iespējamais platformas attīstīšanas virziens nākotnē – metamodeļa specializācijas jēdziens – aprakstīts citā autora publikācijā [46]. Šeit galvenais uzsvars likts uz valodu un rīku izstrādes procesa atvieglošanu valodu/rīku izstrādātājiem. Publikācijā sperts viens solis tālāk ceļā uz atvērtāku grafisku rīku būves platformu, kas ļautu vienkāršā veidā gan koriģēt rīka uzvedību, gan arī piekļūt rīkam no ārējaām lietotnēm.

Šajā publikācijā aprakstītais metamodeļa specializācijas jēdziens ir pati par sevi interesanta lieta. Metamodeļu specializācija definēta neatkarīgi no rīku būves un izmantojama arī citās sfērās. Otrajā publikācijas daļā demonstrēts, kā šādu specializāciju iespējams eleganti izmantot tieši rīku būves procesā. Norādīti arī galvenie šādas pieejas ieguvumi, salīdzinot ar citām pasaulē eksistējošām šādu rīku būves metodēm – dabiska un intuitīvi skaidra rīku

definēšana, stingra rīka loģikas nodalīšana no prezentācijas daļas, kā arī atvērtība gan rīka uzvedības manuālai paplašināšanai, gan arī ārējo lietotņu pieejamībai.

Tā kā šie pētījumi atrodas to sākuma stadijā un tos paredzēts turpināt tālāk pēc promocijas darba pabeigšanas, tad tie šajā darbā sīkāk detalizēti netiks.

8. Nobeigums un secinājumi

Šajā darbā aprakstīti galvenie rezultāti pētījumam, kurā autors piedalījies, strādājot Latvijas Universitātes Matemātikas un informātikas institūtā. Par pētījuma galveno apskates objektu kalpojušas uz metamodeļiem balstītas grafisku problēmorientētu rīku būves platformas un to piedāvātās iespējas. Pētījuma procesā tapusi jauna šāda veida platforma GRAF, kas laika gaitā tikusi apaudzēta ar dažādu papildus funkcionalitāti. Šī darba autors piedalījies platformas izstrādē jau kopš paša tās rašanās brīža un turpina tajā piedalīties vēl joprojām. Kā autora galveno ieguldījumu platformas attīstībā jāmin dažāda veida papildus funkcionalitātes izstrāde, tādējādi atvieglojot platformas lietošanu dažādos aspektos – gan no pašas platformas gala lietotāju (rīku būvētāju), gan arī no ar platformu izstrādāto rīku gala lietotāju (modeļu veidotāju) viedokļa.

Lai darbs būtu lasāms neatkarīgi no lasītāja priekšzināšanām aplūkotajās sfērās, tā sākumā dots neliels ieskats pamata problemātikā, kā arī izskaidroti paši būtiskākie termini, ar kuriem būtu jābūt pazīstamam ikvienam, kas vēlas spēt iedziļināties tālākajās darba nodaļās. Tā kā aplūkotā platforma GRAF pēc savas būtības darbojas ar repozitoriju, kas balstīts uz metamodeļiem, tad vispirms īsi izskaidrots modeļa un metamodeļa jēdziens. Modeļu (un arī metamodeļu) apstrāde tradicionāli notiek, izmantojot modeļu transformācijas, tādēļ dots ieskats arī šajā programmēšanas valodu speciālgadījuma pasaulē. Tāpat dots arī vispārīgs apskats par arhitektūru, kas tiek izmantota tādu sistēmu, kas balstītas uz metamodeļiem, būvē – uz modeļu balstītu arhitektūru (MDA). Visbeidzot – galvenais platformas GRAF uzdevums ir dot gala lietotājam iespēju būvēt savus problēmorientētos rīkus, kas realizē kādas speciālas problēmorientētas valodas. Līdz ar to otrā nodaļa pabeigta ar šādu problēmorientētu valodu aprakstu.

Tālākās nodaļas uzskatāmas par darba pamata sastāvdaļu – tajās aprakstīta platforma GRAF un tās funkcionalitāte, kas attiecas uz augstāk minētā pētījuma daļu, kurā savu ieguldījumu devis promocijas darba autors. Trešajā nodaļā vispārīgi aprakstīta platforma GRAF, tās izstrādes īsa vēsture un tās galvenie darbības principi. Tāpat aprakstītas arī būtiskākās platformas komponentes – uz transformācijām balstītā arhitektūra, rīku definēšanas saskarne un citas. Šeit arī noformulēti galvenie platformas izaicinājumi, kuru risinājumu varianti piedāvāti tālākajās darba nodaļās.

Tālākajās trijās darba nodaļās – ceturtajā, piektajā un sestajā – piedāvāti autora izvirzītie un realizētie risinājumi trešajā nodaļā īsi aprakstītajām problēmām. Šie risinājumi iekļauj sevī

augstāka līmeņa modeļu transformāciju valodu saimes izstrādi un realizāciju, metamodeļa skata jēdziena definēšanu un ieviešanu, kā arī modeļu un metamodeļa eksporta un importa funkcionalitātes realizāciju. Katram no šiem gadījumiem dota motivācija un problēma, ko tie atrisina. Tāpat izdarīti secinājumi par citiem aprakstītā risinājuma iespējamajiem lietojumu variantiem, kas, iespējams, nav tiešā saistībā ar problēmu, kuru tie risina, bet kas attīstījušies kā pozitīvi aplūkotā risinājuma blakusefekti. Kur iespējams, ieskicēti tālāku pētījumu ceļi saistībā ar aplūkoto risinājumu.

Pētījuma veikšanas laikā radušies dažādi secinājumi par uz metamodeļiem balstītu grafisku problēmorientētu rīku būves platformu attīstības ceļiem. Konkrētie secinājumi par aplūkotajām un risinātajām problēmām minēti darba galvenajās nodaļās. Kā vienojošs secinājums par visu platformu kopumā minams tas, ka platformas lietojamība, kā jau vairums datu kvalitātes atribūtu, var vien tiekties uz pilnību, to nekad nesasniedzot. Risinot darbā aprakstītās (un arī citas ar platformu saistītas) problēmas, procesā arvien radās jaunas idejas, kuras būtu labi īstenot, lai platformas lietošanu padarītu vēl ērtāku no gala lietotāja viedokļa. Šāda ideju ģenerēšana notika vēl jo intensīvāk tādēļ, ka vai ikvienā aspektā nepieciešams iedziļināties no divu savstarpēji ļoti atšķirīgu veidu gala lietotājiem. Pirmā lietotāju grupa ir pašas rīku būves platformas lietotāji, kas līdz ar to atbildīgi par konkrētu problēmorientētu rīku izstrādi. Tā kā šajā darbā aplūkota pati platforma un tās iespējas, tad pamatā notikusi koncentrēšanās tieši uz šīs grupas gala lietotājiem – tiem atvieglots transformāciju rakstīšanas process gan ieviešot augstāka līmeņa modeļu transformāciju valodas, gan arī ļaujot definēt skata jēdzienu vieglākai metamodeļu uztverei. Tāpat rīku būvētājiem pieejams arī rīku definīciju eksports no vienas platformas versijas uz citu.

Taču ir vērts pievērst uzmanību arī otrai lietotāju grupai – ar platformu izstrādāto problēmorientēto rīku gala lietotājiem, kas atbildīgi par modeļu veidošanu ar konkrētiem rīkiem. Viņiem atvieglota iespēja apmainīties savā starpā ar izveidotajiem modeļiem vai nu konkrētu diagrammu vai cita veida struktūru izskatā. Tāpat daudzlietotāju režīms, kas balstīts uz eksporta/importa mehānismu atsevišķu diagrammu līmenī, pamatā atvieglo darbu tieši šī tipa gala lietotājiem.

Īpaši vērtīgi ir secinājumi, kas radušies, salīdzinot atsevišķas rīku būves platformas komponentes ar līdzīgām, kuras izstrādātas citu pasaulē esošu pētījumu rezultātā. Tas tikpat lielā mērā attiecas uz visu platformu GRAF kopumā kā uz tām platformas iespējām, kas sīkāk aplūkotas šajā darbā. Konkrēti runājot, dziļāk izpētīti pasaulē eksistējošu metamodeļa skatu

mehānisma analogi. No šiem pētījumiem radušies secinājumi, kādā virzienā būtu derīgi minēto mehānismu attīstīt tālāk. Tie izklāstīti piektās nodaļas nobeigumā.

Atgriežoties pie 3.3. nodaļā minētā secinājuma par platformas stāvokli pirms šī darba tapšanas, kad, izmantojot rīku būves platformu GRAF, ar vienkāršiem līdzekļiem bija iespējams no nulles ārti un ērti izveidot jaunu vienkāršu grafisku rīku, tad jāsecina, ka darba rezultātā sasniegti vēlamie šī stāvokļa uzlabojumi, kuri iztirzāti 3.3. nodaļā, un jaunais stāvoklis varētu tikt formulēts šādā veidā:

Izmantojot rīku būves platformu GRAF, ar vienkāršiem līdzekļiem iespējams no nulles ātri un ērti izveidot un uzturēt jaunus patvaļīgas sarežģītības grafiskus problēmorientētus rīkus

Tātad nu ar platformas palīdzību iespējams veidot ne vien vienkāršus, bet jau krietni vien sarežģītākus rīkus (varbūt *patvaļīgas sarežģītības* skan pārspīlēti, bet faktiski tā ir taisnība – izstrādē izmantojot modeļu transformācijas, rīka sarežģītībai nav formulējama augšējā robeža), kā arī papildus ērtai rīku veidošanai nākusi klāt arī ērtāka to uzturēšana līdz zināmai robežai. Protams, arī te vēl ir vieta uzlabojumiem – to meklēšana un risināšana ir loģisks šī darba turpinājums! Pētījumi šajā sakarā tiek intensīvi turpināti, un kā viens no interesantākajiem virzieniem tālākai platformas lietojamības uzlabošanai, pie kura aktīvi strādā arī promocijas darba autors, minama jauna veida platformas paplašināšanas iespēja, izmantojot papildinātu UML stereotipu veida mehānismu [45, 46].

Autora publikāciju saraksts

1. E. Rencis, J. Barzdins, S. Kozlovics. *Towards Open Graphical Tool-Building Framework.* Scientific Journal of Riga Technical University, "Computer Science", special issue for the 10th International Conference on Perspectives in Business Informatics Research, Riga, Latvia, pp. 80 – 87, 2011.

2. E. Rencis, J. Barzdins. *On the Use of UML Stereotypes to Create Higher-Order Domain Specific Languages and Tools*. Proceedings of MDA & MDSD'2011 Workshop of ENASE 2011, Beijing, China, pp. 14 – 25, 2011.

3. E. Rencis. *On Views on Metamodels*. Databases and Information Systems VI, Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, pp. 94-107.

4. S. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans. *A Kernel-level UNDO/REDO Mechanism for the Transformation-Driven Architecture*. Databases and Information Systems VI, Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, pp. 80-93.

5. A. Sproģis, R. Liepiņš, J. Bārzdiņš, K. Čerāns, S. Kozlovičs, L. Lāce, E. Rencis, A. Zariņš. *GRAF: a Graphical Tool Building Framework.* Proceedings of the Tools and Consultancy Track. European Conference on Model-Driven Architecture Foundations and Applications, Paris, France, pp. 18 – 21, 2010.

6. E. Rencis. *Views on Metamodels: a Different Perception.* Proceedings of Baltic conference on Databases and Information Systems 2010, Riga, Latvia, pp. 343 – 358, 2010.

7. S. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans. *Universal UNDO Mechanism for the Transformation-Driven Architecture.* Proceedings of Baltic conference on Databases and Information Systems 2010, Riga, Latvia, pp. 325 – 340, 2010.

8. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. *A Graph Diagram Engine for the Transformation-Driven Architecture*. Scientific Papers, University of Latvia, "Computer Science and Information Technologies", vol. 756, pp. 139 – 149, 2010.

9. J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. *MDE-based Graphical Tool Building Framework.* Scientific Papers, University of Latvia, "Computer Science and Information Technologies", vol. 756, pp. 121 – 138, 2010.

10. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. *Domain Specific Languages for Business Process Management: a Case Study*. Proceedings of DSM'09 Workshop of OOPSLA 2009, Orlando, Florida, USA, pp. 34 – 40, 2009.

11. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. *A Graph Diagram Engine for the Transformation-Driven Architecture*. Proceedings of MDDAUI'09 Workshop of International Conference on Intelligent User Interfaces 2009, Sanibel Island, Florida, USA, pp. 29 – 32, 2009.

12. J. Barzdins, S. Kozlovics, E. Rencis. *The Transformation-Driven Architecture*. Proceedings of DSM'08 Workshop of OOPSLA 2008, Nashville, Tennessee, USA, pp. 60 – 63, 2008.

13. D. Varro, M. Asztalos, D. Bisztray, A. Boronat, D. H. Dang, R. Geiss, J. Greenyer, P. Van Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, and E. Weinell. *Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools*. Proceedings of 3rd International Workshop Applications of Graph Transformation with Industrial Relevance (AGTIVE '07). Springer, Berlin, LNCS, vol. 5058, pp. 540 – 565, 2008.

14. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, *Model Transformation Languages and their Implementation by Bootstrapping Method*. Pillars of Computer Science, LNCS, vol. 4800, Springer-Verlag, pp. 130 – 145, 2008.

15. E. Rencis, *Model Transformation Languages L1, L2, L3 and their Implementation*. Scientific Papers, University of Latvia, "Computer Science and Information Technologies", vol. 733, pp. 103 – 139, 2008.

16. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis, *GrTP: Transformation Based Graphical Tool Building Platform*. Proc. of Workshop on Model Driven Development of Advanced User Interfaces, MODELS 2007, Nashville, USA.

Izmantotā literatūra

1. MetaEdit+ Workbench User's Guide, Version 4.5, http://www.metacase.com/support/45/manuals/mwb/Mw.html, 2008.
2. S. Cook, G. Jones, S. Kent, A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007.
3. Graphical Modeling Framework (GMF, Eclipse Modeling subproject), http://www.eclipse.org/gmf
4. A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks, J. Barzdins. Building Tools by Model Transformations in Eclipse. Proceedings of DSM'07 workshop of OOPSLA 2007, Montreal, Canada, Jyvaskyla University Printing House, 2007, pp. 194-207.
5. K. Podnieks. Towards a General Definition of Modeling. Science and Religion Dialogue Prints of University of Latvia, http://scireprints.lu.lv/155/1/Podnieks_General_Definition_of_Modeling.pdf
6. J. Bézivin, N. Ploquin, „Tooling the MDA framework: a new software maintenance and evolution scheme proposal", http://www.adtmag.com/joop/article.aspx?id=5736
7. Meta Object Facility (MOF) Core Specification v2.0, OMG, document formal/06-01-01, 2006.
8. Eclipse Modeling Framework (EMF, Eclipse Modeling subproject), http://www.eclipse.org/emf
9. MDA Guide Version 1.0.1. OMG, document omg/03-06-01, 2003.
10. OMG modeling specifications, UML 2.1.1 Superstructure and Infrastructure, http://www.omg.org/docs/formal/07-02-05.pdf
11. OMG modeling specifications, UML 2.1.1 Superstructure and Infrastructure, http://www.omg.org/docs/formal/07-02-05.pdf
12. Object Management Group, Request for Proposal: MOF 2.0 Query / Views / Transformations RFP, 2002. ad/2001-08-04.
13. MOF QVT Final Adopted Specification, OMG, document ptc/05-11-01, 2005.
14. Tefkat, The EMF Transformation Engine, http://tefkat.sourceforge.net
15. Viatra2 Component, http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html
16. A. Agrawal, „Metamodel Based Model Transformation Language", http://www.acm.org/src/subpages/papers/metamodel.pdf
17. ATLAS group, „ATL Transformation Description Template – version 0.1", http://www.eclipse.org/m2m/atl/doc/ATL_Starter_Guide.pdf

18. AGG, The Homebase, http://tfs.cs.tu-berlin.de/agg/index.html
19. Fujaba Tool Suite, http://www.fujaba.de
20. E.D. Willink, „UMLX: A graphical transformation language for MDA", http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/doc/umlx/Oopsla2003.pdf
21. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. Proceedings of MDAFA 2004, Lecture Notes in Computer Science, Vol. 3599, Springer-Verlag, 2005, pp. 62-76.
22. The Base Transformation Language L0, http://lx.mii.lu.lv/L0_plus_CurrVers_2_4.pdf
23. Risto Pohjonen,Steven Kelly, Juha-Pekka Tolvanen, Moving From Coding To Model-Driven Development: Hands-On with MetaEdit+, Conference on Code Generation 2011, Cambridge, UK
24. GRADE tools, http://www.gradetools.com
25. P. Kikusts, P. Rucevskis, Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. Proceedings of Graph Drawing '95, LNCS, vol. 1027, pp. 361–364, Springer-Verlag, 1996.
26. K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. Proceedings of The Latvian Academy of Sciences, Section B, vol. 55, No. 1, pp. 43–51, 2001.
27. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis, GrTP: Transformation Based Graphical Tool Building Platform. Proc. of Workshop on Model Driven Development of Advanced User Interfaces, MODELS 2007, Nashville, USA.
28. A. Rensink, G. Taentzer, AGTIVE 2007 Graph Transformation Tool Contest. Applications of Graph Transformation with Industrial Relevance (AGTIVE), Kassel, Germany, Springer, LNCS, vol. 5088, pp. 487-492, 2007.
29. D. Varro, M. Asztalos, D. Bisztray, A. Boronat, D. H. Dang, R. Geiss, J. Greenyer, P. Van Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, and E. Weinell. Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. Proceedings of 3rd International Workshop Applications of Graph Transformation with Industrial Relevance (AGTIVE '07). Springer, Berlin, LNCS, vol. 5058, pp. 540–565, 2008.
30. J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. MDE-based Graphical Tool Building Framework. Scientific Papers, University of Latvia, "Computer Science and Information Technologies", vol. 756, pp. 121–138, 2010.
31. A. Sproģis, R. Liepiņš, J. Bārzdiņš, K. Čerāns, S. Kozlovičs, L. Lāce, E. Rencis, A. Zariņš. GRAF: a Graphical Tool Building Framework. Proceedings of the Tools and

Consultancy Track. European Conference on Model-Driven Architecture Foundations and Applications, Paris, France, pp. 18–21, 2010.

32. J. Barzdins, S. Kozlovics, E. Rencis. The Transformation-Driven Architecture. Proceedings of DSM'08 Workshop of OOPSLA 2008, Nashville, Tennessee, USA, pp. 60–63, 2008.
33. A. Sprogis. The Configurator in DSL Tool Building. Scientific Papers, University of Latvia, "Computer Science and Information Technologies", Vol. 756, 2010, pp. 121-138.
34. S. Kahle, JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek fur TGraphen, Diplomarbeit, University of Koblenz-Landau, Institute for Software Technology, 2006.
35. Sesame, http://www.openrdf.org
36. J. Barzdins, G. Barzdins, R. Balodis, K. Cerans, A. Kalnins, M. Opmanis, K. Podnieks, „Towards Semantic Latvia". Communications of the 7th International Baltic Conference on Databases and Information Systems (Baltic DB&IS'2006), Vilnius, 2006, pp. 203-218.
37. M. Opmanis, K. Cerans, Multilevel Data Repository for Ontological and Meta-modeling. Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, pp. 125-138,2010.
38. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans. Universal UNDO Mechanism for the Transformation-Driven Architecture. Proceedings of Baltic conference on Databases and Information Systems 2010, Riga, Latvia, pp. 325–340, 2010.
39. S. Kozlovics, E. Rencis, S. Rikacovs, K. Cerans. A Kernel-level UNDO/REDO Mechanism for the Transformation-Driven Architecture. Databases and Information Systems VI, Selected Papers from the Ninth International Baltic Conference, DB&IS, pp. 80-93, 2010.
40. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. Proceedings of MDDAUI'09 Workshop of International Conference on Intelligent User Interfaces 2009, Sanibel Island, Florida, USA, pp. 29–32, 2009.
41. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. Scientific Papers, University of Latvia, "Computer Science and Information Technologies", vol. 756, pp. 139–149, 2010.
42. S. Kozlovics, A Dialog Engine Metamodel for the Transformation-Driven Architecture. Scientific Papers, University of Latvia, vol. 756, pp.151–170, 2010.
43. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. Domain Specific Languages for Business Process

Management: a Case Study. Proceedings of DSM'09 Workshop of OOPSLA 2009, Orlando, Florida, USA, pp. 34–40, 2009.

44. A. Sostaks, A. Kalnins. The Implementation of MOLA to L3 Compiler. Articles of the University of Latvia, "Computer Science and Information Technologies", Riga, Latvia, 2008, pp. 140-178.
45. E. Rencis, J. Barzdins. On the Use of UML Stereotypes to Create Higher-Order Domain Specific Languages and Tools. Proceedings of MDA & MDSD'2011 Workshop of ENASE 2011, Beijing, China, pp. 14–25, 2011.
46. E. Rencis, J. Barzdins, S. Kozlovics. Towards Open Graphical Tool-Building Framework. Scientific Journal of Riga Technical University, "Computer Science", special issue for the 10th International Conference on Perspectives in Business Informatics Research, Riga, Latvia, pp. 80–87, 2011.
47. B. Efron, R.J. Tibshirani, „An Introduction to the Bootstrap", Chapman & Hall/CRC, 1994, 436 p.
48. Eclipse, http://www.eclipse.org
49. S. Rikacovs, The Base Transformation Language L0+ and Its Implementation. Chapters, University of Latvia, Computer Science and Information Technologies, pp. 75-102, 2008.
50. L.M. Garshol, „BNF and EBNF: What are they and how do they work?", http://www.garshol.priv.no/download/text/bnf.html
51. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs, Model Transformation Languages and their Implementation by Bootstrapping Method. Pillars of Computer Science, LNCS, vol. 4800, Springer-Verlag, pp. 130–145, 2008.
52. J. Gallier, „Logic for Computer Science: Foundations of Automatic Theorem Proving", Wiley, 1986.
53. The Lx transformation language set home page, http://Lx.mii.lu.lv
54. A.V. Aho, R. Sethi, J.D. Ullman, „Compilers - Principles, Techniques, and Tools", Addison-Wesley Publishing Company, 1986, 796 p.
55. A. Kalnins, E. Kalnina, E. Celms and A. Sostaks. From requirements to code in a model driven way. J. Grundspenkis, M. Kirikova, Y. Manolopoulus, L. Novickis: Proceedings of Associated Workshops and Doctoral Consortium of the 13th East European Conference, ADBIS 2009, Riga, Lativia, September 7-10, 2009. Revised Selected, Vol 5968, LNCS, Springer, Berlin/Heidelberg, 2010, 161-168.
56. E. Rencis. Views on Metamodels: a Different Perception. Proceedings of Baltic conference on Databases and Information Systems 2010, Riga, Latvia, pp. 343–358, 2010.

57. E. Rencis. On Views on Metamodels. Databases and Information Systems VI, Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, pp. 94-107, 2010.
58. Web Ontology Language (OWL), http://www.w3.org/2004/OWL
59. UML 2.0 OCL Specification, OMG, document ptc/03-10-14, 2003
60. Kermeta, http://www.kermeta.org
61. J. Steel, J. M. Jezequel, Mode Typing for Improving Reuse in Model-Driven Engeneering. Model Driven Engeneering languages and Systems, Lecture Notes in Computer Science, Vol. 3713, Springer-Verlag, pp. 84–96, 2005.
62. J. Steel, J. M. Jezequel, On Model Typing. Journal of Software and Systems Modeling, Vol. 6, No. 4, Springer Berlin, pp. 401–413, 2007.
63. J. R. Romero, J. E. Rivera, F. Duran, A. Vallecillo, Formal and Tool Support for Model Driven Engeneering with Maude. Journal of Object Technology, Vol. 6, No. 9, pp. 187–207, 2007.
64. J. Bezivin, F. Jouault, KM3: a DSL for Metamodel Specification. Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006), Bologna, Italy, 2006, Lecture Notes in Computer Science, Vol. 4037, Springer-Verlag, pp. 171–185, 2006.
65. R. F. Paige, P. J. Brooke, J. S. Ostroff, Metamodel-based model conformance and multiview consistency checking. ACM Trans. Softw. Engin. Method. 16, 3, Article 11 (July 2007), 49 pages
66. J. Jakob, A. Schürr, View Creation of Meta Models by Using Modified Triple Graph Grammars. Electr. Notes Theor. Comput. Sci. 211, pp. 181–190, 2008.