LATVIJAS UNIVERSITĀTE
Datorikas fakultāte

ARTŪRS SPROĢIS

DOMĒNSPECIFISKU RĪKU KONFIGURĀCIJAS VALODA UN TĀS REALIZĀCIJA

Promocijas darbs

Nozare: datorzinātnes
Apakšnozare: programmēšanas valodas un sistēmas

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

Rīga 2013

IEGULDĪJUMS TAVĀ NĀKOTNĒ

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

Saturs

Ievads 6

1 Darba konteksts 9

1.1 Platformas metamodeļi 10

1.1.1 Prezentācijas metamodelis 10

1.1.2 Dialogu logu metamodelis 17

1.1.3 Kokveida diagrammu metamodelis 21

1.2 Lua un lQuery 24

1.2.1 Lua apraksts 24

1.2.2 lQuery bibliotēka 26

1.3 Platformas ietvars un tā realizācija 29

1.4 Secinājumi 32

2 DSML valodu un to rīku specificēšana 33

2.1 Specializācijas metode 34

2.1.1 UML stereotipa mehānisma attīstīšana 34

2.1.2 DSML valodas profils 37

2.1.3 DSML rīku profils 40

2.2 Konkretizācijas metode 44

2.2.1 DSML metamodelis 44

2.2.2 DSML rīku definēšanas metamodelis 47

2.2.3 Blokshēmu redaktora specificēšana ar tipu instancēm 56

2.3 Metožu salīdzinājums 59

2.4 Secinājumi 60

3 Realizācija 61

3.1 Realizācijas metamodelis 61

3.2 Interpretators 63

3.3 Paplašinājuma punktu mehānisms 66

3.3.1 Vienkāršie paplašinājuma punkti 67

3.3.2 Saliktie paplašinājuma punkti 70

3.4 Secinājumi 85

4 Konfigurators 87

4.1 Konfiguratora valoda 87

4.1.1 Box 88

4.1.2 Line 92

4.1.3 Port 94

4.1.4 Specialization 94

4.1.5 Atribūtu specificēšana 94

4.1.6 Stilu pievienošana 97

4.1.7 Dialogu logu konfigurēšana 98

4.1.8 Diagrammas funkcionalitāte 99

4.1.9 Blokshēmu redaktora konfigurēšanas apraksts 101

4.2 Konfiguratora realizācija 102

4.2.1 Konfiguratora specifikācija 102

4.2.2 Interpretatora funkcionalitātes papildinājumi 109

4.3 Rīka versiju un modeļu migrēšana 113

4.3.1 Problēmas nostādne 113

4.3.2 Migrēšanas mehānisms 115

4.3.3 Modeļu migrēšanas piemērs 118

4.3.4 Modeļu migrēšana izpildīšana 122

4.4 Secinājumi 124

5 Lietojumu piemēri un pieredze 125

5.1 Īss platformas attīstības apraksts 125

5.2 BiLingva 126

5.3 ProMod 126

5.4 OWLGrEd 127

5.5 LUMod 128

5.6 Secinājumi 129
6 Pārskats alternatīvām rīku būves pieejām 130
6.1 Eclipse GMF 130
6.2 Microsoft DSL 131
6.3 MetaEdit 132
6.4 ObeoDesigner 133
6.5 EuGENia Live 134
6.6 Graphiti 134
6.7 Generic Modeling Environment (GME) 135
6.8 IBM Rational Software Architect (RSA) 136
6.9 Microsoft Visio 136
6.10 Secinājumi 137
Nobeigums 138
Autora publikāciju saraksts 140
Izmantotā literatūra 142
1. Pielikums 147

Ievads

Domēna specifiskas modelēšanas valodas (*piezīme* – šajā darbā, lai uzsvērtu, ka runa iet par domēna specifiskām valodām, to apzīmēšanai lietosim nevis ierasto saīsinājumu „DSML", bet gan „DSML valodas") [1] un DSML rīki arvien plašāk tiek izmantoti sistēmu modelēšanas kontekstā [2]. Protams, mēs varam izmantot universālas valodas un rīkus, tādus kā – UML [3], BPMN [4], SysML [5], tomēr to lietošanai ir vairākas problēmas. Pirmkārt, šos rīkus ir izstrādājuši IT cilvēki un pieredze rāda, ka tie ir ērti un labi saprotami IT cilvēkiem, bet ne biznesa cilvēkiem. Otrkārt, šie rīki balstās uz universālām valodām, kuru mērķis ir aprakstīt pēc iespējas plašākus priekšmetu apgabalus, bet reālos lietojumos reti tiek izmantotas visas universālo valodu konstrukcijas un tās lieki sarežģī rīka lietojamību. Treškārt, universālajiem rīkiem ir fiksēta funkcionalitāte un reti tiek piedāvāts mehānisms, kā to papildināt ar rīkam specifiskām iespējām, piemēram, modeļu konsistences pārbaudēm, automātisku teksta ģenerēšanu no rīku modeļiem, pieslēgšanos ārējām datu bāzēm, utt. Ceturtkārt, tipiski šo rīku iekšējā repozitorija struktūra nav atvērta ārējām lietojumprogrammām, un tādējādi no ārējām lietojumprogrammām nevar ērti piekļūt modeļu datiem.

DSML rīku lietošana šīs problēmas lielā mērā atrisina, jo katram lietojumu apgabalam tiek izstrādāts jauns rīks ar tam specifisku funkcionalitāti un valodu, kura izmanto tieši šim lietojumu apgabalam specifiskos jēdzienus [6]. Tomēr DSML valodu praktiskai lietošanai ir viena problēma un, proti, katra DSML rīka izstrāde ir sarežģīts un laikietilpīgs process [7].

Veids, kā DSML rīku izstrādes procesu var padarīt vienkāršāku un ātrāku, ir izstrādāt universālu rīku būves platformu. Platformu izstrāde ir sarežģīta un dārga, tādēļ šādu platformu nav daudz, un zināmākās ir Eclipse EMF [8], Microsoft DSL[9], MetaEdit [10] (par esošajiem risinājumiem detalizētāk 6. nodaļā). Tipiski DSML rīku būves platformās tiek izmantotas divas pieejas. Izplatītākā ir pieeja, ko izmanto Eclipse EMF un Microsoft DSL, un tā paredz pirmajā DSML rīka specificēšanas solī definēt DSML valodas abstraktās sintakses modeli, otrajā solī konkrētās sintakses modeli un trešajā solī nodibināt atbilstības starp abiem modeļiem. Šāda platformas arhitektūra ir izvēlēta ar mērķi, lai platformā būtu tieši viens abstraktās sintakses modelis, kuram ir iespējams piesaistīt patvaļīgu skaitu konkrētās sintakses modeļu (attiecīgi katram modelim izpildīt otro un trešo soli). Tipiski šīs platformas piedāvā noteiktu funkcionalitāti, bet, ja nepieciešams kaut kas papildus, tad to ir iespējams noprogrammēt.

No šādas arhitektūras ir divi ieguvumi. Pirmkārt, sarežģītas datu apstrādes ir ērtāk veikt abstraktās sintakses modelī, nevis kādā no konkrētās sintakses modeļiem. Otrkārt, tiek nodrošināta automātiska datu sinhronizācija starp dažādiem konkrētās sintakses modeļiem, proti, datu izmaiņas kādā no konkrētās sintakses modeļiem vispirms tiek ienestas abstraktās

sintakses modelī un pēc tam automātiski sinhronizētas ar pārējiem konkrētās sintakses modeļiem. Acīmredzami šādai pieejai priekšrocības ir gadījumā, ja konkrētās sintakses modeļu ir daudz, vai arī ir nepieciešams veikt sarežģītas datu apstrādes, bet gadījumos, kad konkrētās sintakses modeļu skaits ir mazs, piemēram, viens vai divi un datu apstrāde ir relatīvi vienkārša, kā tas lielākoties ir DSML rīkos, tad pietiek definēt šo vienu vai divus konkrētās sintakses modeļus, bet abstraktās sintakses modelis, līdz ar attiecīgu atbilstību nodibināšanu starp abstraktās un konkrētās sintakses modeļiem, ir nevajadzīgs un nepamatoti sarežģī DSML rīku izstrādi.

Otro pieeju izmanto MetaEdit, un tā paredz DSML rīkus definēt tikai ar konkrētās sintakses modeli un pēc tam šo modeli interpretēt ar interpretatoru. Šī pieeja daudzos gadījumos ir ļoti ērta un ļauj ātri iegūt DSML rīkus, tomēr DSML valodu pasaule nav ierobežota, un aptvert visus gadījumus ar vienu modeli un fiksētu interpretatoru nav iespējams.

Salīdzinot abas pieejas, varam secināt, ka abām metodēm ir savi trūkumi, proti, ar pirmo pieeju varam definēt plašu DSML rīku klasi, bet tas ir sarežģīti, savukārt ar otro pieeju DSML rīkus varam definēt relatīvi vienkārši, bet tie ir ierobežoti ar interpretatora funkcionalitāti. Tādējādi aktuāla problēma ir atrast tādu DSML rīku izstrādes pieeju, kas, pirmkārt, DSML rīkus ļauj izstrādāt ar vienkāršiem līdzekļiem, otrkārt, neaprobežojas ar fiksētu funkcionalitāti, bet ļauj papildus noprogrammēt nestandarta funkcionalitāti pietiekoši ērtā veidā un, treškārt, nodrošina rīka atvērtība pret ārējām lietojumprogrammām, lai tās varētu piekļūt modeļu datiem.

Tādēļ ap 2007. gadu, kad jau bija parādījusies modeļbāzētās izstrādes ideja [11], Latvijas Universitātes Matemātikas un informātikas institūtā (LUMII) tika uzsākts Grade2 projekts (kā turpinājums labi zināmajam Grade [12] projektam), kura ietvaros bija paredzēts izstrādāt jaunas paaudzes rīku būves platformu, izstrādi balstot uz modeļiem un modeļu transformācijām. Šo projektu bija paredzēts realizēt pamatā ar piecu doktorantu spēkiem, un, uzsākot projektu, pienākumi starp doktorantiem tika sadalīti četrās daļās – transformāciju valodu izstrādi uzņēmās S. Rikačovs un R. Liepiņš, platformas realizācijas arhitektūru izstrādāja S. Kozlovičs, skatu mehānismu realizēja E. Rencis, bet rīku definēšana nonāca darba autora pārziņā.

Autors savā promocijas darbā rīku definēšanai ir realizējis tehnoloģiju, ar kuru šīs platformas ietvaros ir iespējams veikt pilnu DSML rīku izstrādes ciklu. Realizētā tehnoloģija balstās uz ideju, ka DSML rīka specifikāciju var uzdot ar vienu universālu metamodeli UML klašu diagrammu formā, kas ļauj ar modeļu transformācijām uzbūvēt universālu interpretatoru tā, lai vienkāršākajos gadījumos, interpretējot metamodeli, strādājošu rīku varētu iegūt bez

papildus programmēšanas darba, bet sarežģītākos gadījumos papildus programmējot tikai rīka specifisko funkcionalitāti. Savukārt, lai vienkāršotu un paātrinātu rīku izstrādes procesu, šim nolūkam ir izstrādāts speciāls DSML rīks jeb konfigurators, kurā rīku specificēšana tiek veikta augstākā abstrakcijas līmenī nekā UML klašu diagrammās, bet, lai konfiguratora specifikācijas varētu interpretēt, t.i., apstrādāt ar universālo interpretatoru, tās tiek automātiski transformētas uz universālo metamodeli. Konfigurators ļauj ne tikai padarīt rīku izstrādi ērtāku, bet arī dod iespēju kontrolēt darbības, kas tiek veiktas, lai no vienas rīka versijas iegūtu nākamo, un tās ierakstīt, lai pēc tam varētu veikt veco modeļu migrāciju uz jaunāku rīka versiju.

Darba struktūra pa nodaļām ir sekojoša. Pirmajā nodaļā ir dots platformas pamatprincipu skaidrojums. Nodaļa sākas ar vispārīgu platformas komponenšu – dziņu, metamodeļu, modeļu transformāciju un to savstarpējās komunikācijas aprakstu. Pēc tam seko katra dziņa un tā metamodeļa skaidrojums, kam seko Lua un lQuery apraksts, un nodaļa beidzas ar platformas ietvara un tā realizācijas skaidrojumu.

Otrā nodaļa ir veltīta universāla rīku definēšanas metamodeļa būvei. Lai šādu metamodeli uzbūvētu, ir apskatītas un analizētas divas alternatīvas pieejas, kā arī, lai dotu pilnīgāku izpratni par rīku specificēšanu ar šīm pieejām, kā piemērs ir aplūkota blokshēmu redaktora specificēšana.

Trešā nodaļa ir veltīta interpretatora aprakstīšanai. Nodaļas lielāko daļu sastāda skaidrojums par to, kā realizēt interpretatora universālās transformācijas un kā tās papildināt, lai iekļautu paplašinājuma punktu mehānismu. Tām universālajām transformācijām, kas satur paplašinājuma punktus, to precīzai izskaidrošanai ir dota vizualizācija ar blokshēmām, kurās precīzi ir norādītas paplašinājuma punktu vietas universālajās transformācijās.

Ceturtajā nodaļā ir aprakstīts konfigurators. Vispirms ir izklāstīta konfiguratora valoda, un pēc tam kā piemērs ir aplūkots otrajā nodaļā specificētā blokshēmu redaktora konfigurēšanas process. Pēc tam seko konfiguratora realizācijas skaidrojumu, parādot, kā ar otrajā un trešajā nodaļā aprakstītajām tehnoloģijām var realizēt konfiguratoru. Nodaļa noslēdzas ar konfiguratorā realizēto migrēšanas mehānisma skaidrojumu, izskaidrojot modeļu datu pārnešanu starp dažādām rīka versijām.

Piektajā nodaļā ir aprakstīti platformas lietojumi un pieredze, kas iegūta izstrādājot reālus rīkus. Kopumā ar šo platformu ir izstrādāti četri rīki – BiLingva, ProMod, OWLGrEd un LuMod, un katra rīka izstrādes gaita ir iztirzāta atsevišķi.

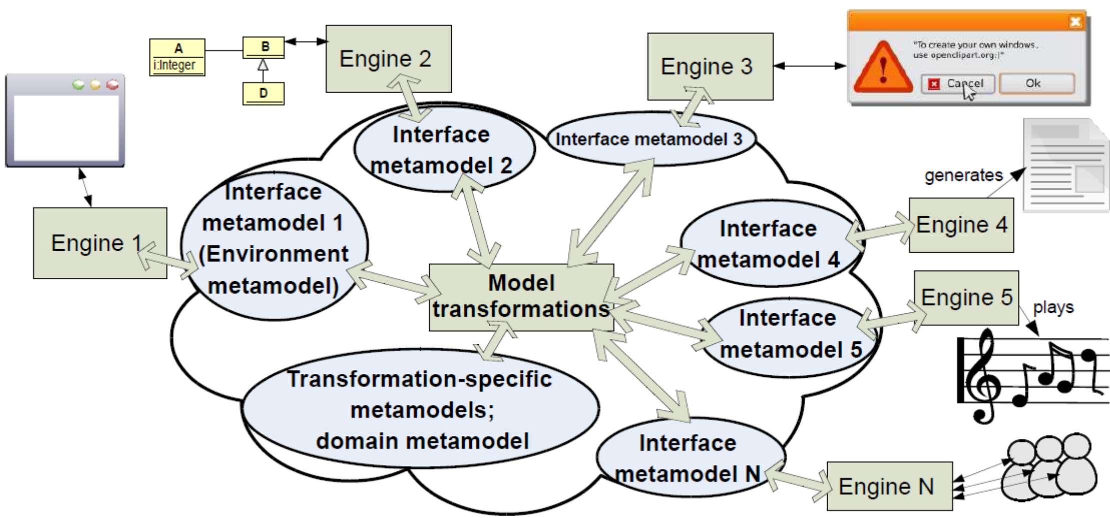
Sestajā nodaļā ir aprakstītas alternatīvas DSML rīku būves platformas un metodes, un dots to salīdzinājums ar darbā aprakstīto risinājumu.

1 Darba konteksts

Kā iepriekš tika minēts, šis promocijas darbs ir lielāka projekta sastāvdaļa un ir šī projekta noslēdzošais posms. Tādējādi šī darba kontekstu (t.i., bāzi, uz kuras balstās šis darbs) veido citu projekta dalībnieku izstrādes, un to izklāsts ir dots šajā nodaļā.

Darba pamata kontekstu veido transformāciju vadītā arhitektūra (TDA) [13] un platformas komponentes, kuras savā promocijas darbā [14] ir izstrādājis S. Kozlovičs, un rīku būves vajadzībām izstrādātā transformāciju valoda lQuery, kuru savā promocijas darbā ir realizējis R. Liepiņš. Jāpiebilst, ka TDA ideja neradās uzreiz, tā tapa platformas izstrādes laikā un ilgāku pētījumu rezultātā, kuros piedalījās arī darba autors [15, 16].

Detalizētāki apskatīsim TDA uzbūvi, kuras shematisks attēlojums ir redzams 1.1. attēlā. TDA galvenās komponentes ir dziņi un metamodeļi. Dziņu mērķis ir nodrošināt rīkā kādu konkrētu servisu, piemēram, prezentācijas dziņa [17] uzdevums ir vizualizēt diagrammas [18] un nodrošināt sadarbību starp rīka lietotāju un rīka diagrammām, bet dialogu dziņa [19] uzdevums ir parādīt dialogu logus un nodrošināt sadarbību starp lietotāju un dialogu logiem. Savukārt metamodeļi ir datu shēmas, kas atspoguļo rīku stāvokli dziņiem saprotamā formā.



1.1. attēls. Transformāciju vadīta arhitektūra (attēla autors S. Kozlovičs)

Tehniski platforma ir realizēta izmantojot modeļu repozitoriju [20], modeļu transformācijas un komandu-notikumu mehānismu. Komandu-notikumu mehānisms ļauj nodrošināt interakciju starp rīka lietotāju, dzini un modeļu transformācijām. Proti, strādājošā rīkā tā darbību nodrošina dzinis un, ja lietotājs rīkā veic kādu darbību, piemēram, peles dubultklikšķi uz kāda diagrammas elementa, tad šo lietotāja darbību „noķer" dzinis un to klasificē kā notikumu, kuram modeļa repozitorijā tiek izveidota atbilstošās klases instance. Katram dzinim ir fiksēts notikumu saraksts un katram notikumam ir piekārtota modeļu

transformācija, kas to apstrādā. Tādējādi, kad dzinis ir „noķēris" lietotāja darbību un ir radījis notikuma instanci repozitorijā, tad rīka vadība tiek nodota tai transformācijai, kura atbild par šī veida notikuma apstrādi.

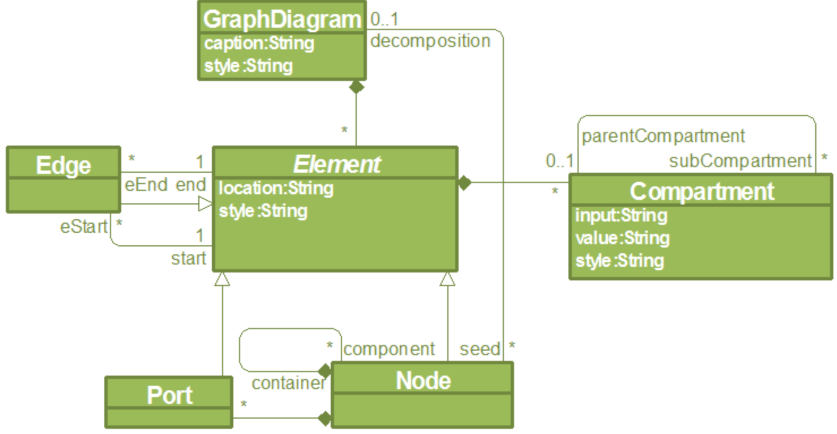
Transformācijas ir tās, kas veic notikumu loģisko apstrādi un šī iemesla dēļ tās var uzskatīt par platformas „smadzenēm". Transformācijām ir atļauts patvaļīgi mainīt repozitorija saturu, piemēram, radīt jaunus diagrammu elementus. Kad transformācija ir savu darbu izpildījusi, rīka vadība tiek atgriezta atpakaļ dzinim, no kura tā vadību saņēma. Gadījumā, ja transformācijai vajag dot pavēli dzinim, piemēram, pavēli par diagrammas pārzīmēšanu (piemēram, ja tā ir radījusi jaunu elementu), tad transformācija repozitorijā izveido jaunu attiecīgās komandas instanci (kad dzinis pavēli ir izpildījis, tas komandas instanci izdzēš). Līdzīgi kā ar notikumiem, dziņiem ir saraksts ar komandām, kuras tas prot izpildīt, un dzinis pavēles saņemšanas brīdī veic nepieciešamās darbības, lai to izpildītu.

Šī arhitektūra paredz, ka uzbūvēt jaunu rīku šajā platformā nozīmē izstrādāt šim rīkam transformāciju komplektu un neko vairāk, jo rīka darbības laikā mainās (atkarībā no rīku lietotāju darbībām) metamodeļu instances, bet ne dziņi. Tas nozīmē, ka, lai varētu uzbūvēt jaunu rīku, rīku izstrādātājiem ir detalizēti jāpārzina platformas metamodeļu uzbūve un dziņu komandas, bet ne pašu dziņu uzbūve.

1.1 Platformas metamodeļi

1.1.1 Prezentācijas metamodelis

Prezentācijas metamodelis priekš dziņa apraksta divas lietas. Pirmkārt, tas norāda, kā vizualizēt repozitorijā esošās diagrammas, un, otrkārt, tas apraksta rīka funkcionalitāti – paletes, rīkjoslas, utt. Prezentācijas metamodeļa kodols jeb klases, ar kurām apraksta vizualizāciju, ir redzams 1.2. attēlā (šeit un turpmākajās vietās ir pieņemts nenorādīt tos lomu vārdus, kuru nosaukums ir vienāds ar klases vārdu sāktu ar mazo burtu).



1.2. attēls. Prezentācijas metamodeļa kodols

Prezentācijas metamodeļa kodolu veido grafa metamodelis, kurš ir pielāgots „stilizētu” grafu attēlošanai. Šis metamodelis paredz, ka repozitorijā var atrasties vairākas diagrammas (klases *GraphDiagram* instances), kur katra diagramma var saturēt patvaļīgu skaitu elementu (klase *Element* instances). Ar atribūtu *caption* tiek norādīts diagrammas nosaukums, bet ar *style* tās izskats.

Elementi satur atribūtus – *style* un *location*. Atribūts *style* norāda, kā konkrētais elements ir vizuāli jāattēlo diagrammā, bet *location* norāda tā atrašanās vietu diagrammā. Dažādu tipu elementiem var būt dažādas īpašības, piemēram, kastes var ielikt kastēs (kompozīcija *component-container*), līnijām obligāti ir jānorāda sākuma un beigu elements (asociācijas *eStart-start* un *eEnd-end*), bet portiem obligāti ir jābūt piestiprinātiem pie kādas kastes (kompozīcija *port-node*). Savukārt ar asociāciju *seed-decomposition* var norādīt, ka kaste detalizē kādu diagrammu.

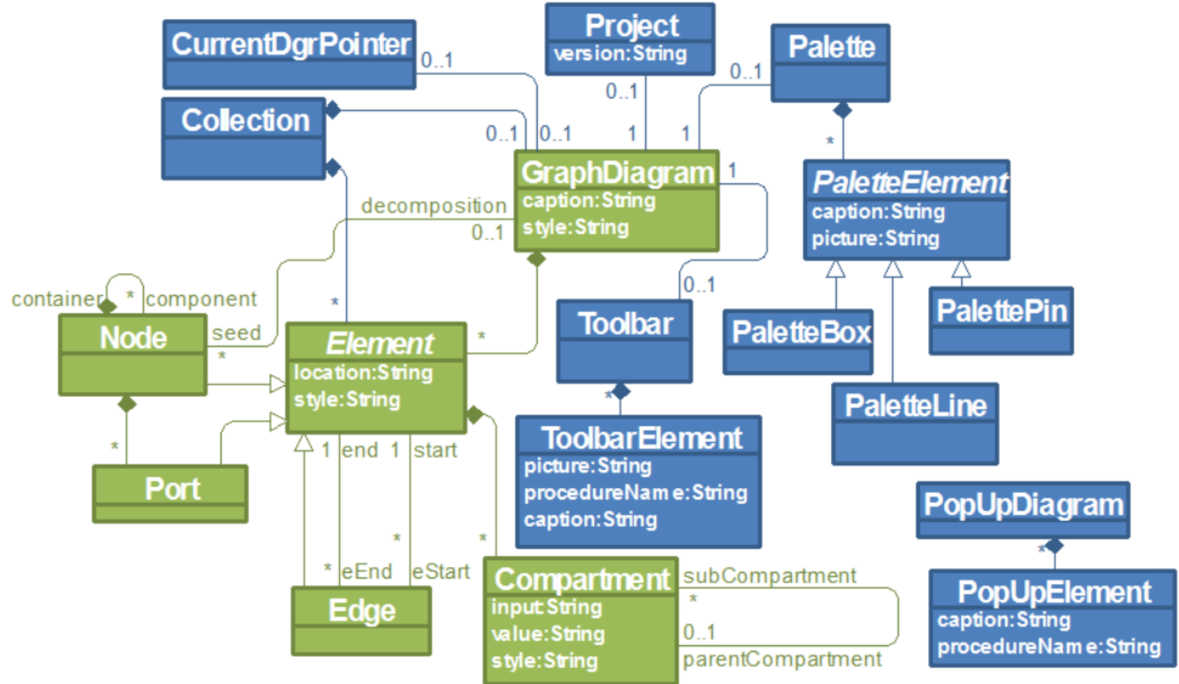
Diagrammas elementiem var būt piesaistīti atribūti (klase *Compartment* un kompozīcija *compartment-element*). Klase *Compartment* satur atribūtus – *input*, *value* un *style*. Elementa atribūta izskatu norāda *style* atribūts. Atribūti *input* un *value* ir savstarpēji saistīti - *value* satur *Compartment* loģisko vērtību, bet *input* satur tā reprezentatīvo vērtību. Bieži šo abu atribūtu vērtības sakrīt, bet ir situācijas, kad tās atšķiras, piemēram, ja vēlamies loģisko vērtību attēlot iekavās, tad atribūts *value* saturēs kādu vērtību - „x”, bet *input* saturēs šo pašu vērtību papildinātu ar prefiksu „(” un sufiksu „)”, kā rezultātā *input* vērtība būs „(x)”.

Klasei *Compartment* pašai uz sevi ir asociācija *subCompartment-parentCompartment*, un tā ļauj veidot atribūtu koku. Šāds koks tiek veidots, lai lietotājam, kā vienu veselu, varētu rādīt atribūtu vērtību, kas ir iegūta, saliekot vairākas apakšatribūtu vērtības. Piemēram, UML klašu diagrammās objekta pilnais nosaukums tiek attēlots formā - <objekta_nosaukums>:<klases_nosaukums>, respektīvi, diagrammās objekta pilno nosaukumu nepieciešams attēlot kā vienu vērtību, kas sastāv no divām neatkarīgām vērtībām - objekta un klases nosaukumiem. Lai šo situāciju varētu realizēt, TDA platformā tiek veidots viens virsatribūts ar diviem apakšatribūtiem, kur viens apakšatribūts satur objekta nosaukumu un otrs klases nosaukumu, bet to virsatribūts satur objekta pilno nosaukumu, kas ir iegūta apvienojot tā apakšatribūtu vērtības. Piemēram, ja objekta nosaukums ir „JānisBērziņš” un klases nosaukums ir „Persona”, tad objekta pilnais nosaukums ir „JānisBērziņš:Persona”.

Ar aprakstīto prezentācijas metamodeļa kodolu ir iespējams aprakstīt „stilizētus” grafus jeb diagrammu datus, bet ne rīka funkcionalitāti. Lai varētu aprakstīt rīka funkcionalitāti, prezentācijas metamodeļa kodols ir papildināts, kā tas ir redzams 1.3. attēlā.

Jaunus diagrammu elementus var veidot ar paletes pogām. Metamodelī paletei atbilst klase *Palette*, bet *PaletteElement* atbilst paletes pogām. Klases *PaletteElement* ir iedalītas

apakšklasēs – *PaletteBox*, *PaletteLine* un *PalettePin* atkarībā no tā, kāda tipa elementus ir paredzēts veidot ar konkrēto paletes pogu. Klasei *PaletteElement* ir divi atribūti – *caption* un *picture*. Atribūts *caption* norāda paletes pogas nosaukumu, bet *picture* pogas ikonas adresi.



1.3. attēls. Vienkāršots prezentācijas metamodelis

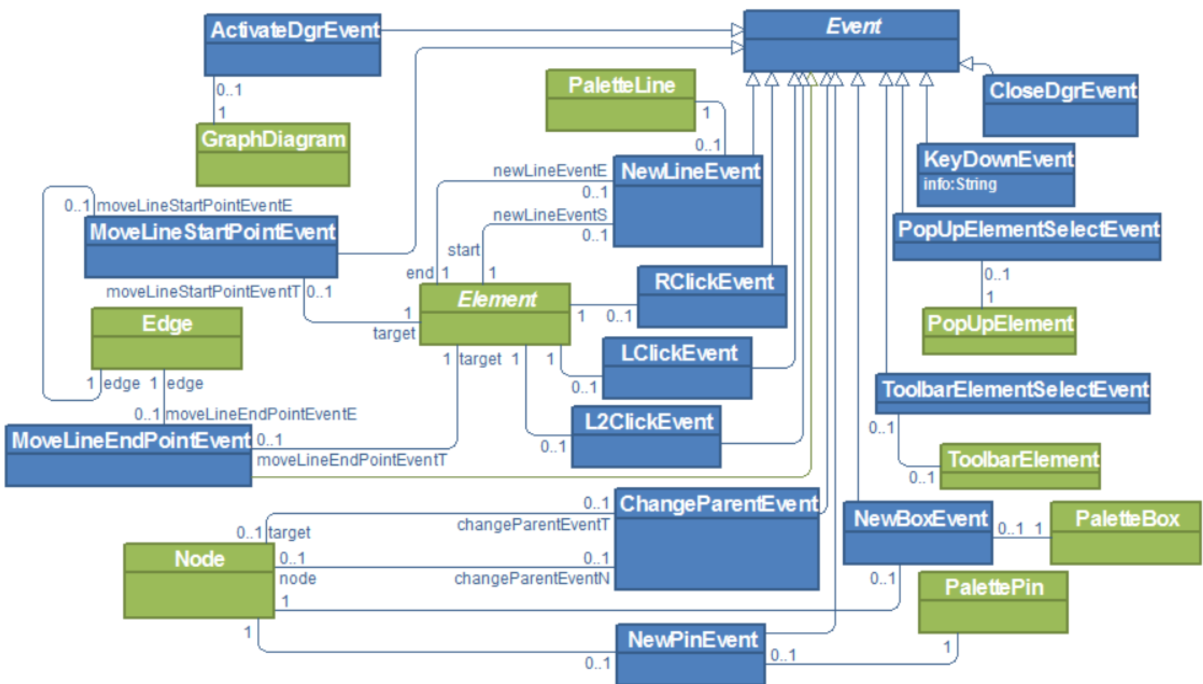
Klase *Toolbar* ļauj diagrammām pievienot rīkjoslu, bet *ToolbarElement* atbilst rīkjoslas pogām. Līdzīgi kā paletes pogām, tā arī klases *ToolbarElement* atribūts *caption* norāda pogas nosaukumu, *picture* norāda pogas ikonas adresi, bet *procedureName* satur transformācijas nosaukumu, kura tiek izpildīta lietotājam nospiežot rīkjoslas pogu.

Klase *PopUpDiagram* atbilst uznirstošajām izvēlnēm, kuras izsauc ar peles labās pogas klikšķi, bet klase *PopUpElement* atbilst uznirstošās izvēlnes operācijām. Līdzīgi kā *PaletteElement* un *ToolbarElement* gadījumā, klases *PopUpElement* atribūts *caption* norāda nosaukumu, bet *procedureName* transformācijas nosaukumu.

Lai norādītu diagrammu stāvokli, prezentācijas metamodelis satur klases *CurrentDgrPointer*, *Collection* un *Project*. Klases *CurrentDgrPointer* uzdevums ir norādīt rīka aktīvo diagrammu, un, lai šo izpildītu, repozitorijā vienmēr ir izveidota tieši viena *CurrentDgrPointer* instance, kura caur asociāciju *currentDgrPointer-graphDiagram* norāda uz aktīvo diagrammu. Klases *Collection* uzdevums ir norādīt diagrammu aktīvos elementus (diagrammās iezīmētās elementu kolekcijas). Tas nozīmē, ka katrai diagrammai ir piesaistīta tieši viena *Collectoin* instance, kura ar asociāciju *collection-element* norāda diagrammas

aktīvos elementus. Savukārt klase *Project* norāda uz diagrammu, kuru projekta atvēršanas brīdi nepieciešams rādīt kā pirmo, bet atribūts *version* norāda rīka versiju.

Kā iepriekš tika minēts, komunikācijai starp dziņiem un modeļu transformācijām tiek izmantots komandu-notikumu mehānisms. Lai nepadarītu metamodeli grūti lasāmu, komandu un notikumu klases nav iekļautas prezentācijas metamodeļa attēlā (t.i., 1.3. attēlā), bet gan izdalītas atsevišķi, saglabājot vien asociācijas uz saistītajām klasēm. Notikumu metamodelis ir redzams 1.4. attēlā (kardinalitātes šeit un turpmāk raksturo repozitorija „momentuzņēmumu”).



1.4. attēls. Notikumu metamodelis

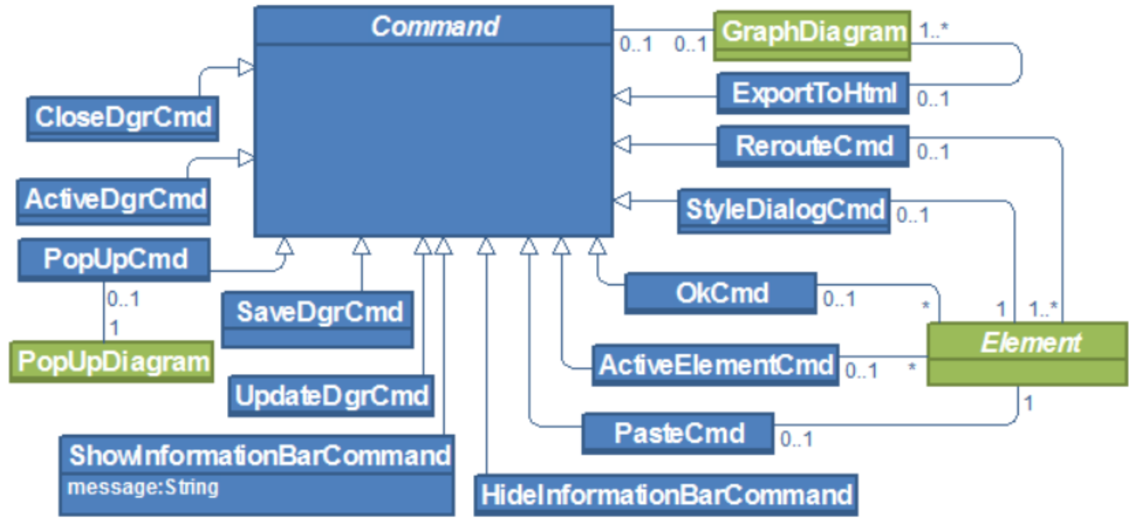
Notikumu klašu skaidrojums ir dots 1.1. tabulā.

***1.1. tabula.* Notikumu apraksts**

Nosaukums	Paskaidrojums
NewBox	Iestājas, kad lietotājs nospiež kastes tipa paletes pogu, un pēc tam ieklikšķina brīvā vietā diagrammā vai kādā elementā (lai veidotu kasti kastē). Notikums norāda uz nospiesto paletes elementu, un, ja ieklikšķināja kādā kastē, tad arī uz šo kasti.
NewLine	Iestājas, kad lietotājs nospiež līnijas tipa paletes pogu, un pēc tām savieno divus diagrammas elementus. Notikums norāda uz nospiesto paletes elementu, un uz līnijas

	sākuma un beigu elementiem.
NewPin	Iestājas, kad lietotājs nospiež porta tipa paletes pogu, un pēc tām uzklikšķina uz kādas kastes. Notikums norāda uz nospiesto paletes elementu un kasti, kurai uzklikšķināja.
MoveLineStartPoint	Iestājas, kad lietotājs līnijas sākuma galu pārceļ pie cita diagrammas elementa. Notikums norāda uz līniju, kuras gals tiek pārcelts, un uz jauno līnijas galapunktu.
MoveLineEndPoint	Iestājas, kad lietotājs līnijas beigu galu pievieno citam diagrammas elementam. Notikums norāda uz līniju, kuras gals tiek pārcelts, un uz jauno līnijas galapunktu.
ChangeParent	Iestājas, kad kasti grib ielikt kādā citā kastē vai arī to izņemt (novietot brīvā vietā diagrammā). Notikums norāda uz kasti, kuru pārvieto, un, ja tāds ir, elementu, kurā grib ievietot.
LClick	Iestājas, kad lietotājs noklikšķina peles kreiso pogu. Notikums norāda uz elementu, uz kura noklikšķināja.
L2Click	Iestājas, kad lietotājs rada dubultklikšķi ar peles kreiso pogu. Notikums norāda uz elementu, uz kura izpildīja dubultklikšķi.
RClick	Iestājas, kad lietotājs noklikšķina peles labo pogu. Notikums norāda uz elementu, uz kura noklikšķināja, vai tā atribūtu, ja noklikšķināja uz atribūta.
ToolbarElementSelect	Iestājas, kad lietotājs nospiež uz kādas no rīkjoslas pogām. Notikums norāda uz nospiesto rīkjoslas pogu.
PopUpElementSelect	Iestājas, kad lietotājs izvēlas kādu no uznirstošās izvēlnes darbībām. Notikums norāda uz izvēlēto izvēlnes darbību.
KeyDown	Iestājas, kad lietotājs nospiež kādu taustiņu kombināciju. Atribūts *info* norāda ievadīto taustiņu kombināciju.
ActivateDgr	Iestājas, kad diagramma tiek aktivizēta.
CloseDgr	Iestājas, kad diagramma tiek aizvērta.

Komandu metamodelis ir redzams 1.5. attēlā.



1.5. attēls. Komandu metamodelis

Komandu klašu skaidrojums ir dots 1.2. tabulā.

***1.2. tabula.* Komandu apraksts**

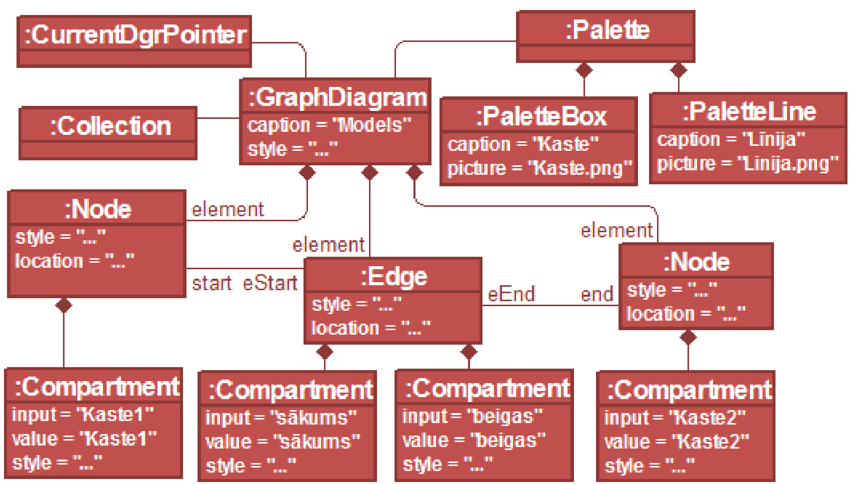
Nosaukums	Paskaidrojums
OK	Pavēle pārzīmēt diagrammu un diagrammas elementus. Komanda norāda uz elementiem, kurus vajag pārzīmēt, vai uz diagrammu, ja vajag pārzīmēt visu diagrammu.
ActiveDgr	Pavēle diagrammas aktivizēšanai. Komanda norāda uz diagrammu.
SaveDgr	Pavēle diagrammas saglabāšanai. Komanda norāda uz diagrammu.
CloseDgr	Pavēle diagrammas aizvēršanai. Komanda norāda uz diagrammu.
UpdateDgr	Pavēle diagrammas nosaukuma nomainīšanai, t.i., dzinis klases *GraphDiagram* instances atribūta *caption* vērtību uzstāda tās atbilstošajam diagrammas logam uz ekrāna. Komanda norāda uz diagrammu.
ActiveElement	Pavēle elementu aktivizēšanai (t.i., iekļaušanai aktīvo elementu kolekcijā, skat. 1.3. attēlu). Komanda norāda uz elementiem.
Paste	Pavēle ielīmēt elementus.

	Komanda norāda uz elementiem, kurus vajag ielīmēt.
PopUp	Pavēle uznirstošās izvēlnes izsaukumam. Komanda norāda uz uznirstošo izvēlni.
ShowInformationBar	Pavēle ziņojuma parādīšanai (ziņojums parādās diagrammas apakšā).
HideInformationBar	Pavēle ziņojuma likvidēšanai.
Reroute	Pavēle izrēķināt līniju optimālo ceļu. Komanda norāda uz līnijām.
StyleDialog	Pavēle, lai atvērtu elementa stila logu, individuālā stila uzstādīšanai. Komanda norāda uz elementu.
ExportToHtml	Pavēle diagrammu eksportēšanai uz HTML formātu. Komanda norāda uz diagrammām, kuras nepieciešams eksportēt.

Lai gūtu pilnīgāku priekšstatu par prezentācijas metamodeli un prezentācijas dzini, aplūkosim vienkāršu diagrammas piemēru ar divām kastēm un līniju starp tām. Piemēra diagramma ir redzama 1.6. attēlā.

1.6. att. Diagrammas piemērs

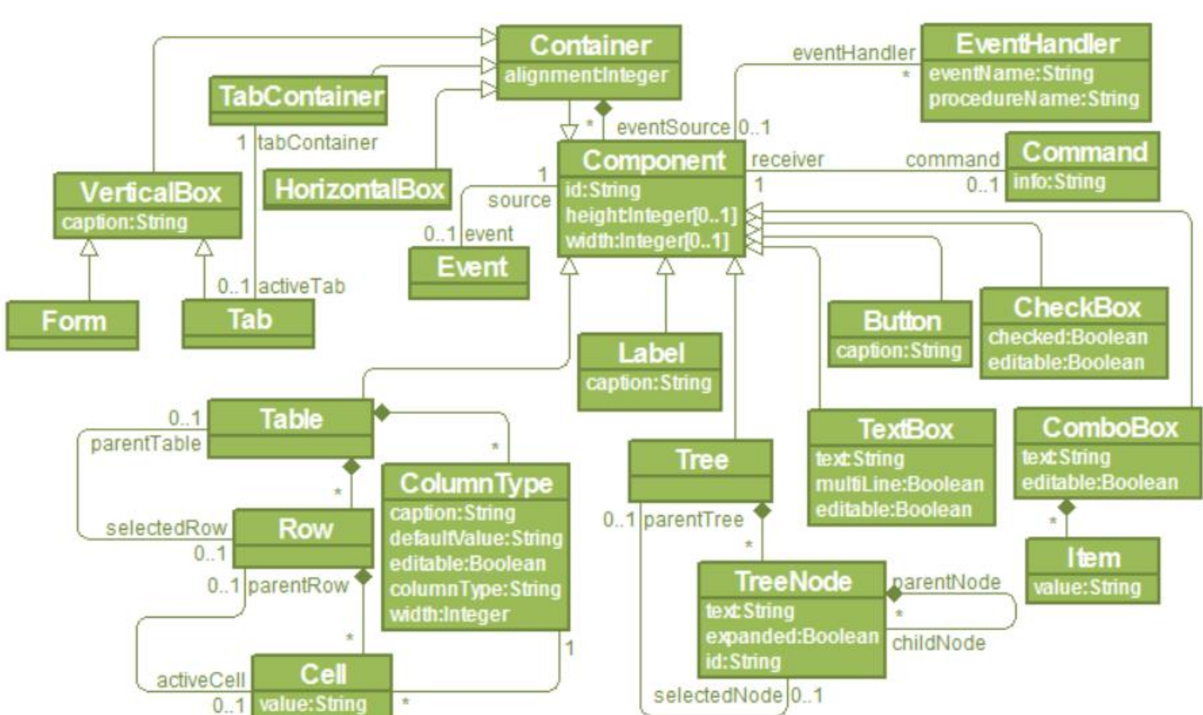
Diagrammas piemēram atbilstošā instanču diagramma ir redzama 1.7. attēlā (šeit un turpmākajās vietās ar apzīmējumu *atribūts* = „...” ir attēlota situācijas, ka dotajam atribūtam ir noteikta vērtība, bet mēs šajā attēlā to nekonkretizējam), kurā ir norādīts, ka katrai kastei ir viens atribūts, kas norāda kastes nosaukumu, bet līnijai katrā tās galā ir pa vienam atribūtam lomas norādīšanai. Bez grafiskajiem elementiem, diagrammai ir arī palete, kurā viena poga ļauj veidot jaunas kastes, bet otra jaunas līnijas. Lai norādītu, ka šī diagramma ir aktīva, diagrammai ir piesaistīta *CurrentDgrPointer* instance, bet iezīmēto elementu norādīšanai diagrammai ir piesaistīta *Collection* instance (tā kā diagrammā nav iezīmēts neviens elements, tad arī no *Collection* instances neiziet neviena saite uz diagrammas elementiem).



1.7. att. Piemēra instanču diagramma

1.1.2 Dialogu logu metamodelis

Dialogu logu metamodeļa instances ir dialogu logu specifikācija, no kuras dialogu dzinis uzģenerē dialogu formas [21]. Dialogu logu metamodelis ir redzams 1.8. attēlā.



1.8. att. Dialoga logu metamodeļa kodols

Dialogu logiem ir kokveida struktūra, kuru sakne ir forma (klase *Form*). Uz formas var atrasties patvaļīgs skaits komponenšu (klase *Component* un kompozīcija *component-*

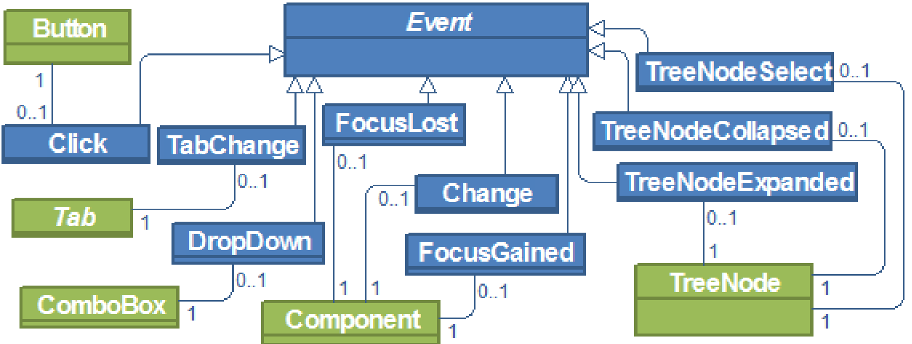
container). Komponentes ir iedalītas divās kategorijās – konteineri (klase *Container*) un kontroļi. Kontroļi ir tādas komponentes, kuras tiek izmantotas interakcijai ar lietotāju un tajās nevar ievietot citas komponentes. Dialogu logu dzinis atbalsta šādus kontroļus - teksta uzraksti (klase *Label*), teksta ievadlauki (klase *TextBox*), ķekša kastes (klase *CheckBox*), izkrītošās izvēlnes (klase *ComboBox*), pogas (klase *Button*), tabulas (klase *Table*) un koki (klase *Tree*).

Tabulas tiek realizētas ar klasi *Table*, bet klases *Row* un *Cell* norāda, ka tabula sastāv no rindām, bet katra rinda no rūtiņām. Asociācija *parentTable-selectedRow* norāda aktīvo rindu, bet aktīvo rindas rūtiņu norāda *parentRow-activeCell*. Klase *ColumnType* ļauj norādīt katrai tabulas rūtiņai tās tipu (caur asociāciju *cell-columnType*), jeb precīzāk sakot, katra rūtiņa atrodas kādā tabulas kolonnā. Atribūts *caption* norāda kolonnas nosaukumu, atribūts *defaultValue* norāda kolonnai piederošo rūtiņu noklusēto vērtību, atribūts *editable* norāda, vai šīs kolonnas rūtiņas ir rediģējamas, atribūts *columnType* norāda rūtiņas ievadlauka tipu (sakrīt ar kontroļu klašu nosaukumu), piemēram, lai kolonnas rūtiņās vērtības varētu ievadīt caur ķekša kasti, tad atribūta *columnType* vērtība ir jānorāda „CheckBox", bet atribūts *width* norāda kolonnas platumu.

Klase *Tree* dialogu logos realizē koka komponenti, kura sastāv no virsotnēm (klase *TreeNode*) patvaļīgā dziļumā (kompozīcija *parentNode-childNode*). Atribūts *text* norāda virsotnes nosaukumu, bet *expanded* norāda, vai virsotne ir izvēsta. Lai norādītu koka aktīvo virsotni tiek lietota asociācija *parentTree-selectedNode*.

Konteineri ir tādas komponentes, kurās var ievietot citas komponentes, un katrs konteiners izvieto komponentes vienu zem otras, vai vienu otrai blakus atkarībā no tā, vai tas ir vertikālais (klase *VerticalBox*), vai horizontālais (klase *HorizontalBox*) konteiners. Forma un cilne (klase *Tab*) ir redzami vertikālie konteineri, un to atribūts *caption* norāda to nosaukumu, bet tie konteineri, kas ir *VerticalBox*, *HorizontalBox* un *TabContainer* tiešās instances ir neredzami, un to vienīgā funkcija ir piedalīties komponenšu izvietošanā.

Sadarbībai starp dialoga loga dzini un transformācijām tiek izmantots komandu-notikumu mehānisms. Notikumiem atbilstošās klases un to saistītās komponentes ir redzamas 1.9. attēlā.



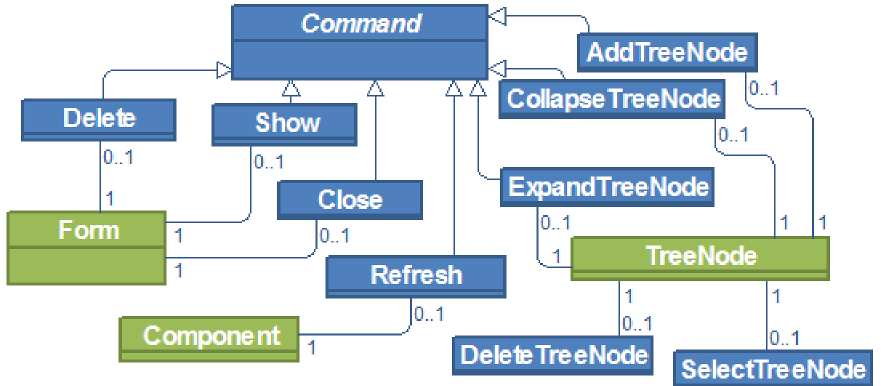
1.9. attēls. Dialogu logu notikumi

Dialogu logu notikumi ir aprakstīti 1.3. tabulā.

1.3. tabula. **Dialogu dziņa notikumu apraksts**

Nosaukums	Paskaidrojums
FocusLost	Iestājas, kad komponente zaudē fokusu. Notikums norāda uz komponenti.
Change	Iestājas, kad komponentei maina vērtību. Notikums norāda uz komponenti.
DropDown	Iestājas, kad *ComboBox* komponentei pieprasa tās saraksta elementus. Notikums norāda uz *ComboBox* komponenti.
Click	Iestājas, kad lietotājs uzklikšķina uz pogas. Notikums norāda uz pogu.
TabChange	Iestājas, kad lietotājs izvēlas citu cilni. Notikums norāda uz jauno cilni.
FocusGained	Iestājas, kad komponente saņem fokusu. Notikums norāda uz komponenti.
TreeNodeSelect	Iestājas, kad koka virsotne tiek aktivizēta. Notikums norāda uz koka virsotni.
TreeNodeExpanded	Iestājas, kad koka virsotne tiek izvērsta. Notikums norāda uz koka virsotni.
TreeNodeCollapsed	Iestājas, kad koka virsotne tiek savilkta. Notikums norāda uz koka virsotni.

Dialogu logu komandu klases ir redzamas 1.10. attēlā.



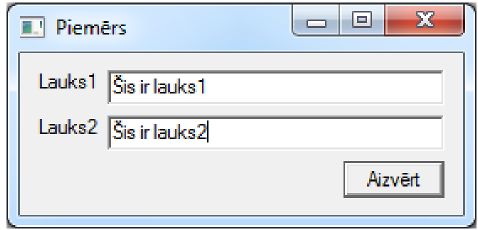
1.10. attēls. Dialogu logu komandas

Dialogu logu dziņa komandas ir aprakstītas 1.4. tabulā.

1.4. tabula. **Dialogu logu komandu apraksts**

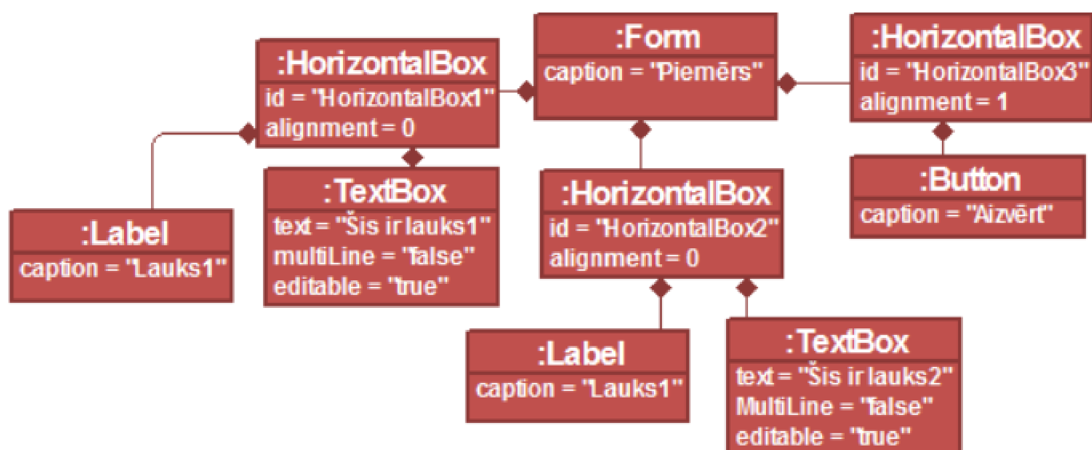
Nosaukums	Paskaidrojums
Show	Pavēle parādīt formu uz ekrāna. Komanda norāda uz formu.
Refresh	Pavēle pārzīmēt kādu formas komponenti. Komanda norāda uz komponenti.
Close	Pavēle aizvērt formu. Komanda norāda uz formu.
Delete	Pavēle izdzēst formu. Komanda norāda uz formu.
AddTreeNode	Pavēle pievienot jaunu virsotni kokā. Komanda norāda uz koka virsotni.
DeleteTreeNode	Pavēle izdzēst virsotni no koka. Komanda norāda uz koka virsotni.
SelectTreeNode	Pavēle aktivizēt koka virsotni. Komanda norāda uz koka virsotni.
ExpandTreeNode	Pavēle izvērst koka virsotni. Komanda norāda uz koka virsotni.
CollapseTreeNode	Pavēle savilkt koka virsotni. Komanda norāda uz koka virsotni.

Kā piemēru aplūkosim 1.11. attēlā redzamo formu, kurai ir divi ievadlauki ar to nosaukumiem katru savā rindā, un poga „Aizvērt”.



1.11. att. Dialoga formas piemērs

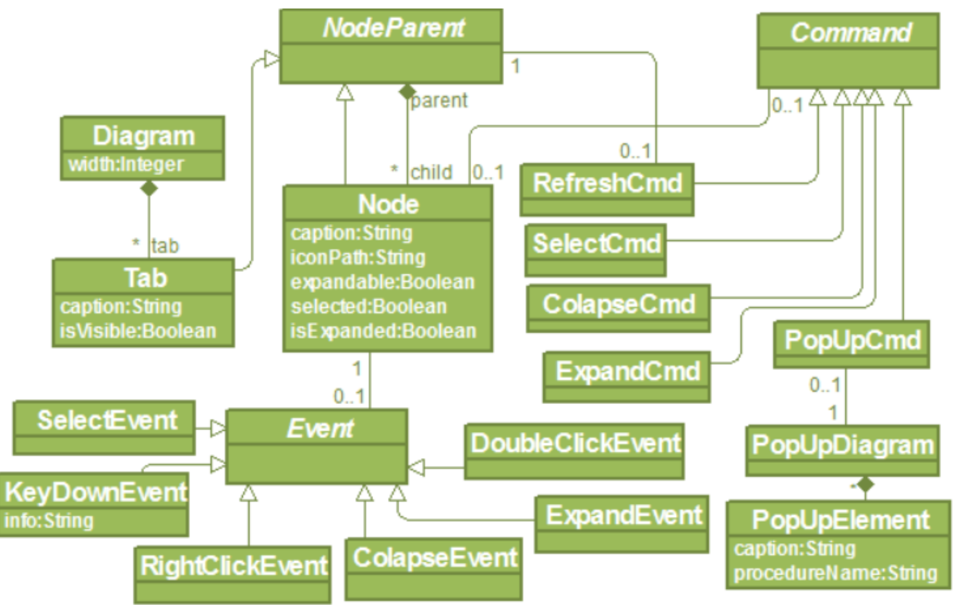
Piemēra formas instanču diagramma ir redzama 1.12. attēlā, kas norāda, ka šī forma sastāv no trīs neredzamiem horizontālajiem konteineriem, kuri uz formas atrodas viens zem otra (pati forma ir vertikālais konteiners). Pirmajos divos horizontālajos konteineros ir ievietoti teksta uzraksti un teksta ievadlauki (izvietoti viens otram blakus), bet trešajā konteinerī - poga, kas ir izlīdzināta pa labi.



1.12. att. Dialoga formas piemēra instanču diagramma

1.1.3 Kokveida diagrammu metamodelis

Kokveida diagrammu metamodeļa instances apraksta kokveida diagrammu struktūru (tām nav tieša sakara ar dialogu logu metamodeļa klasi *Tree*), piemēram, ar kokveida diagrammām var attēlot rīkā esošo grafveida diagrammu hierarhiju. Kokveida diagrammu metamodelis ir redzams 1.13. attēlā.



1.13. att. Kokveida diagrammu metamodelis

Kokveida diagrammu centrālais elements ir virsotne, kurai atbilst klase *Node*. Atribūts *caption* norāda koka virsotnes nosaukumu, *expandable* norāda, vai virsotne ir izvēršama, *isExpanded* norāda, vai virsotne ir izvērsta, *selected* norāda, vai virsotne ir aktivizēta, bet *iconPath* norāda virsotnes ikonas adresi. Katra koka virsotne ir piesaistīta vecākam, kas var būt, vai nu kāda cita virsotne, vai arī cilne (klase *Tab* un kompozīcija *child-parent*). Līdzīgi kā citi dziņi, arī koka dzinis izmanto komandu-notikumu mehānismu un metamodelī tas tiek atspoguļots ar abstraktām klasēm *Command* un *Event*, bet reālās komandu un notikumu klases ir attiecīgi šo abu klašu apakšklases. Komandu un notikumu uzskaitījums ar skaidrojumiem ir aprakstīts tabulās. Kokveida diagrammu notikumi ir aprakstīti 1.5. tabulā.

***1.5. tabula.* Kokveida diagrammu notikumu apraksts**

Nosaukums	Paskaidrojums
Select	Iestājas, kad lietotājs aktivizē koka virsotni (ar peles kreiso pogu uzspiežot uz kādas koka virsotnes). Notikums norāda uz koka virsotni.
KeyDown	Iestājas, kad lietotājs nospiež kādu taustiņu kombināciju uz aktivizētas koka virsotnes. Atribūts *info* norāda ievadīto taustiņu kombināciju. Notikums norāda uz koka virsotni.
RightClick	Iestājas, kad lietotājs veic peles labās pogas klikšķi uz koka virsotnes.

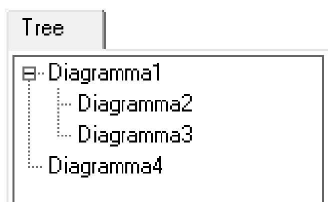
	Notikums norāda uz koka virsotni.
Colapse	Iestājas, kad lietotājs savelk koka virsotni. Notikums norāda uz koka virsotni.
Expand	Iestājas, kad lietotājs izvērš koka virsotni. Notikums norāda uz koka virsotni.
DoubleClick	Iestājas, kad lietotājs veic peles kreisās pogas dubultklikšķi uz koka virsotnes. Notikums norāda uz koka virsotni.

Koka dziņa komandas ir aprakstītas 1.6. tabulā.

***1.6. tabula.* Kokveida diagrammu komandu apraksts**

Nosaukums	Paskaidrojums
Refresh	Pavēle pārzīmēt koka virsotni vai cilni. Komanda norāda uz koka virsotni vai cilni.
Select	Pavēle aktivizēt koka virsotni vai cilni. Komanda norāda uz koka virsotni vai cilni.
Colapse	Pavēle savilkt koka virstoni. Komanda norāda uz koka virsotni.
Expand	Pavēle izvērst koka virsotni. Komanda norāda uz koka virsotni .
PopUp	Pavēle parādīt uznirstošo izvēlni kokā. Komanda norāda uz uznirstošo izvēlni.

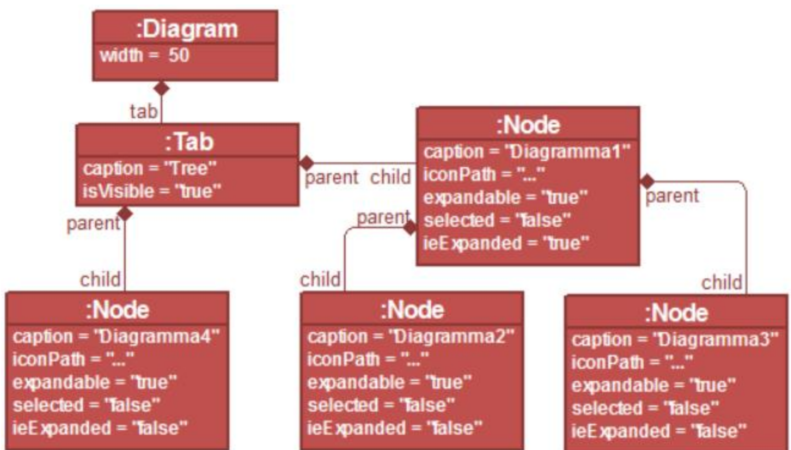
Kā piemēru aplūkosim 1.14. attēlā redzamo kokveida diagrammu.



1.14. att. Kokveida diagrammas piemērs

Šajā diagrammā ir viena cilne un četras virsotnes, kuru nosaukums atbilst diagrammu nosaukumiem, bet virsotņu savstarpējā attiecība atspoguļo diagrammu hierarhiju rīkā

(*Diagramma1* satur elementus, no kuriem var aiznavigēt uz *Diagramma2* un *Diagramma3*). Piemēram atbilstošās metamodeļa instances ir attēlotas 1.15. attēlā.



1.15. att. Kokveida diagrammu metamodeļa instances

1.2 Lua un lQuery

TDA platforma paredz, ka modeļu transformācijas tiek izstrādātas Lua programmēšanas valodā [22], izmantojot šajā valodā realizēto lQuery bibliotēku [23, 24] (autors R. Liepiņš). lQuery bibliotēka (to var arī uztvert kā jaunu transformāciju valodu lQuery) ir speciāli izstrādāta priekš TDA platformas, lai varētu pieslēgties modeļu repozitorijam un veikt tā datu apstrādi. Šāds dalījums, ka tikai modeļu repozitorija datu apstrāde tiek veikta ar speciālas bibliotēkas (transformāciju valodas) funkcijām, bet pārējās darbības tiek pārnestas uz jau realizētu augsta līmeņa programmēšanas valodu [25] Lua, ir ļāvis iegūt (pie tam relatīvi lēti) vienkāršu, bet ļoti spēcīgu modeļu transformāciju izstrādes vidi.

1.2.1 Lua apraksts

Šajā darbā mēs apskatīsim tikai Lua pamatjēdzienus un pamatkonstrukcijas, bet detalizēts Lua apraksts ir dots avotā [22]. Lua valoda ir augsta līmeņa programmēšanas valoda, kas no divu projektu iekšējās lietošanas valodas laika gaitā ir attīstījusies par plaši lietotu programmēšanas valodu apgabalos, kur ir nepieciešama vienkārša, pārnesama un efektīva skriptēšanas valoda. Piemēram, šādi apgabali ir iegultās sistēmas, mobilās iekārtas, web serveri un, it īpaši, spēles. Lua valoda ir izstrādāta ar mērķi, lai tā būtu viegli integrējama ar C programmēšanas valodu, un tādējādi varētu izmantot C valodas stiprās puses, piemēram, augstu veiktspēju, efektīvu zema līmeņa operāciju realizāciju un sadarbību ar trešās puses programmatūru, bet C valodas vājās puses, piemēram, dinamiskas datu struktūras, automātisku atmiņas pārvaldīšanu, kompaktāku kodu, vienkāršāku testēšanu un atkļūdošanu, pārnest uz Lua valodu.

Lua ir interpretēta skriptēšanas valoda (kompilācija notiek izpildes laikā), un, kā daudzām skriptēšanas valodām, arī Lua valoda ir dinamiski tipizēta. Tas nozīmē, ka mainīgo definēšanas laikā netiek norādīts tips, bet tas tiek izrēķināts atkarībā no mainīgajam piešķirtās vērtības. Lua mainīgo vērtības var būt ar šādiem datu tipiem – *nil*, *string*, *number*, *boolean*, *function* un *table*.

Īpašu uzmanību pievērsīsim tabulām (datu tips *table*). Tabulas ir realizētas kā asociatīvie masīvi, t.i., tie ir masīvi, kas var būt indeksēti ne tikai ar veselu skaitli, bet arī ar *string* vai jebkuru citu Lua datu tipu, izņemot *nil*. Pat vairāk, tabulām nav noteikta izmēra, tajās var ievietot patvaļīgu skaitu elementu, turklāt ar atšķirīgiem datu tipiem. Lai gan tabula ir vienīgā Lua datu struktūra, tā ir ļoti spēcīga, un ar to pietiek, lai viegli realizētu tādas tradicionālās datu struktūras kā masīvus, rindas, stekus, kopas, utt.

Tabulas tiek radītas ar atverošo un aizverošo figūriekavu, piemēram, *{}* rada tukšu tabulu, *{1 = „this is string”, „a” = 10}* rada tabulu, kur indeksam 1 atbilst vērtība „this is string” un indeksam „a” atbilst vērtība 10 (pievērsīsim uzmanību, ka vienā tabulā tiek glabātas dažādu datu tipu vērtības, t.i., *string* un *number*, turklāt arī indeksi ir ar dažādiem datu tipiem – *number* un *string*). Lai piekļūtu tabulas *t* vērtībām, raksta *t[1]* vai *t[”a”]*. Lai tabulai *t* pievienotu jaunu vai labotu kādu esošu ierakstu, piemēram, pievienotu vērtību 20 ar indeksu „b”, raksta *t[”b”] = 20*. Savukārt, lai izdzēstu kādu tabulas elementu, attiecīgajam tabulas indeksam piešķir *nil*, piemēram, *t[1] = nil*. Koda fragments, kas ilustrē tabulas pamatoperācijas:

```
t = {1 = „this is string”, „a” = 10}
t[“b”] = 20
print(t[1])    --> „this is string”
print(t[“a”]) --> 10
print(t[“b”]) --> 20
t[1] = nil     -- izdzēš tabulas elementu, kura indekss ir 1
```

Atšķirībā no daudzām citām programmēšanas valodām, mainīgā datu tips var būt arī funkcija (datu tips *function*). Tas nozīmē, ka funkcijas var tikt izveidotas izpildes laikā, tās var piešķirt mainīgajiem, tās var padot kā argumentus citām funkcijām vai atgriezt kā citu funkciju rezultātu, turklāt Lua valodā tiek atbalstītas arī daudzas funkcionālās programmēšanas [26] iespējas. Lai definētu funkciju, piemēram, funkciju, kas argumentam pieskaita vieninieku, jāraksta šādi:

```
function add_one(x)
```

```
return x + 1
end
```

1.2.2 lQuery bibliotēka

lQuery bibliotēka (transformāciju valoda) ir izstrādāta ar mērķi, lai varētu lasīt un rediģēt modeļu repozitorija datus, un tās ideoloģija ir aizgūta no ļoti plaši izmantotās jQuery [27] bibliotēkas programmēšanas valodā JavaScript [28]. Tipiskā scenārijā darbojoties ar modeļu datiem vispirms tiek atlasīta kāda objektu kolekcija, tad ar šīs kolekcijas objektiem kaut ko izdara, pēc tam no atlasītās kolekcijas atlasa citu kolekciju, ar kuru atkal kaut ko izdara, un šādi var turpināt neierobežotu skaitu reižu. lQuery bibliotēkā kolekcijas analogs ir lQuery objekts (jeb, runājot kopu teorijas terminos, multikopa), un visas lQuery bibliotēkas funkcijas ir izstrādātas, lai varētu darboties ar lQuery objektiem. Jāpiebilst, ka lQuery objekti ir realizēti ar tabulām, un tādējādi katra lQuery objekta datu tips ir tabula (bet katra tabula nav lQuery objekts).

lQuery bibliotēka ir ļoti kompakta un tās funkcijas var iedalīt trīs kategorijās. Viena kategorija ir funkcijas, kas no repozitorija atlasa modeļa objektus (repozitorija stāvokli nemainošas funkcijas). Rezultātā tās atgriež lQuery objektu, kas sastāv no atlasītajiem modeļa objektiem. Otrā kategorija ir funkcijas, kas maina repozitorija saturu, bet trešā kategorija ir funkcijas, kas ļauj apsaimniekot un mainīt lQuery objektu saturu, bet nemaina modeļu repozitorija stāvokli.

1.2.2.1 Modeļu datu atlasīšana

Lai varētu veikt modeļu datu atlasīšanu, ir izveidota speciāla vaicājumu valoda lPath, kuras sintakse ir aizgūta no XPath [29] navigēšanas valodas. Pati valoda sastāv no dažiem vienkāršiem primitīviem (1.7. tabulā), kurus kombinējot var izpildīt pat ļoti sarežģītus vaicājumus, nezaudējot vaicājumu lasāmību.

1.7. tabula. **lPath primitīvi**

Sinakse	Skaidrojums
".ClassName"	Atlasa visus klases „ClassName" objektus
"/roleName"	Atlasa objektu kolekciju, uz kuru no sākotnēji dotās kolekcijas objektiem var nokļūt pa saiti ar lomu „roleName"
"[attrName = value]"	Atfiltrē objektus, kuru atribūta „attrName" vērtība sakrīt ar „value"

":has(lPath_expression)"	Ļauj sašaurināt sākotnēji doto kolekciju, kur „lPath_expression" ir lPath izteiksme

Kā piemēru, kas ilustrē visu četru lPath primitīvu lietojumus, aplūkosim vaicājumu, kurā atlasīsim visas klases *Node* instances, kuras atrodas diagrammā ar nosaukumu „Test":

lQuery(".Node:has(/graphDiagram[caption = Test])")

Jāpiezīmē, ka piemērā tiek izmantota funkcija ***lQuery*** no lQuery bibliotēkas, kura kā parametru saņem lPath vaicājuma izteiksmi un pēc tam rezultātu atgriež lQuery objektā. Piemēra pirmajā solī izteiksme lQuery(".Node") atlasa kolekciju, kas satur visus *Node* objektus. Pieliekot izteiksmi ":has(/graphDiagram[caption = Test])" tiek uzlikts filtrs, kas sašaurina pirmajā solī atlasīto kolekciju atbilstoši lPath izteiksmei ("/graphDiagram[caption = Test]"), respektīvi, kolekcijā atstāj tikai tos *Node* objektus, no kuriem pa lomu „graphDiagram" var sasniegt objektus, kuru *caption* vērtība ir „Test".

Otra repozitorija stāvokli nemainoša funkcija ir ***attr***, kura ļauj iegūt repozitorija objekta atribūta vērtību (respektīvi, tās rezultāts ir *string* tipa vērtība). Lai šo funkciju varētu izpildīt, lQuery objektam ir jāsatur tieši viens repozitorija objekts (jeb kolekcijas izmēram ir jābūt viens). Piemēram, izteiksme:

lQuery(".GraphDiagram[caption = Test]"):attr("caption")

atgriež lQuery kolekcijā atlasītā klases *GraphDiagram* objekta atribūta *caption* vērtību (šeit tiek izmantota Lua notācija ar simbolu „:").

1.2.2.2 Repozitorija stāvokli mainošās funkcijas

lQuery bibliotēkā ir četras repozitorija stāvokli mainošas funkcijas. Funkcija ***delete*** izdzēš lQuery kolekcijā esošos repozitorija objektus. Piemēram, lai izdzēstu visus *Element* objektus, jāraksta:

lQuery(".Element"):delete()

Funkcija ***remove_link*** izdzēš saites, kas saista divas lQuery kolekcijas. Šī funkcija saņem divus parametrus – viens ir lomas vārds un otrs lQuery kolekcija. Piemēram, ja ir atlasīta klases *Element* objektu kolekcija un šiem objektiem atbilstošā *GraphDiagram* kolekcija, tad, lai izdzēstu saites starp šīm kolekcijām, ir jāraksta šādi:

elems = lQuery(".Element") --atlasa Element kolekciju

dgrs = elements:find("/graphDiagram") --atlasa GraphDiagram kolekciju

```
elems:remove_link(”graphDiagram”, dgrs)
```

Funkcijai ***attr*** lQuery bibliotēkā ir divas nozīmes. Vienā gadījumā ***attr*** atgriež atribūta vērtību (iepriekš aplūkotais gadījums, kad ***attr*** ir repozitorija stāvokli nemainoša operācija), bet otrā gadījumā ***attr*** uzstāda atribūta vērtību. Gadījumā, ja ***attr*** saņem *string* tipa parametru, tiek atgriezta atribūta vērtība, bet, lai ar ***attr*** uzstādītu objektam atribūta vērtības, kā funkcijas parametrs ir jāpadod tabula, kuras indeksi ir atribūta nosaukumi un indeksu vērtības ir atribūta vērtības. Piemēram, lai klases *GraphDiagam* objektam, kura *caption* vērtība ir „Test”, uzstādītu atribūta *caption* vērtību „Jauna vērtība”, jāraksta:

```
lQuery(”.GraphDiagram[caption = Test]”):attr({caption = ‘Jauna vērtība‘})
```

Lai varētu izveidot jaunus repozitorija objektus, ir jāizpilda funkcija ***lQuery.create***, kurai viens parametrs ir klases nosaukums un otrs ir Lua tabula, kuras indeksi ir atribūta nosaukumi un to vērtības ir atribūtu vērtības. Piemēram, lai izveidotu jaunu *GraphDiagram* instanci, kurai atribūta *caption* vērtība ir „Test”, jāraksta:

```
lQuery.create(”GraphDiagram”, {caption = ‘Test’})
```

1.2.2.3 lQuery kolekciju funkcijas

lQuery bibliotēkā ir realizētas funkcijas ***each***, ***find*** un ***filter***, kas ļauj apsaimniekot lQuery kolekcijas, bet nemaina repozitorija stāvokli.

Funkcija ***each*** ļauj iterēt pa visiem lQuery kolekcijas objektiem. Šī funkcija kā pirmo argumentu saņem iterācijas funkciju, kas tiek izpildīta uz katra kolekcijas objekta (tādējādi tiek izmantots apstāklis, ka funkcija ir Lua valodas primitīvs), bet pārējie argumenti (patvaļīgs skaits) tiks nodoti iterācijas funkcijai kā argumenti. Piemēram, ja ir definēta funkcija, kas objekta atribūta *caption* vērtību pārveido uz lielajiem burtiem:

```
function caption_in_all_caps(obj)
        old_value = obj:attr(”caption”)
        new_value = string.upper(old_value)
        obj:attr({caption = new_value})
end,
```

tad ar šo funkciju var visiem klases *GraphDiagram* objektiem pārmainīt atribūta *caption* vērtības:

```
lQuery(”.GraphDiagram”):each(caption_in_all_caps)
```

Funkcija ***find*** ļauj no vienas lQuery kolekcijas iegūt citu kolekciju un kā parametru saņem lPath izteiksmi, ar kuru navigē no vienas kolekcijas uz otru. Piemēram, ja ir dota kolekcija x, kas sastāv no visiem *Element* objektiem, tad, pielietojot šai kolekcijai find(”/graphDiagram”), iegūst, kolekciju y, kas sastāv no *GraphDiagram* objektiem, uz kuriem var nokļūt pa lomu „graphDiagram” no *Element* objektiem:

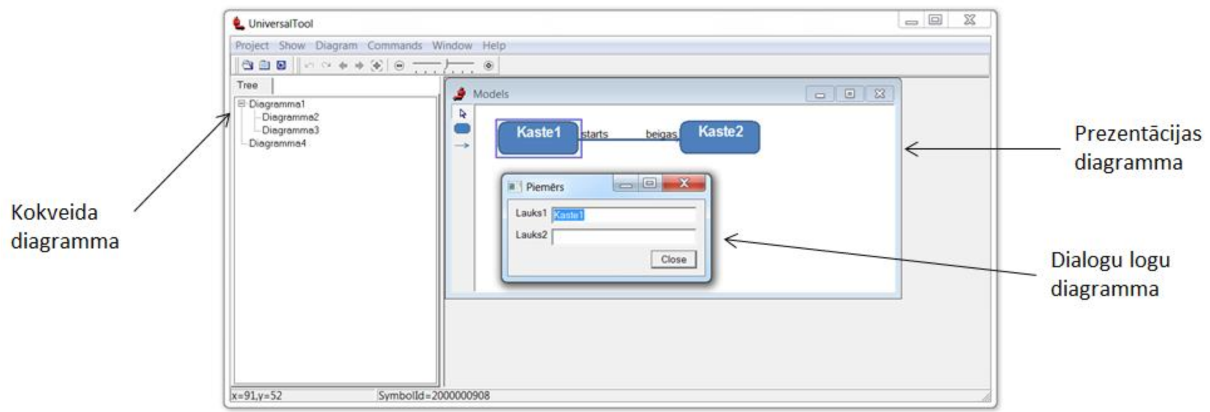
```
x = lQuery(”.Element”) -- atalasa Element kolekciju
y = x:find(”/graphDiagram”)
```

Funkcija ***filter*** ļauj sašaurināt dotu lQuery kolekciju. Piemēram, ja ir dota visu *Element* objektu kolekcija, tad pielietojot funkciju filter(”.Node”) iegūst kolekciju, kas satur *Node* objektus:

```
lQuery(”.Element”):filter(”.Node”).
```

1.3 Platformas ietvars un tā realizācija

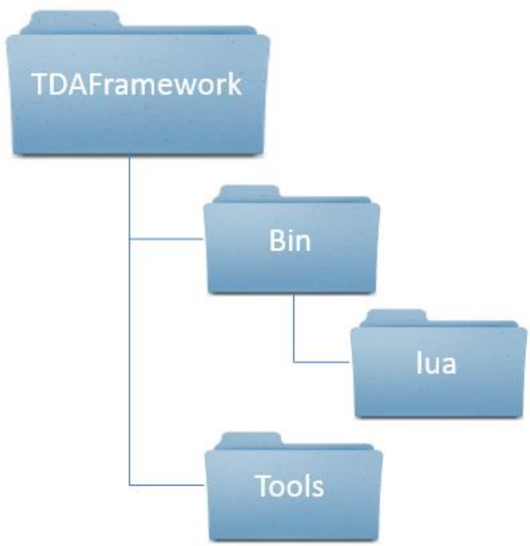
Lai varētu praktiski izmantot iepriekš aprakstītās TDA sastāvdaļas, ir izstrādāts TDA platformas ietvars, ar kura palīdzību tiek realizēti visi TDA izstrādātie rīki. Jeb tehniski runājot, šis ietvars apvieno visas iepriekš aprakstītās platformas sastāvdaļas vienā sistēmā, lai uzbūvētu pilnībā funkcionējošu rīku. 1.16. attēlā ir redzams viena realizēta rīka piemērs, izmantojot šo ietvaru.



1.16. att. TDA ietvars

Ietvara ideja ir „pirms iekavām” iznesta visiem rīkiem kopīgo funkcionalitāti, piemēram, projekta saglabāšanu, diagrammu drukāšanu, utt., lai tās nebūtu katru reizi jāprogrammē no jauna, bet to funkcionalitāti, kas ir atkarīga no konkrēta rīka, realizēt caur dziņiem un modeļu transformācijām, respektīvi, modeļu transformācijas veic rīkam specifisko lietotāju darbību apstrādi un attiecīgo modeļu datu izveidošanu, bet ietvarā šos datus dziņi attēlo uz ekrāna.

Lai varētu izstrādāt modeļu transformācijas un attiecīgi pašu rīku, rīku izstrādātājiem ir nepieciešams zināt platformas direktoriju un datņu struktūru, kura ir parādīta 1.17. attēlā, bet viena konkrēta rīka realizācija atbilstoši šai struktūrai ir parādīta 1.19. attēlā. Visas platformas datnes atrodas *TDAFramework* direktorijā, kura sastāv no divām apakšdirektorijām. Direktorija *Tools* satur rīkus, kas ir izveidoti ar šo platformu (sākotnēji tā ir tukša). Savukārt direktorijā *Bin* atrodas visas platformas datnes, kas nodrošina platformā izstrādāto rīku darbību. Šajā direktorijā ir apakšdirektorija *lua*, kura ir paredzēta transformāciju datņu glabāšanai, un pēc noklusējuma tajā atrodas viena datne *root*, kura satur projekta inicializācijas transformāciju *create_project*, kas izpildās, kad pirmo reizi atver kādu projektu.



1.17. att. TDA datņu struktūru

Tādējādi jauna rīka izstrāde sākas ar to, ka mēs izveidojam jaunu projektu TDA ietvarā. Jaunā projekta nosaukums sakrīt ar izstrādājamā rīka nosaukumu, piemēram, *DemoTool*, un par šī projekta atrašanās vietu ir jānorāda *Tools* direktorija. Tā rezultātā *Tools* direktorijā platforma izveidos apakšdirektoriju, kas atbilst jaunajam projektam (mūsu gadījumā tas nosaukums būs *DemoTool*), un tad ietvarā atvērsies jaunais projekts, kura repozitorijā atradīsies visu dziņu metamodeļi, t.i., prezentācijas, dialogu logu un kokveida diagrammu metamodeļi. Pēc tam izpildīsies transformācija *create_project*, kuras saturs ir jāizstrādā rīka izstrādātājam, un tās primārais uzdevums ir katram dzinim norādīt transformācijas, kas veiks tā notikumu apstrādi. Lai varētu norādīt šīs transformācijas, ietvarā katram dzinim ir izveidota viena klase, kuras atribūti atbilst dziņa notikumiem, kā tas ir parādīts 1.18. attēlā.

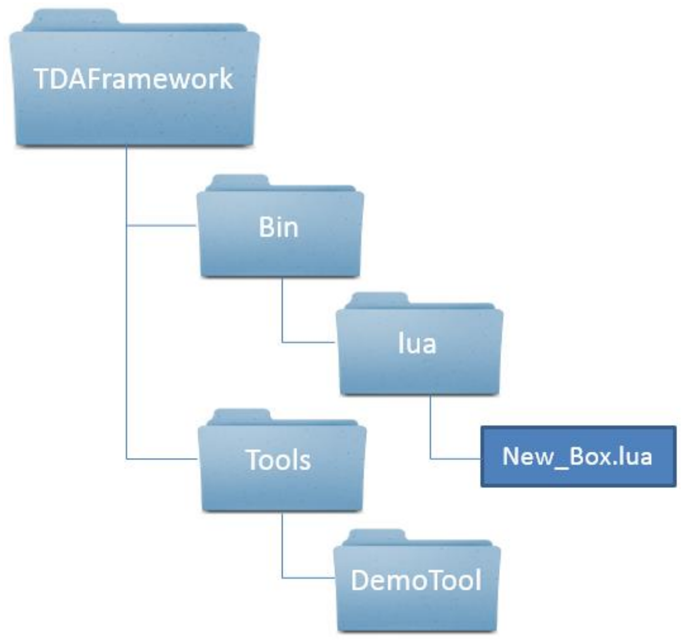
PresentationEngine
NewBox:String
NewLine:String
NewPin:String
MoveLineStartPoint:String
MoveLineEndPoint:String
ChangeParent:String
LClick:String
L2Click:String
RClick:String
ToolbarElementSelect:String
PopUpElementSelect:String
KeyDown:String
ActivateDgr:String
CloseDgr:String

DialogEngine
FocusLost:String
Change:String
DropDown:String
Click:String
TabChange:String
FocusGained:String
TreeNodeSelect:String
TreeNodeExpanded:String
TreeNodeCollapsed:String

TreeEngine
Select:String
KeyDown:String
RightClick:String
Colapse:String
Expand:String
DoubleClick:String

1.18. att. Dziņu notikumu klases

Tādējādi, lai specificētu konkrētu rīku, katrai 1.18. attēla klasei mums ir jāizveido viena instance un to atribūtos ir jānorāda notikumus apstrādājošās transformācijas. Piemēram, pieņemsim, ka mēs gribam priekš prezentācijas dziņa *NewBox* notikuma apstrādes izstrādāt transformāciju, kuras nosaukums ir *JaunaKaste*. Tādā gadījumā mums *lua* direktorijā ir jāizveido jauna datne, piemēram, *New_Box.lua* (skat. 1.19. attēlu), kura satur funkciju ar nosaukumu *JaunaKaste*. Pēc tam, lai norādītu, ka šī transformācija ir atbildīga par *NewBox* notikuma apstrādi, klases *PresentationEngine* atribūtā *NewBox* ir jānorāda vērtība, kas atbilst transformācijas *JaunaKaste* relatīvajai adresei no *lua* direktorijas (vispārīgā gadījumā *lua* direktorijai drīkst veidot arī apakšdirektorijas, un adreses formāta regulārā izteiksme ir šāda – *(direktorijas_nosaukums.)*datnes_nosaukums.funkcijas_nosaukums*), t.i., šajā piemērā transformācijas relatīvā adrese ir „New_Box.JaunaKaste”.



1.19. att. TDA datņu struktūru pēc New_Box.lua pievienošanas

Vispārīgi skatoties, ja nepieciešams, transformācija *create_project* projekta veidošanas brīdī drīkst veikt arī vēl citas darbības (transformācijām ir pieejams viss repozitorijs), piemēram, izveidot domēna metamodeli vai pievienot jaunas instances. Tādējādi pēc transformācija *create_project* izpildes (jeb jaunā rīka inicializācijas) repozitorijā katram platformas notikumam ir jābūt piekārtotai kādai transformācijai, kas atbild par šī notikuma apstrādi, un, kad tas ir izdarīts, šo projektu vajag saglabāt, un rīks ir „gatavs”. Citiem vārdiem sakot, mēs varam izveidot jaunus šī rīka projektus, un to atvēršanas brīdī tiem būs tieši tāds repozitorija stāvoklis, kādu mēs iepriekš saglabājām.

1.4 Secinājumi

Ar šajā nodaļā aprakstītajām platformas komponentēm teorētiski pietiek, lai varētu būvēt DSML rīkus, proti, katram rīkam ir jāizstrādā savs modeļu transformāciju komplekts, kas veic lietotāju notikumu apstrādi atbilstoši rīka specifikācijai. Taču tas ir darbietilpīgs process un tādēļ izvirzās jautājums, vai šo procesu nevar būtiski vienkāršot. Promocijas darbā ir izstrādāts šīs problēmas risinājums, un tā plāns paredz sekojošas lietas:

- izstrādāt formālu DSML rīku specificēšanas valodu;
- izstrādāt šīs valodas interpretatoru kā universālu modeļu transformācijas kopumu, kas dotu rīka specifikāciju pārvērš strādājošā rīkā;
- ātrākai un ērtākai specifikāciju uzdošanai izstrādāt konfiguratoru.

2 DSML valodu un to rīku specificēšana

Šīs nodaļas mērķis ir atrast formālu specificēšanas valodu (jeb metodi), ar kuru varētu aprakstīt plašu DSML valodu saimi un to atbilstošos rīkus. Ja mums izdotos atrast šādu formālu valodu, tad mēs varētu uzbūvēt universālu interpretatoru (vai kompilatoru), kas katru šīs valodas specifikāciju (izmantojot, piemēram, TDA platformu) pārvērš strādājošā rīka. Tā rezultātā, lai iegūtu strādājošu rīku, mums pietiktu padot šim interpretatoram konkrētā DSML rīka specifikāciju minētajā valodā.

Lai labāk saprastu iepriekš minētā uzdevuma sarežģītību, apskatīsim vienkāršāku uzdevumu, proti, meklēsim formālu specificēšanas metodi, ar kuru var definēt plašu DSML valodu saimi (bez to atbilstošajiem rīkiem). Galvenā problēma ir apstāklī, ka šobrīd ir zināmas vienlīmeņa metamodelēšanas valodas, t.i., ir labi zināmas metodes, kā ar metamodeļu palīdzību definēt konkrētas valodas, piemēram, UML aktivitāšu diagrammas, UML klašu diagrammas, blokshēmu diagrammas, utt., bet nav formālas metodes, kas ļautu pacelties vienu meta līmeni augstāk, t.i., nevis definēt vienu konkrētu valodu, bet veselu valodu saimi. Citiem vārdiem sakot, ir labi atrisināta vienlīmeņa metamodelēšanas problēma, bet ne divlīmeņu problēma DSML valodu kontekstā.

Par vienu no pirmajiem mēģinājumiem šīs problēmas risināšanā var uzskatīt UML stereotipu [30] mehānismu, proti, ņemot par pamatu konkrētas UML diagrammas, ar UML stereotipu palīdzību varam definēt dažādus profilus, kas no modelētāja viedokļa tradicionālās UML diagrammas pārvērš par citām modelēšanas valodām. Vienīgi jāatzīmē, ka tradicionālais UML stereotipu mehānisms neļauj elastīgi regulēt elementa grafisko formu, t.i., tradicionālais stereotipa mehānisms „dzīvo" abstraktās sintakses līmenī (ja neskaita, ka stereotipa vārdu var aizstāt ar ikonu). Tādēļ 2.1. nodaļā „Specializācijas metode" [31] mēs attīstīsim tālāk šo UML stereotipu (profilu) mehānismu, lai ar tā palīdzību definētu pietiekoši plašu grafisku DSML valodu saimi. Šajā sakarā mums ir jāatrisina divas problēmas. Pirmkārt, jāizvēlas pietiekoši laba grafisko diagrammu bāzes modelēšanas valoda (UML diagrammas šim nolūkam nav pietiekoši ērtas, jo tās savā metamodelī neietver grafikas daļu). Otrkārt, priekš stereotipu lietojumiem jāizstrādā daudz ērtāks un lasāmāks attēlojumu veids nekā tas ir UML.

Savukārt 2.2. nodaļā „Konkretizācijas metode" mēs piedāvāsim alternatīvu (oriģinālu) DSML valodu definēšanas valodu, kura balstās uz tipu metamodeļa jēdziena. Abas šīs metodes no modelēšanas iespēju viedokļa ir līdzvērtīgas, taču 2.2. nodaļā piedāvātajai metodei ir vairākas priekšrocības (par tām sīkāk 2.3. nodaļā), un tāpēc priekš realizācijas ir izvēlēta šī metode.

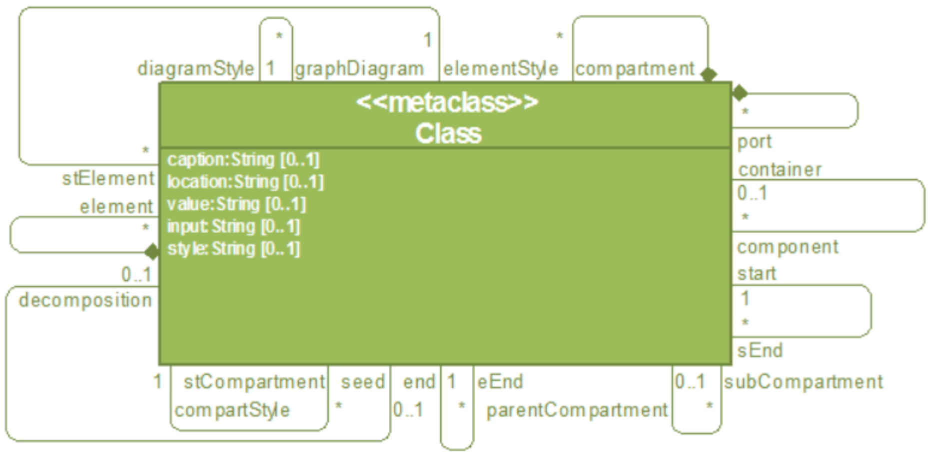
Atcerēsimies, ka, lai specificētu DSML rīkus, mums bez DSML valodas ir jāspecificē arī atbilstošo rīku uzvedība, piemēram, paletes jaunu elementu veidošanai, rīkjoslas, uznirstošās izvēlnes, utt. Tāpēc nodaļās 2.1 un 2.2 ir risināta ne tikai valodas definēšanas problēma, bet arī to atbilstošo rīku specificēšanas problēma, proti, nodaļā 2.1.3. ir attīstīta rīku definēšanas metode kā papildinājums valodas profilam, bet nodaļā 2.2.2. ir attīstīta rīku definēšanas metode kā papildinājums valodas tipu metamodelim.

Izvirzās jautājums, kā DSML rīku specificēšana tiek risināta citās rīku būves platformās. Sīkāk par to ies runa 6. nodaļā, šeit atzīmēsim pamatatšķirību. Lai definētu DSML rīkus, visas platformas piedāvā kaut kādas konfigurēšanas iespējas, un tās veiksmīgi strādā līdz dotajā platformā ir nepieciešams izstrādāt nerealizētu funkcionalitāti. Tādā gadījumā šo funkcionalitāti ir nepieciešams papildus noprogrammēt, bet, lai to izdarītu, ir nepieciešams iejaukties platformas iekšējā realizācijā, kas ir sarežģīti. Tāpēc šajā nodaļā abas aprakstītās specificēšanas metodes pretendē uz to, ka, pirmkārt, tās dod formālu DSML rīku specifikāciju UML klašu diagrammu formā, otrkārt, tās nodrošina rīku atvērtību, jo to rīku metamodeļi kalpo arī par rīka iekšējo datu struktūru un, treškārt, tās ļauj relatīvi viegli pielikt dažādus papildinājumus un padarīt rīkus atvērtus pret dažādām ārējām lietojumprogrammām.

2.1 Specializācijas metode

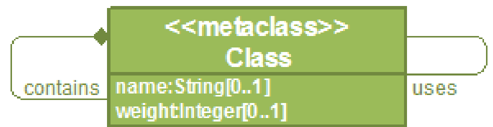
2.1.1 UML stereotipa mehānisma attīstīšana

Tradicionāli UML stereotipi tiek lietoti, lai paplašinātu esošus UML metamodeļus un to rīkus padarītu domēnspecifiskus. Lai gan no DSML rīku skatpunkta šī pieeja ir ierobežojoša, tajā pašā laikā UML stereotipu ideja ir spēcīgāka nekā tās tradicionālie lietojumi. Tādēļ šajā nodaļā attīstīsim UML stereotipu ideju, un sagatavosim bāzi DSML rīku definēšanai.



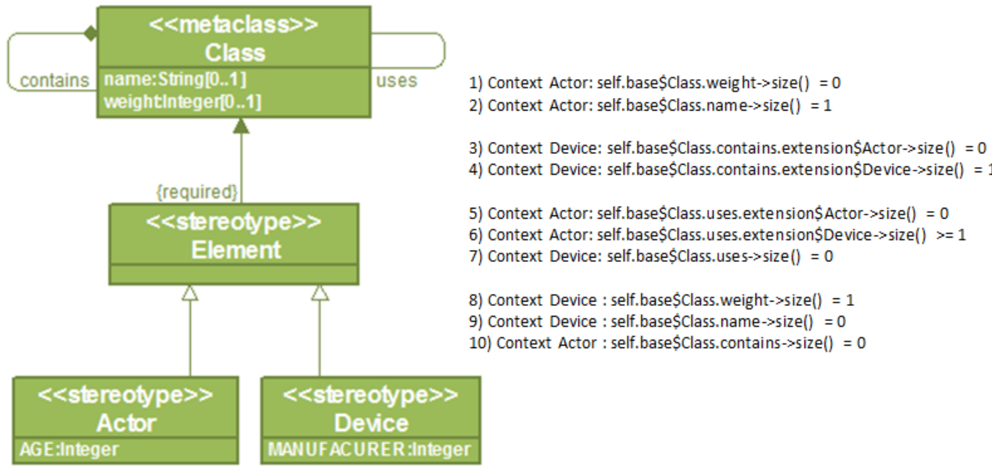
2.1. att. Stereotipu bāzes metamodelis

Priekš mūsu mērķa par stereotipu definēšanas bāzes metamodeli ņemsim metamodeli, kas sastāv tikai no vienas klases un ir redzams 2.1. attēlā. Šo klasi sauksim par meta-klasi, un tā satur visus nepieciešamos atribūtus un asociācijas priekš DSML valodas definēšanas (atribūtu un asociāciju semantika būs izskaidrota vēlāk).



2.2. att. Vienkāršs metamodeļa piemērs

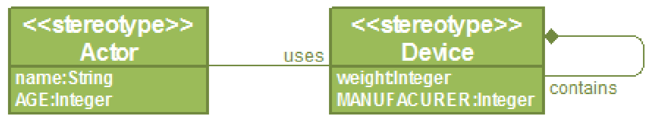
Pirms turpinām ar stereotipu pielietojumu mūsu uzdevumam, ieviesīsim saīsināto stereotipu un profila notāciju. Lai izskaidrotu šo notāciju, apskatīsim vienkāršotu bāzes metamodeli, kas ir redzams 2.2. attēlā. Balstoties uz šo metamodeli definēsim 2.3. attēlā redzamo profilu, izmantojot oficiālo UML notāciju. Profils satur vairākus ierobežojumus. Pirmkārt, iezīme *required* norāda, ka klase *Element* ir obligāta, un tas nozīmē, ka šim profilam atbilstošie modeļi var saturēt tikai klases ar stereotipiem *Actor* un *Device*. Otrkārt, klasēm ar stereotipu *Actor* ir jāsatur atribūts *name*, bet ne atribūts *weight* (apgalvojumi (1+2)). Treškārt, klasēm ar stereotipu *Device* ir jāsatur *weight*, bet ne *name* (apgalvojumi (3+4)). Ceturtkārt, asociācija *uses* drīkst savienot tikai stereotipu klasi *Actor* ar stereotipa klasi *Device* (apgalvojumi (5+6+7)). Pietkārt, asociācija *contains* drīkst savienot tikai stereotipa klases *Device* (apgalvojumi (8+9+10)).



2.3. att. Profila piemērs

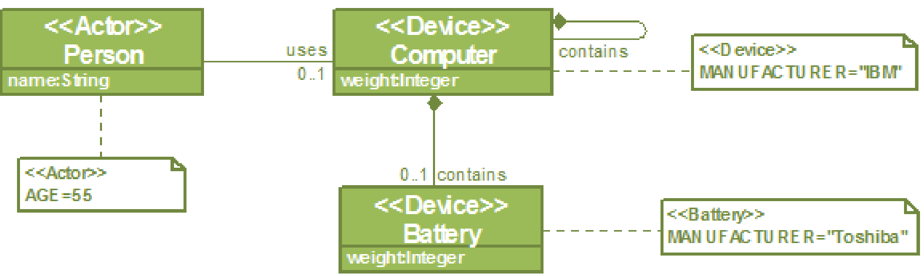
Acīmredzot UML oficiālā stereotipu notācija ne vienmēr ir viegli uztverama, tāpēc 2.4. attēlā ir nodemonstrēta darbā piedāvātā saīsinātā notācija. Lai atšķirtu stereotipu atribūtus (AGE, MANUFACTURER) no atribūtiem, kas ir doto stereotipu klašu atribūti, pēdējie vienmēr būs rakstīti ar mazajiem burtiem (šie atribūti tiek mantoti no meta-klases), bet stereotipa atribūti vienmēr būs rakstīti ar lielajiem burtiem. Šāda saīsinātā profila notācija precīzi parāda, kādus atribūtus un ar kādu kardinalitāti ir atļauts lietot, veidojot jaunas atbilstošā stereotipa klases (t.i., dotā profila modeļus). Otrs pieņēmums attiecībā uz saīsināto notāciju ir tāds, ka ierobežojums *required* ir noklusētais ierobežojums.

Patiesībā iepriekš aprakstītā saīsinātā profila notācija jau satur visu informāciju, ko mēs „paņēmām" no bāzes metamodeļa klases. Tādējādi bāzes metamodeļa klase (2.2. attēlā) ir lieka, lai gan, no otras puses, to var uzskatīt par neatkarīgu otrā līmeņa modelēšanas valodu.



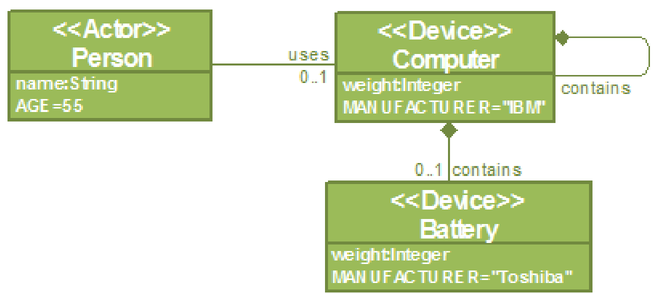
2.4. att. Profila saīsinātā pieraksta notācija

Turpmāk modeļiem dotajā profilā mēs lietosim vēl vienu saīsināto notāciju. 2.5. attēlā ir redzams piemērs, kura modelis ir uzdots atbilstoši oficiālajai UML sintaksei.



2.5. att. Modelis UML sintaksē

Tā kā mums ir notācija, kas ļauj viennozīmīgi atšķirt stereotipu atribūtus no meta-klašu atribūtiem, mēs varam stereotipa atribūtus rakstīt kastes iekšpusē, kā tas ir parādīts 2.6. attēlā.

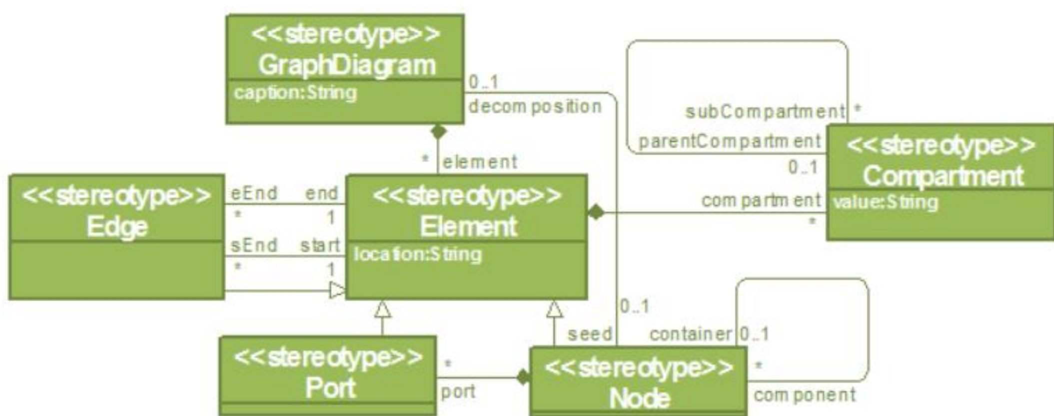


2.6. att. Piemērs saīsinātajai pieraksta notācijai

Tādējādi, pievienojot oficiālajai UML profila notācijai OCL [32] ierobežojumus, mēs iegūstam divas lietas. Pirmkārt, mēs katram meta-klases atribūtam varam precīzi norādīt tā piederību konkrētai stereotipa klasei. Otrkārt, pievienojot OCL ierobežojumus, mēs katrai meta-klases asociācijai varam precīzi norādīt tās stereotipa klases, kuras tā saista, un tādā veidā apiet oficiālās notācijas ierobežojumu, ka meta-klases asociācijas nedrīkst izmantot profilā (bet drīkst izmantot OCL apgalvojumos).

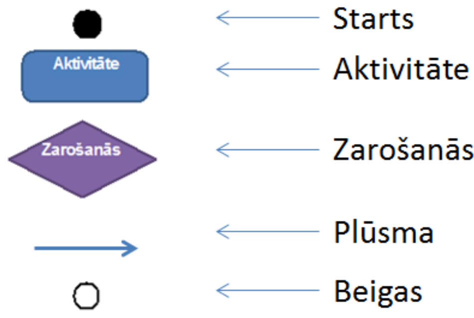
2.1.2 DSML valodas profils

DSML rīku specifikācijai ir jāapraksta divas lietas - izstrādājamā rīka valoda un uzvedība. Šajā sadaļā apskatīsim valodas abstraktās sintakses specificēšanu ar 2.7. attēlā parādīto profilu, kura semantika ir tāda pati kā 1.2. attēlā redzamajam prezentācijas metamodeļa kodolam.



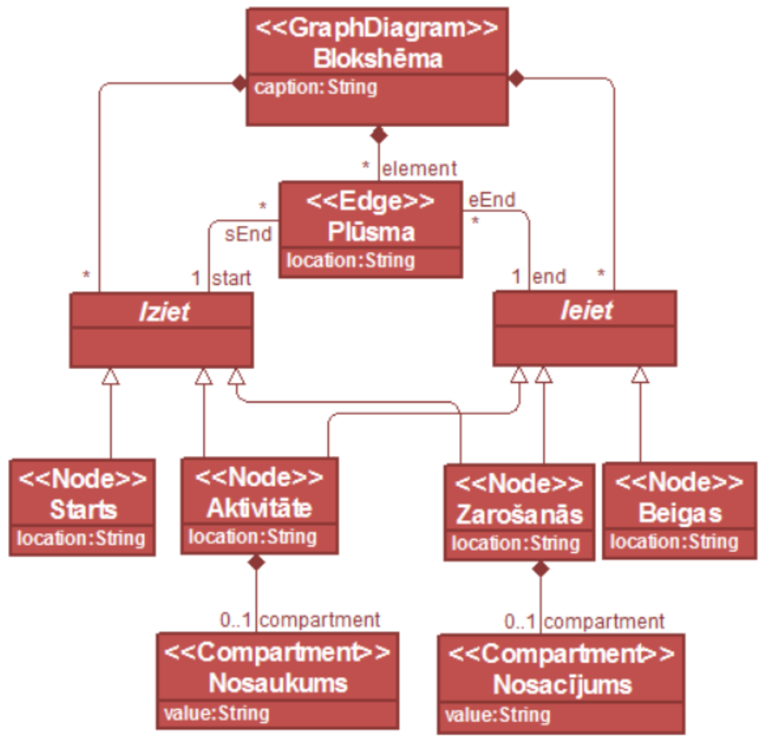
2.7. att. DSML valodu abstraktās sintakses profils

Lai nodemonstrētu profila lietojumu, specificēsim blokshēmu valodu, kuras piemērs ir redzams 2.8. attēlā.



2.8. att. Blokshēmu valodas elementi

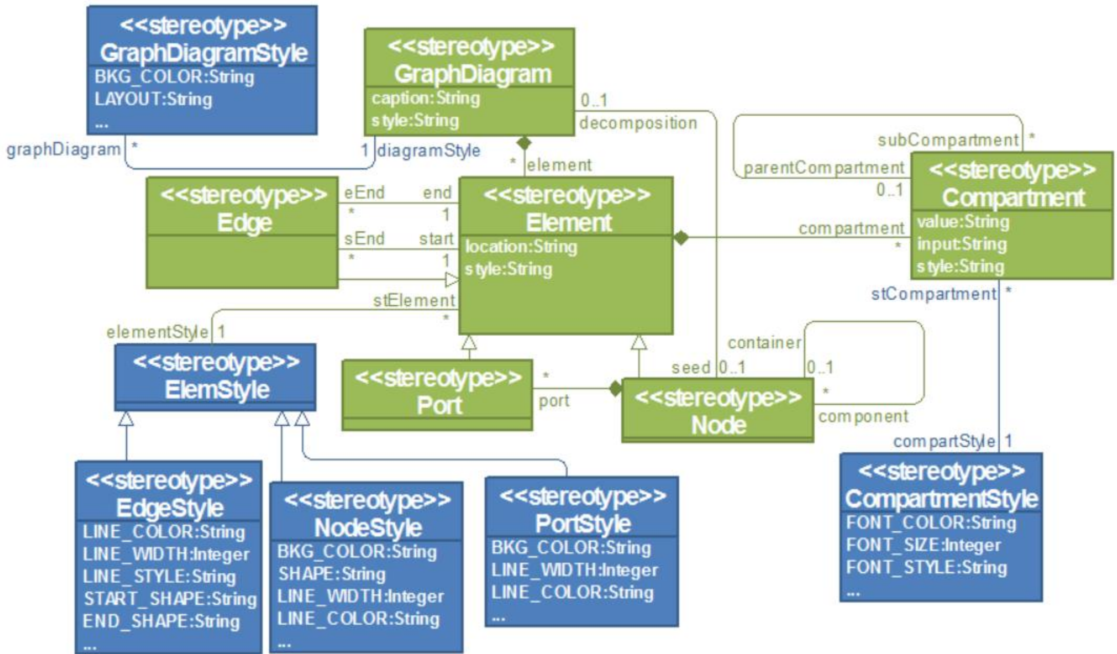
Šī valoda sastāv no šādiem elementiem „Starts” (melnais riņķis), „Aktivitāte” (taisnstūri ar noapaļotajiem stūriem), „Zarošanās” (rombs), „Beigas” (riņķis ar „tukšu vidu”) un „Plūsma” (līnijas, kas savieno blokshēmas elementus), un to specifikācija ar profilu ir redzama 2.9. attēlā.



2.9. att. Blokshēmu valodas abstraktās sintakses specifikācija

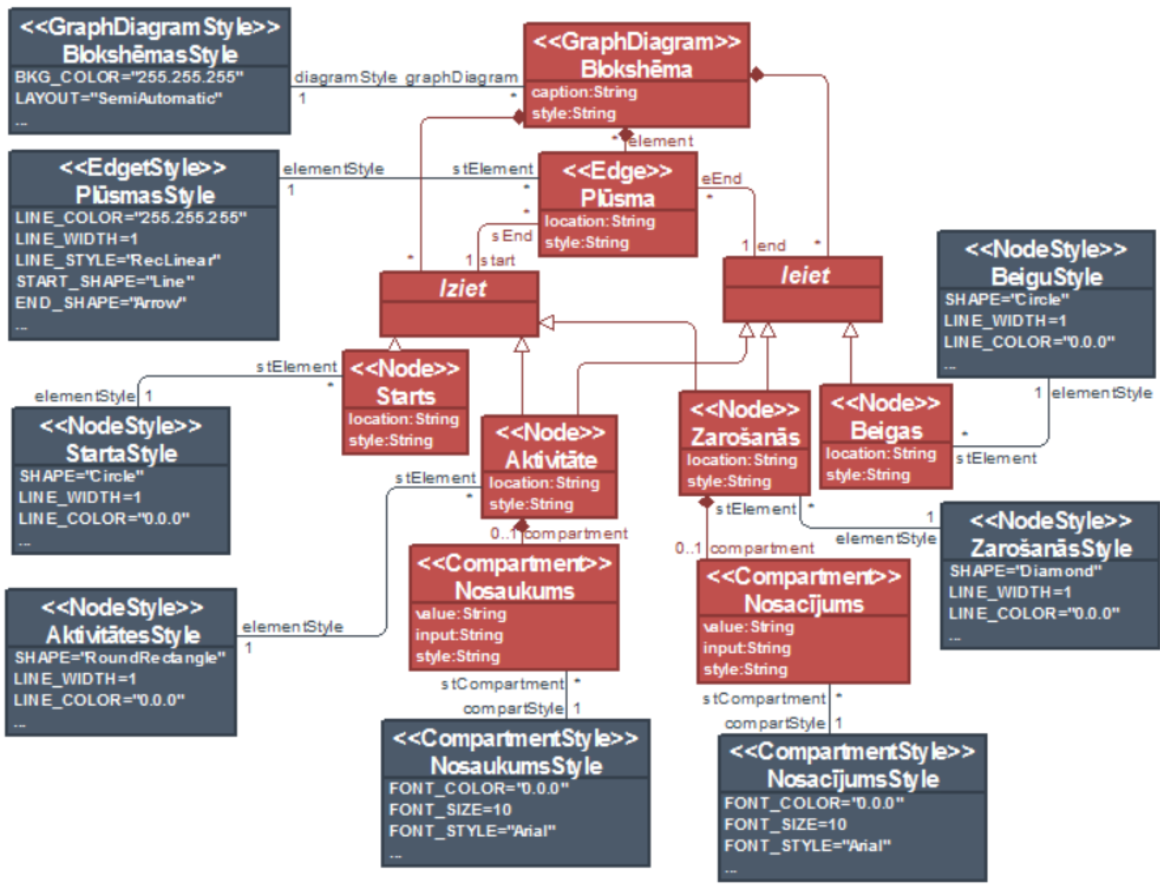
Iepriekš aprakstījām, kā var specificēt valodas abstrakto sintaksi, bet, lai varētu specificēt valodas konkrēto sintaksi, ir jānorāda stili. Lai to izdarītu, abstraktās sintakses profils ir papildināts ar stereotipu klasēm - *CompartmentStyle*, *ElemStyle*, *LineStyle*, *BoxStyle*, *PortStyle* un *GraphDiagramStyle*, kuras ļauj norādīt noklusētos stilus, un vēlāk to vērtības (noteiktā kodējumā) ienest *style* atribūtā. Savukārt stereotipa klases *Compartment* atribūts

input ļauj norādīt atribūtu reprezentatīvo vērtību (atribūts *value* norāda loģiskās vērtības). DSML valodu konkrētās sintakses modelis ir redzams 2.10. attēlā.



2.10. att. DSML valodu konkrētās sintakses profils

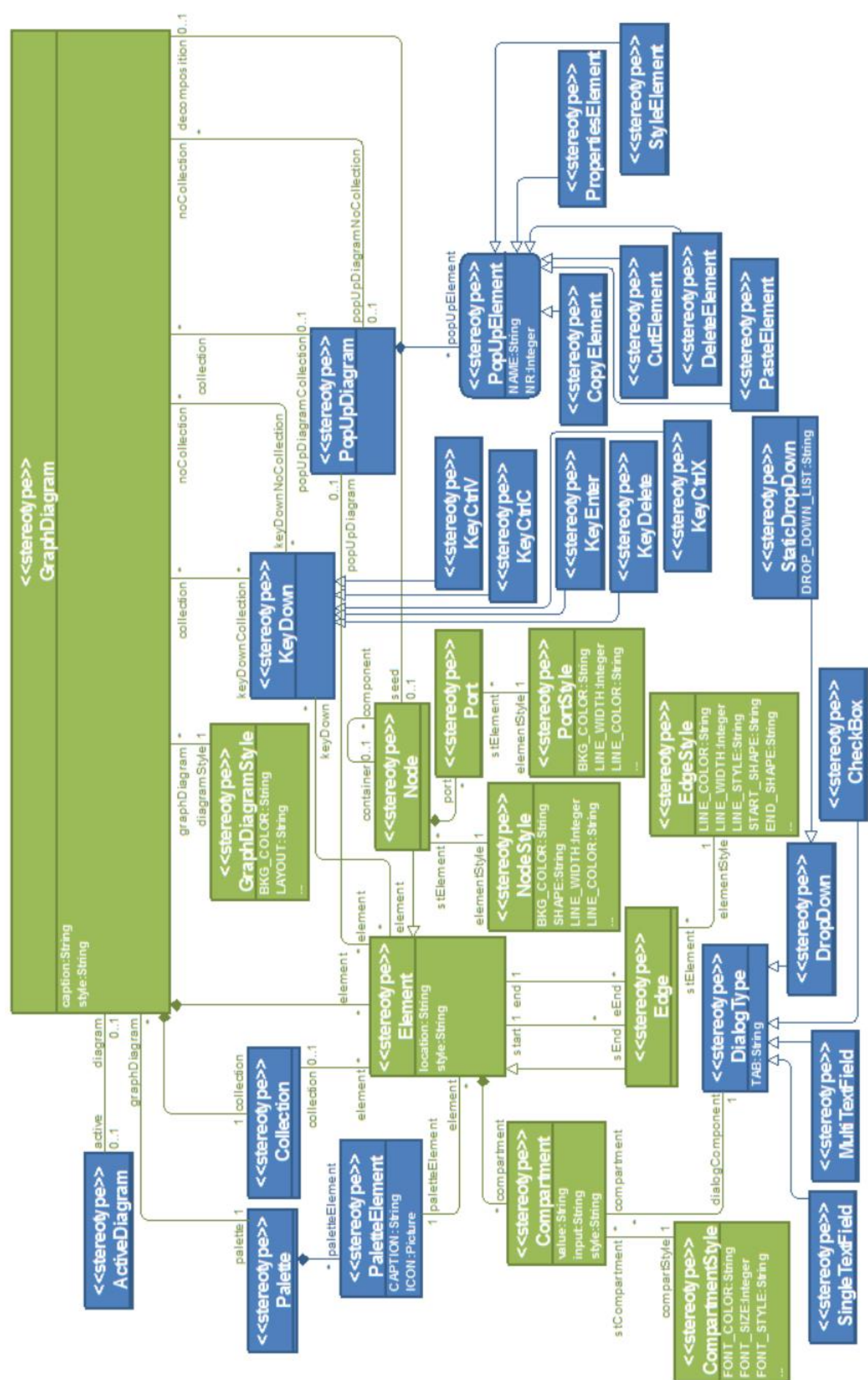
Blokshēmu valodas konkrētās sintakses specifikācija ir redzama 2.11. attēlā.



2.11. att. Blokshēmu valodas konkrētās sintakses specifikācija

2.1.3 DSML rīku profils

Kā iepriekš tika minēts, DSML rīku specifikācija sastāv no divām daļām – rīka valodas un uzvedības specifikācijām. Lai iegūtu DSML rīku profilu, DSML valodas profils ir papildināts ar jaunām klasēm un asociācijām, kā tas ir redzams 2.12. attēlā (lai nesarežģītu 2.1. attēlu, šīs jaunās asociācijas tajā nav attēlotas).



2.12. att. DSML rīku profils

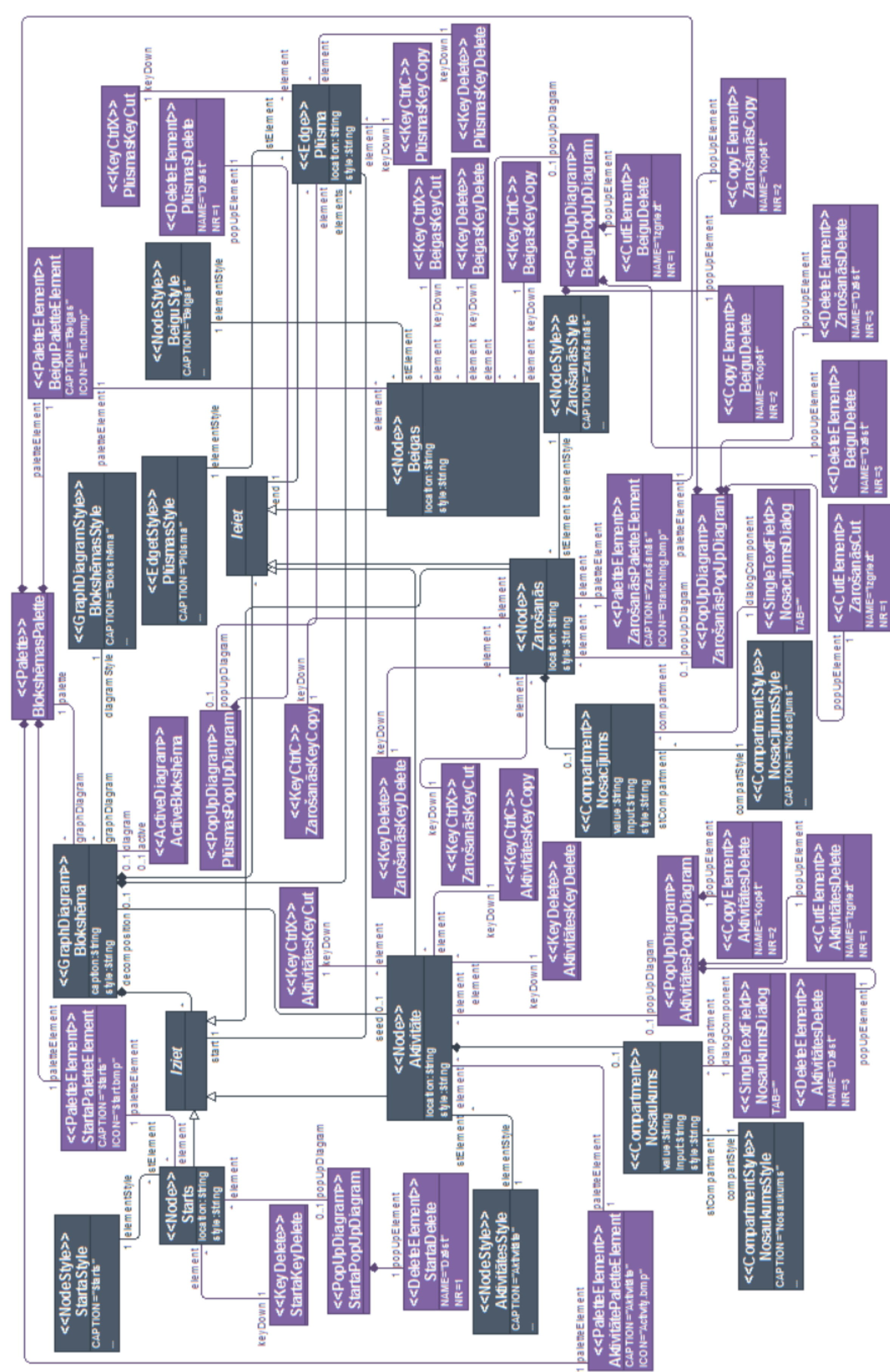
Stereotipu klases *Palette* un *PaletteElement* ļauj veidot jaunus diagrammas elementus, atribūts CAPTION norāda paletes elementa nosaukumu, ICON norāda ceļu uz paletes elementa ikonu. Asociācija *paletteElement-element* norāda diagrammas elementam atbilstošo paletes elementu.

Stereotipu klases *PopUpDiagram* un *PopUpElement* ļauj būvēt uznirstošās izvēlnes. Uznirstošās izvēlnes var būt piekārtotas elementiem, diagrammu kolekcijām (satur vairāk nekā vienu elementu) vai tukšai kolekcijai (neviens diagrammas elements nav iezīmēts). *PopUpElement* atribūts NAME norāda izvēlnes elementa nosaukumu, NR izvēlnes elementa numuru. *CutElement*, *CopyElement*, *PasteElement*, *DeleteElement*, *PropertiesElement* un *StyleElement* ir priekš definētas uznirstošo izvēļņu elementi, kas nodrošina pamatfunkcionalitāti – izgriezt, kopēt, ielīmēt, dzēst, atvērt dialogu logu un stila dialogu logu.

Stereotipa klase *KeyDown* ļauj norādīt nospiesto taustiņu kombināciju apstrādi. Līdzīgi kā uznirstošajām izvēlnēm, taustiņu kombināciju apstrādes ir piesaistītas vienam elementam, elementu kolekcijai vai tukšai kolekcijai. Tradicionāli rīkos pēc noklusējuma tiek atbalstītas šādas taustiņu kombinācijas - Ctrl+X, Ctrl+C, Ctrl+V, Delete un Enter, kas nodrošina šādu funkcionalitāti – izgriezt, kopēt, ielīmēt, dzēst un atvērt dialogu logu. Profilā šīm darbībām atbilst klases *KeyCtrlX*, *KeyCtrlC*, *KeyCtrlV*, *KeyDelete* un *KeyEnter*.

Atribūtu vērtības tiek ievadītas caur dialogu logu formām, un stereotipa klase *DialogType* un tās apakšklases *SingleTextFiled*, *MultiTextField*, *StaticDropDown* un *CheckBox* ļauj norādīt atribūta ievadlauka tipu uz dialogu logu formas.

Bez minētājām klasēm, profils vēl satur stereotipa klases *ActiveDiagram* un *Collection*. *ActiveDiagram* mērķis ir norādīt aktīvo diagrammu (asociācija *active-diagram*). Savukārt, *Collection* caur asociāciju *collection-element* ļauj norādīt aktīvos diagrammas elementus. 2.13. attēlā ir redzams pilna blokshēmu redaktora specifikācija ar DSML rīku profilu.



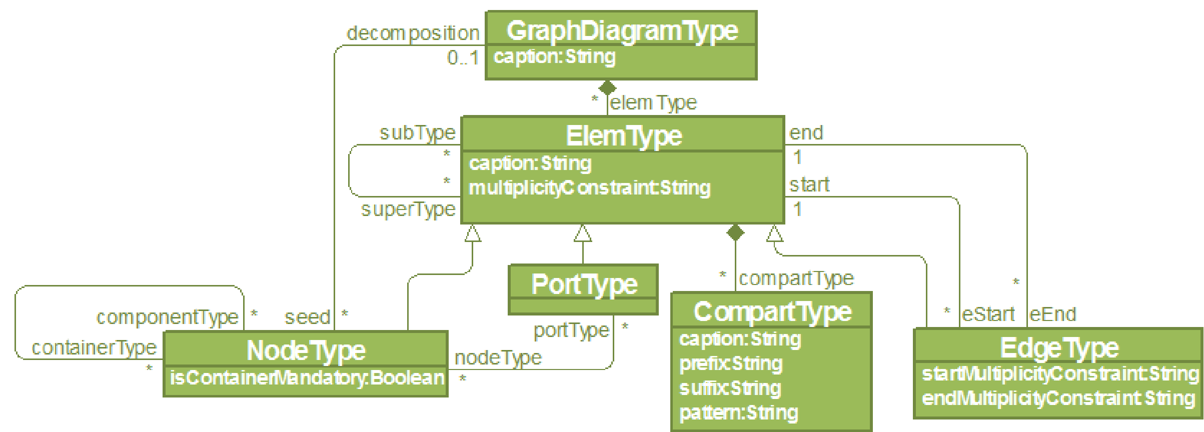
2.13. att. Blokshēmu redaktora specifikācija

2.2 Konkretizācijas metode

Šajā sadaļā mēs ieviesīsim citu DSML rīku definēšanas metodi, kuru sauksim par konkretizācijas metodi. Atbilstoši šai metodei ir izstrādāts universāls DSML rīku definēšanas metamodelis. Tas nozīmē, ka katra rīku definēšanas metamodeļa instance definē vienu konkrētu DSML rīku. Tā kā rīku definīcijas sastāv no rīka valodas un uzvedības definīcijām, tad arī šī metamodeļa izklāstu sadalīsim attiecīgi divās daļās.

2.2.1 DSML metamodelis

Šajā nodaļā aprakstīsim valodas metamodeli, kas ļauj uzdot DSML specifikācijas. Šī metamodeļa izklāstu sadalīsim vairākās loģiskās daļās un sāksim ar valodas abstraktās sintakses metamodeli, kas ir redzams 2.14. attēlā.



2.14. att. DSML valodu abstraktās sintakses metamodelis

Šajā metamodelī valodas tiek specificētas ar klasi *GraphDiagramType*, un tās atribūts *caption* norāda valodas nosaukumu. Savukārt, valodas elementu specificēšanai ir paredzēta klase *ElemType*, kuras atribūts *caption* norāda elementa nosaukumu, bet to, kurai valodai elements pieder, norāda kompozīciju *elemType-graphDiagramType*. Klasei *ElemType* ir ieviestas apakšklases *NodeType*, *PortType* un *EdgeType*, kas attiecīgi norāda, ka elementa tips ir kaste, ports vai līnija. Klases *NodeType* asociācija *containerType-componentType* norāda, ka viena tipa kastes var ievietot cita tipa kastē. Atribūts *isContainerMandatory* norāda, vai šī tipa kastei vienmēr ir jāatrodas ievietotai kādā citā kastē. Asociācija *portType-nodeType* (savieno klases *PortType* un *NodeType*) norāda, ka portam vienmēr ir jābūt piesaistītam noteikta tipa kastei.

Katra līnija savieno divus elementus, un tādēļ, lai definētu līniju, mums ir jānorāda, no kura elementa līnijai ir jāiziet un kurā jāieiet, un šo informāciju metamodelī norāda klases *EdgeType* asociācijas *start-eStart* un *end-eEnd*. Savukārt, atribūti *startMultiplicityConstraint*

un *endMultiplicityConstraint* norāda līnijas sākuma un beigu kardinalitātes, proti, šīs kardinalitātes sāk darboties, kad tām ir norādīta skaitliskā vērtība (pretējā gadījumā maksimālā skaita ierobežojumi nepastāv), un to uzdevums ir norādīt attiecīgi maksimālo izejošo līniju skaitu no sākuma elementa un maksimālo ieejošo līniju skaitu beigu elementā.

Ar klases *ElemType* atribūtu *multiplicityConstraint* var norādīt maksimālo pieļaujamo elementu skaitu vienā diagrammā, un, līdzīgi kā ar līnijas kardinalitātēm, tas sāk darboties, kad ir norādīta skaitliskā vērtība.

Ar asociāciju *subType-superType* klasei *ElemType* var norādīt, ka kāds elements ir virstips kādam citam elementam, tādējādi ļaujot veidot elementu tipu hierarhiju, kurā apakštipi manto virstipu ierobežojumus. Piemēram, tipu metamodelis nosaka, ka katrai līnijai ir tieši viens sākuma un tieši viens beigu elements, respektīvi, ar šo konstrukciju var pateikt, ka „X" tipa līnijas savieno „A" tipa elementus ar „B" tipa elementiem, bet, ja mēs gribam, lai „X" tipa līnijas varētu savienot arī „A" tipa elementus ar „C" tipa elementiem, tad ir jāizveido jauns elementa tips „D" un jānorāda, pirmkārt, ka „X" tipa līnijas savieno „A" tipa elementus ar „D" tipa elementiem, un, otrkārt, ka „B" un „C" tipa elementi ir jāpadara par „D" tipa elementa apakštipiem (tas nozīmē, ka „B" un „C" mantos iespēju no „D" būt par „X" tipa līnijas galapunktu). Tā rezultātā tipu hierarhija ļauj norādīt, ka viena tipa līnijām ir iespējami vairāki sākuma un beigu elementu tipi. Tādējādi šī konstrukcija ļauj instanču līmenī simulēt UML klašu diagrammās izmantoto vispārināšanas attiecības ideju, kur apakšklases manto virsklases īpašības.

Jāpiezīmē, ka katram elementa tipam var būt ne tikai patvaļīgs skaits apakštipu, bet arī patvaļīgs skaits virstipu, kas atbilst UML klašu diagrammās izmantotajai daudzkāršās mantošanas [29] idejai, ļaujot norādīt, ka viens elementa tips var mantot vairāku virstipu ierobežojumus.

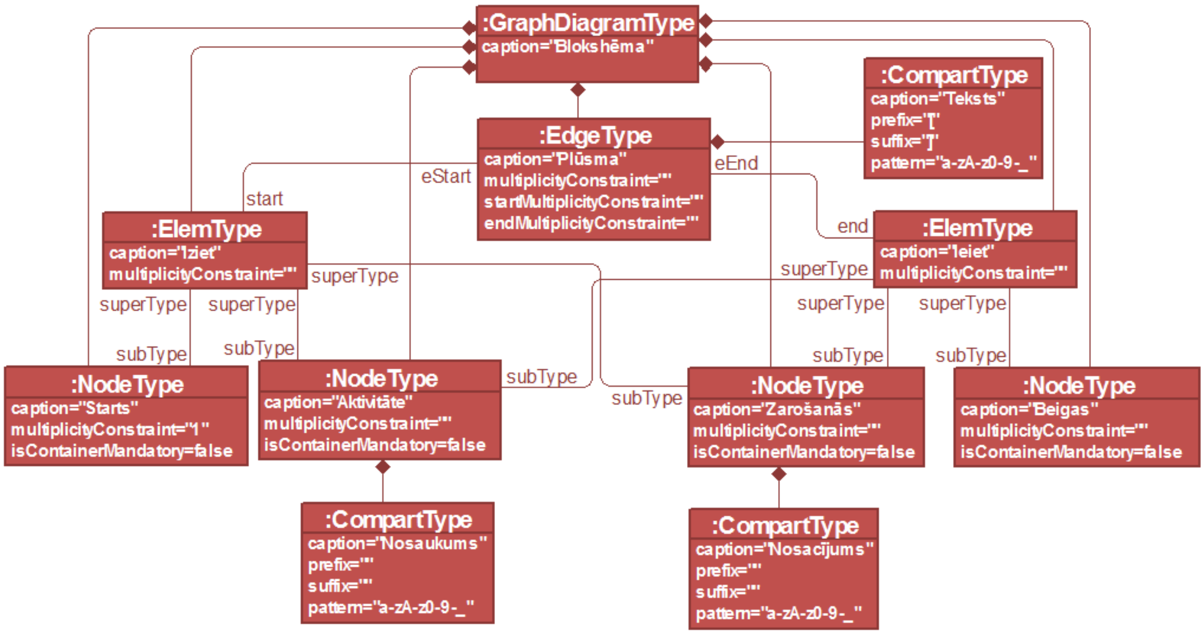
Šis metamodelis paredz, ka ar klases *ElemType* tiešajām instancēm tiek definēti abstrakti elementi, jeb precīzāk sakot, tie ir tādi elementi, kuriem nav norādīts elementa tips (respektīvi, kaste, ports vai līnija). Tādējādi šādus elementus diagrammās nevar izveidot, bet tos var izmantot, lai veidotu elementu hierarhiju, kas ir līdzīgi kā tas ir ar abstraktajām klasēm UML klašu diagrammās, kuras izmanto klašu hierarhiju veidošanai, bet tām nav tiešo instanču (ir to apakšklasēm).

Metamodelī ir iestrādāta iespēja caur elementu detalizēt diagrammas. Tas tiek panākts ar asociāciju *seed-decomposition* starp *ElemType* un *GraphDiagramType*, kas norāda, ka viena tipa elementi detalizē noteikta tipa diagrammas.

Klase *CompartType* ļauj specificēt elementu atribūtus, tās atribūts *caption* norāda atribūta nosaukumu, bet to, kuram elementam tie pieder, norāda kompozīciju *compartType-elemType*.

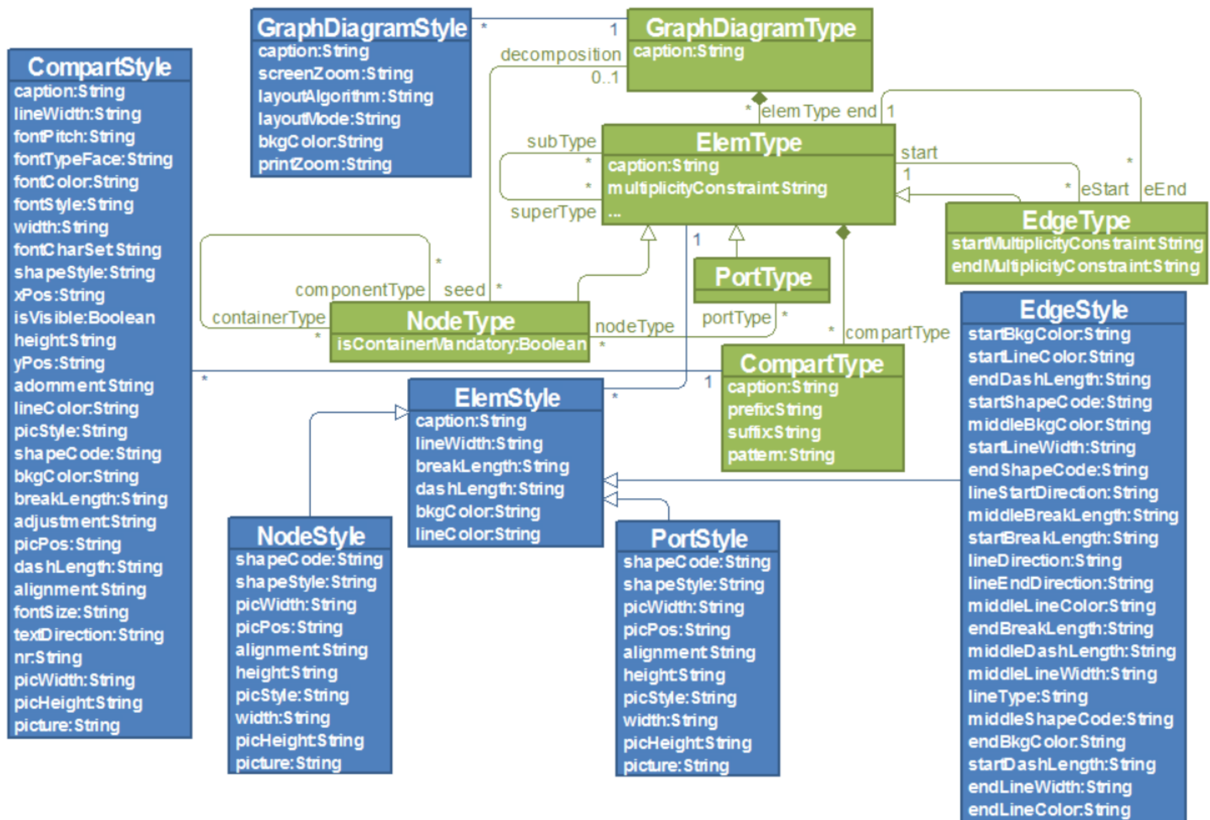
Atribūti *prefix* un *suffix* norāda atribūta prefiksu un sufiksu, piemēram, ja prefikss ir „<<” un sufikss ir „>>”, tad atribūta vērtība ir formā - „<<šī ir atribūta vērtība>>”. Savukārt atribūts *pattern* ļauj norādīt atribūtu vērtībās pieļaujamos simbolus. Šīs vērtības ir jānorāda līdzīgi kā regulārās izteiksmes, piemēram, lai norādītu, ka atribūtā ir pieļaujami mazie burti, lielie burti, cipari, atribūtam *pattern* ir jāpiešķir „a-zA-Z0-9”, vai, ja gribam pieļaut atribūtu vērtībās ciparus, domuzīmes, apakšsvītras un burtu „a”, tad atribūtam *pattern* ir jāpiešķir „0-9-_a”.

Lai nodemonstrētu šī metamodeļa lietojumu, 2.15. attēlā ir parādīts, kā ar šo metamodeli var definēt blokshēmu valodas abstrakto sintaksi.



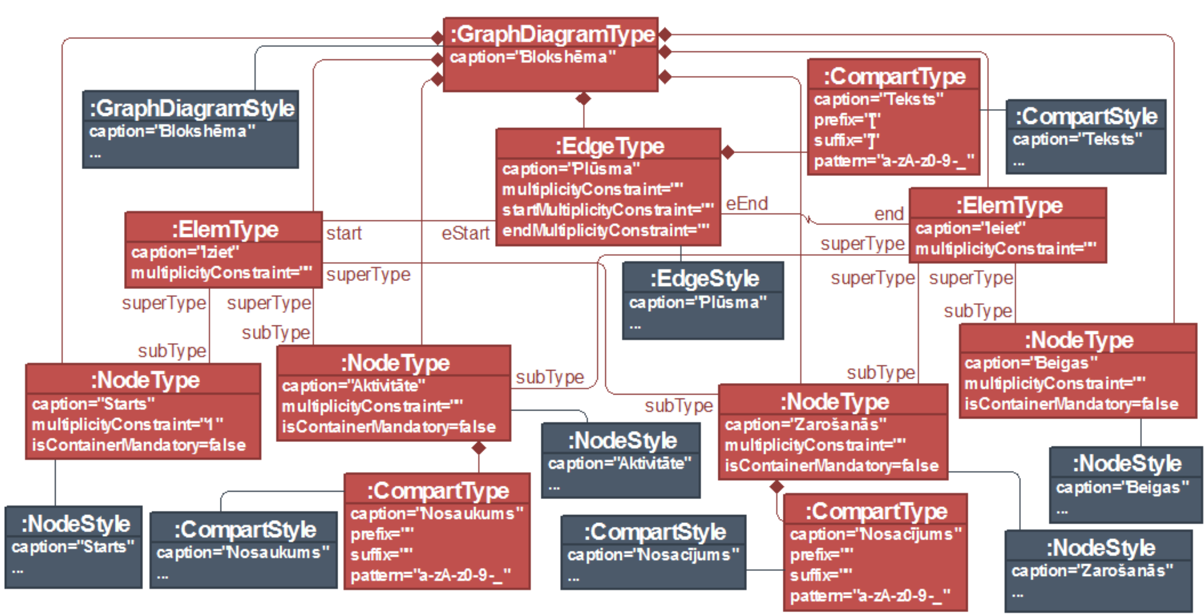
2.15. att. Blokshēmu valodas abstraktās sintakses specifikācija

Iepriekš aprakstījām, kā var specificēt valodas abstrakto sintaksi, bet, lai varētu specificēt valodas konkrēto sintaksi, ir jānorāda stili. Lai to izdarītu, abstraktās sintakses metamodelis ir papildināts ar stila klasēm. Klase *GraphDiagramStyle* norāda diagrammas stilu, klase *ElemStyle* un tās apakšklases *NodeStyle*, *PortStyle* un *EdgeStyle* norāda stilu attiecīgi kastēm, portiem un līnijām, bet *CompartStyle* norāda atribūtu stilus. Pilns valodas metamodelis ir redzams 2.16. attēlā.



2.16. att. DSML valodu metamodelis

Blokshēmu valodas konkrētās sintakses instanču diagramma ir redzama 2.17. attēlā.



2.17. att. Blokshēmu valodas specifikācija

2.2.2 DSML rīku definēšanas metamodelis

Lai specificētu DSML rīku, valodas specifikācija ir jāpapildina ar rīka uzvedības specifikāciju. Tas nozīmē, ka ir jāpapildina valodas metamodelis, un, lai šie papildinājumi

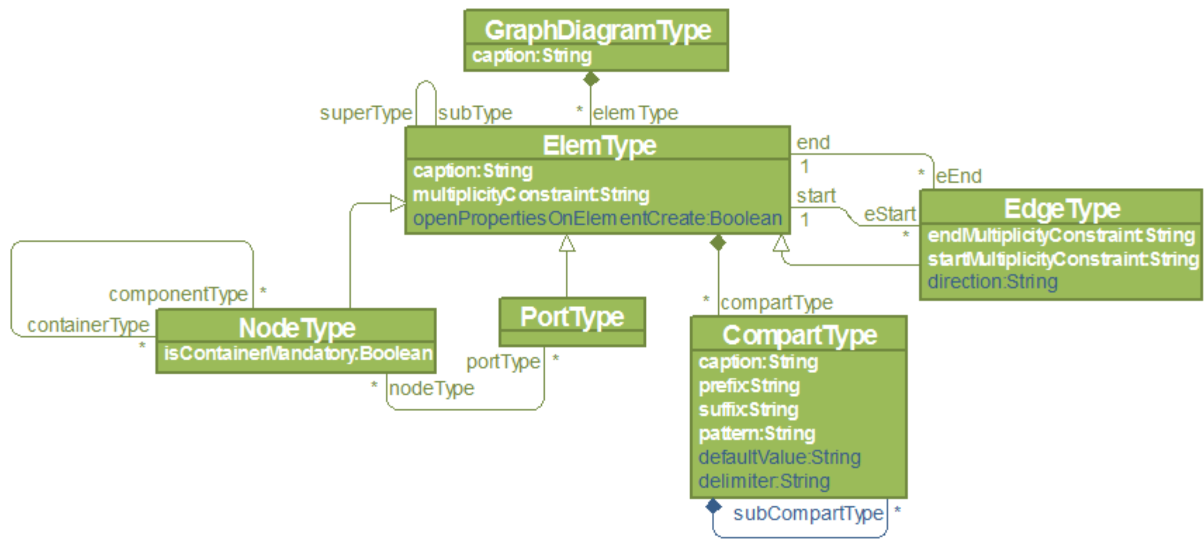
būtu vieglāk saprotami, to izklāsts ir sadalīts pa fragmentiem, bet pilnais DSML rīku definēšanas metamodelis ir redzams 2.26. attēlā.

2.2.2.1 Valodas metamodeļa papildinājumi

Pirmajā solī valodas metamodelis ir papildināts ar četriem jauniem atribūtiem un vienu kompozīciju (skat. 2.18. attēlu). Klasei *ElemType* ir pievienots jauns atribūts *openPropertiesOnElementCreate*, kas norāda, vai jauna elementa veidošanas laikā ir jāatver dialogu logs. Klasei *EdgeType* ir pievienots jauns atribūts *direction*, kas norāda līnijas orientāciju un tam ir iespējamas trīs vērtības – „UniDirectional", „BiDirectional" un „ReverseBiDirectional". Noklusētā vērtība ir „UniDirectional" un tā norāda, ka līnija ir orientēta, t.i., līniju var novilkt no viena elementa uz otru, bet ne otrādāk. Vērtība „BiDirectional" norāda, ka līnija nav orientēta, t.i., līniju var vilkt no viena elementa uz otru un otrādāk. Vērtība „ReverseBiDirectional" norāda, ka līniju var vilkt abos virzienos, bet tās orientācija būs vienmēr nemainīga neatkarīgi no vilkšanas virziena, piemēram, ja ar bultu tiktu savienoti elementi A un B, tad bulta ietu no A uz B, bet, ja bultu vilktu no B uz A, tad bulta tiktu „pagriezta" pretējā virzienā un rezultātā tā ietu no A uz B.

Klasei *CompartType* ir pievienoti atribūti *defaultValue* un *delimiter*, kā arī kompozīcija *compartType-subCompartType*. Atribūts *defaultValue* norāda atribūta noklusēto vērtību, kas automātiski ir jāuzstāda uzreiz pēc atribūta izveidošanas. Kompozīcija *compartType-subCompartType* norāda, ka atribūtam var būt apakšatribūti „patvaļīgā dziļumā", kas veido atribūtu koku. Šāds koks tiek veidots, lai lietotājam, kā vienu veselu, varētu rādīt atribūtu vērtību, kas ir iegūta, saliekot kopā apakšatribūtu vērtības. Savukārt, atribūts *delimiter* kalpo kā atdalītājs starp apakšatribūtiem, kad no apakšatribūtu vērtībām tiek būvēta virsatribūta vērtība.

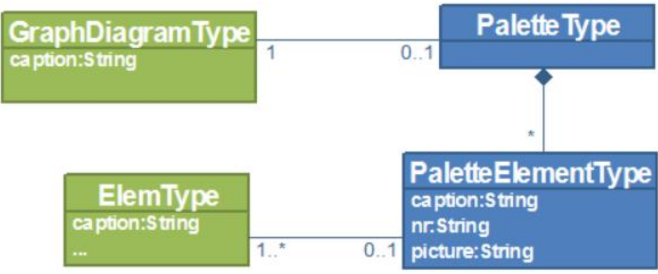
Vispārīgā gadījumā šīs iespējas tiek kombinētas. Gadījumā, ja ir vairāki apakšatribūti, tad tos atdala ar atdalītāju, un, ja kādam apakšatribūtam ir norādīts prefikss vai sufikss, tad šīs vērtības arī tiek uzstādītas. Piemēram, ja ir dots atribūts, kuram ir divi apakšatribūti un pirmajam apakšatribūtam prefikss ir „<<" un sufikss ir „>>", un atribūta atdalītājs ir „:", tad atribūta vērtība būs formā – „<<apakšatribūts1>>:apakšatribūts2", bet katra apakšatribūtu vērtības būs attiecīgi „<<apakšatribūts1>>" un „apakšatribūts2".



2.18. att. Papildināts valodas metamodelis

2.2.2.2 Palete

Jaunu elementu veidošana ir paredzēta ar paletes pogām, un tādēļ metamodelī ir ieviestas jaunas klases *PaletteType* un *PaletteElementType* (skat. 2.19. attēlu), kuras attiecīgi specificē paleti un tās pogas. To, kuram diagrammas tipam pieder palete, norāda asociācija *graphDiagramType-paletteType*, bet to, kuru elementu veidot ar paletes pogu norāda *elemType-paletteElementType*.

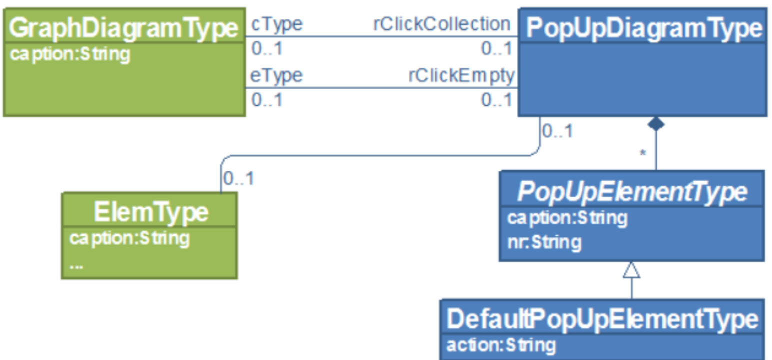


2.19. att. Paletes metamodeļa fragments

Klasei *PaletteElementType* ir atribūti – *caption*, *nr* un *picture*. Atribūts *caption* norāda paletes pogas nosaukumu, *nr* norāda tās kārtas numuru uz paletes un *picture* norāda ikonas adresi.

2.2.2.3 Uznirstošās izvēlnes

Lai realizētu uznirstošās izvēlnes, kuras parādās, kad lietotājs nospiež peles labo taustiņu, metamodelī ir ieviestas jaunas klases un asociācijas, kā tas ir redzams 2.20. attēlā.



2.20. att. Uznirstošo izvēlņu metamodeļa fragments

Izvēlnei metamodelī atbilst klase *PopUpDiagramType*, bet izvēlnes elementiem klase *PopUpElementType*, kuras atribūts *caption* norāda elementa nosaukumu, bet *nr* elementa kārtas numuru uz izvēlnes. Lai piedāvātu noklusētos izvēlnes elementus, metamodelī ir ieviesta klase *DefaultPopUpElementType*, kuras atribūts *action* norāda elementa darbību, un pieļaujamo darbību saraksts ir dots 2.1. tabulā.

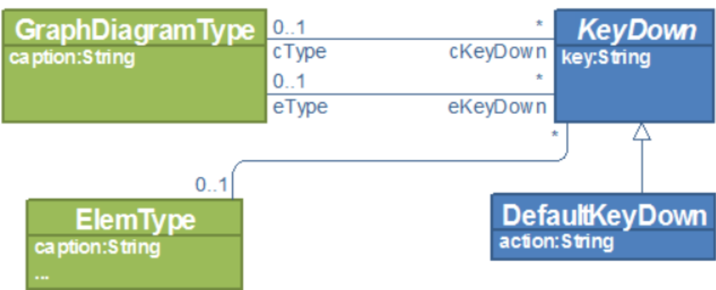
***2.1. tabula.* Noklusētās izvēļņu elementu darbības**

Darbība	Paskaidrojums
Cut	Realizē viena vai vairāku elementu izgriešanu (atkarībā no tā, vai elementu kolekcijā ir viens vai vairāki elementi).
Copy	Realizē viena vai vairāku elementu kopēšanu (atkarībā no tā, vai elementu kolekcijā ir viens vai vairāki elementi).
Paste	Realizē nokopēto elementu ielīmēšanu.
Delete	Realizē viena vai vairāku elementu dzēšanu (atkarībā no tā, vai elementu kolekcijā ir viens vai vairāki elementi).
Properties	Realizē elementa dialogu loga atvēršanu.
SymbolStyle	Realizē elementa stila nomaiņu.
Reroute	Realizē līnijas trajektorijas pārrēķināšanu.
Detail	Realizē detalizācijas diagrammas pievienošanu.

Lai parādītu uznirstošo izvēlni, atkarībā no konteksta tiek šķiroti trīs gadījumi. Viens gadījums ir, kad lietotājs ir nospiedis uz kāda elementa, un tādā gadījumā, lai uzbūvētu izvēlni, tiek izmantota asociācija *popUpDiagramType-elemType*. Otrs gadījums ir, kad lietotājs ir nospiedis uz tukšas vietas diagrammā, un tādā gadījumā ir jāizmanto asociācija *rClickEmpty-eType*. Trešais gadījums ir, kad lietotājs ir nospiedis uz elementu kolekcijas, un tādā gadījumā transformācija izmanto *rClickCollection-cType*.

2.2.2.4 Taustiņu kombinācijas

Lai realizētu lietotāju nospiesto taustiņu kombināciju apstrādi, metamodelī ir ieviestas jaunas klases un asociācijas, kā tas ir redzams 2.21. attēlā.



2.21. att. Taustiņu kombināciju metamodeļa fragments

Klase *KeyDown* un tās atribūts *key* specificē nospiesto taustiņu kombināciju. Analoģiski uznirstošajām izvēlnēm, lai piedāvātu noklusētās taustiņu kombināciju darbības, metamodelī ir ieviesta klase *DefaultKeyDown* un tās atribūts *action* norāda izpildāmo darbību un pieļaujamo darbību saraksts ir dots 2.2. tabulā.

***2.2. tabula.* Noklusētās taustiņu darbības**

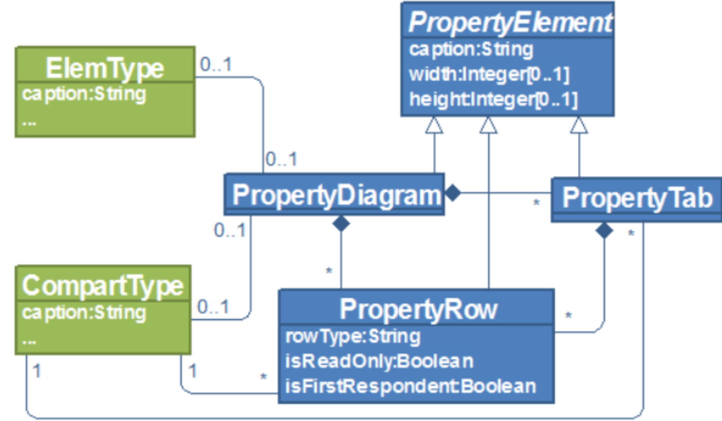
Taustiņu kombinācija	Darbība	Paskaidrojums
Ctrl X	Cut	Realizē viena vai vairāku elementu izgriešanu (atkarībā no tā, vai elementu kolekcijā ir viens vai vairāki elementi).
Ctrl C	Copy	Realizē viena vai vairāku elementu kopēšanu (atkarībā no tā, vai elementu kolekcijā ir viens vai vairāki elementi).
Ctrl V	Paste	Realizē nokopēto elementu ielīmēšanu.
Delete	Delete	Realizē viena vai vairāku elementu dzēšanu (atkarībā no tā, vai elementu kolekcijā ir viens vai vairāki elementi).
Enter	Properties	Realizē elementa dialogu loga atvēršanu.

Tāpat kā uznirstošās izvēlnes gadījumā, atkarība no konteksta tiek izšķirti trīs gadījumi, kad taustiņu kombinācija tika nospiesta – uz elementa, tukšas vietas diagrammā vai elementu kolekcijas. Ja taustiņu kombinācija tika nospiesta uz elementa, tad izmanto asociāciju

keyDown-elemType. Ja uz tukšas vietas, tad izmanto *eKeyDown-eType*, bet, ja uz kolekcijas, tad *cKeyDown-cType*.

2.2.2.5 Dialogu logi

Elementu atribūtu vērtību ievadīšana ir paredzēta caur dialogu logiem. Lai šos dialogu logus varētu uzbūvēt ģenēriski, metamodelis ir papildināts ar jaunām klasēm un asociācijām, kā tas ir redzams 2.22. attēlā.



2.22. att. Metamodeļa fragments dialogu logu specifikācijai

Klase *PropertyElement* ir abstrakta klase, kas norāda, ka visām dialogu logu komponentēm ir atribūts *caption*, lai norādītu komponentes nosaukumu, un atribūti *height* un *width*, lai norādītu komponentes minimālo augstumu un platumu.

Dialogu formām atbilst klase *PropertyDiagram*, kas sastāv no rindām vai cilnēm (metamodelī atbilst klases *PropertyRow* un *PropertyTab*). Katra dialogu logu rinda (*PropertyRow*) tiek attēlota kā rindas nosaukums un ievadlauks. Rindas nosaukumam atbilst *caption* vērtība (atribūts mantots no klases *PropertyElement*), bet ievadlauka tipu nosaka *rowType* vērtība (iespējamie ievadlauku veidi un to izskati ir aprakstīti 2.3. tabulā). Atribūts *isFirstRespondent* norāda to ievadlauku, uz kura jābūt uzstādītam fokusam formas atvēršanas brīdī (iespējams uzstādīt tikai vienam laukam), bet atribūts *isReadOnly* norāda, vai šīs rindas ievadlauks ir vai nav rediģējams.

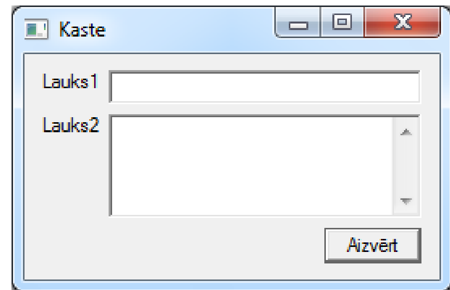
***2.3. tabula.* Ievadlauku veidi**

Lauka tips	Izskats	Paskaidrojums
„InputField”		Vienkāršs teksta lauks.
„ComboBox”		Izkrītošās izvēlnes lauks.

„CheckBox”		Ķekša kastes lauks.
„TextArea”		Liels teksta lauks.
„Label”	Label	Uzraksts.
„InputField+Button”		Vienkāršs teksta lauks ar pogu paredzēts, lai ievadītu saliktas atribūtu vērtības. Poga paredzēta, lai atvērtu papildus formu apakšatribūtu vērtību ievadei.
„TextArea+Button”		Teksta lauks ar pogu. Katra teksta lauka rinda atbilst saliktai apakšatribūtu vērtībai. Ar pogu tiek atvērta papildus forma apakšatribūtu ievadei.

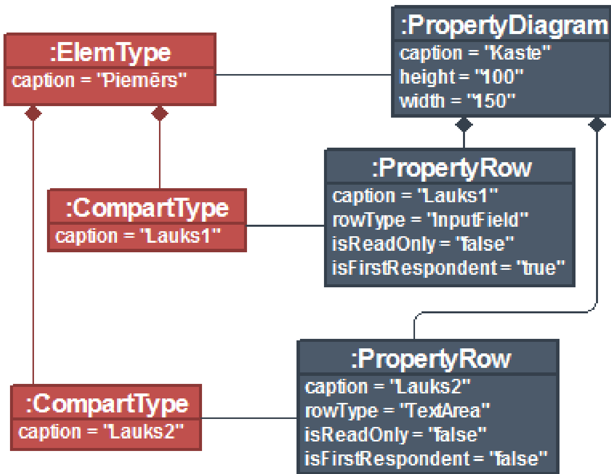
Klase *PropertyTab* ļauj uz dialogu loga formas būvēt cilnes, tādējādi rindas var atrasties tieši uz formas, ko norāda asociācija *propertyRow-propertyDiagram*, vai uz cilnes, ko norāda asociācija *propertyRow-propertyTab*.

Dialogu logu metamodeļa klases ar elementu un atribūtu specificējošajām klasēm ir sasaistīts caur asociācijām *compartType-propertyRow*, *compartType-propertyDiagram* un *elemType-propertyDiagram*. Asociācija *elemType-propertyDiagram* norāda elementa tipam atbilstošo dialogu logu formu, asociācija *compartType-propertyRow* norāda atribūtam piesaistīto dialogu logu rindu, bet ar *compartType-propertyDiagram* norāda atribūtam piesaistīto papildus dialogu logu formu apakšatribūtu vērtību ievadei.



2.23. att. Ģenerētā dialogu logu forma

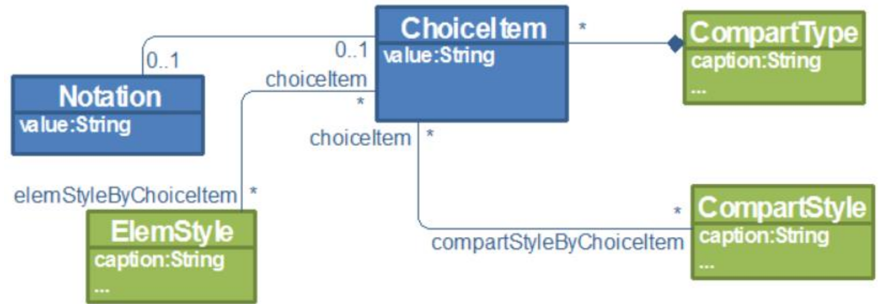
Kā piemēru aplūkosim 2.23. attēlā redzamo formu, kuras atbilstošā instanču diagramma ir redzama 2.24. attēlā. Formas nosaukums ir „Kaste” un tā sastāv no divām rindām ar šādiem ievadlaukiem - „InputField” un „TextArea”. Jāpiezīmē, ka uz formas ir arī poga „Aizvērt”, kā arī pogas formas minimizēšanai (Minimize), izmēru atjaunošanai (Restore Down) un aizvēršanai (Close), kuras metamodelī netiek attēlotas, jo ģenēriski veidotajām formām tās tiek pievienotas vienmēr.



2.24. att. Dialogu loga instanču diagramma

2.2.2.6 Atribūtu loģiskās un reprezentatīvās vērtības

Dažās situācijās vērtības, kuras ievada caur dialogu logu atšķiras no tām, kas tiek rādītas lietotājiem diagrammās. Piemēram, ja dialogu logā atribūta vērtību ievada caur ķekša kasti, tad ir iespējamas divas vērtības – „true” vai „false”. Ja diagrammā rādīs kādu no šīm vērtībām, tad lietotājam būs grūti saprast, kura tipa atribūtam šī vērtība atbilst un ko tā apraksta. Lai lietotājiem diagrammās varētu rādīt atribūtu reprezentatīvās vērtības, bet dialogu logos lietotāju ievadītās (piemēram, „true” vai „false”), metamodelis ir papildināts ar jaunām klasēm un asociācijām, kā tas ir redzams 2.25. attēlā.



2.25. att. Atribūtu reprezentatīvo vērtību un stilu metamodeļa fragments

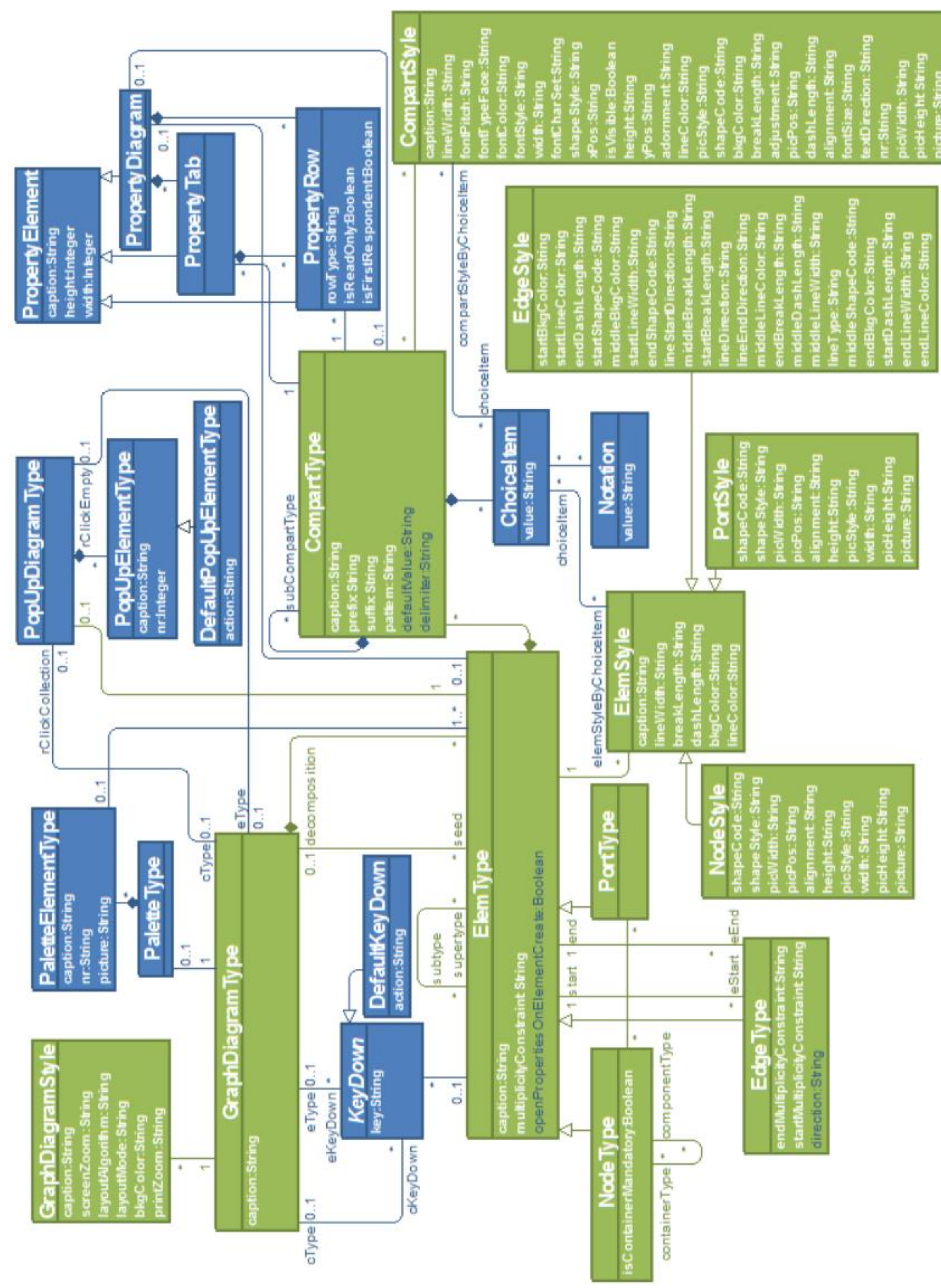
Klase *ChoiceItem* norāda vienu iespējamo atribūta vērtību dialogu logā, bet klase *Notation* norāda tā reprezentatīvo nosaukumu un abas šīs klases saista asociācija *choiceItem-notation*. Ar kompozīciju *compartType-choiceItem* norāda, kuram atribūta tipam vērtības ir piesaistītas.

Klases *ChoiceItem* vērtības tiek izmantotas arī priekš izkrītošo izvēļņu sarakstu specificēšanas, respektīvi, ja formas ievada lauks ir izkrītošā izvēlne, tad katra *ChoiceItem* instance tiek attēlota kā viens elements izkrītošās izvēlnes sarakstā.

Tomēr ir situācijas, kad ar reprezentatīvo vērtību uzstādīšanu nepietiek, un ir vajadzība mainīt atribūta vai pat elementa stilu. Piemēram, ja gribam, lai pie noteiktām vērtībām, atribūtu tekstu vai pašu elementu rāda citā krāsā. Metamodelī šī iespēja tiek realizēta ar asociācijām *choiceItem-compartStyleByChoiceItem* un *choiceItem-elemStyleByChoiceItem*. Tātad, kad lietotājs dialoga logā ievada kādu vērtību un, ja tā atbilst kādai no *ChoiceItem* vērtībām, tad kā atribūta vērtība ir jāuzstāda atbilstošā *Notation* vērtība un, ja ir norādīti stili, tad ir jāuzstāda arī *ChoiceItem* instancei piesaistītos atribūta un elementa stilus.

2.2.2.7 Pilnais DMSL rīku metamodelis

Pilnais DSML rīku metamodelis ir redzams 2.26. attēlā, un tas ir iegūts apvienojot valodu metamodeli un rīku uzvedību aprakstošās klases vienā metamodelī.



2.26. att. DSML rīku definēšana metamodelis

2.2.3 Blokshēmu redaktora specificēšana ar tipu instancēm

Lai nodemonstrētu DSML rīku definēšanas metamodeļa lietojumu, specificēsim iepriekš apskatīto blokshēmu redaktoru. Lai iegūtu blokshēmu redaktora specifikāciju, valodas specifikācija (skat. 2.17. attēlu) ir papildināta ar redaktora uzvedības specifikāciju, kā tas ir redzams 2.27. attēlā.

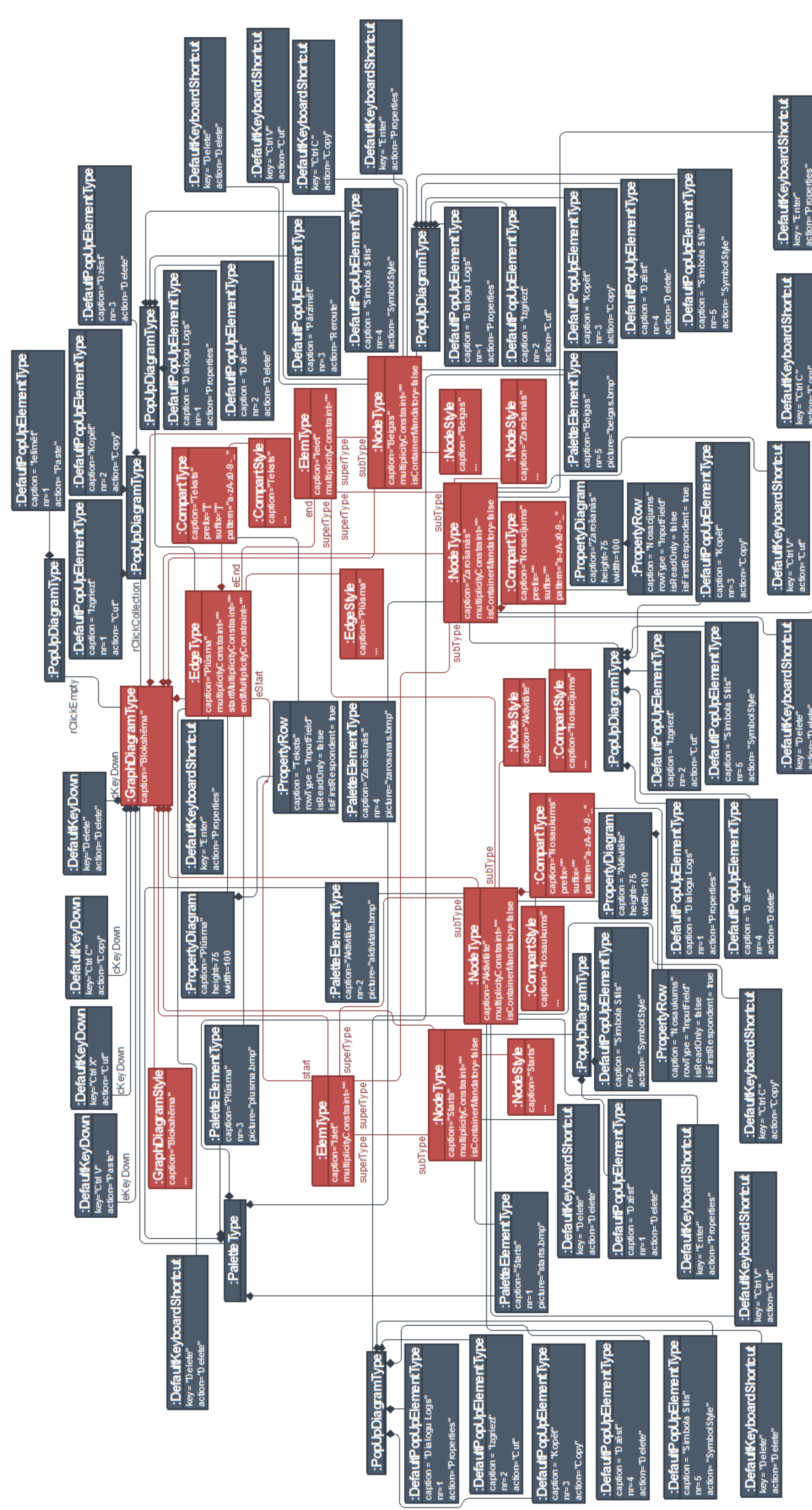
Šī specifikācija paredz, ka blokshēmu diagrammās, nospiežot peles labo taustiņu uz vairāku elementu kolekcijas, tiek piedāvāta uznirstošā izvēlne ar trīs operācijām – izgriezt, kopēt un dzēst, bet, ja peles labo taustiņu nospiež uz tukšas vietas diagrammā, tad tiek piedāvāta izvēlne ar vienu operāciju – ielīmēt. Taustiņu kombināciju apstrāde ir specificēta līdzīgi. Vairāku elementu kolekcijai ir specificētas trīs taustiņu kombinācijas – „Ctrl X", kas atbilst izgriešanas operācijai, „Ctrl C", kas atbilst kopēšanas operācijai, un „Delete", kas atbilst dzēšanas operācijai, bet tukšai vietai diagrammā ir specificēta viena taustiņu kombinācija - „Ctrl V", kas atbilst ielīmēšanas operācijai. Savukārt, lai varētu pievienot jaunus elementus, šī specifikācija paredz, ka diagrammās ir palete, ar kuras elementiem var izveidot jaunus diagrammas elementus.

Valodā ir specificēti šādi kastes tipa elementi – „Starts", „Aktivitāte", „Zarošanās" un „Beigas". Tā kā elements „Starts" diagrammā var būt tikai viens, un tam nav neviena atribūta, tad tam ir specificēta tikai uznirstošā izvēlne ar divām operācijām - dzēst un nomainīt stilu un viena taustiņu kombinācija – „Delete", kas atbilst dzēšanas operācijai.

Elementiem „Aktivitāte" un „Zarošanās" specificētās uznirstošās izvēlnes un taustiņu kombinācijas piedāvā plašāku funkcionalitāti. Uznirstošās izvēlnes šiem elementiem sastāv no piecām operācijām – atvērt dialogu logu, izgriezt, kopēt, dzēst un nomainīt stilu, un četrām taustiņu kombinācijām – „Delete", kas atbilst dzēšanas operācijai, „Ctrl X", kas atbilst izgriešanas operācijai, „Enter", kas atbilst dialogu loga atvēršanas operācijai un „Ctrl C", kas atbilst kopēšanas operācijai. Katram no šiem elementiem ir viens atribūts, un attiecīgi to vērtību ievadīšanai katram elementam ir specificēts dialogu logs ar vienu rindu.

Elementa „Beigas" specificētā funkcionalitāte ir līdzīga elementiem „Aktivitāte" un „Zarošanās", bet, tā kā šim elementam nav atribūtu, tad šim elementa nav nepieciešams specificēt dialogu logu, kā arī izvēlnes operāciju atvērti dialogu logu un taustiņu kombināciju „Enter" neeksistējošā dialogu loga atvēršanai.

Vienīgais līnijas tipa elements blokshēmu valodā ir „Plūsma". Šim elementam ir viens atribūts un tā vērtību ievadīšanai ir specificēts dialogu logs ar vienu rindu. Savukārt tā uznirstošā izvēlne sastāv no četrām operācijām - dzēst, atvērt dialogu logu, pārzīmēt līniju un nomainīt stilu, bet specificētās taustiņu kombinācijas ir „Delete", kas atbilst dzēšanas operācijai, un „Enter", kas atbilst dialogu loga atvēršanas operācijai.



2.27. att. Blokshēmu redaktora specifikācija

2.3 Metožu salīdzinājums

Salīdzinot abas metodes, var redzēt, ka no rīku specificēšanas viedokļa, abas šīs metodes ir līdzvērtīgas, to galvenā atšķirība ir tā, ka specializācijas metode rīkus definē ar klasēm, bet konkretizācijas metode ar instancēm. Tomēr šajā apstāklī ir viena būtiska nianse, proti, tas, ka specializācijas metode rīkus definē ar klasēm, nozīmē to, ka klašu instances atbilst konkrētiem rīka objektiem, piemēram, blokshēmu redaktorā stereotipa klases *Aktivitāte* instances atbilst konkrētiem diagrammas elementiem, un tādējādi specializācijas metode apvieno divus vienā – tā kalpo ne tikai par rīka specifikāciju, bet arī par rīka datu struktūru, kas apraksta rīka izpildes laika datus.

Šī nianse ir būtiska no rīku versiju pārvaldīšana viedokļa, jo tipiska DSML rīku izstrāde ir iteratīva, kas nozīmē, ka tiek izstrādāta viena rīka versija, pēc tam tā tiek uzlabota un iegūta otra versija, tad trešā, ceturtā, utt. līdz iegūst apmierinošu risinājumu. Šādas izstrādes problēma ir tā, ka katrā solī ar viena rīka versiju tiek izveidoti modeļi, kurus nepieciešams lietot arī katrā nākamajā rīka versijā. Tas nozīmē, ka platformā ir nepieciešams mehānisms, kas ļauj modeļu datus, kas izstrādāti ar vecāku rīka versiju, migrēt uz jaunākām rīka versijām.

Migrēšanas mehānisma realizāciju tiešā veidā ietekmē tas, kuru no piedāvātajām metodēm mēs izvēlamies par realizācijas variantu, un tādēļ salīdzināsim šīs metodes migrēšanas iespēju aspektā. Lai to nodemonstrētu, apskatīsim vienkāršotu piemēru un pieņemsim, ka esam realizējuši iepriekš specificēto blokshēmu redaktoru, t.i., šī ir rīka pirmā versija, bet rīka otrajā versijā elementu „Aktivitāte" esam aizstājuši ar „Activity". Tādā gadījumā ar specializācijas metodi izstrādātajos metamodeļos stereotipu klase *Aktivitāte* ir jāaizstāj ar klasi *Activity*, kas nozīmē, ka, pirmkārt, modelī ir jāizveido jauna klase *Activity*, otrkārt, klasei *Activity* ir jāpievieno tieši tās pašas asociācijas, kas bija klasei *Aktivitāte* (tas nebūtu grūti, jo profils specificē visas iespējamās asociācijas) un, treškārt, klases *Aktivitāte* instances (tās, kas raksturo rīka izpildes laika stāvokli) ir jāpadara par *Activity* instancēm. Tā kā tradicionālās transformāciju valodas nepiedāvā iespēju „pa tiešo" nomainīt instancēm to klasi, tad šajā gadījumā vienīgais veids, kā to izdarīt, ir izveidot šo instanču kopijas ar jaunās klases tipu, turklāt pārvelkot arī visas oriģinālu saites.

Savukārt ar konkretizācijas metodi izstrādātajā rīku definēšanas metamodelī elementu „Aktivitāte" specificē klases *NodeType* instance, un, lai šo *NodeType* instanci aizstātu ar tādu, kas specificē jauno elementu „Activity", mums atliek izveidot vienu *NodeType* instanci un uz šo jauno instanci pārsaistīt saites no elementa „Aktivitāte" atbilstošās *NodeType* instances.

Tādējādi modeļu datu migrēšana starp dažādām rīka versijām, ar konkretizācijas metodi ir vienkāršāka, kam par iemeslu ir tas, ka rīku definēšanas metamodelis apraksta tikai rīka

specifikāciju, bet ne rīka izpildes laika datus (par to, kā izpildes laika dati saistās ar specifikāciju ir aprakstīts 3. nodaļā).

2.4 Secinājumi

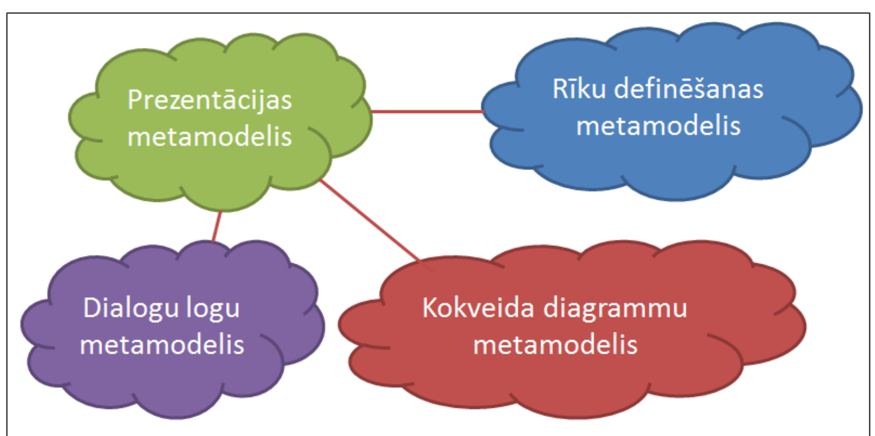
Šajā nodaļā ir risināta DSML rīku specificēšanas problēma un ir piedāvātas divas alternatīvas metodes - specializācijas un konkretizācijas metodes. Lai izvēlētos realizācijas variantu, abas metodes tika salīdzinātas, un tika secināts, ka no rīku specificēšanas viedokļa, abas šīs metodes ir līdzvērtīgas, un to galvenā atšķirība ir tā, ka specializācijas metode rīkus definē ar klasēm, bet konkretizācijas metode ar instancēm. Savukārt apstāklis, ka konkretizācijas metode (pretēji specializācijas metodei) apraksta tikai rīka specifikāciju, bet ne rīka izpildes laika datus, ļauj vieglāk realizēt modeļu migrēšanu starp dažādām rīka versijām (detalizētāk 4.3. nodaļā), kas ir galvenais arguments, lai par realizācijas variantu izvēlētos konkretizācijas metodi.

3 Realizācija

Iepriekš aprakstītais rīku definēšanas metamodelis ir uzbūvēts tā, lai visa informācija, kas nepieciešama konkrēta DSML rīka specificēšanai, būtu iegūstama kā rīku definēšana metamodeļa instance (ja neskaita platformas ietvara funkcionalitāti, kas ir kopīga visiem rīkiem). Tas nozīmē, ka balstoties uz šo informāciju, var izstrādāt interpretatoru (jeb universālu modeļu transformāciju kopumu), kas dotu rīka specifikāciju pārvērš par strādājošu rīku, un tādējādi ļauj rīku izstrādi reducēt uz rīku definēšanas metamodeļa instanču izveidošanu.

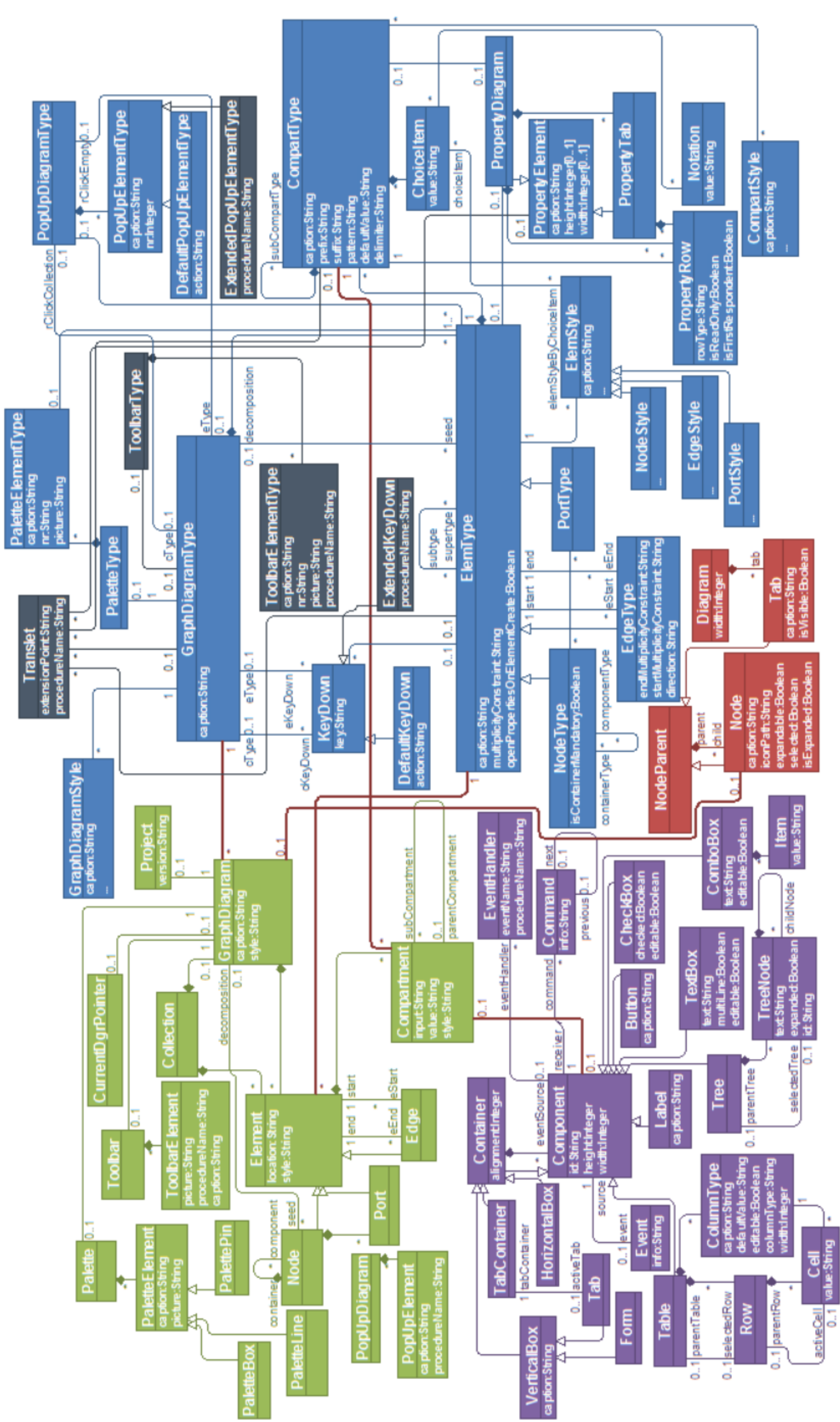
3.1 Realizācijas metamodelis

Pirms mēs apskatām, kas būtu jādara, lai izveidotu interpretatoru, mums ir nepieciešams rīku definēšanas metamodeli integrēt ar platformā esošo dziņu metamodeļiem, t.i., ar prezentācijas, dialogu logu un kokveida diagrammu metamodeļiem. Visi šie metamodeļi kopā papildināti ar sastatņu asociācijām veido realizācijas metamodeli, kura shematisks attēlojums ir redzams 3.1. attēlā.



3.1. att. Shematisks realizācijas metamodeļa attēlojums

Pilnais realizācijas metamodelis ir redzams 3.2. attēlā (metamodelī tā apjoma dēļ nav attēlotas notikumu un komandu klases, bet klašu *Translet*, *ExtendedKeyDown*, *ExtendedPopUpElement*, *ToolbarType* un *ToolbarElementType* semantika būs izskaidrota 3.3. nodaļā). Šajā metamodelī ir ieviestas piecas jaunas sastatņu asociācijas. Asociācijas *graphDiagram-graphDiagramType*, *element-elemType* un *compartment-compartType* ļauj norādīt kāds ir katras diagrammas, elementa un atribūta tips. Asociācija *component-compartment* ļauj norādīt katram dialogu formas ievadlaukam tā atbilstošo atribūtu. Savukārt asociācija *node-graphDiagram* katram kokveida diagrammas elementam tā atbilstošo diagrammu.



3.2. att. Realizācijas metamodelis

3.2 Interpretators

Darba 1. nodaļā tika minēts, ka konkrēta rīka realizēšana nozīmē izstrādāt rīkam specifisku transformāciju komplektu. Savukārt, ja mums ir dota rīka specifikācija, tas mums ļauj priekš katra platformas notikuma izstrādāt vienu universālu transformāciju un tādējādi rīkiem specifiskās transformācijas aizstāt ar universālajām.

Lai mēs varētu šādi rīkoties, mums ir jāņem katra dziņa notikumu saraksts, un priekš katra notikuma jāizstrādā viena universāla transformācija. Tā kā platformā ir trīs dziņi, tad tieši šo notikumu saraksti mums ir jāaplūko, un precīzi jāspecificē, kā veikt katra notikuma apstrādi.

Prezentācijas dziņa notikumu saraksts un to apstrādājošo transformāciju specifikācijas ir aprakstītas 3.1. tabulā.

3.1. tabula. **Prezentācijas dziņa notikumi un transformāciju specifikācijas**

Notikums	Transformācijas specifikācija
NewBox	Transformācijai vispirms ir jāpārbauda metamodelī norādītie kastes ierobežojumi, un, ja tie ir apmierināti, tad jāizveido jauna kaste ar tās atribūtiem (klase *Compartment*) un to sākotnējām vērtībām (ja tādas ir norādītas). Pēc tam, ja nepieciešams, tiek atvērts dialogu logs kastes atribūtu vērtību ievadīšanai.
NewLine	Transformācijai vispirms ir jāpārbauda metamodelī norādītie līnijas ierobežojumi, un, ja tie ir apmierināti, tad ir jāizveido jauna līnija ar tās atribūtiem un to sākotnējām vērtībām (ja tādas ir norādītas). Pēc tam, ja nepieciešams, tiek atvērts dialogu logs līnijas atribūtu vērtību ievadīšanai.
NewPin	Transformācijai vispirms ir jāpārbauda metamodelī norādītie porta ierobežojumi, un, ja tie ir apmierināti, tad ir jāizveido jauns ports ar tā atribūtiem un to sākotnējām vērtībām (ja tādas ir norādītas). Pēc tam, ja nepieciešams, tiek atvērts dialogu logs porta atribūtu vērtību ievadīšanai.
MoveLineStartPoint	Transformācijai vispirms ir jāpārbauda, vai drīkst pārvietot līnijas sākuma galu, un, ja drīkst, tad jāpārvieto.
MoveLineEndPoint	Transformācijai vispirms ir jāpārbauda, vai drīkst pārvietot līnijas beigu galu, un, ja drīkst, tad jāpārvieto.
ChangeParent	Lai kādu kasti ievietotu citā kastē, transformācijai vispirms ir

	jāpārbauda, vai doto kasti drīkst ievietot norādītajā kastē. Ja drīkst, tad ir jāpārvieto, ja nedrīkst, tad ir jāizdod paziņojums.
LClick	Universālā transformācija nereaģē uz šo notikumu.
L2Click	Transformācijai ir jāatver dialogu logs, vai arī, ja elements ir detalizēts ar diagrammu, tad jātver šī diagramma.
RClick	Transformācijai atbilstoši metamodelim ir jāuzbūvē uznirstošā izvēlne.
ToolbarElementSelect	Transformācijai ir jāizpilda izvēlētā rīkjoslas elementa transformācija.
PopUpElementSelect	Transformācijai ir jāizpilda izvēlētā uznirstošās izvēlnes elementa transformācija.
KeyDown	Transformācijai atbilstoši metamodelim ir jāizpilda taustiņu kombinācijai atbilstošā darbība.
ActivateDgr	Universālā transformācija nereaģē uz šo notikumu.
CloseDgr	Universālā transformācija nereaģē uz šo notikumu.

Dialogu logu dziņa notikumu saraksts un to transformāciju specifikācijas ir aprakstītas 3.2. tabulā. Jāpiezīmē, ka dialogu dzinis ļauj notikumus apstrādāt divos veidos. Pirmais veids paredz apstrādi veikt atbilstoši iepriekš minētajam universālajam scenārijam, kad katram notikumu tipam ir piekārtota viena universāla transformācija, kas veic šī notikuma apstrādi. Otrs veids ļauj katrai komponentei norādīt, ka noteikta tipa notikumus attiecībā uz šo komponenti vajag apstrādāt nevis universāli, bet gan specifiski. Lai to izdarītu, dialogu logu metamodelī ir klase *EventHandler*, un, lai komponentei piekārtotu specifisko transformāciju, ir jāizveido klases *EventHandler* instance, kuras atribūtā *eventName* ir jānorāda notikuma nosaukums, bet *procedureName* ir jānorāda transformācijas nosaukums, kura veiks konkrētā notikuma apstrādi. Tādējādi, kad iestāsies konkrētā tipa notikums, tad dialogu dzinis vispirms pārbaudīs, vai komponentei ir piesaistīta attiecīgā klases *EventHandler* instance. Ja šāda instance ir, tad notikuma apstrādi veiks specifiskā transformācija, bet, ja nav, tad notikuma apstrādi veiks universālā transformācija.

3.2. tabula. **Dialogu dziņa notikumi un transformāciju specifikācijas**

Notikums	Transformācijas specifikācija
FocusLost	Transformācijai ir atribūtam jāuzstāda dialogu loga laukā ievadītā vērtība.

Change	Universālā transformācija nereaģē uz šo notikumu.
DropDown	Transformācijai ir jāizveido izkrītošās izvēlnes elementi.
Click	Transformācijai ir jāaizver dialogu logs, ja nospieda aizvēršanas pogu, vai arī jāatver papildus dialoga logs, ja nospieda lauka detalizēšanas pogu.
FocusGained	Universālā transformācija nereaģē uz šo notikumu.
TabChange	Universālā transformācija nereaģē uz šo notikumu.
TreeNodeSelect	Universālā transformācija nereaģē uz šo notikumu.
TreeNodeExpanded	Universālā transformācija nereaģē uz šo notikumu.
TreeNodeCollapsed	Universālā transformācija nereaģē uz šo notikumu.

Kokveida diagrammu dziņa notikumu saraksts un to transformāciju specifikācijas ir aprakstītas 3.3. tabulā.

3.3. tabula. **Kokveida diagrammu dziņa notikumi un transformāciju specifikācijas**

Nosaukums	Paskaidrojums
Select	Universālā transformācija nereaģē uz šo notikumu.
KeyDown	Universālā transformācija nereaģē uz šo notikumu.
RightClick	Universālā transformācija nereaģē uz šo notikumu.
Colapse	Universālā transformācija nereaģē uz šo notikumu.
Expand	Universālā transformācija nereaģē uz šo notikumu.
DoubleClick	Transformācijai ir jāatver koka virsotnei atbilstošā diagramma.

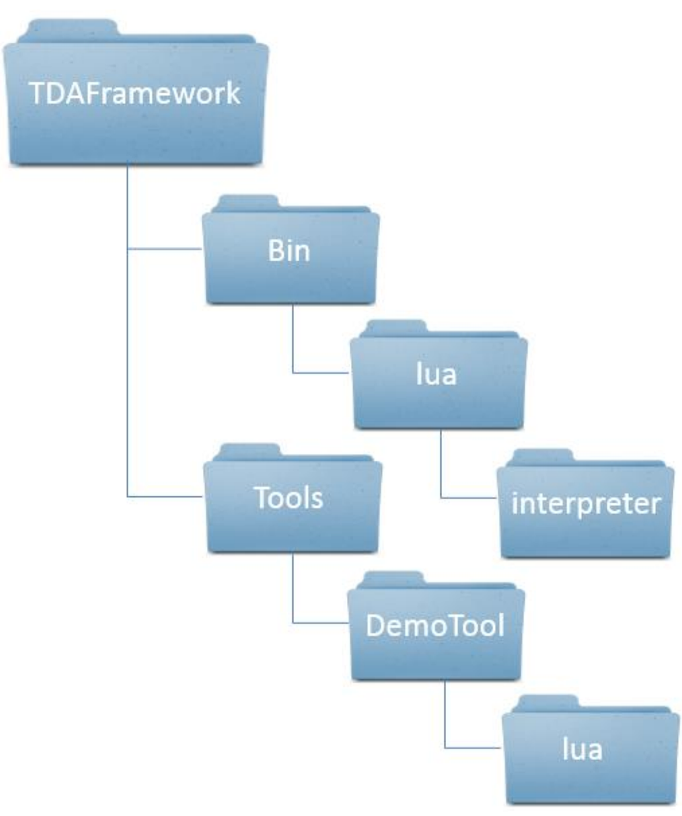
Platformā bez dziņu notikumiem, ir realizētas arī priekšdefinētas darbības un šo darbību realizējošo transformāciju specifikācijas ir aprakstītas 3.4. tabulā.

3.4. tabula. **Priekšdefinētas darbības un transformāciju specifikācijas**

Nosaukums	Paskaidrojums
Copy	Transformācijai ir jāuzģenerē kods, kas ļauj izveidot iezīmēto elementu klonus.
Paste	Transformācijai ir jāizpilda kopēšanas laikā uzģenerētais kods.
Delete	Transformācijai ir jāizdzēš iezīmētie elementi.
Cut	Transformācijai ir secīgi jāizpilda iezīmēto elementu kopēšana un dzēšana.

Properties	Transformācijai atbilstoši metamodelim ir jāuzbūvē dialogu logs.
SymbolStyle	Transformācijai ir jāatver dialogu logs elementa stila mainīšanai.
Reroute	Transformācijai ir jānodrošina iezīmēto līniju pārzīmēšana.
Detail	Transformācijai atbilstoši metamodelim ar diagrammu jādetalizē elements.

Lai tehniski realizētu interpretatoru, 1.3. nodaļā aprakstītā datņu struktūrā direktorijai *lua* (adrese *../TDAFramework/Bin/lua*) ir izveidota jauna apakšdirektorija *interpreter*, kurā ir ievietotas visas universālo transformāciju datnes, bet priekš transformācijām, kas realizē rīkam specifisko funkcionalitāti (par šīm transformācijām sīkāk 3.3. nodaļā), rīka direktorijā (direktorijā *DemoTool*) ir izveidota vēl viena *lua* apakšdirektorija. Papildinātā datņu struktūra ir redzama 3.3. attēlā.



3.3. att. Papildinātā datņu struktūra

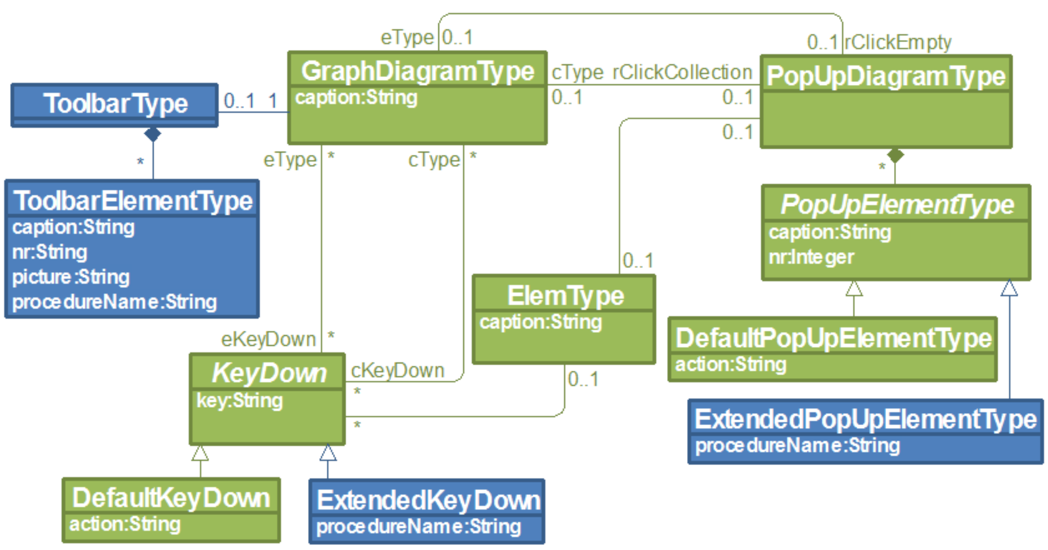
3.3 Paplašinājuma punktu mehānisms

Ne visas DSML rīku būvētāju vēlmes attiecībā uz rīka funkcionalitāti var aprakstīt ar vienu universālu metamodeli un fiksētu interpretatoru. Tādēļ platformā ir realizēts paplašinājuma punktu mehānisms, ar kura palīdzību rīka izstrādātājs var pievienot specifiski rīkam izstrādātas transformācijas, un tādējādi papildināt rīka funkcionalitāti.

Paplašinājuma punkti ir iedalīti divās kategorijās - vienkāršie un saliktie paplašinājuma punkti. Vienkāršie paplašinājuma punkti atbilst konkrētām lietotāju darbībām, piemēram, rīkjoslas pogas nospiešanai, bet saliktie ļauj iejaukties universālo transformāciju izpildes laikā un papildināt to funkcionalitāti.

3.3.1 Vienkāršie paplašinājuma punkti

Lai ieviestu vienkāršos paplašināšanas punktus, rīku definēšanas metamodelis ir papildināts, kā tas ir redzams 3.4. attēlā.

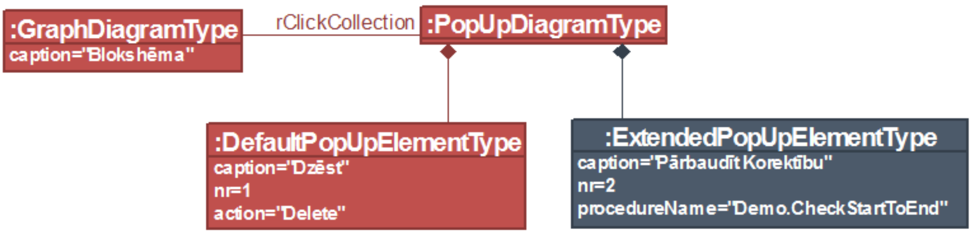


3.4. att. Rīku definēšanas metamodeļa fragments

Klase *ToolbarType* un *ToolbarElementType* specificē rīkjoslu un tās pogas. Klasei *ToolbarElementType* ir atribūti – *caption*, *nr*, *picture* un *procedureName*. Atribūts *caption* norāda rīkjoslas pogas nosaukumu, *nr* norāda tās kārtas numuru uz rīkjoslas, *picture* norāda ikonas adresi, bet *procedureName* transformācijas nosaukumu, kura tiks izpildīta nospiežot rīkjoslas pogu. Šai un citām paplašinājuma punktu transformācijām ir pieejams viss realizācijas metamodelis, kā arī domēna metamodeļi (ja tādi ir), un tā var brīvi mainīt šo metamodeļu saturu. Pie tam, lai atvieglotu specifisko transformāciju izstrādi, ir izveidotas transformāciju bibliotēkas, kas sastāv no transformācijām ar biežāk lietoto funkcionalitāti (skat. 1. pielikumu).

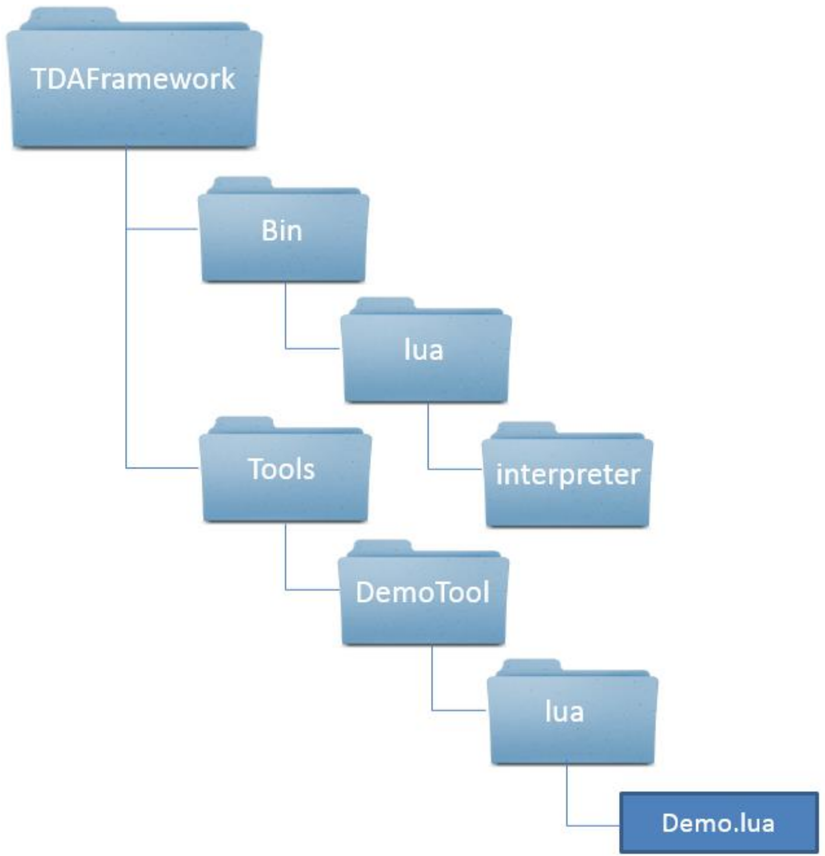
Lai uz taustiņu kombinācijām un ar uznirstošo izvēļņu elementiem varētu specificēt ne tikai priekšdefinētas darbības, bet arī rīkam specifiskas darbības, metamodelim ir pievienotas klases *ExtendedKeyDown* un *ExtendedPopUpElementType*, kuru atribūts *procedureName* ļauj norādīt izpildāmās transformācijas nosaukumu.

Lai nodemonstrētu paplašinājuma punkta lietojumu, papildināsim iepriekš specificētā blokshēmu redaktora (skat. instanču diagrammu 2.25. attēlā) funkcionalitāti ar iespēju pārbaudīt, vai plūsmas, kas iziet no starta simbola, aizved līdz beigu simbolam. Lai to varētu izdarīt, mums ir jāizstrādā transformācija, kas veiks šo pārbaudi, kā arī jāpapildina blokshēmu redaktora specifikācija. Ja pieņemam, ka šo transformāciju, mēs gribam izpildīt caur uznirstošo izvēlni, kas ir iegūta, nospiežot peles labo pogu uz tukšas vietas diagrammā, tad blokshēmu redaktora specifikācijas instanču diagramma ir jāpapildina, kā tas ir redzams 3.5. attēlā.



3.5. att. Bloskhēmu redaktora instanču diagrammas fragments

Šī specifikācija paredz (to norāda 3.5. attēla klases *ExtendedPopUpElementType* atribūta *procedureName* vērtība „Demo.CheckStartToEnd”), ka transformācija, kas veiks pārbaudi saucās „CheckStartToEnd”, un tā atrodas datnē „Demo”, kura ir novietota *lua* apakšdirektorijā, kā tas ir redzams 3.6. attēlā.



3.6. att. Blokshēmu redaktora datņu struktūra

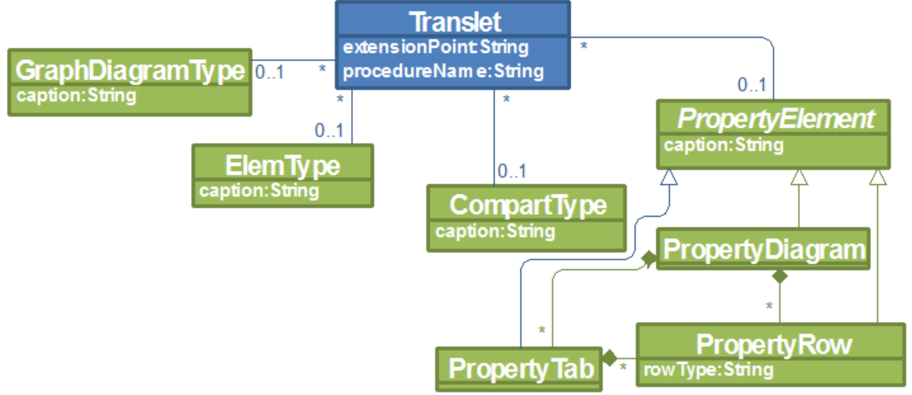
Savukārt transformācijas „CheckStartToEnd” kods ir redzams 3.7. attēlā.

```
function CheckStartToEnd ()
	visited = { } -- apstaigāto virsotņu saraksts, sākumā tukšs

	diagram = utilities.current_diagram() -- atrod aktīvo diagrammu, ar funkcija no funkciju bibliotēkas
	start = diagram:find("/element:has(/elemType[caption = 'Starts'])") -- sameklē starta elementu
	status = true -- diagrammas statuss norāda, vai diagramma ir korekta

	check_pathes(start, visited) -- izpilda funkciju, kas pārbauda blokshēmu

	-- izdrukā uz ekrāna, vai blokshēma ir korekta
	if status then
		print("Blokshēma ir korekta")
	else
		print("Blokshēma nav korekta")
	end
end

function check_pathes(current_node, visited) -- apstaigā blokshēmu, izmantojot Depth-first search algoritmu
	neighbours = current_node:find("/eStart/end")
	if neighbours:is_empty() then
		if not(is_beigas(current_node)) then
			print("in false")
			status = false
			return false
		end
	else
		neighbours:each(function(node)
			if visited[node:id()] ~= true then
				if not(is_beigas(node)) then
					visited[node:id()] = true
					return check_pathes(node, visited)
				end
			end
		end)
	end
end

function is_beigas(node)  --funkcija pārbauda, vai virsotne ir beigu elements
	if node:find(":has(/elemType[caption = 'Beigas'])"):is_empty() then
		return false
	else
		return true
	end
end
```

3.7. att. Transformācijas *CheckStartToEnd* kods

3.3.2 Saliktie paplašinājuma punkti

Salikto paplašinājuma punktu būtība ir ļaut universālo transformāciju izpildes laikā noteiktās situācijās iejaukties ar specifiskām transformācijām un mainīt to uzvedību. Tehniski šie paplašinājuma punkti ir realizēti, kā transformāciju izsaukumi noteiktās vietās universālajās transformācijās. Metamodelī paplašinājuma punkti tiek kodēti ar klasi *Translet* kā pāris, kur atribūts *extensionPoint* norāda paplašinājuma punkta nosaukumu, bet *procedureName* norāda izpildāmās procedūras nosaukumu. Metamodelī asociācijas *translet-graphDiagramType*, *translet-ElemType, translet-compartType* un *translet-propertyElement* ļauj norādīt, ka „translets" ir piekārtots attiecīgi diagrammai, elementam, atribūtam vai dialoga loga komponentei. Papildinātais metamodeļa fragments ir redzams 3.8. attēlā.



3.8. att. Papildināts rīku definēšanas metamodeļa fragments ar klasei *Translet*

3.3.2.1 Paplašinājuma punktu uzskaitījums

Lai varētu precīzi izskaidrot, kuros brīžos katra konkrētā paplašinājuma punkta transformācija tiek izpildīta, ir nepieciešams zināt universālo transformāciju uzbūvi. Lai to izskaidrotu, katrai universālajai transformācijai, kas satur paplašinājuma punktu, ir dota blokshēma, kas precīzi norāda, kādi soļi ir veikti pirms konkrētā paplašinājuma punkta transformācijas izpildes un kādas darbības tiks veiktas pēc tam. Bet pirms apskatām paplašinājumu punktu blokshēmas, dosim visu paplašinājuma punktu uzskaitījumu un to vispārīgu skaidrojumu.

3.5. tabulā ir apkopoti un skaidroti diagrammai piesaistītie paplašinājuma punkti.

3.5. *tabula.* Diagrammas paplašinājuma punkti

Nosaukums	Skaidrojums
DynamicPopUpE	Paplašinājuma punkts iestājas uz peles labās pogas nospiešanas tukšā vietā diagrammā, un tā mērķis ir ļaut dinamiski radīt uznirstošo izvēlni (skat. 3.13. attēlu).
DynamicPopUpC	Paplašinājuma punkts iestājas uz peles labās pogas nospiešanas vairāku elementu kolekcijai, un tā mērķis ir ļaut dinamiski radīt uznirstošo izvēlni (skat. 3.13. attēlu).

Paplašinājuma punkti, kas ir piekārtoti elementiem, ir aprakstīti 3.6. tabulā.

3.6. *tabula.* Elementa paplašinājuma punkti

Nosaukums	Konteksts	Skaidrojums
Precondition	Elements	Paplašinājuma punkts iestājas, kad tiek veidots jauns elements, un tā mērķis ir dinamiski pārbaudīt elementa veidošanas pieļaujamos nosacījumos (skat.

		3.9. attēlu).
Properties	Elements	Paplašinājuma punkts iestājas, kad tiek veidots jauns elements, un tā mērķis ir ļaut dinamiski izveidot dialogu logus (skat. 3.9. attēlu).
DynamicPopUp	Elements	Paplašinājuma punkts iestājas uz peles labās pogas nospiešanas, un tā mērķis ir ļaut dinamiski radīt uznirstošo izvēlni (skat. 3.13. attēlu).
DynamicDoubleClick	Elements	Paplašinājuma punkts iestājas uz peles kreisās pogas dubultklikšķa un tā mērķis ir ļaut dinamiski radīt uznirstošo izvēlni (skat. 3.12. attēlu).
CreateElementDomain	Elements	Paplašinājuma punkts iestājas, kad ir izveidots jauns elements, un tā mērķis ir radīt izveidotā elementa apkārtni (skat. 3.9. attēlu).
DeleteElementDomain	Elements	Paplašinājuma punkts iestājas, kad tiek izpildīta dzēšana, un tā mērķis ir ļaut izdzēst dzēšamā elementa apkārtni (skat. 3.19. attēlu).
CopyDomain	Elements	Paplašinājuma punkts iestājas, kad tiek izpildīta kopēšana, un tā mērķis ir ļaut nokopēt kopējamā elementa apkārtni (skat. 3.18. attēlu).
ContainerChanged	Kaste	Paplašinājuma punkts iestājas, kad mainās kastes vecāks (skat. 3.11. attēlu).
MoveLine	Līnija	Paplašinājuma punkts iestājas, kad tiek pārcelts kāds no līnijas galiem (skat. 3.10. attēlu).

Paplašinājuma punkti, kas ir piekārtoti atribūtiem, ir aprakstīti 3.7. tabulā.

3.7. *tabula.* Atribūta paplašinājuma punkti

Nosaukums	Skaidrojums
GenerateDefaultValue	Paplašinājuma punkts ļauj dinamiski izrēķināt sākotnējo atribūta vērtību (skat. 3.9. attēlu).
GetPrefix	Paplašinājuma punkts ļauj dinamiski izrēķināt atribūta prefiksu (skat. 3.9. attēlu).
GetSuffix	Paplašinājuma punkts ļauj dinamiski izrēķināt atribūta sufiksu (skat. 3.9. attēlu).

ValueChanged	Paplašinājuma punkts ļauj reaģēt uz atribūta vērtību izmaiņām (skat. 3.17. attēlu).
CreateCompartmentDomain	Paplašinājuma punkts ļauj izveidot atribūta apkārtni (skat. 3.9. attēlu).
DeleteCompartmentDomain	Paplašinājuma punkts ļauj izdzēst atribūta apkārtni (skat. 3.19. attēlu).
UpdateCompartmentDomain	Paplašinājuma punkts ļauj mainīt atribūta apkārtni (skat. 3.17. attēlu).

Paplašinājuma punkti, kas ir piekārtoti dialogu logu komponentēm, ir aprakstīti 3.8. tabulā.

3.8. ***tabula.*** **Dialogu logu paplašinājuma punkti**

Notikums	Komponente	Skaidrojums
Open	Forma	Paplašinājuma punkts iestājas, kad dialogu logu forma ir uzbūvēta, bet vēl nav parādīta uz ekrāna (skat. 3.20. attēlu).
Close	Forma	Paplašinājuma punkts iestājas, kad dialogu logs ir aizvērts, bet vēl nav izdzēsti tā objekti no repozitorija (skat. 3.15. attēlu).
Show	Cilne	Paplašinājuma punkts iestājas, kad konkrētā cilne tiek aktivizēta (skat. 3.16. attēlu).
IsReadOnly	Ievadlauks	Paplašinājuma punkts iestājas, kad lauks tiek veidots, un tā mērķis ir izrēķināt lauka rediģējamību vai nerediģējamību (skat. 3.20. attēlu).
CheckEnteredField	Ievadlauks	Paplašinājuma punkts iestājas, kad ir ievadīta lauka vērtība, un tā mērķis ir pārbaudīt, vai ievadītā vērtība ir korekta (skat. 3.17. attēlu).
GenerateItems	Izkrītošā izvēlne	Paplašinājuma punkts ļauj dinamiski izveidot uznirstošās izvēlnes sarakstu (skat. 3.14. attēlu).

3.3.2.2 Paplašināto universālo transformāciju blokshēmas

Lai konkrēti saprastu, kad un kā tiek izpildītas paplašinājuma punktu transformācijas, vajag precīzi zināt transformāciju darbību secību. Tāpēc priekš transformācijām, kas satur paplašinājuma punktus, ir izveidotas blokshēmas.

3.3.2.2.1 Prezentācijas dziņa notikumu transformācijas

3.3.2.2.1.1 NewBox, NewLine un NewPin

Neatkarīgi no elementa tipa, jaunu elementu veidošanas transformācijas ir līdzīgas, atšķirīgas ir katra elementa tipa ierobežojumu pārbaudes, un tāpēc visu trīs transformāciju izklāstu var viegli apvienot. Elementa veidošanas transformācija tiek izpildīta, kad lietotājs no paletes veido jaunu elementu. Prezentācijas dzinis izveido attiecīgā notikuma (attiecīgi *NewBox*, *NewLine* vai *NewPin*) instanci un novelk saiti uz paletes elementu, ar kuru tiek radīts jaunais elements, kā arī atkarībā no elementa tipa, dzinis no notikuma instances novelk vēl papildus saites sekojoši:

- gadījumā, ja jaunam elementam ir kastes tips un jaunā kaste tiek ievietota kādā citā kastē, tad no notikuma instances novelk papildus saiti uz kasti, kurā jauno kasti ievieto;
- gadījumā, ja jaunajam elementam ir līnijas tips, tad no notikuma instances novelk saites uz līnijas sākuma un beigu galu elementiem;
- gadījumā, ja jaunajam elementam ir porta tips, tad no notikuma instances novelk saiti uz kasti, kurai portu pievieno.

Pēc tam neatkarīgi no elementa tipa tiek izpildīta paplašinājuma punkta „Precondition” transformācija. Šis paplašinājuma punkts ļauj dinamiski pārbaudīt, vai tiek izpildīti visi nosacījumi, lai varētu radīt jauno elementu. Ja nosacījumi ir izpildīti, tad „Precondition” transformācija atgriež „true”, ja nav, tad „false”. Gadījumā, ja paplašinājuma punkta transformācija nav norādīta, tiek uzskatīts, ka atgrieztā vērtība ir „true”. Pēc dinamisko ierobežojumu pārbaudes, universālā transformācija pārbauda katra elementa statiskos ierobežojumus.

Ja tiek radīta jauna kaste, transformācija, pirmkārt, pārbauda, vai netiek pārkāpti skaitliskie ierobežojumi (klases *ElemType* atribūts *multiplicityConstraint*). Otrkārt, platformā tiek izšķirts vai kaste tiek veidota diagrammā „pa tiešo”, vai arī tā tiek veidota kādā jau esošā kastē, un šo informāciju norāda notikuma instance, respektīvi, ja lietotājs jauno kasti veido kādā jau esošā kastē, tad no notikuma instances vēl papildus ir novilkta saite uz esošo kasti, bet, ja jauno kasti veido tieši diagrammā, tad saite netiek radīta. Pēc tam, ņemot vērā šo informāciju, transformācija pārbauda jaunās kastes ierobežojumus. Ja kaste tiek veidota jau

kādā esošā kastē, tad pārbauda, vai šī tipa kasti drīkst ievietot norādītajā kastē (asociācija *containerType-componentType*). Bet, ja kaste tiek veidota tieši uz diagrammas, tad transformācija pārbauda, vai šī tipa kastei nav obligāti jābūt ievietotai kādā citā kastē (klases *NodeType* atribūts *isContainerMandatory*).

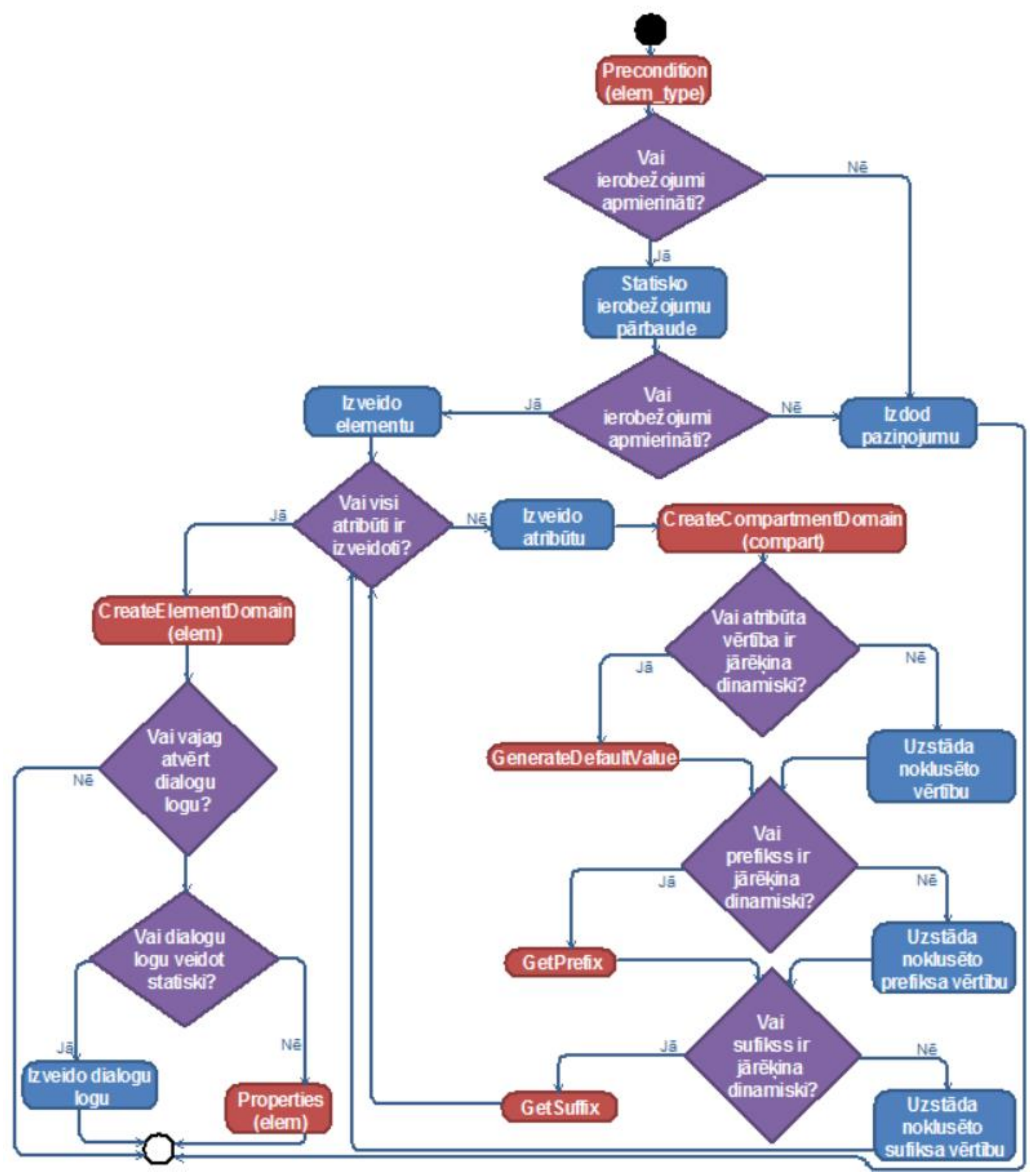
Ja tiek radīta jauna līnija, transformācija, pirmkārt, pārbauda skaitliskos ierobežojumus līniju galiem (klases *EdgeType* atribūti *startMultiplicityConstraint* un *endMultiplicityConstraint*) un pašai līnijai (klases *ElemType* atribūts *multiplicityConstraint*). Otrkārt, pārbauda, vai veidojamā līnija drīkst savienot sākuma un beigu elementus, uz kuriem norāda notikuma instance.

Ja tiek radīts jauns ports, transformācija, pirmkārt, pārbauda, vai netiek pārkāpti skaitliskie ierobežojumi. Otrkārt, pārbauda, vai jauno portu drīkst pievienot kastei, uz kuru norāda notikuma instance.

Ja jaunā elementa ierobežojumi ir apmierināti, tad transformācija izveido jauno elementu. Pēc tam transformācija izveido elementa atribūtus. Elementa atribūti veido koka struktūru, un atribūti tiek radīti secīgi, t.i., tiek radīts atribūts, kas ir atribūtu koka sakne, tad šī atribūta apakšatribūti un tā rekursīvi turpina līdz ir radīts viss atribūtu koks.

Apskatīsim viena atribūta koka veidošanu. Atribūtu koks tiek veidots identiski atbilstošajai atribūtu tipu struktūrai. Veidojot katru atribūtu, transformācija uzstāda tā noklusēto vērtību. Noklusētā vērtība tiek ņemta no klases *CompartType* atribūta *defaultValue*, vai arī tiek izpildīta paplašinājuma punkta „GenerateDefaultValue” transformācija, kas dinamiski izrēķina noklusēto vērtību. Pēc tam tiek uzstādīti atribūta prefikss un sufikss. Prefiksu un sufiksu var uzstādīt statiski ņemot to vērtības no *CompartType* atribūtiem *prefix* un *suffix*, vai arī izpildot paplašinājuma punktu „GetPrefix” un „GetSuffix” transformācijas. Pēc tam universālā transformācija pēc šī paša mehānisma rekursīvi rada atribūta apakšatribūtus līdz ir uzbūvēts atribūtu koks simetriski to atribūtu tipiem.

Pēc tam, kad ir radīti elementa atribūti, tiek izpildīta paplašinājuma punkta „CreateElementDomain” transformācija. Pēc šīs transformācijas izpildes universālā transformācija pārbauda, vai konkrētā tipa elementam ir nepieciešams atvērt dialogu logu formu atribūtu vērtību ievadīšanai (klases *ElemType* atribūts *openPropertiesOnElementCreate*). Ja dialogu logu formu ir nepieciešams atvērt, tad to var izdarīt divos veidos. Ja ir norādīta paplašinājuma punkta „Properties” transformācija, tad dialogu logs tiek uzbūvēts dinamiski, bet, ja nav, tad dialogu logs tiek uzbūvēts no metamodeļa instancēm. Jauna elementa veidošanas blokshēma ir redzama 3.9. attēlā.

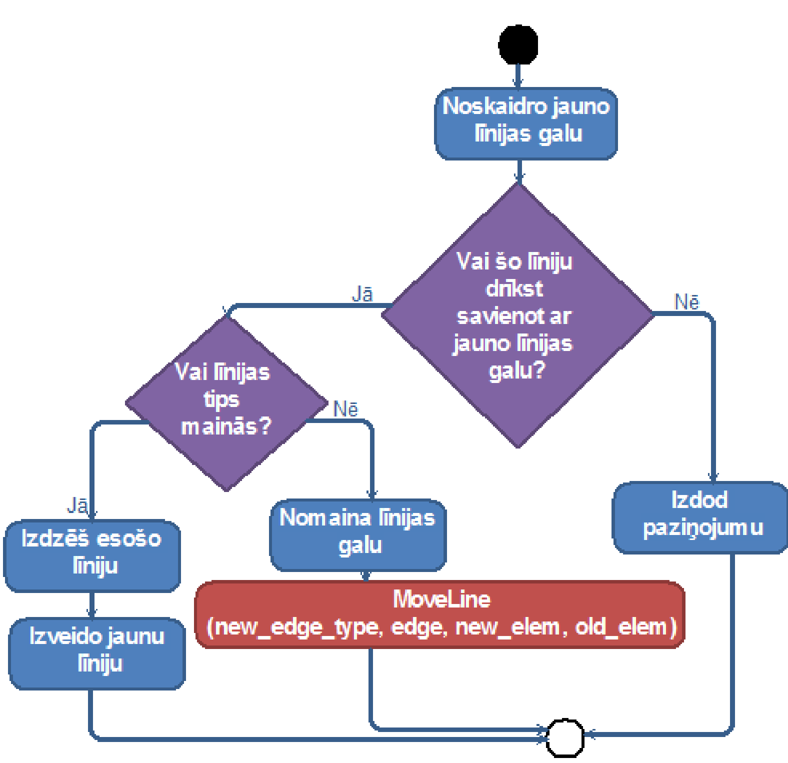


3.9. att. Notikumu *NewBox*, *NewLine* un *NewPort* apstrādes blokshēma

3.3.2.2.1.2 MoveLineStartPoint un MoveLineEndPoint

Notikumu *MoveLineStartPoint* un *MoveLineEndPoint* apstrāde ir līdzīga, tāpēc arī šo transformāciju izklāstu var viegli apvienot. Transformācija izpildās, kad lietotājs pārceļ kādu no līnijas galiem. Izveidojot notikuma instanci, prezentācijas dzinis novelk vēl trīs saites. Vienu saiti novelk uz *Edge* instanci, kuras gals tika pārvietots, otru saiti novelk uz *Element* instanci, ar kuru līnijas gals bija savienots (vecais galapunkts), un trešo saiti novelk uz *Element* instanci, kurai lietotājs līniju vēlas pievienot (jaunais galapunkts). Pēc tam transformācija no galapunktu atbilstošajiem tipiem izsecina, vai šī tipa līniju drīkst savienot ar

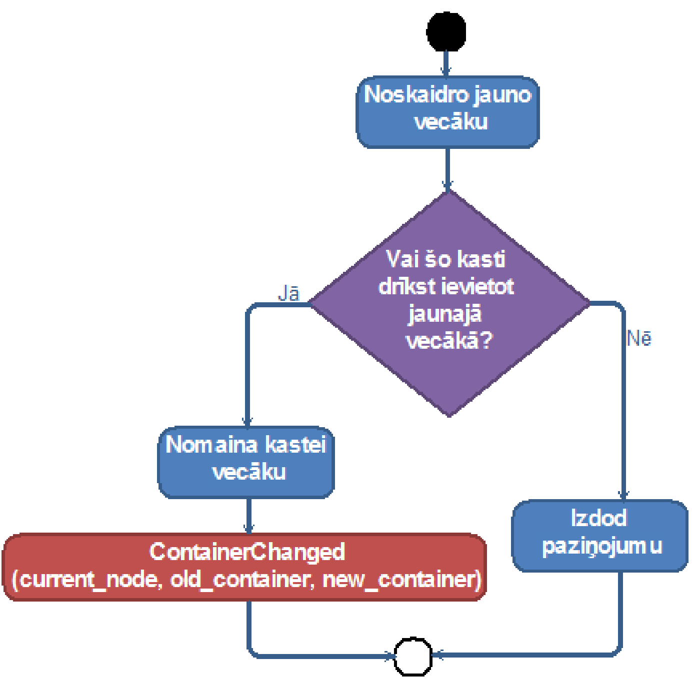
norādīto elementu. Ja drīkst, tad līnijas gals tiek pārcelts. Līnijas pārcelšana var notikt divos veidos. Gadījumā, ja līnijas tips nemainās, tad tiek nomainīts līnijas gals un pēc tam tiek izpildīta paplašinājuma punkta „MoveLine" transformācija. Gadījumā, ja līnijas tips mainās, tad līnija tiek izdzēsta un tiek radīta jauna. Savukārt, ja līniju nedrīkst savienot ar norādīto elementu, tad līnijas gals netiek pārcelts un transformācija izdod paziņojumu. Pirms transformācija beidz darbu, tā izdzēš notikuma instanci. Līnijas gala pārcelšanas transformācijas blokshēma ir redzama 3.10. attēlā.



3.10. att. Notikumu *MoveLineStartPoint* un *MoveLineEndPoint* apstrādes blokshēma

3.3.2.2.1.3 ChangeParent

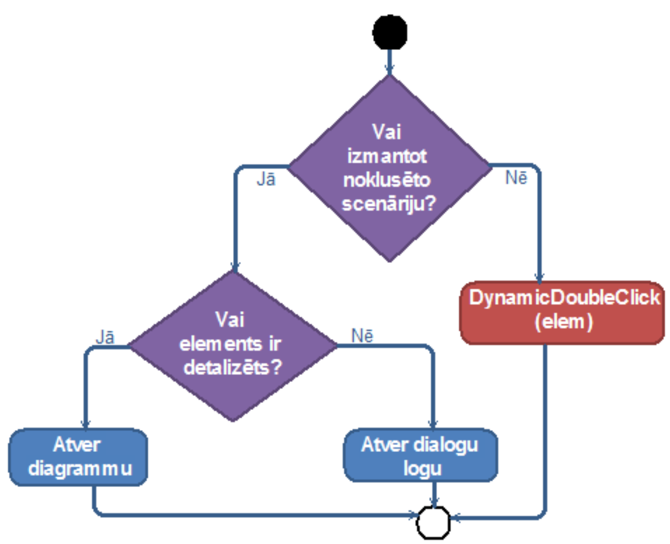
Notikums *ChangeParent* iestājas, kad lietotājs kasti mēģina ievietot kādā citā kastē, vai arī no kādas kastes izņemt. Izveidojot *ChangeParent* instanci, prezentācijas dzinis novelk vienu saiti uz *Node* instanci, kuru lietotājs pārvieto, un otru saiti uz *Node* instanci, kurā lietotājs kasti vēlas ievietot. Gadījumā, ja kaste tiek izņemta no kādas kastes un tiek pārvietota uz diagrammu, tad dzinis otro saiti neveido. Pēc tam transformācija no atbilstošajiem tipiem izsecina, vai pārvietoto kasti drīkst ievietot norādītā tipa kastē. Ja drīkst, tad kaste tiek ievietota norādītajā kastē un tiek izpildīta paplašinājuma punkta „ContainerChanged" transformācija, kas kā parametrus saņem minētās trīs instances. Ja nedrīkst, tad transformācija izdod paziņojumu. Pirms transformācija beidz darbu, tā izdzēš notikuma instanci. Kastes vecāka mainīšanas blokshēma ir redzama 3.11. attēlā.



3.11. att. Notikuma *ChangeParent* apstrādes blokshēma

3.3.2.2.1.4 L2Click

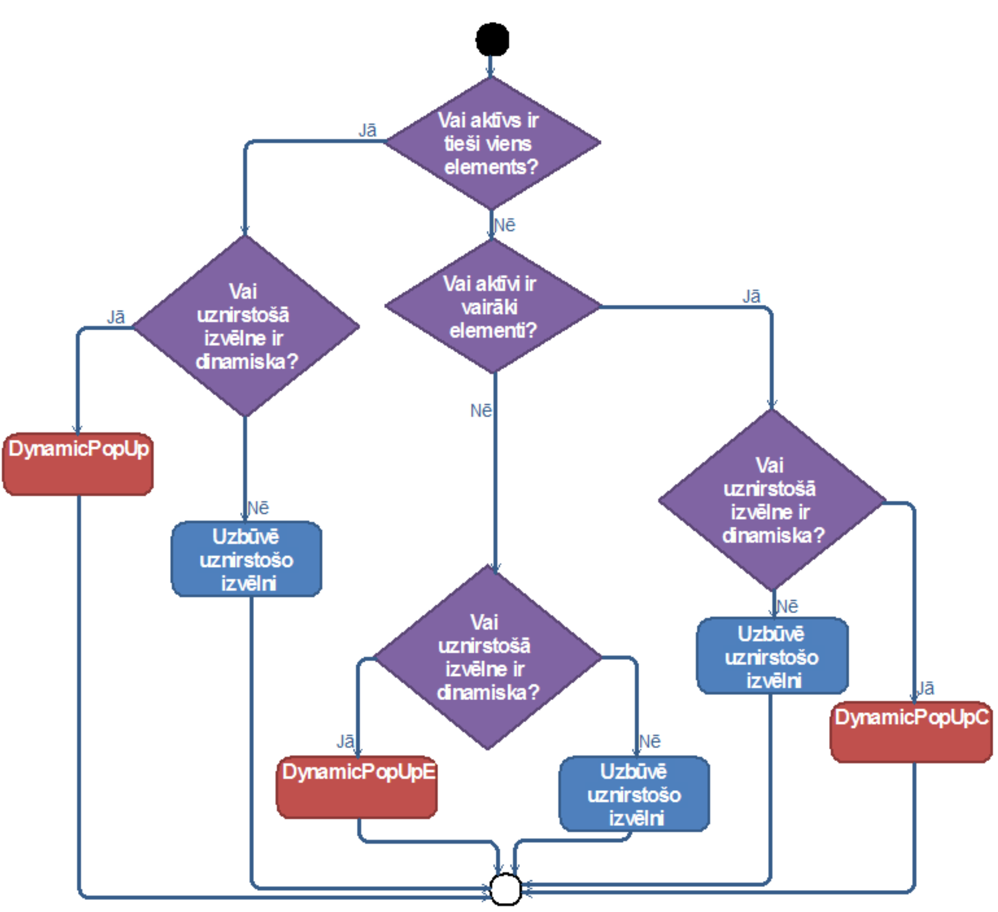
Notikums *L2Click* iestājas, kad lietotājs uz kāda elementa izpilda peles kreisās pogas dubultklikšķi. Izveidojot *L2Click* notikuma instanci, prezentācijas dzinis novelk saiti uz *Element* instanci, uz kuras lietotājs uzklikšķināja. Pēc tam transformācija noskaidro, vai elementam ir piesaistīta paplašinājuma punkta „DynamicDoubleClick" transformācija, un, ja ir, tad izpilda šo transformāciju, bet, ja nav, tad izpilda noklusēto scenāriju, kas paredz, vispirms pārbaudīt, vai elements ir detalizēts. Ja elements ir detalizēts, tad atver tam piesaistīto diagrammu, bet, ja nav, tad no metamodeļa instancēm uzbūvē dialogu logu. Peles kreisās pogas dubultklikšķa apstrādes blokshēma ir redzama 3.12. attēlā.



3.12 att. Notikuma *L2Click* apstrādes blokshēma

3.3.2.2.1.5 RClick

Notikums *RClick* iestājas, kad lietotājs nospiež peles labās pogas taustiņu, un transformācija izšķir trīs gadījumus. Vienā gadījumā peles labā poga var tikt nospiesta uz tieši viena elementa. Tādā gadījumā transformācija no elementa tipa meklē paplašinājuma punkta „DynamicPopUp" transformāciju. Ja šī transformācija ir norādīta, tad tā būvē dinamisku uznirstošo izvēlni. Ja šāda transformācija nav norādīta, tad universālā transformācija uznirstošo izvēlni būvē statiski no tipu instancēm, proti, no elementa tipa sameklē *PopUpDiagramType* instanci un attiecīgi izveido uznirstošo izvēlni ar operācijām, kuras atbilst *PopUpElementType* instancēm. Otrajā gadījumā peles labā poga var tikt nospiesta uz vairāku elementu kolekcijas. Tādā gadījumā transformācija no aktīvās diagrammas tipa meklē paplašinājuma punkta „DynamicPopUpC" transformāciju. Tālākās darbības ir tādas pašas kā pirmajā gadījumā – uznirstošo izvēlni būvē dinamiski ar paplašinājuma punkta transformāciju (ja ir norādīts) vai statiski no tipu instancēm. Trešajā gadījumā peles labā poga var tikt nospiesta uz tukšas vietas diagrammā. Tādā gadījumā no aktīvās diagrammas tipa meklē paplašinājuma punkta „DynamicPopUpE" transformāciju un tālākās darbības ir tādas pašas kā iepriekšējos gadījumos. Peles labās pogas apstrādes blokshēma ir redzama 3.13. attēlā.

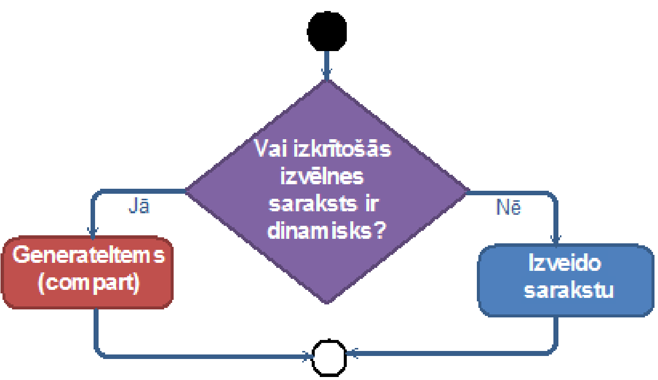


3.13. att. Notikuma *RClick* apstrādes blokshēma

3.3.2.2.2 Dialogu logu dziņa notikumu transformācijas

3.3.2.2.2.1 DropDown

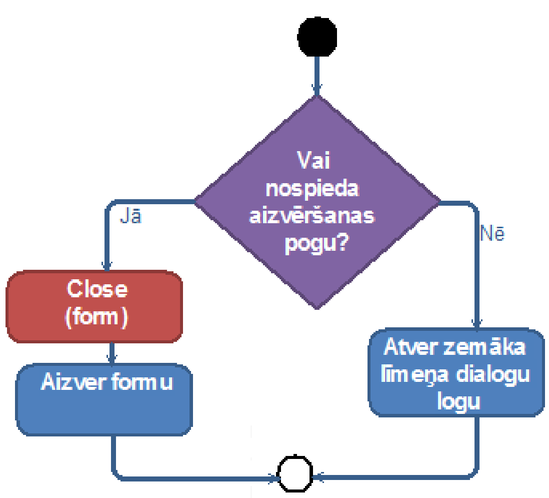
Notikums *DropDown* iestājas, kad *ComboBox* komponentei pieprasa tā saraksta elementus. Transformācija noskaidro, vai elementu saraksts ir jāveido statisks vai dinamisks. Ja ir jāveido dinamisks, tad izpilda paplašinājuma punkta „GenerateItems" transformāciju, bet, ja nav, tad atbilstoši metamodelim statiski veido elementu sarakstu. Transformācijas blokshēma ir redzama 3.14. attēlā.



3.14. att. DropDown blokshēma

3.3.2.2.2.2 Click

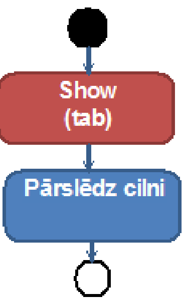
Kad notikums iestājas, tad universālā transformācija izšķir, vai tika nospiests uz aizvēršanas pogas, un tad vispirms izpilda paplašinājuma punkta „Close" transformāciju un pēc tam aizver formu, vai arī, ja nospieda uz lauka detalizācijas pogu, tad atver zemāka līmeņa dialogu logu. Transformācijas blokshēma ir redzama 3.15. attēlā.



3.15. att. Click blokshēma

3.3.2.2.2.3 TabChange

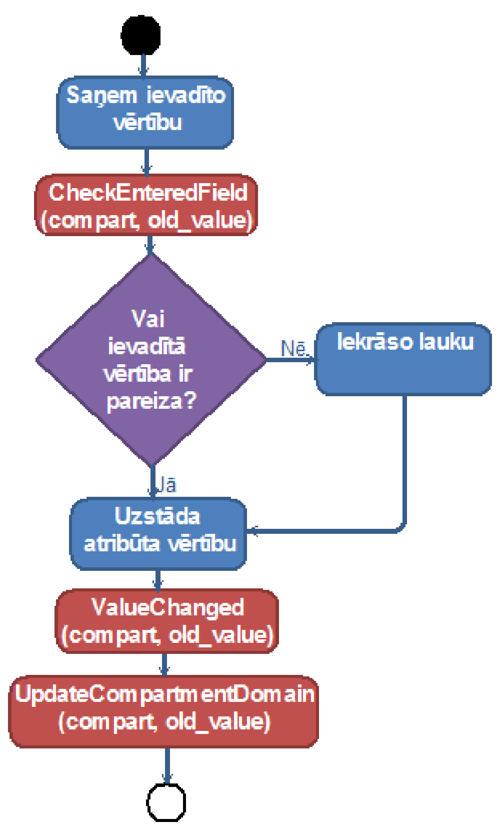
Notikums iestājas, kad lietotājs aktivizēta kādu cilni. Pirms šīs cilnes parādīšanas tiek izpildīta paplašinājuma punkta „Show” transformācija. Transformācijas blokshēma ir redzama 3.16. attēlā.



3.16. att. TabChange blokshēma

3.3.2.2.2.4 FocusLost

Kad notikums iestājas, universālā transformācija fokusu zaudējušā lauka vērtību uzstāda lauka atbilstošajam atribūtam, un pēc tam izpilda paplašinājuma punkta „CheckEnteredField” transformāciju, kas pārbauda ievadītās vērtības korektību. Vērtību korektību pārbaude ir realizēta „brīdinošā formā”, proti, ja lietotājs ir ievadījis laukam neatbilstošo vērtību, laukam tiek uzstādīta sarkana kontūra, bet pats lauks vai formas pogas netiek bloķētas un neatkarīgi no ievadītās vērtības, tā vienmēr tiek atribūtam uzstādīta. Pēc atribūta vērtību uzstādīšanas tiek izpildītas paplašinājumu punktu „ValueChanged” un „UpdateCompartmentDomain” transformācijas. Šīs transformācijas ļauj veikt papildus izmaiņas diagrammā, piemēram, dinamiski pārrēķināt kāda elementa stilu. Transformācijas blokshēma ir redzama 3.17. attēlā.



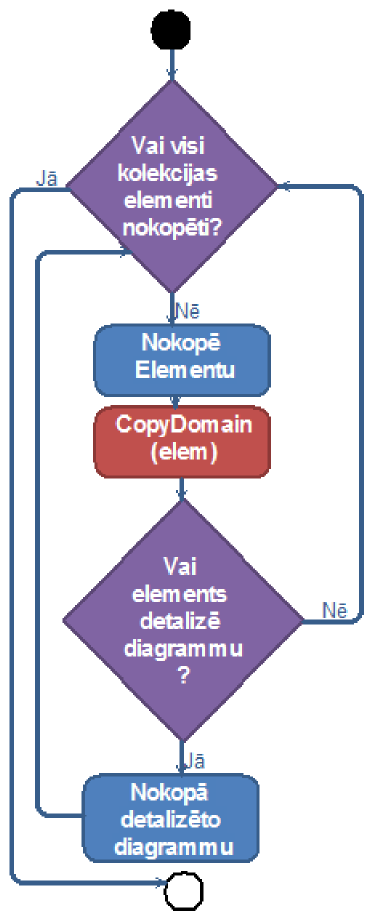
3.17. att. FocusLost blokshēma

3.3.2.2.3 Priekšdefinēto darbību transformācijas

3.3.2.2.3.1 Copy

Šī transformācija nodrošina prezentācijas objektu kopēšanu, un tās rezultāts ir Lua kods, kuru izpildot, tiek radītas nokopēto elementu kopijas. Transformācija pa vienam kopē elementus, kas ir aktīvās diagrammas kolekcijā. Elementa kods, kas attiecas uz prezentācijas metamodeļa instancēm, tiek uzģenerēts universāli, bet, lai nokopētu elementa apkārtni, kas nav prezentācijas metamodeļa instances, tiek izpildīta paplašinājuma punkta „CopyDomain" transformācija. Universālā kopēšanas transformācija izpildās rekursīvi „dziļumā" pa *seed-decomposition* saiti, respektīvi, tā nokopē arī elementu detalizējošās diagrammas ar visiem tās elementiem.

Kad transformācija ir uzģenerējusi modeļa fragmenta kopējošās transformācijas kodu, tas tiek ievietots starpliktuvē (*clipboard*). Ievietošana starpliktuvē ļauj kodu izpildīt starp vairākiem atvērtiem rīkiem. Transformācijas blokshēma ir redzama 3.18. attēlā.

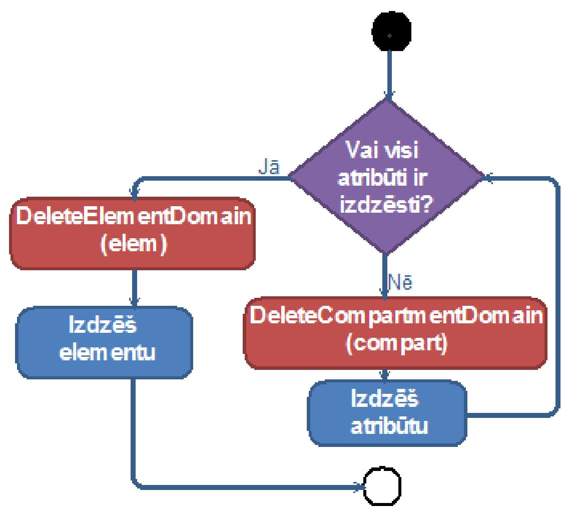


3.18. att. Copy blokshēma

3.3.2.2.3.2 Delete

Šī transformācija nodrošina prezentācijas elementu dzēšanu. Šī transformācija dzēš pa vienam visus elementus, kas ir aktīvās diagrammas kolekcijā. Elementa dzēšanas pirmajā solī tiek dzēsti tā atribūti. Atribūti tiek dzēsti pa vienam, un dzēšana izpildās „dziļumā”, t.i., vispirms tiek dzēsts viss viena atribūta koks ar visiem tā apakšatribūtiem, pēc tam nākamais atribūts ar tā apakšatribūtiem, utt., līdz tiek izdzēsti visi atribūti. Pirms katra atribūta dzēšanas tiek izpildīta paplašinājuma punkta „DeleteCompartmentDomain” transformācija, ar kuru var izdzēst atribūta apkārtni.

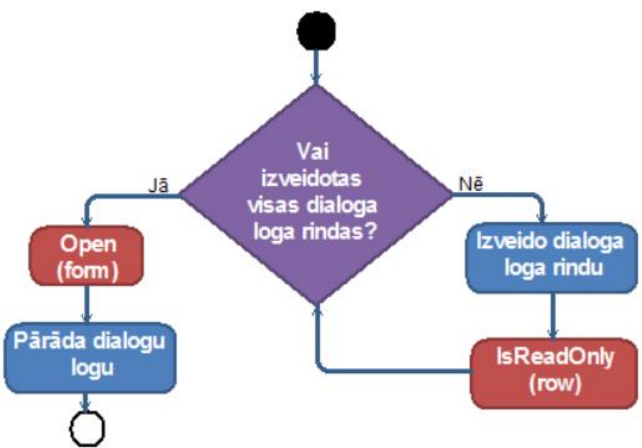
Otrajā solī, kad atribūti ir izdzēsti, tiek izpildīta paplašinājuma punkta „DeleteElementDomain” transformācija. Šī transformācija ļauj rīka izstrādātājam specifiski dzēst tikai elementa apkārtni, bet paša elementa dzēšanu atstāt universālās transformācijas ziņā. Transformācijas blokshēma ir redzama 3.19. attēlā.



3.19. att. *Delete* blokshēma

3.3.2.2.3.3 Properties

Šī transformācija pirmajā solī uzbūvē dialogu loga formu, veidojot pa vienam katru dialogu logu rindu un katrai rindai izpildot paplašinājuma punkta „IsReadOnly” transformāciju, kas ļauj norādīt, vai šīs rindas lauks ir rediģējams vai nerediģējams. Kad visas dialogu logu rindas ir izveidotas, tiek izpildīta paplašinājuma punkta „Open” transformāciju, ar kuru var veikt izmaiņas uzbūvētajā formā, piemēram, pēc kādiem kritērijiem pārrēķināt formas izmērus vai padarīt kādu lauku neaktīvu. Pēdējā solī uzbūvēto formu parāda uz ekrāna. Transformācijas blokshēma ir redzama 3.20. attēlā.

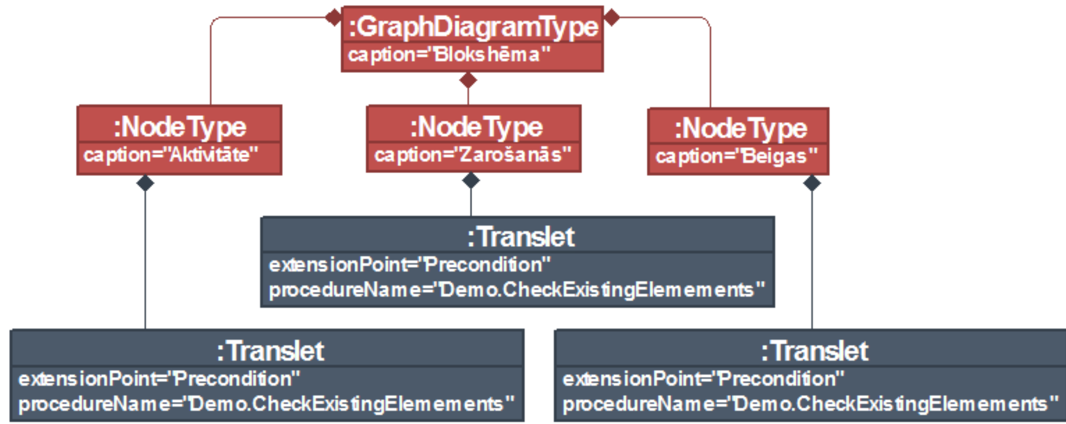


3.20. att. *Properties* blokshēma

3.3.2.3 Piemērs

Lai nodemonstrētu kāda paplašinājuma punkta lietojumu, pieņemsim, ka blokshēmu redaktorā katrā blokshēmā kā pirmais ir jāizveido starta simbols. Tas nozīmē, ka mums ir jāizmanto paplašinājuma punkts „Precondition”, un attiecīgi jāpapildina blokshēmu redaktora

specifikācija (skat. instanču diagrammu 2.25. attēlā) ar klases *Translet* instancēm, kā tas ir parādīts 3.21. attēlā.



3.21. att. Delete blokshēma

Šī specifikācija paredz, ka transformācija, kas veiks pārbaudi saucās „CheckExistingElements”, un tā atrodas datnē „Demo” (datņu struktūru skatīt 3.6. attālu). Savukārt transformācijas „CheckExistingElements” kods ir redzams 3.22. attēlā.

```
function CheckExistingElemements()
		diagram = utilities.current_diagram()
		start = diagram:find("/element:has(/elemType[caption = 'Starts'])")
		if start:is_not_empty() then
			return false
		else
			return true
		end
end
```

3.22. att. Transformācijas *CheckExistingElements* kods

3.4 Secinājumi

Šajā nodaļā ir aprakstīts interpretators, kas dotu rīku definēšanas metamodeļa instanci pārvērš strādājošā rīkā. Savukārt, lai varētu neaprobežoties ar fiksētu interpretatora funkcionalitāti, interpretatorā ir realizēts paplašinājuma punktu mehānisms, kas ļauj konkrētā rīkā realizēt tam specifisku papildfunkcionalitāti, izstrādājot rīkam specifiskas transformācijas. Lai varētu izstrādāt šīs specifiskās transformācijas, DSML rīku būvētājiem ir jāzina tikai 3.2. attēlā parādītā datu struktūra (uz metamodeli var skatīties arī kā uz datu struktūru). Lai arī šis metamodelis nav vienkāršs, tā apjoms nav pārāk liels (satilpst uz vienas A4 lapas) un tādēļ vēl ir samērā viegli saprotams. Tādējādi ar interpretatoru un realizācijas metamodeli var izstrādāt dažādas sarežģītības rīkus, sākot ar tik vienkāršiem kā blokshēmu redaktors, kura realizēšanai pietiek ar specifikācijas uzdošanu un papildus transformāciju programmēšana nav

nepieciešama, un beidzot ar tādiem rīkiem, ar kuriem var izstrādāt citus DSML rīkus (detalizētāk 4.2. nodaļā).

4 Konfigurators

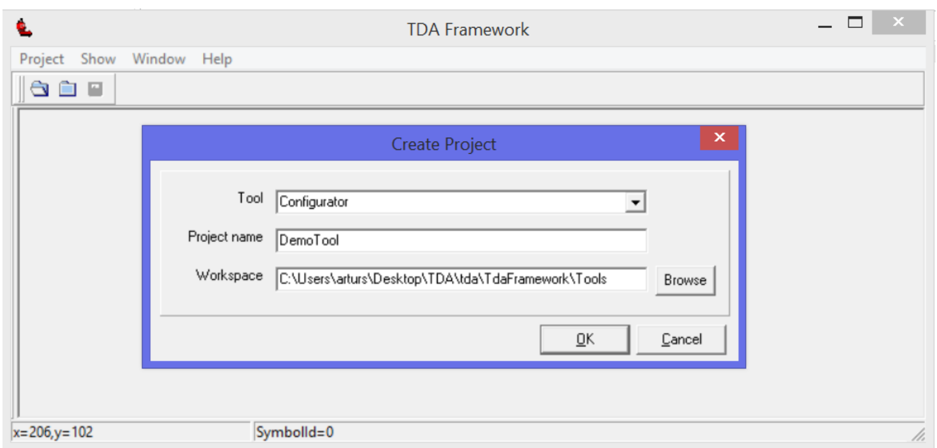
Kā iepriekš noskaidrojām, ar rīku definēšanas metamodeļa instancēm var specificēt DSML rīkus un pēc tam šo specifikāciju izmantot, lai to pārvērstu strādājošā rīkā. Tādējādi par aktuālu problēmu DSML rīku izstrādē kļūst rīku definēšanas metamodeļa instanču radīšana, un tas, cik ātri un ērti tās var radīt, ietekmē DSML rīku izstrādes laiku.

Šim nolūkam ir izstrādāts konfigurators [33, 34, 35] jeb rīks, ar kuru tiek radītas un uzturētas rīku definēšanas metamodeļa instances. Konfigurators ir izstrādāts kā grafisks DSML rīks (kad lejupielādē platformu, konfigurators jau pēc noklusējuma ir iekļauts platformā), ar kuru jauni DSML rīki tiek specificēti kā konfiguratora grafiskie modeļi. Konfiguratora grafisko valodu var uzskatīt par augstāku rīku definēšanas metamodeļa abstrakciju, un tās galvenā priekšrocība ir daudz kompaktākie modeļi salīdzinājuma ar UML klašu diagrammām, jo elementu stili tiek definēti ar tā izskatu un daļa informācijas tiek ievadīta caur dialogu logu formām.

Šādas rīku izstrādes ieguvums ir tas, ka konfigurators automātiski savus modeļus saglabā kā rīku definēšanas metamodeļa instances, un tā rezultātā tas nepieprasa rīku izstrādātājiem pārzināt rīku definēšanas metamodeli, dažādas tā noklusētās vērtības, kā arī zināt kādi modeļi ir korekti un kādi nav. Tādējādi, specificējot rīkus ar konfiguratoru, tiek ievērojami samazināts rīku definēšanas metamodeļa instances izstrādes laiks, un tiek izslēgta iespējamība uzbūvēt tādu interpretāciju, kuru neatbalsta interpretators.

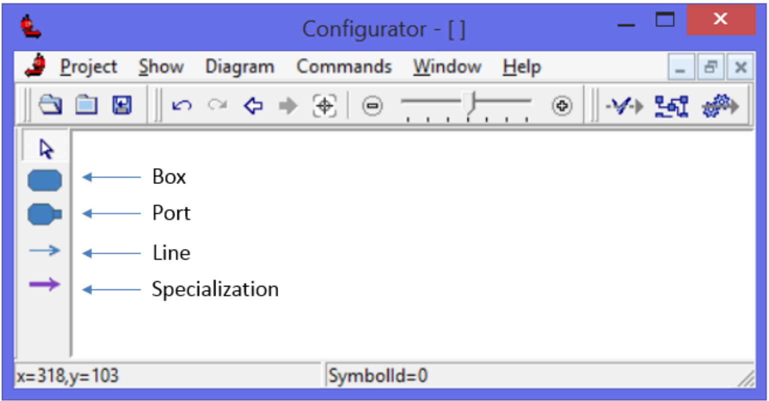
4.1 Konfiguratora valoda

Jauna DSML rīka konfigurēšana sākas ar jauna projekta izveidošanu, ar 4.1. attēlā redzamo logu.



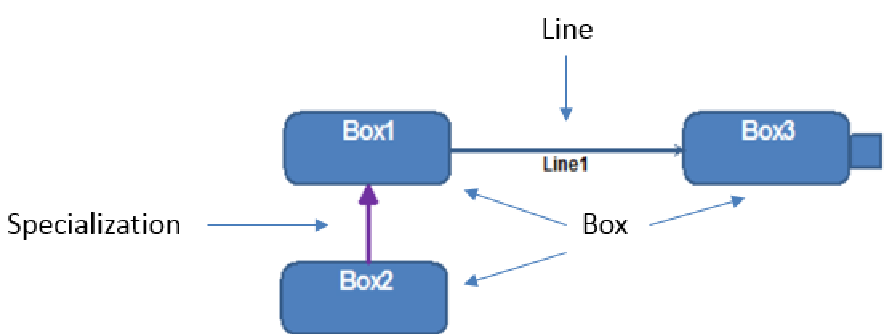
4.1. att. Jauna projekta izveidošana

Lai izvēlētos konfiguratoru, laukā *Tool* lietotājam ir jānorāda „Configurator”, laukā *Project Name* ir jāievada projekta nosaukums, kas sakritīs ar izstrādājamā rīka nosaukumu, piemēram, „DemoTool”, bet laukā *Workspace* ir jānorāda projekta atrašanās vieta platformas *Tools* direktorijā. Pēc jauna projekta izveides atveras projektu diagramma, bet, nospiežot taustiņu „C”, atveras 4.2. attēlā redzamā konfiguratora diagramma.



4.2. att. Konfiguratora diagramma

Jaunus elementus konfiguratora diagrammā veido ar paleti, kura sastāv no četrām pogām, un ar katru pogu var izveidot kādu no 4.3. attēlā redzamajiem konfiguratora grafiskajiem elementiem – *Box*, *Line*, *Port* vai *Specialization*.

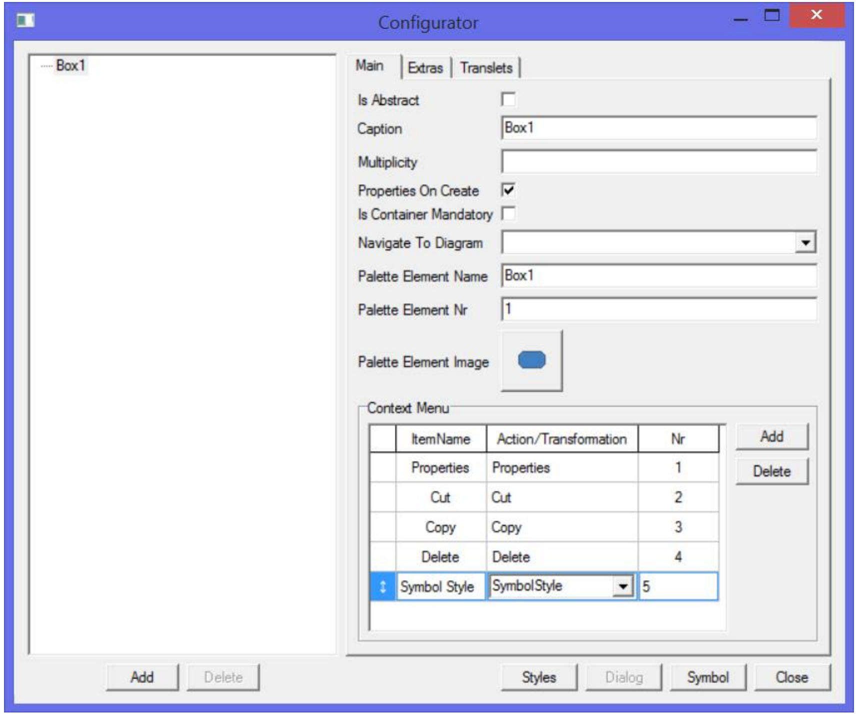


4.3. att. Konfiguratora grafiskās valodas elementi

Katrs valodas elements, izņemot *Specialization*, specificē kādu izstrādājamā DSML rīka grafisko elementu, respektīvi, elements *Box* specificē kastes tipa elementus, *Line* specificē līnijas tipa elementus un *Port* specificē porta tipa elementus.

4.1.1 Box

Box elementu veidošana notiek ar paletes elementu *Box*, un tā vērtības tiek ievadītas caur dialogu logu, kas ir redzams 4.4. attēlā.



4.4. att. *Box* dialogu loga cilne *Main*

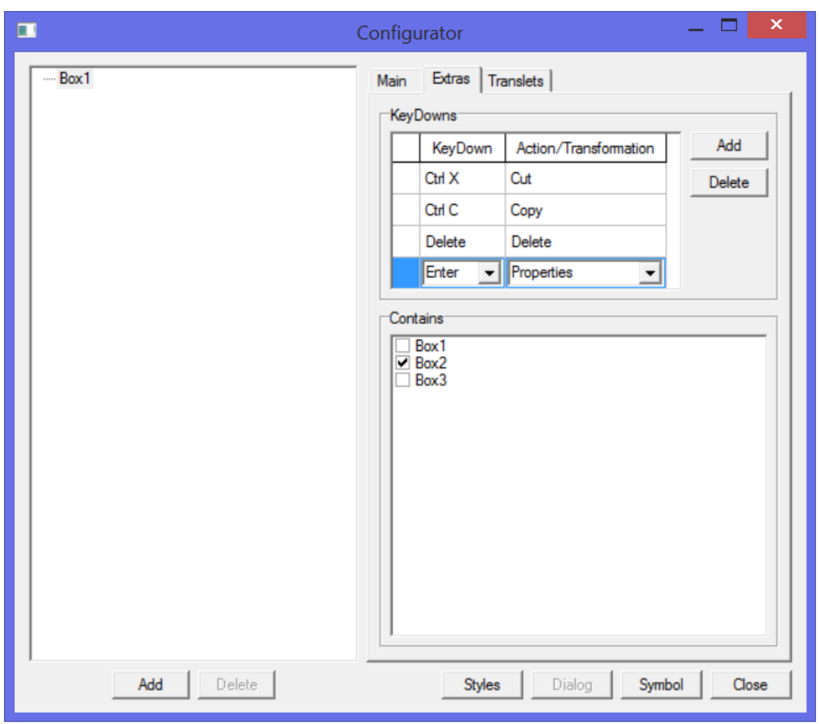
Dialogu loga kreisajā pusē atrodas koks ar elementa un tā atribūtu nosaukumiem (pagaidām neviena atribūta nav), bet labajā pusē atrodas trīs cilnes – *Main*, *Extras* un *Translets*. Cilnes *Main* laukā *Caption* ir jānorāda elementa nosaukums (jābūt diagrammā unikālam). Laukos *Palette Element Name*, *Palette Element Nr* un pogu *Palette Element Image* var ievadīt atbilstošā paletes elementa īpašības – nosaukumu, numuru paletē un paletes ikonas adresi. Ja kādu iemeslu dēļ jaunu elementu nevajag veidot ar paletes elementu, tad lauks *Palette Element Name* ir jāatstāj tukšs un atbilstošais paletes elements netiks izveidots.

Laukā *Open Properties On Element Create* var norādīt, vai pēc jauna elementa izveidošanas nepieciešams atvērt dialogu logu, bet laukā *Is Container Mandatory* var norādīt, ka šī tipa kastēm ir vienmēr jāatrodas ievietotām kādā kastē. Lai norādītu maksimāli pieļaujamo viena tipa kastu skaitu diagrammā, laukā *Multiplicity* ir jānorāda to ierobežojošais skaits. Ja šis lauks ir atstāts tukšs, tad maksimālais ierobežojums nepastāv. Lauks *Navigate To Diagram* ļauj norādīt, ka, veidojot jaunu kasti, kastei vēl nepieciešams piesaistīt jaunu diagrammu. Piemēram, diagrammas piesaistīšana ir nepieciešama elementu detalizēšanai ar diagrammu.

Tabulā *Context Menu* var definēt kastes uznirstošās izvēlnes operācijas. Katra tabulas rindiņa atbilst vienai uznirstošās izvēlnes operācijai, un katra operācija tiek specificēta, ievadot operācijas nosaukumu, norādot vai nu izpildāmo darbību (izpildāmās darbības tiek piedāvātas no saraksta), vai transformācijas nosaukumu (tā, kas realizēs šo operāciju), kā arī operācijas kārtas numuru uz izvēlnes.

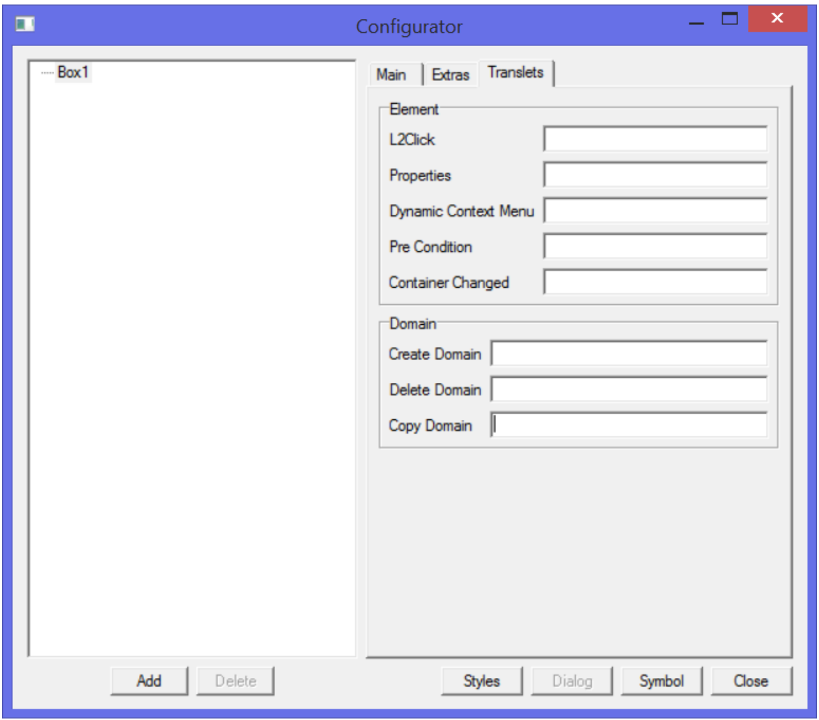
Cilnē *Extras*, kas ir redzama 4.5. attēlā, rīka izstrādātājs var ievadīt īpašības, kuras tiek reti lietotas, vai arī reti mainītas. Cilne sastāv no divām daļām - tabulas *KeyDowns* un saraksta *Contains*. Tabulā *KeyDowns* var norādīt elementam piekārtotās taustiņu kombinācijas un tām atbilstošās transformācijas, kas tiek izpildītas pēc attiecīgās taustiņu kombinācijas ievades. Katra taustiņu kombinācija atbilst vienai tabulas rindiņai un pēc noklusējuma katrai kastei ir piesaistītas četras taustiņu kombinācijas „Ctrl X”, „Ctrl C”, „Delete” un „Enter” ar tām atbilstošajām noklusētajām darbībām - izgriezt, kopēt, dzēst un atvērt dialogu logu.

Sarakstā *Contains* ir norādīti visi diagrammā esošie kastes tipa elementi. Ja kādu saraksta elementu ieķeksē, tad ar to tiek norādīts, ka specificējamā tipa kaste drīkst saturēt ieķeksētā tipa kastes. Piemēram, šajā logā ir norādīts, ka *Box1* tipa kastes drīkst saturēt *Box2* tipa kastes.



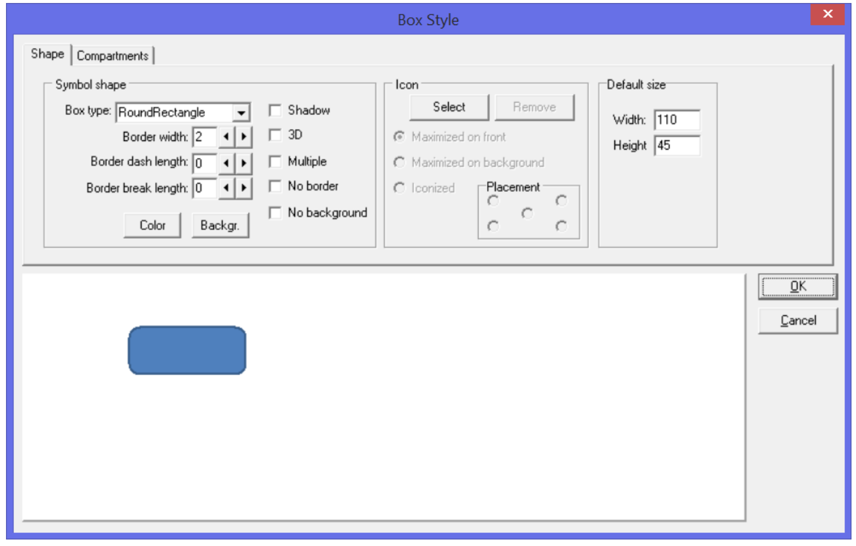
4.5. att. *Box* dialogu loga cilne *Extras*

Cilne *Translets* ir redzama 4.6. attēlā. Šīs cilnes laukos var norādīt elementa paplašinājuma punktu transformācijas, kur lauks „L2Click” atbilst paplašinājuma punktam *DynamicDoubleClick*, „Properties” atbilst *Properties*, „Dynamic Context Menu” atbilst *DynamicPopUp*, „Pre Condition” atbilst *PreCondition*, „Container Changed” atbilst *ContainerChanged*, „Create Domain” atbilst *CreateElementDomain*, „Delete Domain” atbilst *DeleteElementDomain* un „Copy Domain” atbilst *CopyDomain* (paplašinājuma punktu saraksts un to skaidrojums ir 3.6. tabulā).



4.6. att. *Box* dialogu loga cilne *Translets*

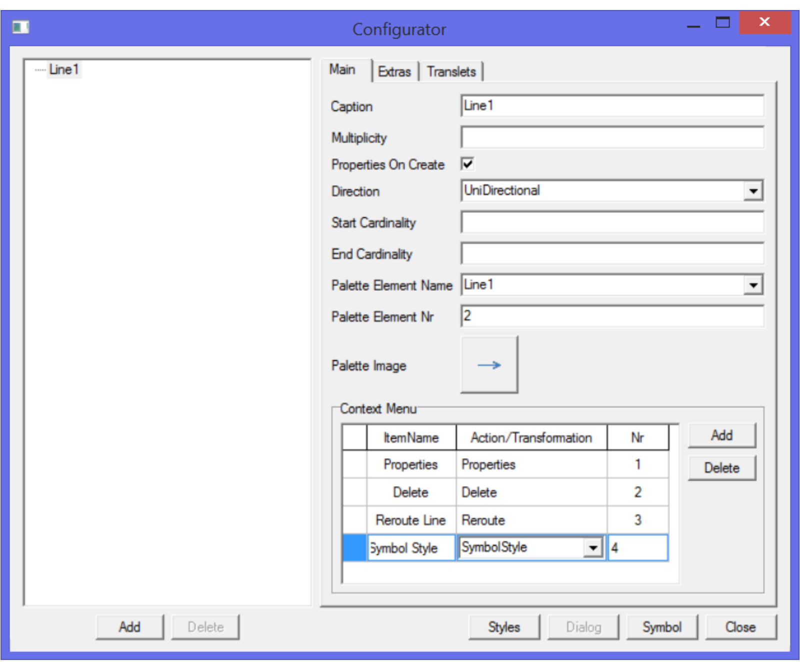
Pēc kastes īpašību ievadīšanas ir jāspecificē kastes izskats. To var izdarīt nospiežot pogu *Symbol*, kas atver 4.7. attēlā redzamo dialoga logu. Cilnē *Shape* ir paredzēta kastes stila definēšana un tā ir sadalīta četrās grupās – *Symbol shape*, *Icon*, *Default size* un lauks, kurā var redzēt kastes izskatu. Svarīgākā grupa kastes stila definēšanā ir *Symbol shape*. Šīs grupas lauks *Box type* ļauj norādīt kastes formu, piedāvājot sarakstu ar iepriekš definētām formām, piemēram, taisnstūri, elipsi, trapeci utt. Laukā *Border width* ir jānorāda kasti ietverošā rāmja līnijas biezums, kas pēc noklusējuma ir „1”, bet vispārīgā gadījumā – jo lielākā vērtība, jo platāka līnija. Lauki *Border dash length* un *Border break length* ļauj norādīt, ka ietverošā rāmja līnija ir raustīta. Pēc noklusējuma abu lauku vērtība ir „0”, kas norāda, ka līnija nav raustīta. Ja kādu no šīm vērtībām palielina, tad līnija kļūst raustīta. *Border dash length* ļauj norādīt, cik gara ir raustītā līnijas, bet *Border break length* ļauj norādīt, cik garš ir tukšums uz līnijas. Ar pogu *Color* var uzstādīt ietverošā rāmja līnijas krāsu, bet ar pogu *Backgr*. var uzstādīt kastes krāsu. Pārējie lauki ļauj uzstādīt dažādus papildus specefektus. Grupa *Icon* ļauj kasei piesaistīt bildi. Ar pogu *Select* var bildi pievienot, bet ar *Remove* noņemt. Piesaistītās bildes pozīciju diagrammā var norādīt *Placement* apakšgrupā kopā ar iezīmētu *Iconized* lauku. Ja ir iezīmēts *Maximized on front*, tad bilde pārklāj visu kasti ar visiem atribūtiem, bet, ja ir iezīmēts *Maximized on background*, tad bilde pārklāj visu kasti, atribūtus atstājot virspusē. Savukārt grupa *Default size* ir paredzēta noklusētā kastes platuma un augstuma uzstādīšanai.



4.7. att. Box stila dialogu logs

4.1.2 Line

Ja izvēlas paletes elementu *Line* un pēc tam izveido līniju, kas savieno divus citus diagrammas elementus, tad ar to tiek pateikts, ka par šī tipa līnijas galapunktiem drīkst būt norādītā tipa elementi, bet līnijas īpašības ievada caur 4.8. attēlā redzamo dialogu logu.



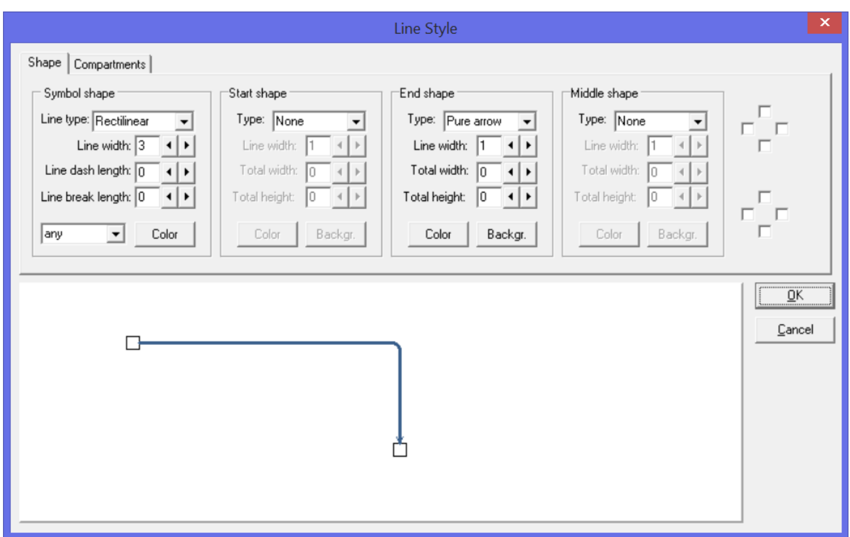
4.8. att. Line dialogu loga cilne *Main*

Lai arī *Line* dialogu loga *Main* cilne ir līdzīga ar *Main* cilni *Box* elementam, tām ir vairākas atšķirības. Pirmkārt, nav lauku *Is Container Mandatory* un *Navigate To Diagram*. Otrkārt, ir jauns lauks *Direction*. Šis lauks ļauj norādīt līnijas orientāciju, un pieļaujamās vērtības ir – „UniDirectional”, „BiDirectional” un „ReverseBiDirectional” (tās atbilst iepriekš

skaidrotajām klases *EdgeTpye* atribūta *direction* vērtībām). Treškārt, ir jauni lauki *Start Cardinality* un *End Cardinality*, kas norāda līnijas galu elementiem maksimālo izejošo un ieejošo viena tipa līniju skaitu. Ceturtkārt, *Palette Element Name* lauka tips ir izkrītošā izvēlne, nevis vienkāršs teksta lauks. Izvēlne sastāv no visu valodā esošo līniju paletes elementiem, un, izvēloties kādu no tiem, tiek norādīts, ka ar izvēlēto paletes elementu tiks veidotas specificējamā tipa līnijas, respektīvi, ar šo mehānismu ir iespējams norādīt, ka dažādu tipu līnijas tiek veidotas ar vienu paletes elementu. Savukārt, ja laukā ievada izvēlnē neesošu nosaukumu, tad tiek izveidots jauns paletes elements.

Cilnēs *Extras* un *Translets* ievadāmā informācija ir līdzīga *Box* dialogu logos ievadāmajai. Cilnē *Extras* var norādīt elementam piekārtotās taustiņu kombinācijas, bet cilnē *Translets* var norādīt transformācijas, kas attiecas uz līnijas paplašinājuma punktiem.

Line stilu definēšanas logs ir redzams 4.9. attēlā. Tā cilne *Shape* sastāv no grupām *Symbol shape*, *Start shape*, *End shape*, *Middle shape* un lauka, kurā var redzēt līnijas izskatu. Grupa *Symbol shape* attiecas tieši uz līnijas stila definēšanu. Laukā *Line type* var norādīt līnijas tipu, laukā *Line width* var norādīt līnijas biezumu, bet laukos *Line dash length* un *Line break length* var norādīt, ka līnija ir raustīta, bet ar pogu *Color* var norādīt līnijas krāsu. Grupās *Start shape*, *End shape* un *Middle shape* ir lauks *Type*, kas ļauj norādīt līnijas sākuma, beigu un vidus simbolu, piemēram, bultu, trijstūri, riņķi utt. *Line width* ļauj norādīt simbolu ierobežojošās līnijas biezumu, bet *Total width* un *Total height* norāda simbola platumu un garumu. Poga *Color* ļauj norādīt simbolu ierobežojošās līnijas krāsu, bet *Backgr.* ļauj norādīt simbola krāsu.



4.9. att. Line stila dialogu logs

4.1.3 Port

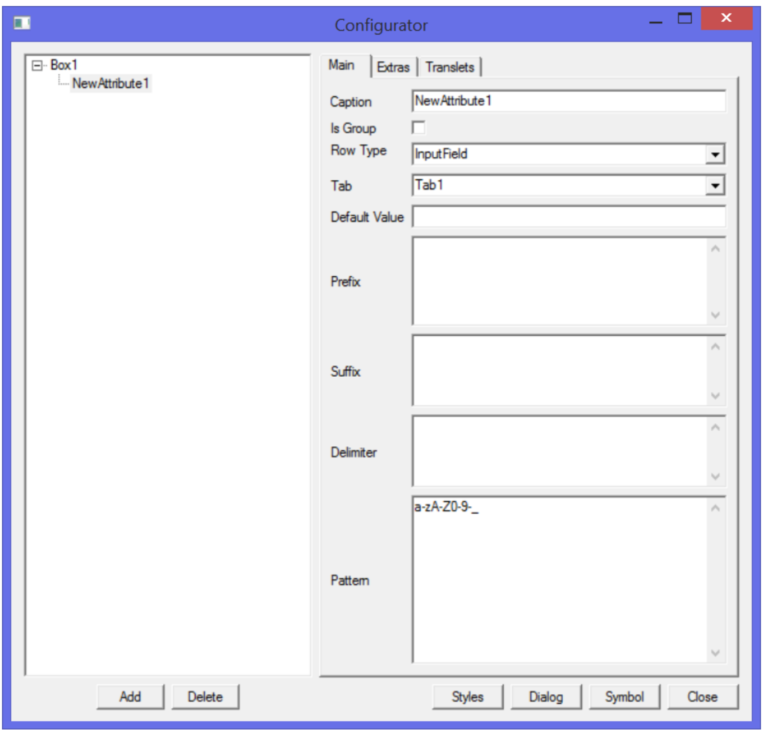
Ja nospiež paletes pogu *Port* un pēc tam nospiež uz kāda *Box* elementa, tad tiek izveidots jauns *Port* elements (*port* tipa elementiem ir vienmēr jābūt piesaistītiem kādam kastes tipa elementam), un ar to tiek pateikts, ka jaunajā rīkā tiek izveidots jauns porta tipa elements, kuru drīkst pievienot šī tipa kastei. Tā kā šī elementa īpašībās ievada caur dialogu logu, kura struktūra sakrīt ar citu elementu dialogu logu struktūru, un stilu definēšanas logs ir ļoti līdzīgs ar *Box* stila logu, tad šos logus detalizētāk neaplūkosim.

4.1.4 Specialization

Ja nospiež paletes pogu *Specialization* un pēc tam izveido līniju, kas savieno divus diagrammas elementus, tad tiek izveidots jauns *Specialization* elements. Ar šo elementu tiek norādīts, ka viens diagrammas elements ir apakštips otram.

4.1.5 Atribūtu specificēšana

Atribūtu specificēšana visiem elementiem neatkarīgi no to tipa ir vienāda un tādēļ apskatīsim tikai atribūta specificēšanu kastes gadījumā. Jauni atribūti tiek pievienoti ar pogu *Add*, un tā rezultātā zem iepriekš aktīvās koka virsotnes tiek izveidota jauna koka virsotne, kura automātiski aktivizējas, un pēc tam parādās 4.10. attēlā redzamais dialogu logs atribūta īpašību ievadīšanai.

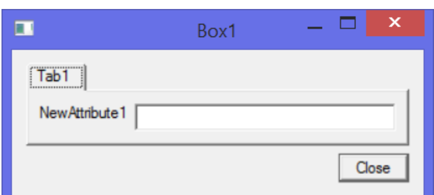


4.10. att. Atribūta dialogu loga cilne *Main*

Līdzīgi kā elementiem, atribūtiem laukā *Caption* ir jānorāda tā nosaukums (viena līmeņa atribūtiem jābūt unikāliem). Atribūta dialogu logā tiek norādītas divu veidu īpašības. Vienas specificē atribūta vērtību veidošanu, piemēram, noklusēto vērtību vai prefiksu, bet otras specificē informāciju dialogu logu veidošanai. Lauki *Row Type* un *Tab* attiecas uz dialogu logu veidošanu. Laukā *Row Type* ir jānorāda ievadlauka tips atribūta vērtību ievadīšanai. Šim laukam ir izkrītošās izvēlnes tips, kur izvēlne sastāv no 2.1. tabulā uzskaitītajiem ievadlauku nosaukumiem.

Savukārt, ja nepieciešams dialoga loga komponentes grupēt, tad to var izdarīt ar cilnēm. Laukā *Tab* ir jānorāda cilnes nosaukums. Gadījumā, ja norādītais cilnes nosaukums jau eksistē, tad atribūta lauks tiek piesaistīts jau esošajai cilnei. Gadījumā, ja norādītais cilnes nosaukums neeksistē, tad tiek izveidota jauna cilne un atribūta lauks tiek piesaistīts no jauna izveidotajai cilnei.

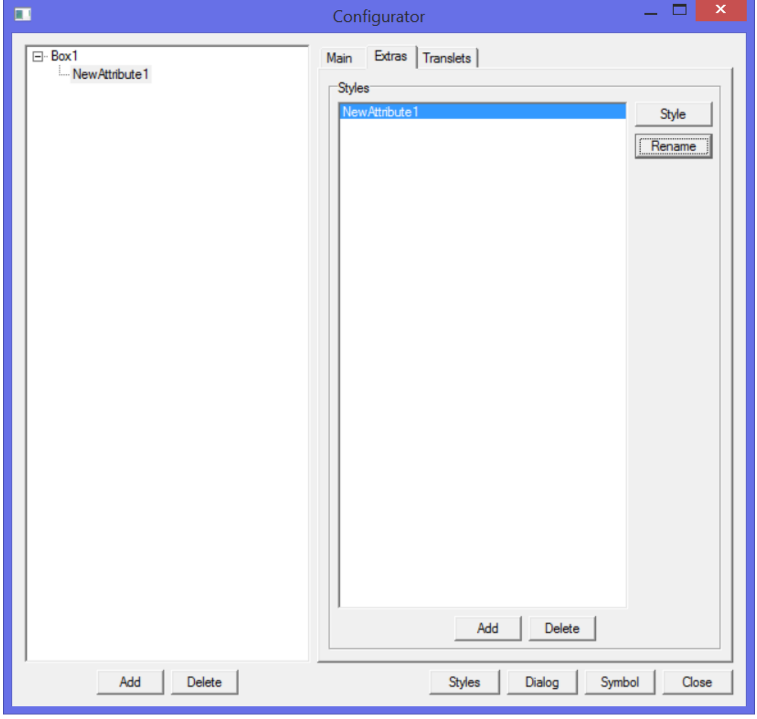
Piemēram, 4.10. attēlā redzamajā dialogu logā ir norādīts, ka atribūta „NewAttribute1" ievadlauka tips ir „InputField" un cilnes nosaukums ir „Tab1". Šai specifikācijai atbilstošais dialogu logs ir redzams 4.11. attēlā.



4.11. att. Specificētais dialogu logs

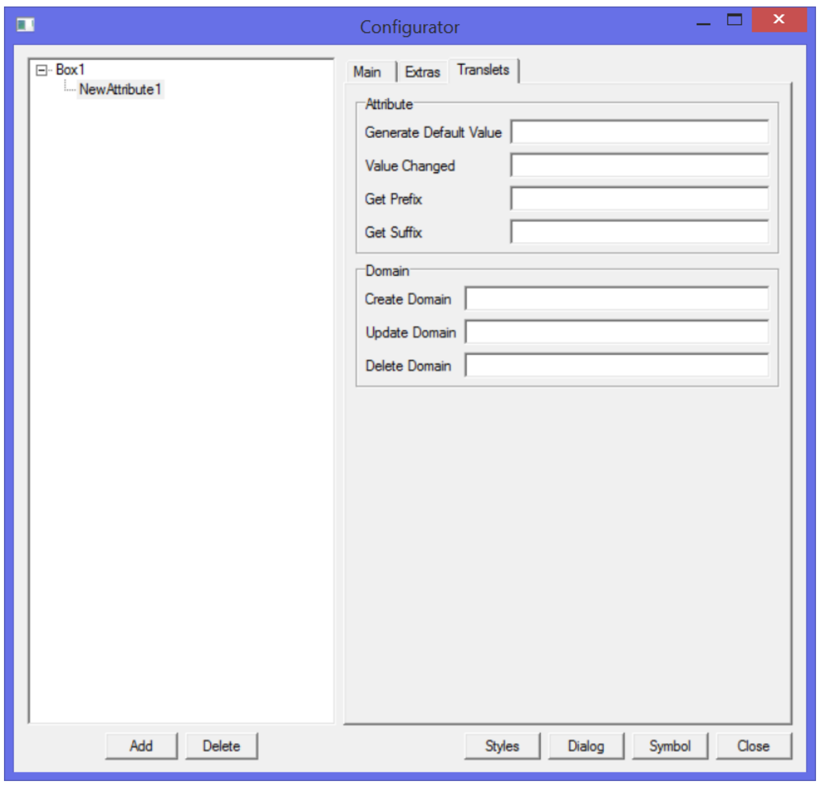
Atlikušo lauku vērtības atbilst klases *CompartType* atribūtu vērtībām. Laukā *Default Value* ir iespējams norādīt atribūta noklusēto vērtību, kas metamodelī atbilst klases *CompartType* atribūta *defaultValue* vērtībai, laukos *Prefix* un *Suffix* var norādīt atribūta prefiksa un sufiksa vērtības un tās atbilst atribūtu *prefix* un *suffix* vērtībām, laukā *Delimiter* var norādīt apakšatribūtu atdalītāja vērtību un tā atbilst atribūta *delimiter* vērtībai, laukā *Pattern* var norādīt atribūta vērtībā pieļaujamos simbolus un tas atbilst atribūta *pattern* vērtībai.

Cilne *Extras* ir redzama 4.12. attēlā. Šajā cilnē atribūtam var piesaistīt jaunus stilus (jaunu stilu pievienošana detalizētāk ir aprakstīta 4.1.6. nodaļā).



4.12. att. Atribūta dialogu loga cilne *Extras*

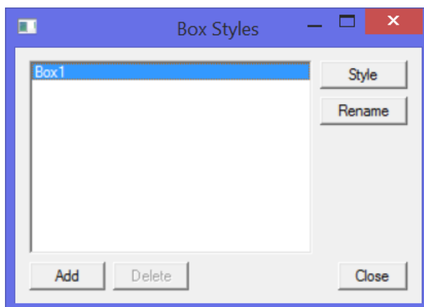
Cilne *Translets* ir redzama 4.13. attēlā. Šīs cilnes laukos var norādīt atribūta paplašinājuma punktu transformācijas, kur lauks „Generate Default Value" atbilst *GenerateDefaultValue*, „Value Changed" atbilst *ValueChanged*, „Get Prefix" atbilst *GetPrefix*, „Get Suffix" atbilst *GetSuffix*, „Create Domain" atbilst *CreateCompartmentDomain*, „Update Domain" atbilst *UpdateCompartmentDomain* un „Delete Domain" atbilst *DeleteCompartmentDomain* paplašinājuma punktam (paplašinājuma punktu saraksts un to skaidrojums ir 3.7. tabulā).



4.13. att. Atribūta dialogu loga cilne *Translets*

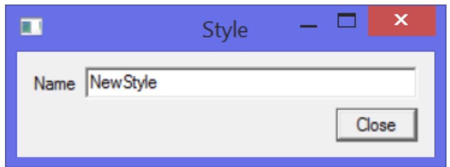
4.1.6 Stilu pievienošana

Katram elementam un atribūtam pēc noklusējuma ir piesaistīts tieši viens stils. Gadījumā, ja ir vajadzība pēc vairākiem stiliem, tad elementa dialogu logā ir jānospiež *Styles* poga, bet atribūta gadījumā jāpārslēdzas uz cilni *Extras*. Tā rezultātā priekš elementa parādīsies 4.14. attēlā redzamais logs, bet priekš atribūta 4.12. attēlā redzamais logs.



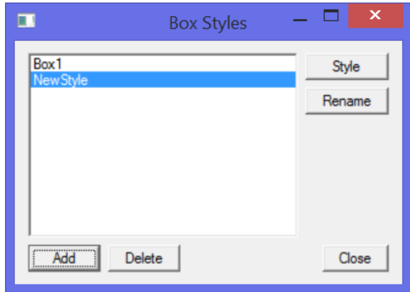
4.14. att. Stila specificēšanas logs

Tā kā gan elementu, gan atribūtu stilu pievienošana notiek pēc viena principa, tad apskatīsim tikai elementa stila pievienošanu. Lai pievienotu jaunu stilu, ir jāspiež poga *Add* un tad parādīsies 4.15 attēlā redzamais logs, kurā ir jānorāda stila nosaukums.



4.15. att. Stila nosaukuma ievade

Stāvoklis pēc stila pievienošanas ir redzams 4.16. attēlā.



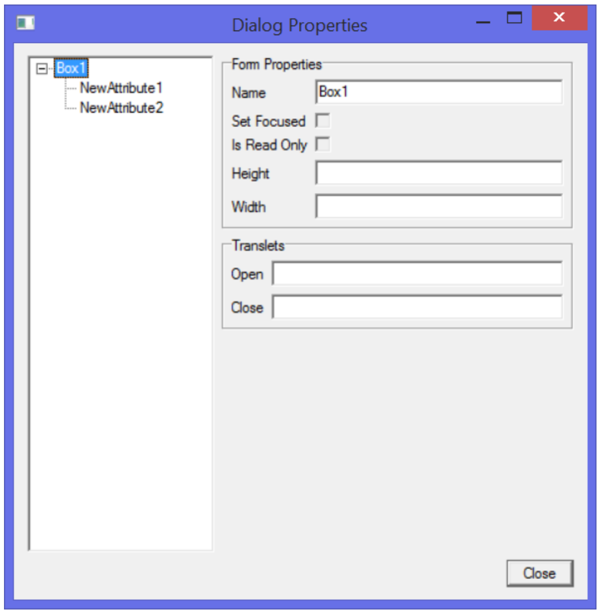
4.16. att. Stila specificēšanas logs pēc stila pievienošanas

Lai uzstādītu stila parametrus, ir jānospiež poga *Style* un tad parādīsies elementa tipam atbilstošais dialogu logs, piemēram, ja elements ir *Box*, tad parādīsies 4.7. attēlā redzamais logs, vai, ja elements ir *Line*, tad parādīsies 4.9. attēlā redzamais logs.

4.1.7 Dialogu logu konfigurēšana

Ja elementu dialogu logā nospiež pogu *Dialog*, parādās 4.17. attēlā redzamais dialogu logs. Dialogu loga kreisajā pusē ir koks, kura saturs atbilst specificētā elementa dialogu loga struktūrai. Piemēram, attēlā redzamā dialogu loga koks reprezentē formu ar diviem laukiem „NewAttribute1” un „NewAttribute2”.

Sākotnējā dialogu logu komponenšu secība atbilst to veidošanas secībai, bet, ja nepieciešams secību mainīt, to var izdarīt, ar peli mainot koka virsotnes pozīciju viena vecāka ietvaros.

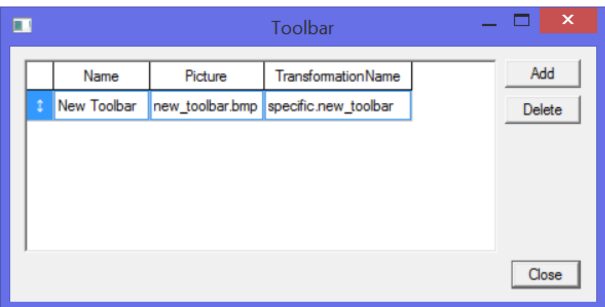


4.17. att. Dialogu loga konfigurēšanas logs

Dialogu loga labajā pusē var norādīt aktīvās dialogu loga komponentes īpašības. Piemēram, attēlā redzamajā logā aktīvā komponente ir forma. Grupā *Form Properties* var mainīt dažādus komponentes parametrus, bet grupā *Translets* var norādīt komponentes paplašinājuma punktu transformācijas, kas atbilst 3.2. tabulā aprakstītajiem.

4.1.8 Diagrammas funkcionalitāte

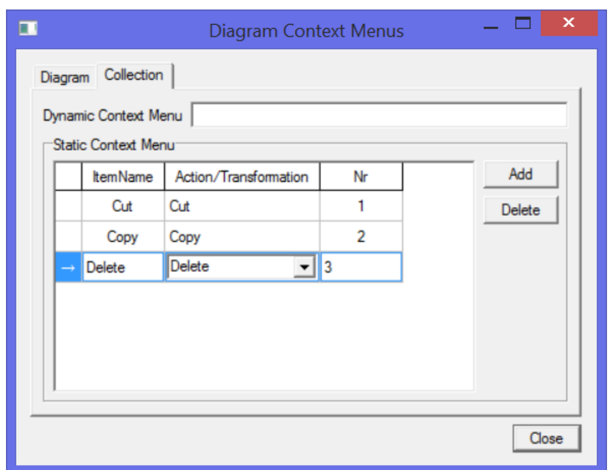
Nospiežot peles labo pogu uz tukšas vietas konfiguratora diagrammā, parādās uznirstošā izvēlne, kuras operācijas – *Add Toolbar*, *Add Context Menu* un *Add Key Downs* specificē diagrammas funkcionalitāti. Ja no izvēlnes izvēlas operāciju *Add Toolbar*, tad parādās 4.18. attēlā redzamais dialogu logs. Šajā logā katra tabulas rinda specificē vienu rīku joslas pogu. Piemēram, attēlā redzamajā logā ir specificēta rīku josla ar vienu pogu „New Toolbar".



4.18. att. Rīku joslas specificēšanas logs

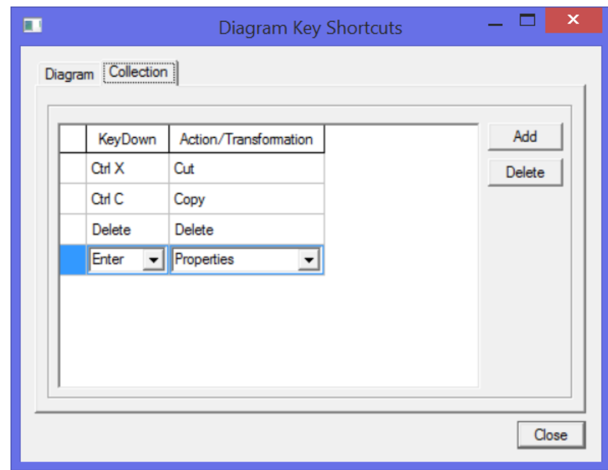
Savukārt, ja no sākotnējās uznirstošās izvēlnes izvēlas operāciju *Add Context Menu*, tad parādās 4.19. attēlā redzamais logs. Šajā logā var specificēt diagrammas tipa uznirstošās

izvēlnes un tiek izšķirtas divas situācijas. Vienā situācijā uznirstošā izvēlne tiek specificēta tukšai kolekcijai un šai situācijai atbilst cilne *Diagram*, otrā situācijā uznirstošā izvēlne tiek specificēt vairāku elementu kolekcijai un šai situācijai atbilst cilne *Collection*. Attēlā redzamajā logā ir specificēta statiskā uznirstošā izvēlne ar trīs operācijām vairāku elementu kolekcijai, bet, lai specificētu dinamisku uznirstošo izvēlni, laukā *Dynamic Context Menu* ir jānorāda atbilstošās transformācijas nosaukums.



4.19. att. Uznirstošās izvēlnes specificēšanas logs

Visbeidzot, ja no izvēlnes izvēlas operāciju *Add Key Downs*, tad parādās 4.20. attēlā redzamais dialogu logs. Šajā logā var specificēt diagrammai piesaistītās taustiņu kombinācijas un arī šajā gadījumā tiek izšķirtas divas situācijas. Vienā gadījumā taustiņu kombinācijas tiek specificētas tukšai kolekcijai un šai situācijai atbilst cilne *Diagram*, otrā gadījumā taustiņu kombinācijas tiek specificētas vairāku elementu kolekcijai un šai situācijai atbilst attēlā redzamā cilne *Collection*, kura specificē trīs taustiņu kombināciju apstrādi.



4.20. att. Taustiņu kombināciju specificēšanas logs

4.1.9 Blokshēmu redaktora konfigurēšanas apraksts

Šajā nodaļā aplūkosim, kā ar konfiguratoru var specificēt to pašu blokshēmu redaktoru, kurš tika specificēts ar rīku definēšanas metamodeļa instancēm 2. nodaļā. Blokshēmu redaktoru specificēšana sākas ar jauna platformas projekta izveidi, kas tiek izdarīts norādot attiecīgo informāciju 4.1. attēlā redzamajā dialogu logā. Pēc tam, nospiežot taustiņu „C”, atveras konfiguratora diagramma.

Šajā diagrammā vispirms specificēsim kastes tipa elementu, kas atbilst aizmetnim. Lai to izdarītu, ir jāizveido jauns *Box* tipa elements. Tā rezultātā atvērsies *Box* dialogu logs (4.4. attēls), kurā ir jāievada aizmetņa elementu specificējošās vērtības. Laukos *Caption* un *Palete Element Name* ir jāievada elementa nosaukums - „Blokshēma”. Laukā *Palette Element Nr* ir jānorāda paletes elementa numurs (piemēram, „1”), laukā *Palette Element Image* paletes ikonas adrese, bet laukā *Navigate To Diagram* ir jāievada nosaukums „Blokshēma”, kas norāda, ka aizmetnis detalizē blokshēmu diagrammas, proti, veidojot aizmetni, interpretators automātiski izveidos aizmetnim atbilstošo blokshēmas diagrammu. Kad ir specificēts aizmetņa elements, tad, uz šī elementa veicot peles kreisās pogas dubultklikšķi, atvērsies jauna konfiguratora diagramma, kurā ir jāspecificē blokshēmas elementi (t.i., šī diagramma ir blokshēmas specifikācija).

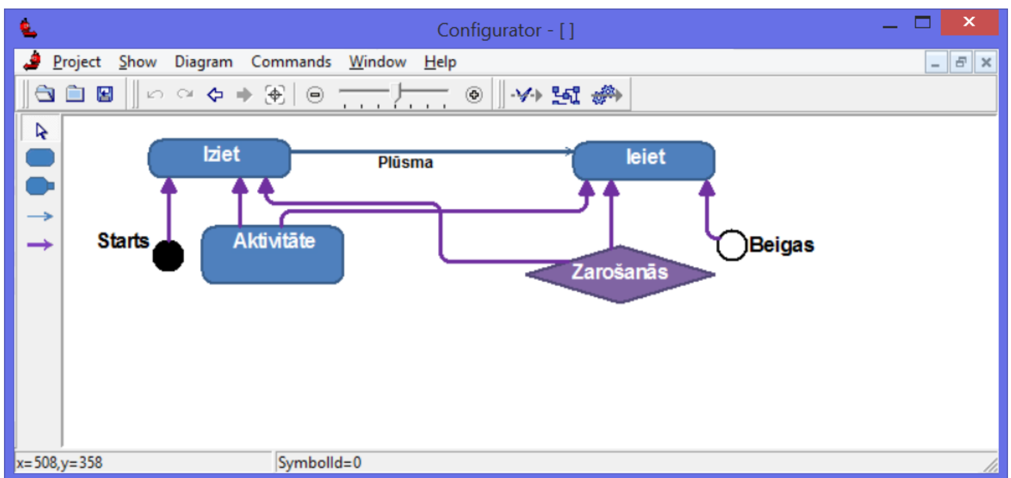
Tā kā visu blokshēmas elementu specificēšanas ir līdzīga, detalizētāk apskatīsim „Aktivitāte” specificēšanu. Šī un citu elementu specificēšana ir tāda pati kā aizmetņa specificēšana, t.i., lai specificētu „Aktivitāte”, pirmajā solī ir jāizveido *Box* elements. Tā rezultātā atvērsies *Box* dialogu logs (4.4. attēls), kurā ir jāievada elementu „Aktivitāte” specificējošās vērtības. Laukos *Caption* un *Palete Element Name* ir jāievada elementa nosaukums - „Aktivitāte”, bet laukos *Palette Element Nr* un *Palette Element Image* attiecīgi paletes elementa numurs un paletes ikonas adrese.

Savukārt, ja gribam pievienot 3.3.2.3. nodaļas piemērā izveidoto transformāciju, kas pārbauda, vai starta simbols jau eksistē, mums ir jāpārslēdzas uz cilni *Translets* (4.5. attēls) un laukā „Pre Condition” ir jāievada vērtība „Demo.CheckExistingElements”.

Lai specificētu „Aktivitāte” noklusēto stilu, jānospiež poga *Symbol*. Tā rezultātā parādīsies 4.7. attēlā redzamais dialogu logs, un tajā jānorāda attiecīgie stila parametri. Pēc „Aktivitāte” funkcionalitātes un stilu uzstādīšanas, ir jāspecificē elementa „Aktivitāte” atribūts „Nosaukums”. Lai specificētu „Nosaukums”, ir jānospiež poga *Add*, un tā rezultātā parādīsies atribūta dialogu logs (4.10. attēls), kura laukā *Caption* jāievada vērtība „Nosaukums”, bet laukā *Row Type* jāizvēlas vērtība „InputField”.

Pārējo blokshēmas redaktora elementu specificēšana ir līdzīga, atšķirīgāka vien ir abstrakto elementu „Iziet” un „Ieiet” veidošana. Lai šos elementus padarītu abstraktus, ir

jāieķeksē to dialogu logu lauks *Is Abstract*. Savukārt, lai norādītu, ka abstraktie elementi ir blokshēmas „īsto” elementu virstipi vēl papildus, ir jānovelk *Specialization* līnijas starp abstraktajiem un blokshēmas „īstajiem” elementiem. Rezultātā iegūtā blokshēmu redaktora konfigurācijas grafiskais modelis ir redzams 4.21. attēlā.



4.21. att. Blokshēmu redaktora konfigurācija

Kad konfigurācija ir pabeigta, ir jānospiež rīkjoslas poga *New Version Created* un pēc tam projekts ir jāsaglabā un jāaizver. Tā rezultātā ir izstrādāts blokshēmu redaktors, un, lai ar šo redaktoru varētu izveidot blokshēmu modeļus, ir jāizveido jauns blokshēmas redaktora projekts, t.i., veidojot jaunu projektu parādīsies 4.1. attēlā redzamais dialogu logs, kura laukā *Tools* ir jānorāda, ka rīks būs „BloskhēmuRedaktors”, bet laukā *Workspace* jānorāda projekta atrašanās vietā, un tad atvērsies blokshēmu redaktors, ar kuru var zīmēt blokshēmas.

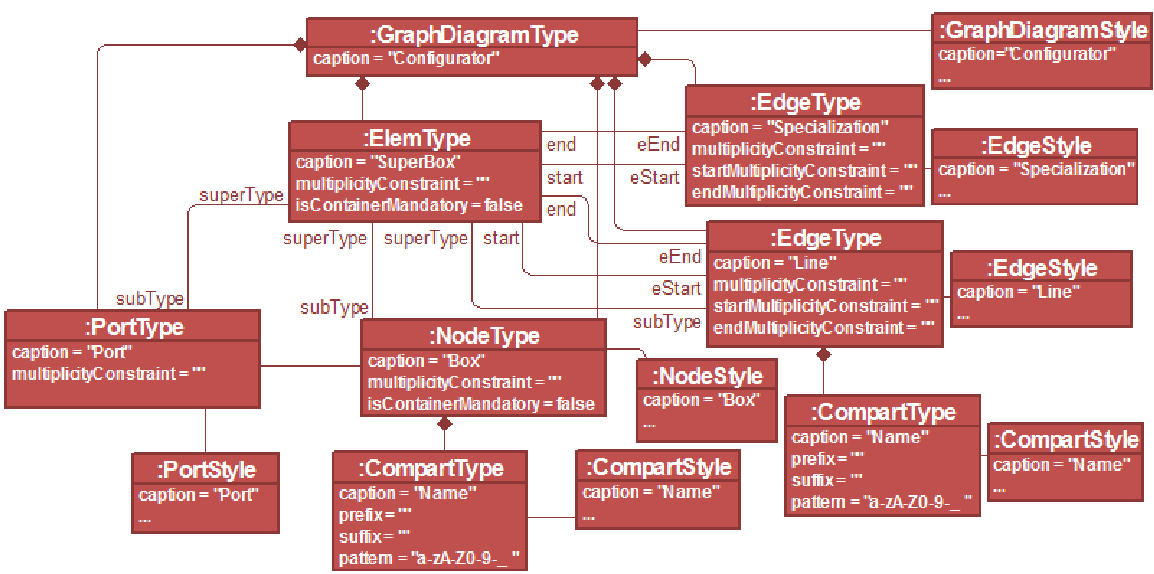
4.2 Konfiguratora realizācija

Konfigurators ir realizēts kā DSML rīks, un tā realizācijai ir izmantotas TDA komponentes, lietojot to pašu mehānismu, ko citu DSML rīku būvei. Proti, rīka specifikācijas uzdošanai ir izmantots rīku definēšanas metamodelis, kuru par strādājošu rīku pārvērš interpretators ar papildinātu funkcionalitāti (caur paplašinājuma punktiem).

4.2.1 Konfiguratora specifikācija

Kā 4.1. nodaļā tika minēts, konfiguratora valoda sastāv no četriem elementiem – *Box*, *Line*, *Port* un *Specialization*. Tādējādi, lai specificētu šos elementus, mums ir jāizveido tiem atbilstošās specifikācijas ar rīku definēšanas metamodeļa instancēm. Specifikācijas uzdošanu, līdzīgi kā iepriekš, sadalīsim divās daļās. Vispirms specificēsim konfiguratora valodu, un pēc tam rīka uzvedību.

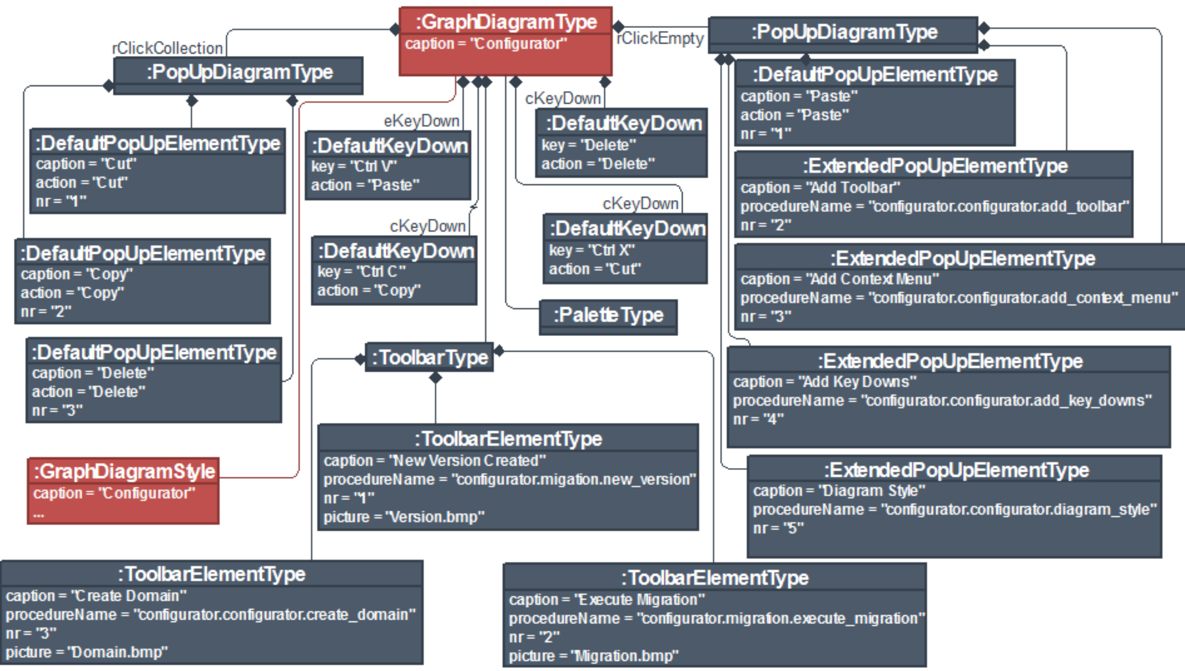
Konfiguratora valodu specificējošā instanču diagramma ir redzama 4.22. attēlā. Jāpiezīmē, ka bez minētajiem valodas elementiem šī specifikācija satur vienu abstraktu elementu „SuperBox” (klases *ElemType* instance), lai norādītu, ka *Line* un *Specialization* drīkst patvaļīgi savienot *Box*, *Line* un *Port* tipa elementus.



4.22. att. Konfiguratora valodas specifikācija

Nākamais solis ir specificēt katra valodas elementa un to diagrammas uzvedību. Sāksim ar diagrammu (skat. 4.23. attēlu). Konfiguratora diagrammās ir realizētas divu veidu uznirstošās izvēlnes. Viena izvēlne ir gadījumiem, ja lietotājs ar peles labo pogu nospiež uz vairāku elementu kolekcijas, un tā sastāv no operācijām – *Cut* (izgriezt), *Copy* (kopēt) un *Delete* (dzēst). Otra ir gadījumiem, kad lietotājs ar peles labo pogu nospiež uz tukšas vietas diagrammā, un tā sastāv no operācijām – *Paste* (ielīmēt), *Add Toolbar* (ļauj pievienot rīkjoslas pogas), *Add Context Menu* (ļauj pievienot uznirstošās izvēlnes), *Add Key Downs* (ļauj pievienot taustiņu kombināciju apstrādes) un *Diagram Style* (ļauj nomainīt diagrammas stilu). Savukārt, elementu kolekcijai ir piesaistītas šādas taustiņu kombinācijas - „Ctrl X”, „Ctrl C” un „Delete” (izpilda izgriešanas, kopēšanas un dzēšanas operācijas), bet gadījumam, kad neviens diagrammas elements nav aktīvs, ir piesaistīta „Ctrl V” (izpilda ielīmēšanas operāciju).

Konfiguratora diagrammu rīkjoslā ir specificētas trīs pogas. Pogas „New Version Created” un „Execute Migration” attiecas uz modeļu datu migrēšanu (par šīm pogām detalizētāk 4.2.3. nodaļā), bet poga „Create Domain” atver dialogu logu, kurā var norādīt transformācijas nosaukumu, kas pievienos domēna metamodeli.

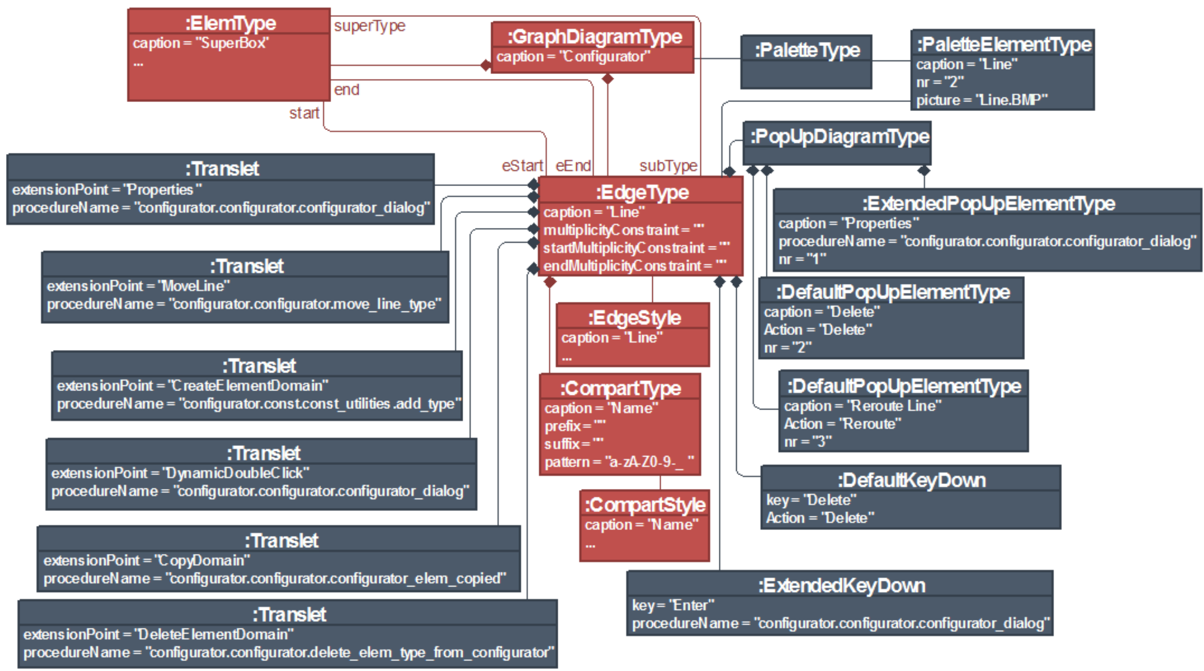


4.23. att. Konfiguratora diagrammas tipa specifikācija

Elementa *Line* specifikācija ir redzama 4.24. attēlā. Konfiguratora specifiskās funkcionalitātes nodrošināšanai *Line* ir piesaistītas sešas paplašinājumu punktu transformācijas. Paplašinājuma punktu *Properties* un *DynamicDoubleClick* transformācijas nodrošina dialogu loga (skat. 4.8. attēlu) izveidošanu, *CreateElementDomain* un *DeleteElementDomain* transformācijas attiecīgi nodrošina *Line* atbilstošo rīku definēšanas metamodeļa instanču izveidošanu un izdzēšanu, *MoveLine* transformācijas sinhronizē *Line* instanču stāvokli pēc līnijas gala pārcelšanas un *CopyDomain* transformācija nodrošina *Line* atbilstošo instanču kopēšanu.

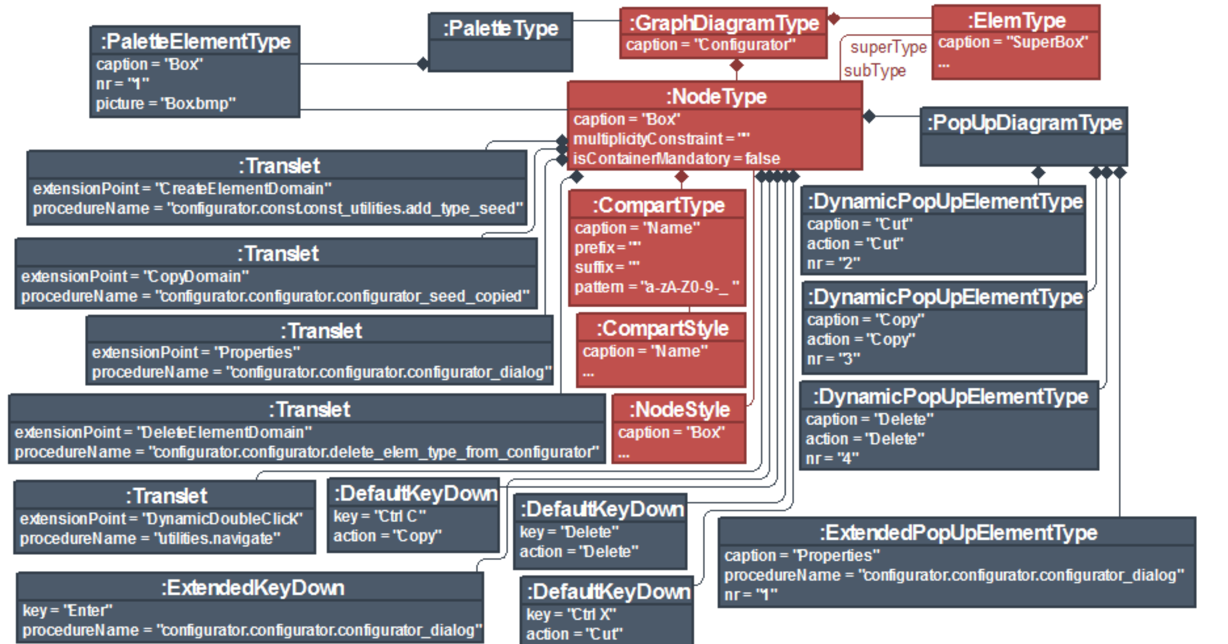
Line uznirstošā izvēlne sastāv no trīs operācijām *Delete* (dzēst), *Properties* (atvērt dialogu logu) un *Reroute Line* (pārzīmēt līniju), bet līnijai piesaistītās taustiņu kombinācijas ir „Delete” un „Enter” (izpilda dzēšanas un dialogu loga atvēršanas operācijas).

Par *Line* sākuma un beigu elementu ir norādīts abstraktais elements *SuperBox*, tādējādi nodrošinot, ka elementu *Line* ir iespējams visās kombinācijās savienot ar jebkuriem diviem *SuperBox* apakštipiem. Savukārt, lai konfiguratora diagrammā parādītu *Line* nosaukumu, elementam *Line* ir piekārtots atribūts *Name*.



4.24. *att. Line* specifikācija

Elementa *Box* specifikācija ir redzama 4.25. attēlā. Atskaitot paplašinājuma punkta *MoveLine* transformācijas, atlikušās piecas *Box* piesaistītās transformācijas ir tās pašas, kas elementam *Line*, un nodrošina to pašu funkcionalitāti.



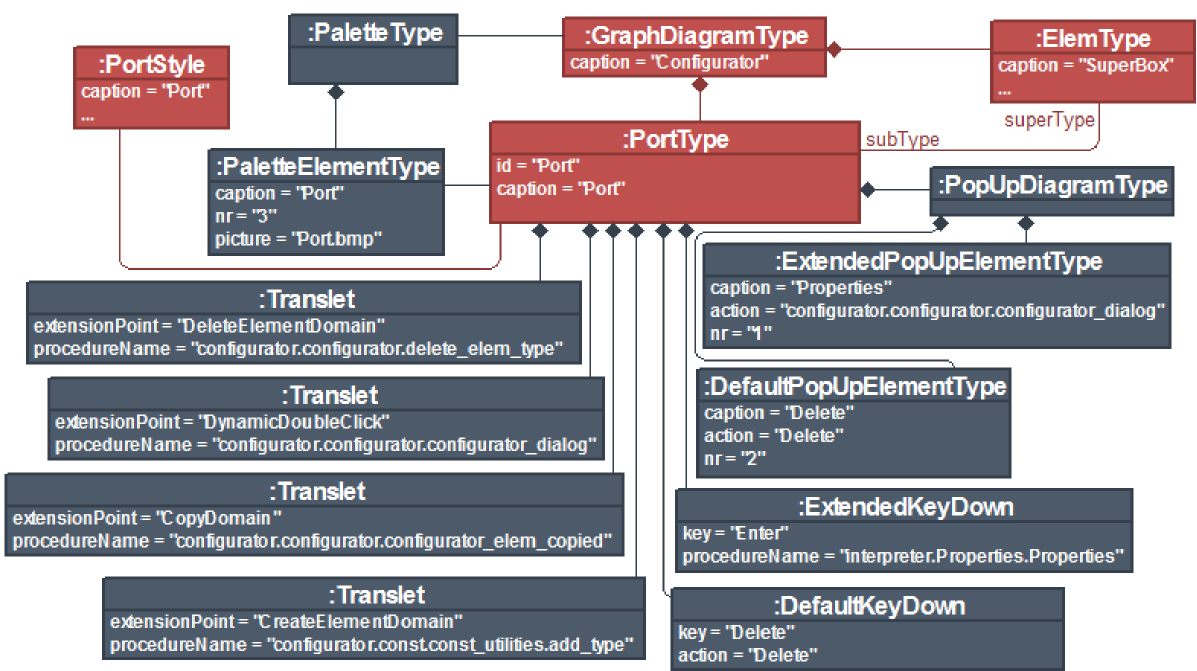
4.25. *att. Box* specifikācija

Box uznirstošā izvēlne sastāv no operācijām *Properties* (atvērt dialogu logu), *Cut* (izgriezt), *Copy* (kopēt) un *Delete* (dzēst), bet elementam piesaistītās taustiņu kombinācijas ir

„Enter”, „Ctrl X”, „Ctrl C” un „Delete” (attiecīgi izpilda atvērt dialogu logu, izgriezt, kopēt un dzēst operācijas).

Box ir norādīts par *SuperBox* apakštipu, tādējādi tas manto *SuperBox* ieejošās un izejošās līnijas (t.i., iespēju būt pievienotam *Line*). Līdzīgi kā *Line*, *Box* arī ir atribūts *Name* elementa nosaukuma rādīšanai.

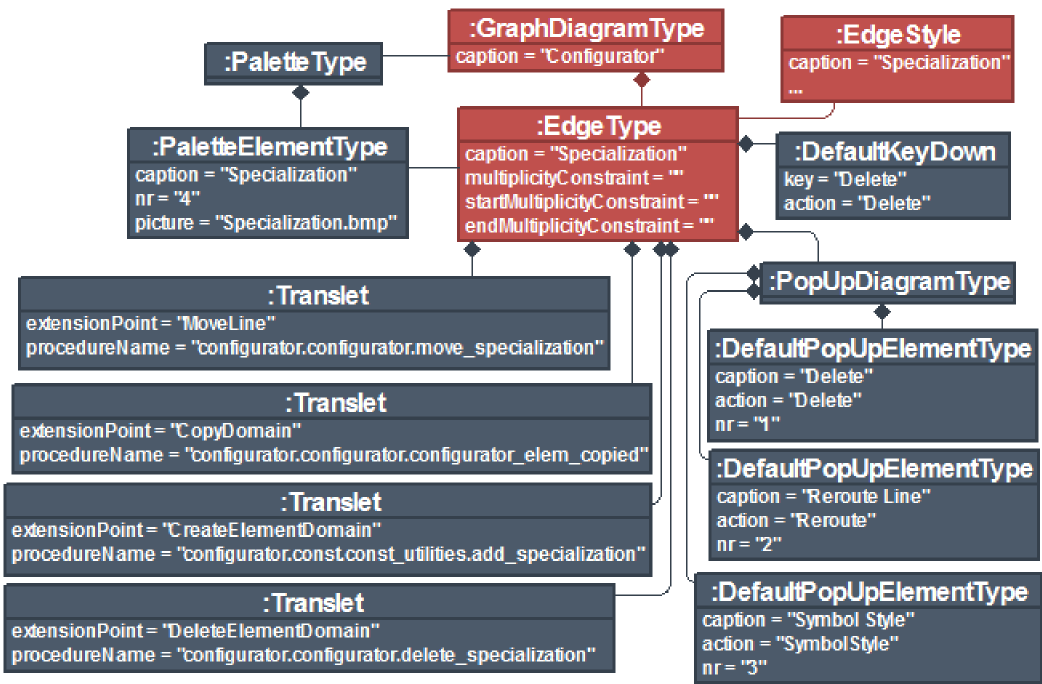
Elementa *Port* specifikācija ir redzama 4.26. attēlā. Šim elementam ir piesaistītas tās pašas paplašinājuma punktu transformācijas, kas elementam *Box*. *Port* uznirstošā izvēlne sastāv no operācijām *Properties* (atvērt dialogu logu) un *Delete* (dzēst), bet elementam piesaistītās taustiņu kombinācijas ir - „Enter” un „Delete” (attiecīgi izpilda atvērt dialogu logu un dzēst operācijas). Līdzīgi kā *Box*, *Port* ir *SuperBox* apakštips, lai varētu mantot *SuperBox* ieejošās un izejošās līnijas.



4.26. att. Port specifikācija

Elementa *Specialization* specifikācija ir redzama 4.27. attēlā. Šim elementam ir piesaistītas tikai četras no minētajām sešām *Line* paplašinājuma punktu transformācijām (nav *Properties* un *DynamicDoubleClick*, jo *Specialization* elementam nav dialogu logu).

Specialization uznirstošā izvēlne sastāv no trīs operācijām *Delete* (dzēst), *Reroute Line* (pārzīmēt līniju) un *Symbol Style* (uzstādīt simbola stilu), bet elementam piesaistīta ir tikai viena taustiņu kombinācija „Delete”, kas atbilst dzēšanas operācijai.

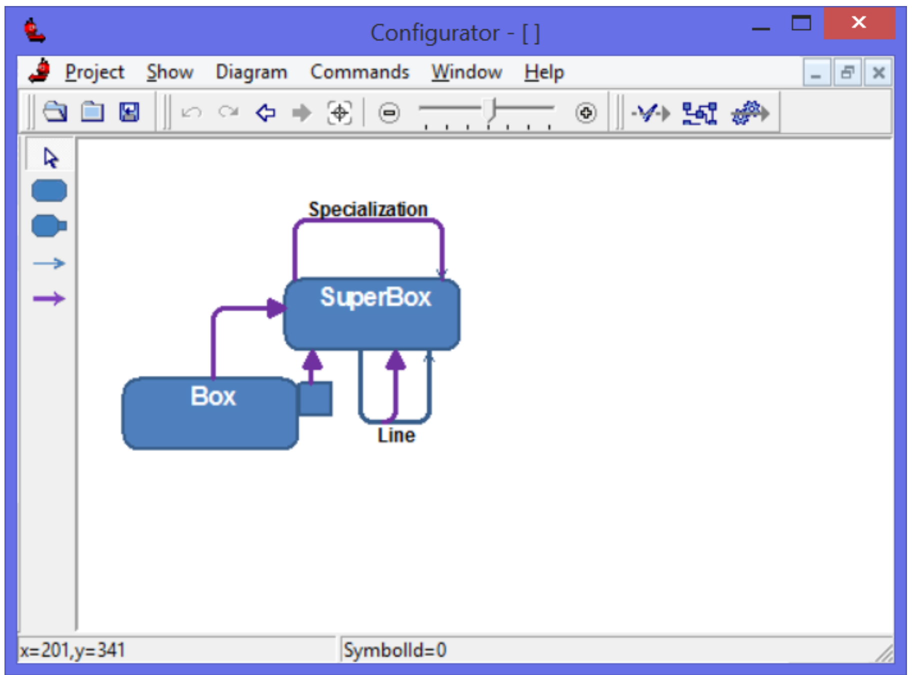


4.27. att. Specialization specifikācija

Pilna konfiguratora specifikācija ar rīku definēšanas metamodeļa instancēm ir redzama 4.28. attēlā.

4.28. att. Pilna konfiguratora specifikācija

Tā kā konfigurators ir specificēts ar rīku definēšanas metamodeļa instanci, šo pašu specifikāciju var uzdod ar konfiguratoru, respektīvi, ar konfiguratoru ir iespējams specificēt pašu konfiguratoru (sāknēšanas metode [36]), un šādas specifikācijas atbilstošais grafiskais modelis ir redzams 4.29. attēlā.



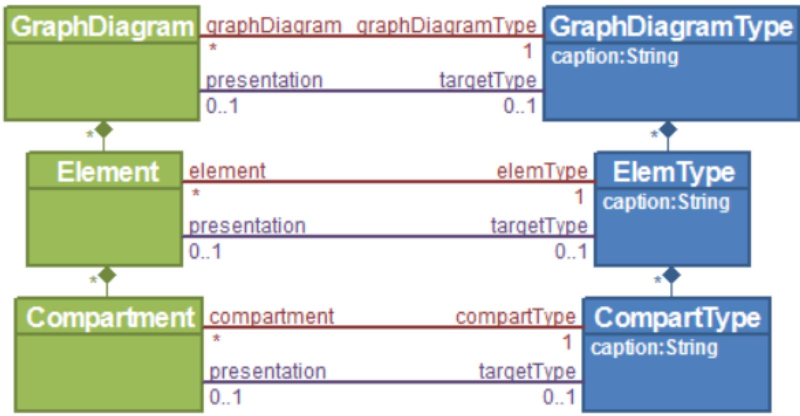
4.29. att. Konfiguratora specifikācijas grafiskais modelis ar konfiguratoru

4.2.2 Interpretatora funkcionalitātes papildinājumi

Nākamais solis pēc specifikācijas uzdošanas ir doto specifikāciju interpretēt. Šajā gadījumā realizētā interpretatora funkcionalitāte ir nepietiekoša, tādēļ tā ir papildināta ar sešām transformācijām šādos paplašinājuma punktos – *Properties*, *DynamicDoubleClick*, *CreateElementDomain*, *DeleteElementDomain*, *CopyDomain* un *MoveLine*, un šajā nodaļā detalizēti apskatīsim katras transformācijas rezultātu.

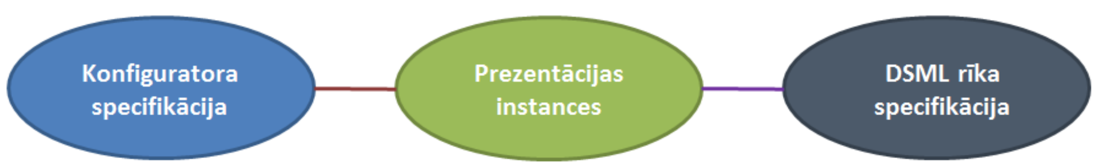
4.2.2.1 Realizācijas metamodeļa papildinājumi

Pirms apskatām minēto transformāciju rezultātus, realizācijas metamodeli papildināsim ar trīs sastatņu asociācijām *presentation-targetType* starp klasēm *GraphDiagram* un *GraphDiagramType*, *Element* un *ElemType*, *Compartment* un *CompartType*. Papildinātais realizācijas metamodeļa fragments ir redzams 4.30. attēlā.



4.30. att. Papildinātais realizācijas metamodeļa fragments

Šo asociāciju mērķis ir norādīt, ka prezentācijas metamodeļa instance specificē kādu jaunā rīka tipu instanci. Shematisks repozitorija stāvoklis pēc rīka specificēšanas ar konfiguratoru ir redzams 4.31. attēlā.



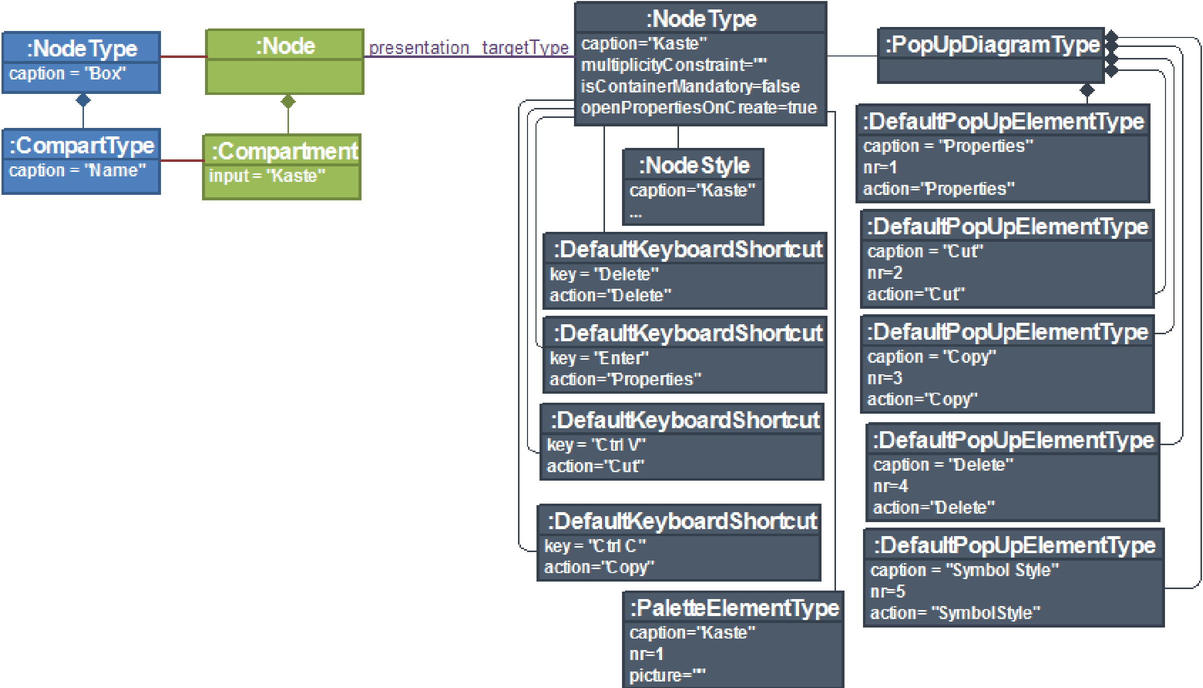
4.31. att. Shematiski attēlots konfigurācijas rezultāts

4.2.2.2 Konfiguratora paplašinājuma punktu transformācijas

4.2.2.2.1 Paplašinājuma punkts CreateElementDomain

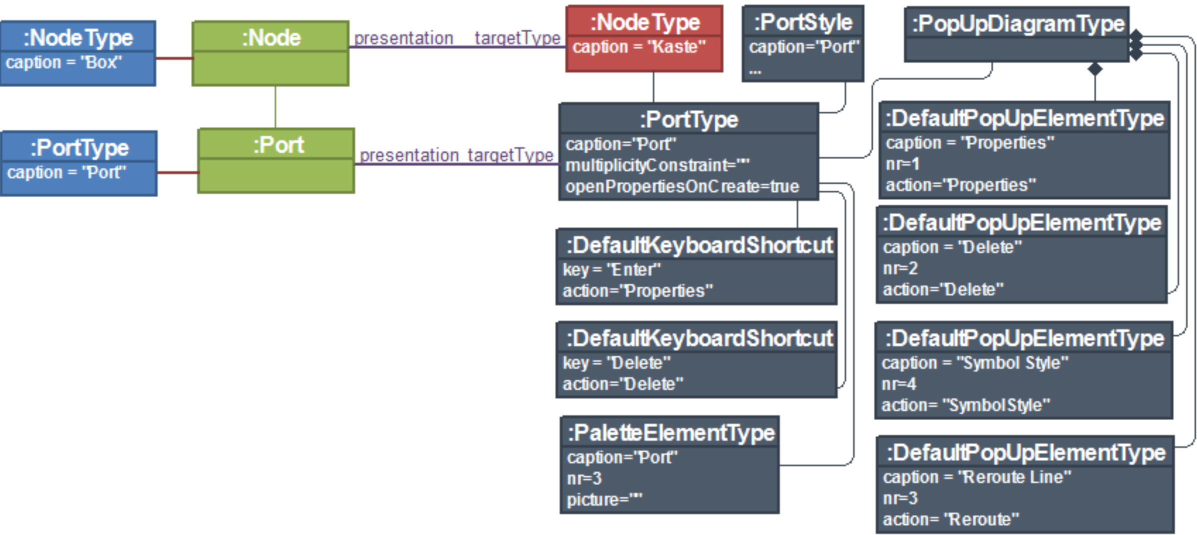
Šī paplašinājuma punkta transformācija tiek izpildīta, kad tiek veidots jauns elements, jeb precīzāk sakot, kad universālā transformācija repozitorijā ir izveidojusi prezentācijas metamodeļa instances (skat. 3.9. attēlu), un tās uzdevums ir izveidot elementam atbilstošās rīku definēšanas metamodeļa instances (šajā gadījumā rīku definēšanas metamodelis kalpo arī par domēna metamodeli). Tā kā konfiguratorā ir četru veidu elementi, tad katram elementam izveidotās instances apskatīsim detalizētāk.

Gadījumā, ja tiek veidots jauns *Box* elements ar nosaukumu „Kaste”, tad paplašinājuma punkta transformācija universālās transformācijas izveidotajai prezentācijas metamodeļa *Node* instancei caur *presentation-targetType* saiti pievieno 4.32. attēlā redzamās rīku definēšanas metamodeļa instances. Vispirms paplašinājuma punkta transformācija izveido *NodeType* un *NodeStyle* instances, kuras definē pašu elementu un tā noklusēto stilu, pēc tam tā definē elementa uzvedību, un pēc noklusējuma transformācija izveido paletes elementam, uznirstošajai izvēlnei un taustiņu kombinācijām atbilstošās instances.



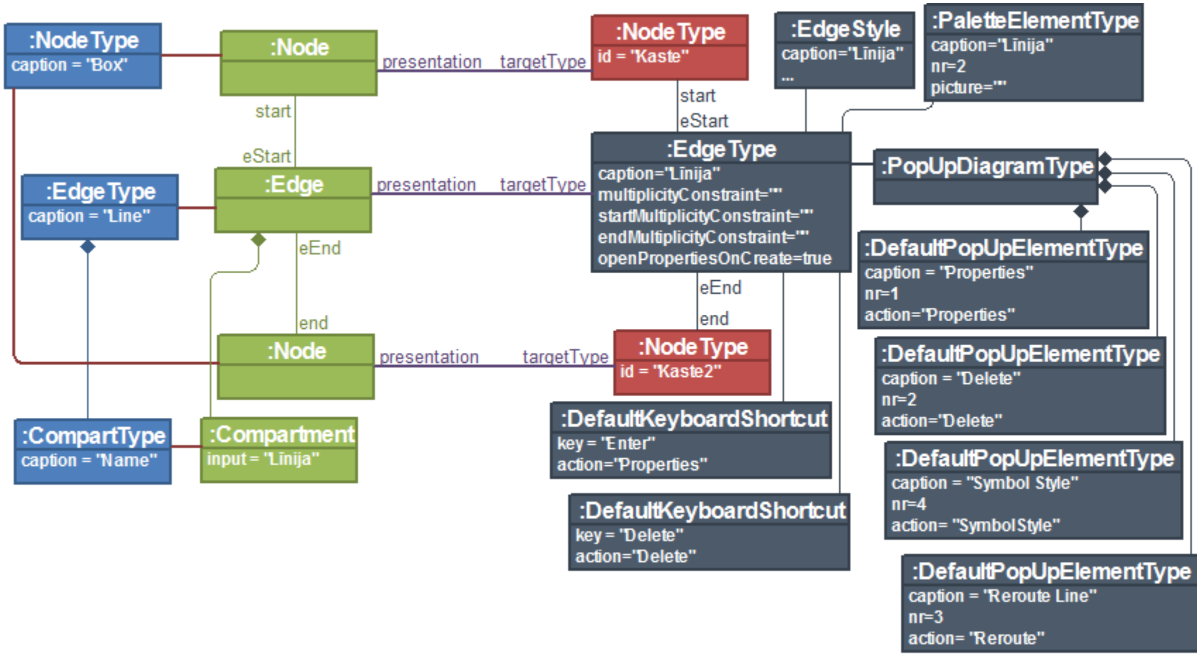
4.32. att. „Kaste" specifikācija ar rīku definēšanas metamodeļa instancēm

Gadījumā, ja tiek veidots jauns *Port* elements, tad transformācija atbilstošajai prezentācijas metamodeļa *Port* instancei caur *presentation-targetType* saiti pievieno 4.33. attēlā redzamās rīku definēšanas metamodeļa instances. Jāpiezīmē, ka porti tipa elementi nevar būt „paši par sevi", tiem ir vienmēr jābūt piesaistītiem pie kādas kastes (t.i., *Node* instances). Līdzīgi kā iepriekš, vispirms transformācija izveido *PortType* un *PortStyle* instances, lai definētu elementu un tā stilu, bet pēc tam elementa uzvedību definējošās instances.



4.33. att. „Port" specifikācija ar rīku definēšanas metamodeļa instancēm

Gadījumā, ja tiek veidots jauns *Line* elements ar nosaukumu „Līnija”, tad transformācija atbilstošajai prezentācijas metamodeļa *Edge* instancei caur *presentation-targetType* saiti pievieno 4.34. attēlā redzamās rīku definēšanas metamodeļa instances. Jāpiezīmē, ka arī līnijas tipa elementi nevar būt „pašas par sevi”, tiem ir vienmēr jāsavieno divi diagrammas elementi, piemēram, divas kastes (t.i., *Node* instances). Līdzīgi kā iepriekš, vispirms transformācija izveido *EdgeType* un *EdgeStyle* instances, lai definētu elementu un tā stilu, bet pēc tam elementa uzvedību definējošās instances.



4.34. att. „Līnija” specifikācija ar rīku definēšanas metamodeļa instancēm

4.2.2.2.2 Paplašinājuma punkti Properties un L2Click

Šajos paplašinājuma punktos tiek radīti konfiguratora elementiem specifiski izstrādāti dialogu logi (logu piemēri redzami 4.4. un 4.8. attēlos) valodas elementu un to uzvedības definēšanai. Logiem ir tieša atbilstība ar rīku definēšanas metamodeli, un katras izmaiņas logā nozīmē izmaiņas šī metamodeļa instancēs. Piemēram, ja laukā *Caption* ievada nosaukumu „Kaste”, tad atbilstošās *ElemType* instances atribūtam *caption* tiek uzstādīta vērtība „Kaste”, vai, ja pārslēdzās un cilni *Translets* un kādā no tās laukiem tiek ievadīts transformācijas nosaukums, tad repozitorijā tiek izveidota *Translet* instance ar attiecīgajām atribūtu *extensionPoint* un *procedureName* vērtībām, un pēc tam šī instance tiek piesaistīta atbilstošajai *ElemType* instancei.

Tādējādi šādā veidā tiek „apsaimniekotas" rīku definēšanas metamodeļu instances un stingri ievērota atbilstība starp konfiguratora modeļiem un rīku definēšanas metamodeli, kurš šajā gadījumā kalpo arī par konfiguratora datu struktūru.

4.2.2.2.3 Paplašinājuma punkts DelteteElementDomain

Šī paplašinājuma punkta transformācija tiek izpildīta elementa dzēšanas laikā un tās uzdevums ir izdzēst prezentācijas instancei (pa *targetType*) piesaistītās rīku definēšanas metamodeļa instances.

4.2.2.2.4 Paplašinājuma punkts CopyDomain

Šī paplašinājuma punkta transformācija tiek izpildīta elementa kopēšanas laikā, un tās uzdevums ir uzģenerēt transformācijas kodu, kas rada prezentācijas instancei (pa *targetType*) piesaistītās rīku definēšanas metamodeļa instances.

4.2.2.2.5 Paplašinājuma punkts MoveLine

Šī paplašinājuma punkta transformācija tiek izpildīta, kad tiek pārcelts kāds no līnijas galiem, un šī transformācija ir piekārtota abiem līnijas tipa elementiem *Line* un *Specialization*. Šīs transformācijas uzdevums ir veikt atbilstošās izmaiņas rīku definēšanas metamodeļa instancēs, respektīvi, atbilstošajai *EdgeType* instancei nomainīt tai pa lomu *start* vai *end* sasniedzamo *ElemType* instanci.

4.3 Rīka versiju un modeļu migrēšana

Pieredze rāda, ka DSML rīku izstrāde ir iteratīvs process. Tas nozīmē, ka vispirms tiek izstrādāta viena rīka versija, pēc tam tā tiek papildināta un iegūta otrā versija, tad trešā, ceturtā, utt. līdz iegūst vēlamo rezultātu. Šādas izstrādes problēma ir tā, ka katrā solī ar vienu rīka versiju tiek izveidoti modeļi, kurus nepieciešams lietot arī katrā nākamajā rīka versijā. Tas nozīmē, ka platformā ir nepieciešams mehānisms, kas ļauj modeļu datus, kas izstrādāti ar vecāku rīka versiju, pārmigrēt uz jaunākām rīka versijām.

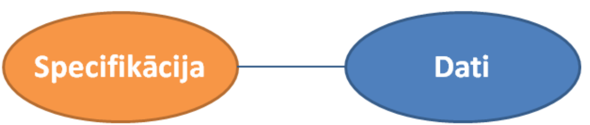
4.3.1 Problēmas nostādne

Vispirms precizēsim, ka ar „rīku" un „rīka versija" mēs saprotam tā atbilstošo specifikāciju ar rīku definēšanas metamodeļa instanci, jeb citiem vārdiem sakot, konkrēta rīka versija ir viena šī metamodeļa instance, bet ar „modeļu datiem" mēs saprotam prezentācijas metamodeļa instances. Tādējādi „izstrādāt jaunu rīka versiju" nozīmē izveidot jaunu rīku definēšanas metamodeļa instanci, bet „migrēt modeļu datus no vecākas rīka versijas uz jaunāku" nozīmē panākt situāciju, ka modeļu dati ir piesaistīti jaunajai rīka specifikācijai, jeb metamodeļa terminos runājot, prezentācijas instances ir piesaistītas jaunajai rīku definēšanas metamodeļa

instancei. Lai šādu situāciju panāktu, ir iespējami divi scenāriji. Viens scenārijs ir veco rīka specifikāciju modificēt uz jauno, bet otrs ir aizstāt veco specifikāciju ar jauno.

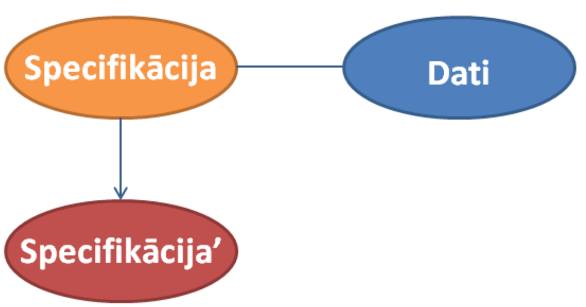
Apskatīsim katru no šiem scenārijiem detalizētāk. Lai varētu izpildīt modificēšanas scenāriju, mums ir jāizmanto divi apstākļi. Pirmkārt, rīku evolūcija notiek par bāzi ņemot kādu jau izstrādātu rīka versiju, no kuras, secīgi izpildot kaut kādu darbību ķēdi, iegūst rīka nākamo versiju. Otrkārt, rīku specificēšana notiek caur konfiguratoru, līdz ar to mums ir iespējams šo darbību ķēdi saglabāt. Tādējādi, izmantojot šos divus apstākļus, mēs varam ierakstīt visas rīka izstrādātāja izpildītās darbības, kuras no vienas rīka versijas ļāva iegūtu nākamo, jeb, precīzākos terminos runājot, iegūt deltu starp rīka veco un jauno versiju, un pēc tam no iegūtās deltas uzbūvēt transformāciju, kas veco rīka versiju pārveido par jauno.

Lai aprakstītu aizstāšanu scenāriju, apskatīsim to pa soļiem. Kopumā šis process sastāv no diviem soļiem, un sākotnējais repozitorija stāvoklis pirms migrēšanas atbilst „normālam" rīka stāvoklim, t.i., repozitorijā ir rīka specifikācija un modeļu dati, kā tas shematiski ir attēlots 4.35. attēlā.



4.35. att. Stāvoklis pirms migrēšanas

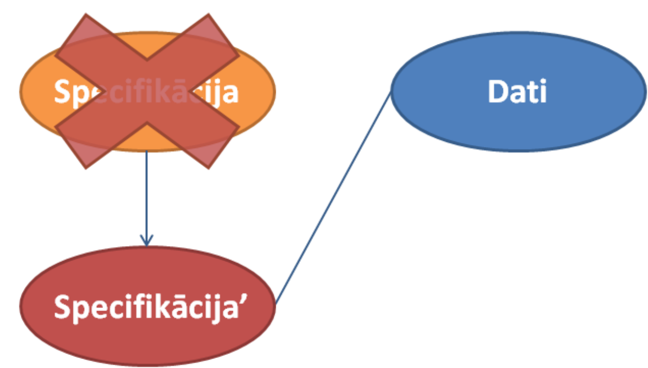
Pirmajā solī repozitorijā ir jāizveido jaunā rīku definēšanas metamodeļa instance, kā rezultātā repozitorijā vienlaicīgi atrodas vecā un jaunā rīku definēšanas metamodeļa instance, bet prezentācijas metamodeļa instance vēl joprojām ir sasaistīta ar veco rīku definēšanas metamodeļa instanci. Pirmā migrēšanas soļa rezultāts atbilst 4.36. attēlā redzamajam.



4.36. att. Stāvoklis pēc jauno tipu ielādes

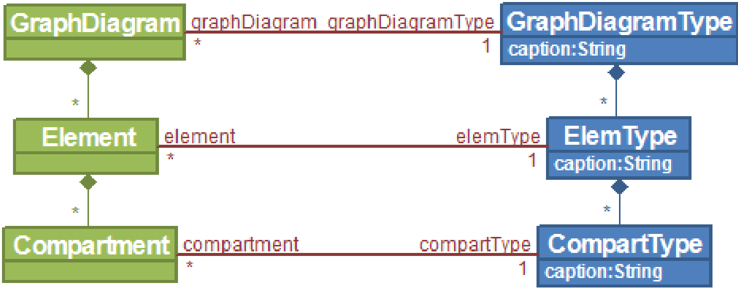
Otrajā solī prezentācijas metamodeļa instance vispirms ir jāatsaista no vecās rīku definēšanas metamodeļa instances, bet pēc tam jāpiesaista jaunajai, un, kad tas ir izdarīts,

vecā rīku definēšanas metamodeļa instance ir jāizdzēš, iegūstot, ka repozitorija stāvoklis pēc migrēšanas beigām ir atbilstoši 4.37. attēlam (dati pēc migrēšanas netiek pārrēķināti).



4.37. att. Stāvoklis pēc migrēšanas

Salīdzinot abas pieejas, redzam, ka modificēšanas scenārijs paredz pilnīgi visu darbību ierakstīšanu, kas tika veiktas, lai no vienas versijas iegūtu otru. Citiem vārdiem sakot, mums ir jāfiksē katra rīku definēšanas metamodeļa instanču izmaiņa, un, ņemot vērā rīku definēšanas metamodeļa apjomu, šī scenārija realizācija būtu tehniski sarežģīta. Savukārt, izpildot aizstāšanas scenāriju, ir jāspēj tikai pārmigrēt saites starp modeļu datiem un jauno specifikāciju. Tā kā rīku definēšanas metamodeli un prezentācijas metamodeli saista vien trīs sastatņu asociācijas (skat. 4.38. attēlu), tad izsekot izmaiņām, kas attiecas uz minēto asociāciju saišu atjaunošanu, tehniski ir ievērojami vieglāk realizēt, nekā izsekot katrai rīku definēšanas metamodeļa instances izmaiņai, un tāpēc aizstāšanas scenārijs ir izvēlēts par realizācijas variantu.



4.38. att. Rīku definēšanas metamodeļa un prezentācijas metamodeļa sasaiste

4.3.2 Migrēšanas mehānisms

Kā iepriekš tika minēts, rīku evolūcija notiek par bāzi ņemot kādu jau izstrādātu rīka versiju, no kuras, secīgi izpildot kādu darbību ķēdi, tiek iegūta rīka nākamā versija. Ņemot vērā, ka rīku definēšanas metamodeli un prezentācijas metamodeli saista vien trīs asociācijas, proti, 4.38. attēlā ir parādīts, ka saistību starp prezentācijas metamodeli un rīku definēšanas

metamodeli nodrošina asociācijas *graphDiagram-graphDiagramType*, *element-elemType* un *compartment-compartType*, tad rīka migrēšanas uzdevumu mēs varam reducēt uz šo trīs asociāciju saišu atjaunošanu, jo visa sarežģītība ir izpildīt migrēšanas otro soli, kad pirmajā solī izveidotās rīku definēšanas metamodeļa instances ir jāsasaista ar esošo modeļu datiem. Tā kā tikai klases *GraphDiagramType*, *ElemType* un *CompartType* vienlaicīgi ir piesaistītas minētajām asociācijām un apraksta rīka specifikāciju, tad, lai atjaunotu minēto asociāciju saites, tieši šo klašu instanču izmaiņām mums ir jāseko līdzi (jeb, precīzāk sakot, šo klašu atribūta *caption* vērtību maiņām).

Aplūkojot, kādas vispār darbības var veikt ar šo klašu instancēm (lai iegūtu nākamo rīka versiju), varam redzēt, ka tādas ir trīs – radīt, dzēst un mainīt, un, lai realizētu migrēšanu, mums ir jāapskata, kā katra no šīm darbībām ietekmē migrēšanas procesu. Citiem vārdiem sakot, mums ir nepieciešams apskatīt, kas mums būtu jādara, lai pārmigrētu saites starp prezentācijas metamodeļa instancēm un jaunās versijas rīku definēšanas metamodeļa instancēm, kad tiek izpildīta katra no minētajām darbībām.

Gadījumā, ja notiek jaunas instances radīšana (t.i., tiek radīta klases *GraphDiagramType*, *ElemType* vai *CompartType* instance), piemēram, tiek radīta jauna instance „A”, tad attiecībā uz saišu pārmigrēšanu, mums būtu jāmeklē instancei „A” atbilstošā instance vecajā rīka specifikācijā, un tā kā tādas nav, tad nav arī prezentācijas instanču, ko pārmigrēt. Tā rezultātā jaunas instanču pievienošanas gadījumā attiecībā uz saišu pārmigrēšanu nekas nav jādara.

Gadījumā, ja instance tiek dzēsta, piemēram, tiek dzēsta kāda instance „B”, tad vecā specifikācija satur instanci „B”, bet jaunā to nesatur. Tādā gadījumā rīkos, kas ir izstrādāti ar jauno rīka specifikāciju, nevar izveidot „B” tipa elementus un, tādēļ, šī migrēšanas mehānisma ietvaros ir pieņemts, ka migrēšanas laikā „B” instancei atbilstošās prezentācijas instances ir jāizdzēš.

Gadījumā, ja instance tiek mainīta (t.i., tiek mainīts tās nosaukums), piemēram, kāda instance „C” tika pārsaukta par „D”, tādā gadījumā, lai varētu veikt saišu pārmigrēšanu, mums vecajā rīka specifikācijā ir jāsameklē instance „C” un visas tai piesaistītās prezentācijas instances, un tad prezentācijas instances jāpiesaista jaunās specifikācijas instancei „D”. Tādējādi, katru reizi, kad tiek izpildīta mainīšanas darbība, mums ir jāsaglabā informācija par to, kā atrast atbilstošo instanci vecajā rīka specifikācijā, t.i., mums ir jāuztur atbilstību saraksts, kas satur veco un jauno instanču nosaukumus.

Minēto instanču izmaiņu saglabāšanas (t.i., atbilstību saraksta veidošanas) kontekstā, ir jāatzīmē, ka katru klašu *GraphDiagramType*, *ElemType* un *CompartType* instanci var unikāli identificēt pēc atribūta *caption* vērtībām. Precīzāk sakot, vienā rīkā var būt realizēti vairāki diagrammu tipi un katram diagrammas tipam atbilst *GraphDiagramType* instance ar unikālu

caption vērtību. Katra tipa diagramma satur dažādus elementu tipus, kur katram elementa tipam atbilst *ElemType* instance ar unikālu *caption* vērtību, respektīvi, elementu tipu *caption* vērtību unikalitātes apgabals ir tās diagrammas tips, kurai tas pieder. Līdzīgi ir ar atribūtu tipiem. Katram atribūta tipam atbilst *CompartType* instance un tās *caption* vērtībai ir jābūt unikālai atribūta vecāka apgabalā.

Uzskatāmības labad apskatīsim piemēru, kur jau iepriekš redzētajā blokshēmu redaktorā sameklēsim starta simbolu specificējošo instanci. Tādā gadījumā mums vispirms ir jāsameklē *GraphDiagramType* instance ar *caption* vērtību „Blokshēma" un pēc tam no šīs instances atbilstošā *ElemType* instance ar *caption* vērtību „Starts".

Tā kā katra instance ir unikāli identificējama, tad mūsu atbilstību sarakstā mēs varam viennozīmīgi saglabāt informāciju par to, kāda bija instances vecā *caption* vērtība un kāda ir jaunā. Jāpiezīmē, ka šī informācija tiek saglabāta pie katras *caption* vērtību maiņas, bet, ja *caption* vērtība tiek mainīta vairākas reizes vienai instancei, piemēram, ja sākotnējā vērtība bija „A", pēc tam to pārsauca par „B" un vēl pēc tam par „C", tad saglabātas tiek tikai pēdējās izmaiņas, respektīvi, ka „A" tika pārsaukta par „C" (mūs interesē tikai vecā un jaunā *caption* vērtība).

Tātad saglabātais atbilstību saraksts mums viennozīmīgi ļauj noteikt atbilstību starp veco un jauno specifikāciju instancēm, un, izmantojot šo informāciju, mēs varam sākt migrēšanas procesu, t.i., pirmajā solī repozitorijā ir jāizveido jaunās specifikācijas instances, bet otrajā ir jāatsaista prezentācijas metamodeļa instances no vecās specifikācijas instancēm un jāpiesaista jaunajām.

Pirmais solis ir tehniski vienkāršs, tāpēc apskatīsim tikai otro soli. Sāksim ar *GraphDiagramType* instanču pārmigrēšanu, proti, vispirms no atbilstību saraksta iegūsim jauno un veco diagrammu tipu *caption* vērtības. Pēc tam „vecajos" tipos sameklē *GraphDiagramType* instanci ar veco *caption* vērtību un „jaunajos" tipos sameklē *GraphDiagramType* instanci ar jauno *caption* vērtību. Tad no „vecās" *GraphDiagramType* instances sameklē (pa saiti *graphDiagramType-graphDiagram*) visas tai atbilstošās *GraphDiagram* instances. Iegūtās *GraphDiagram* instances atsaista no „vecās" *GraphDiagramType* instances un piesaista „jaunai" *GraphDiagramType* instancei. Tā rezultātā ir nomigrētas visas viena tipa diagrammas.

Nākamais solis ir apstrādāt šo diagrammu tipu elementu tipus. To apstrāde ir identiska ar diagrammu tipu apstrādi, proti, tiek iegūtas jauno un veco elementu tipu *caption* vērtības, pēc tam no „vecās" *GraphDiagramType* instances tiek sameklēta „vecā" *ElemType* instance un no „jaunās" *GraphDiagramType* instances tiek sameklēta „jaunā" *ElemType* instance. Tad no „vecās" *ElemType* instances sameklē (saite *elemType-element*) *Element* instances, un *Element*

instances atsaista no „vecās” *ElemType* instances un piesaista „jaunajai” *ElemType* instancei. Rezultātā ir nomigrēti visi viena tipa elementi.

Pēc identiska scenārija rīkojas ar atribūtiem. No „vecās” *ElemType* instances sameklē „veco” *CompartType* instanci un no „jaunās” *ElemType* instances „jauno” *CompartType* instanci un pēc tam pārsaista *Compartment* instances.

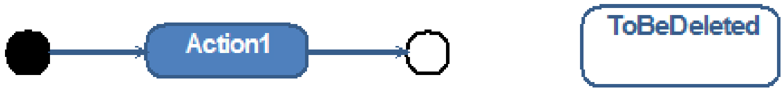
Kad ir pabeigta modeļu datu migrēšana, vajag atbrīvoties no vecajām tipu instancēm. Pievērsīsim uzmanību, ka prezentācijas metamodeļa instances tiek pārsaistītas, ja ir atrasts gan „jaunais”, gan „vecais” tips. Gadījumā, ja rīka jaunajā versijā kāds tips ir izdzēsts, tad tā prezentācijas instances ir piesaistītas „vecajam” tipam un no tā netika atsaistītas. Tādēļ dzēšot „vecos” tipus tiek dzēstas arī to atbilstošās prezentācijas metamodeļa instances, jo nepārsaistītas ir palikušas vienīgi izdzēstajiem tipiem atbilstošās prezentācijas instances.

4.3.3 Modeļu migrēšanas piemērs

Migrēšanas procesu aplūkosim uz blokshēma redaktora piemēra. Pieņemsim, ka mēs ar konfiguratoru esam izveidojuši blokshēmu redaktora versiju, kurā visi elementu nosaukumi ir angļu valodā, un rīka nākamajā versijā mēs gribam, lai šie nosaukumi būtu latviešu valodā. Tādā gadījumā, atbilstoši iepriekš aprakstītajai modeļu migrēšanas shēmai, angliskā redaktora versija kļūst par bāzes versiju, no kuras, veicot rīku definēšanas metamodeļa instanču pārsaukšanu, tiek radīta blokshēmu redaktora latviskā versija, proti, tiek izpildītas vairākas datu mainīšanas darbības.

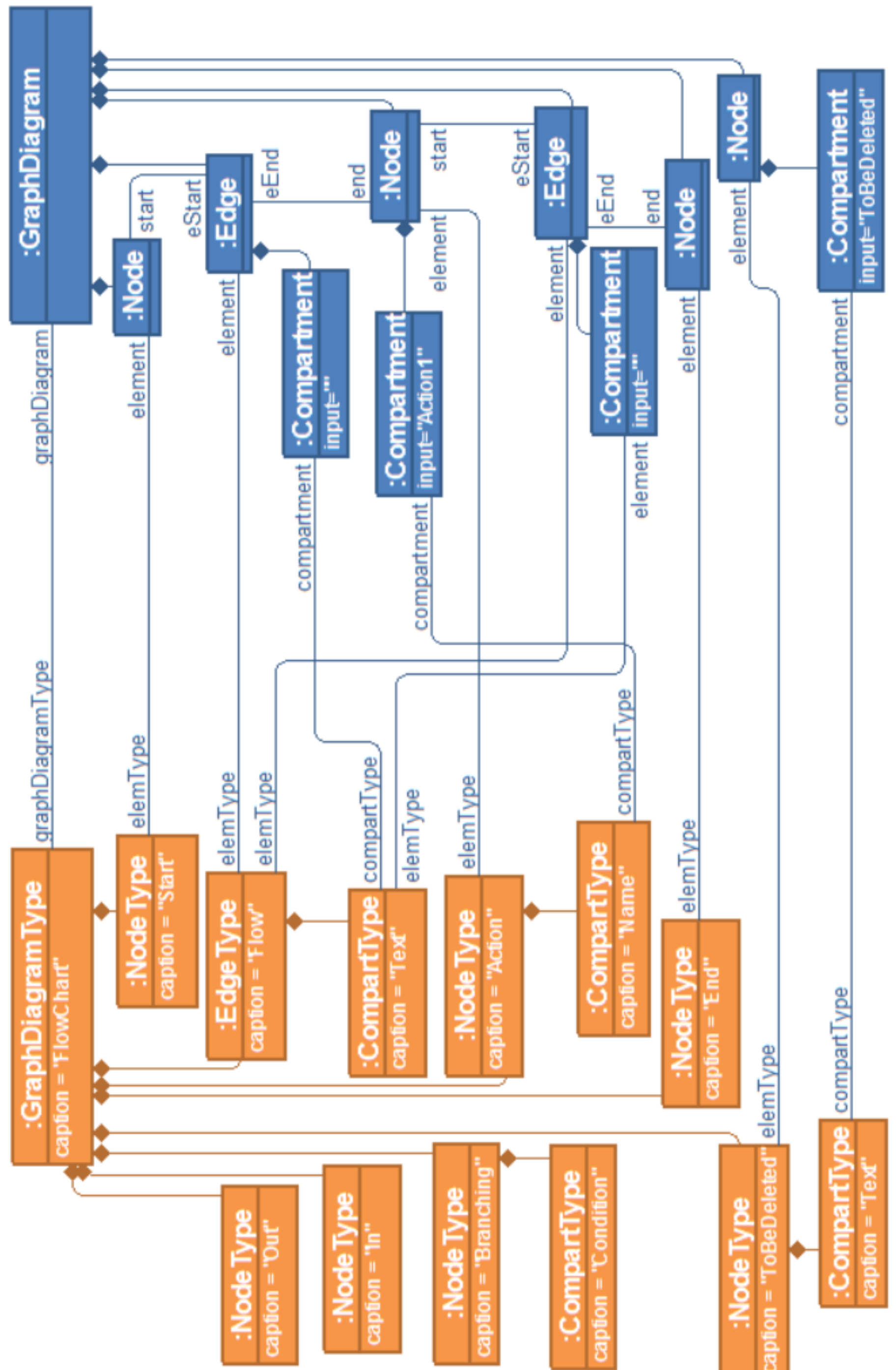
Lai nodemonstrētu modeļu datu migrēšanu arī ar dzēšanas un pievienošanas darbībām, blokshēmu redaktora valoda ir papildināta ar „īstajai” valodai nepiederošiem elementiem. Dzēšanas operācijas demonstrēšanas nolūkos rīka angliskajā versijā ir izveidots elements „ToBeDeleted”, kurš latviskajā rīka versijā būs izdzēsts ar visām tam atbilstošajām instancēm. Pievienošanas operācijas demonstrēšanas nolūkos latviskajā rīka versijā ir pievienots jauns elements „JaunsElements”.

Kā migrējamos modeļu datus ņemsim blokshēmu, kura sastāv no viena starta, viena aktivitātes, viena beigu simbola un viena „ToBeDeleted” elementa, kā arī diviem plūsmas simboliem. Šīs blokshēmas grafiskā reprezentācija ir redzama 4.39. attēlā.



4.39. att. Blokshēmas datu piemērs

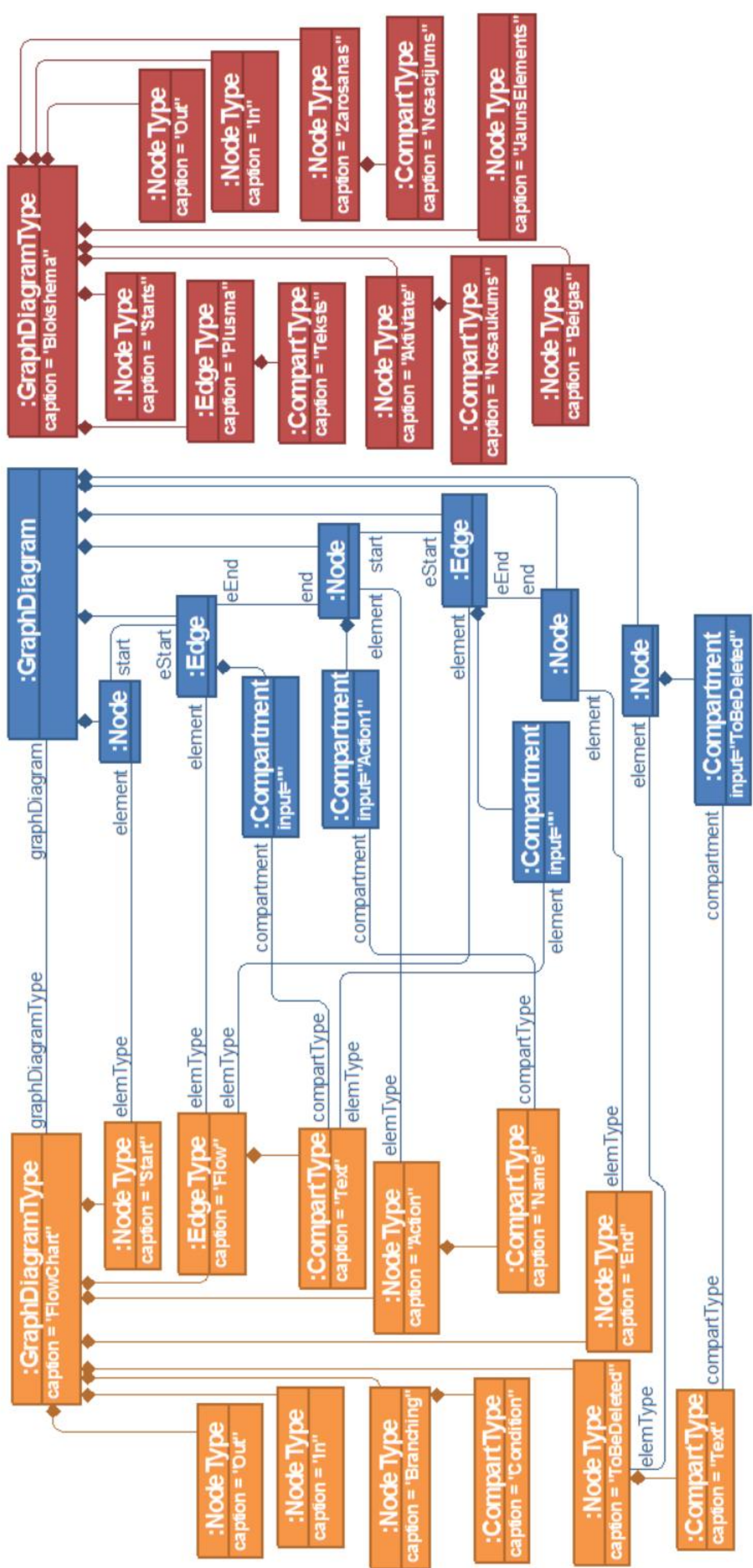
Šīs pašas blokshēmas stāvoklis repozitorijā ar atbilstošajām rīku definēšanas metamodeļa instancēm ir attēlots 4.40. attēlā (shematiski šī situācija atbilst 4.35. attēlam).



4.40. att. Blokshēmu redaktora modelis pirms migrēšanas

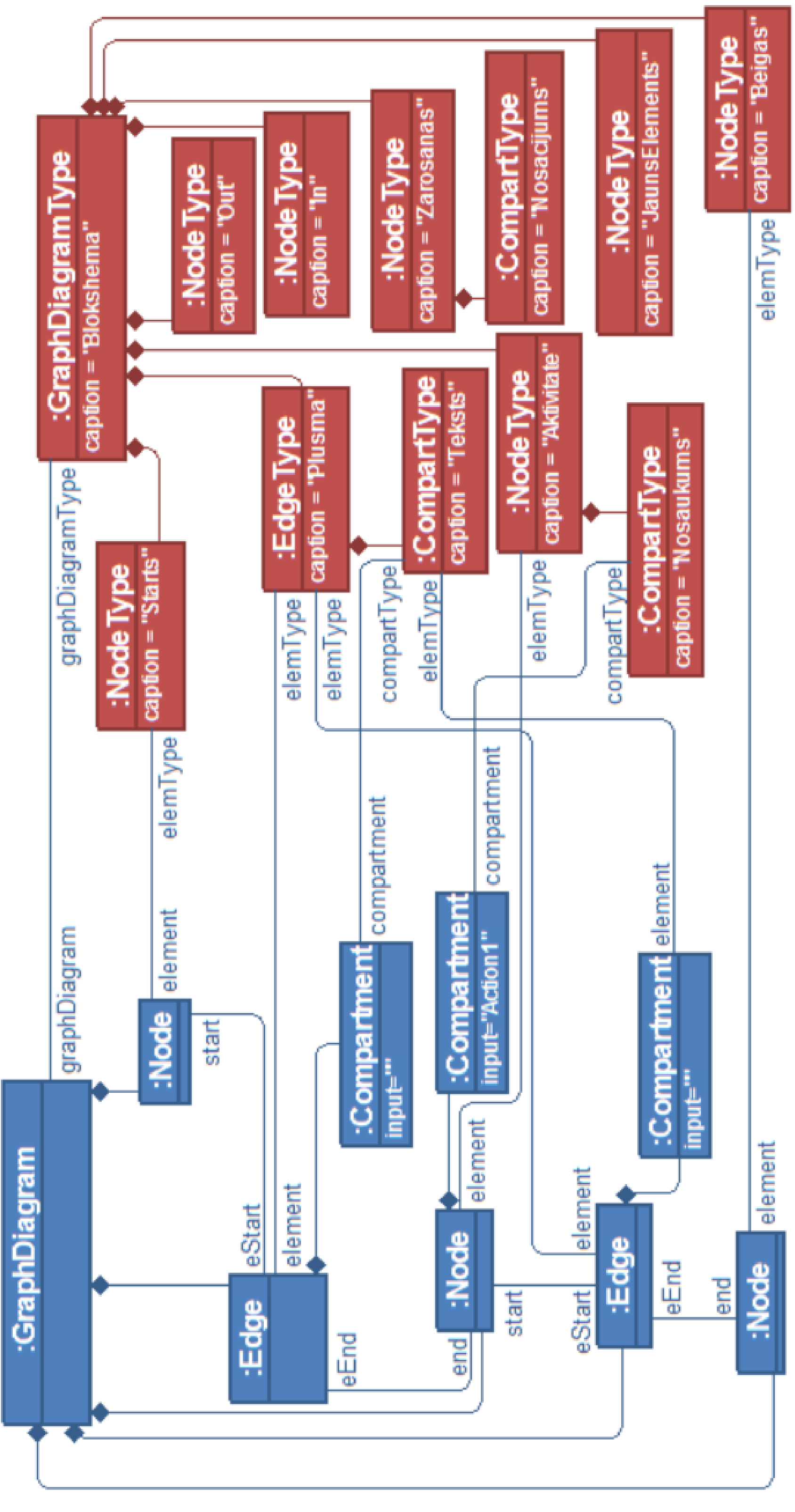
Pieņemsim, ka ar konfiguratoru esam veikuši visas darbības, lai iegūtu latvisko blokshēmu redaktora versiju. Tā rezultātā esam ieguvuši migrēšanas transformāciju, kura, kā iepriekš tika minēts, sastāv no diviem soļiem. Pirmais migrēšanas solis repozitorijā izveido

jaunās specifikācijas instances, un repozitorija stāvoklis pēc šī transformācijas soļa izpildes ir redzams 4.41. attēlā (shematiski šī situācija atbilst 4.36. attēlam).



4.41. att. Blokshēmu redaktora modelis pēc migrēšanas pirmā soļa

Transformācijas otrajā solī modeļu dati tiek piesaistīti jaunajai specifikācijai un pēc tam vecā specifikācija tiek izdzēsta. Repozitorija stāvoklis pēc migrēšanas beigām ir redzams 4.42. attēlā (shematiski šī pati situācija ir attēlota 4.37. attēlā).

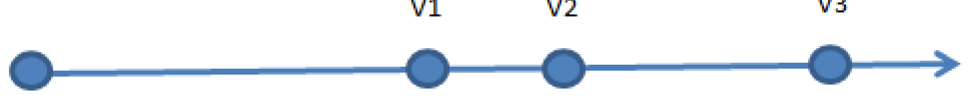


4.42. att. Blokshēmu redaktora modelis pēc migrēšanas otrā soļa

Kā redzams, atšķirībā no sākotnējās rīka versijas, jaunajā rīka versijā rīku definēšanas metamodeļa instances ir pārsauktas latviskajos terminos, ir radies jauns elements „JaunsElements”, kā arī tajā vairs nav elementa „ToBeDeleted” un tam atbilstošo prezentācijas instanču.

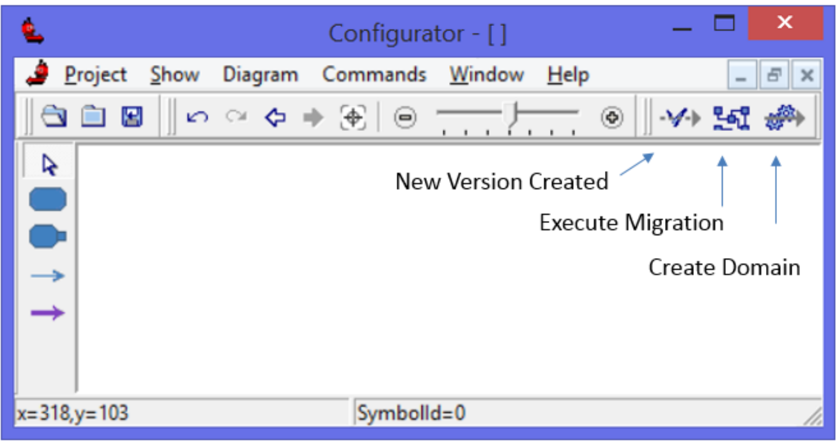
4.3.4 Modeļu migrēšana izpildīšana

Rīku izstrādes process notiek laikā, un tas paredz, ka rīka izstrāde sākas nulles punktā, un pēc tam rīka izstrādātājs konfigurē rīka pirmo, otro, utt. versiju. Šī situācija grafiski ir attēlota 4.43. attēlā.



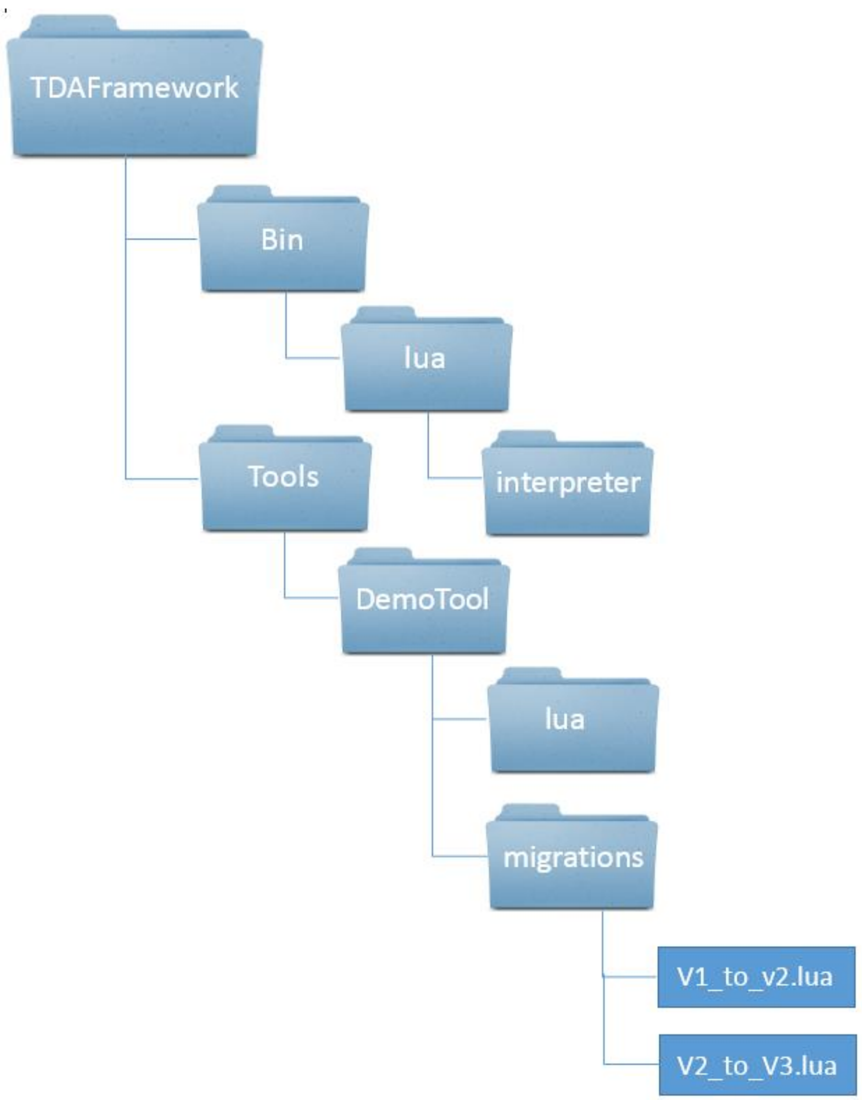
4.43. att. Rīka izstrādes process laikā

Lai konfigurēšanas procesā izstrādātājs varētu paziņot, ka ir radīta jauna versija, konfiguratora rīkjoslā ir izveidota poga „New Version Created” (skat. 4.44. attēlu). Pēc šīs pogas nospiešanas tiek izveidota migrēšanas transformācija, kas ļauj vecās versijas modeļu datus pārnest uz jauno.



4.44. att. Rīkjoslas pogas konfiguratora diagrammā

Tehniski runājot, katra rīka *Tools* direktorijā tiek izveidota direktorija *migration*, kurā katra datne satur vienu migrācijas transformāciju, un tā nosaukums ir izveidots formā <vecās_versijas_nosaukums>_to_<jaunās_versijas_nosaukums>.lua, kā tas ir redzams 4.45. attēlā.



4.45. att. Rīka datņu struktūra

Tātad katru reizi, kad tiek nospiesta poga „New Version Created”, tiek izveidota migrēšanas datne. Izņēmums ir rīka pirmā versija, jo, tad modeļus nevajag migrēt, un attiecīgi arī migrēšanas datne netiek veidota. Tā rezultātā *migration* direktorijā atrodas visas migrēšanas datnes, uzkrājot pilnu vēsturi par rīka evolūciju (ja tika izstrādātas n rīka versijas, tad *migration* direktorijā atradīsies n-1 migrēšanas datnes). Piemēram, ja rīks ir izstrādāts atbilstoši 4.43. attēlā redzamajam scenārijam, tad *migration* direktorijā atradīsies divas datnes – „V1_to_V2.lua” un „V2_to_V3.lua”, un katra no tām viennozīmīgi nosaka, starp kurām rīka versijām tā veic migrēšanu.

Lai migrēšanas transformācijas izpildītu, rīka (tā, kura modeļu datus migrēs) *migration* direktorijā ir jāiekopē attiecīgās migrāciju datnes, piemēram, jau minētās datnes „V1_to_V2.lua” un „V2_to_V3.lua”, un, tad jāatver projekts, kuru migrēs. Kad projekts ir atvērts, ir jāatver konfigurators (nospiežot taustiņu „C”), un tur jānospiež rīkjoslas poga „Execute Migration”, kas piedāvās sarakstu ar iekopēto datņu nosaukumiem no „migration”

direktorijas (mūsu piemēra gadījumā tie būs - „V1_to_V2.lua" un „V2_to_V3.lua"), un izvēloties vienu, tiks izpildīta šajā datnē esošā migrēšanas transformācija.

4.4 Secinājumi

Šajā nodaļā ir aprakstīts konfigurators, jeb rīks, kas ļauj grafiskā formā specificēt DSML rīkus, nenolaižoties līdz rīku definēšanas metamodeļu instancēm. Turklāt konfiguratora lietošana ļauj DSML rīku izstrādi padarīt iteratīvu, jo konfiguratorā realizētais modeļu migrēšanas mehānisms ļauj modeļu datus pārnest starp dažādām rīka versijām.

Jāpiebilst, ka pats konfigurators arī ir realizēts TDA platformā kā DSML rīks, un darbā ir parādīts, kā ar konfiguratoru var definēt pašu konfiguratoru, tādējādi, apliecinot, ka ar to var realizēt plašu DSML rīku klasi.

5 Lietojumu piemēri un pieredze

Šajā nodaļā ir īsi aprakstīti platformas attīstības posmi un pieredze, kas tika iegūta izstrādājot praktiski lietojumus rīkus, no kuriem nozīmīgākie ir - BiLingva, ProMod, OWLGrEd un LuMod.

5.1 Īss platformas attīstības apraksts

Rīku būves platformas attīstību kopš projekta uzsākšanas var iedalīt divos lielos posmos. Pirmajā posmā katram rīkam tika izstrādāts savs metamodelis, bet transformācijas tika izstrādātas tieši šim rīkam transformāciju valodā L0 [37, 38]. Izstrādājot vairākus rīkus, tika konstatēts, ka daudzu transformāciju teksti atkārtojas, un tādēļ tās vajag padarīt universālas.

Tas noveda pie otrā platformas attīstības posma, kurā, lai uzdotu rīku specifikācijas, tika izstrādāts rīku definēšanas metamodelis, un interpretators, kas doto specifikāciju pārvērš strādājošā rīkā. Savukārt, lai rīku definēšanas metamodeļa instances varētu izveidot ērtāk un ātrāk, tika izstrādāts konfigurators. Tā rezultātā šajā posmā rīka specifikācija tiek uzdota ar konfiguratoru, bet interpretators to realizē strādājošā rīkā.

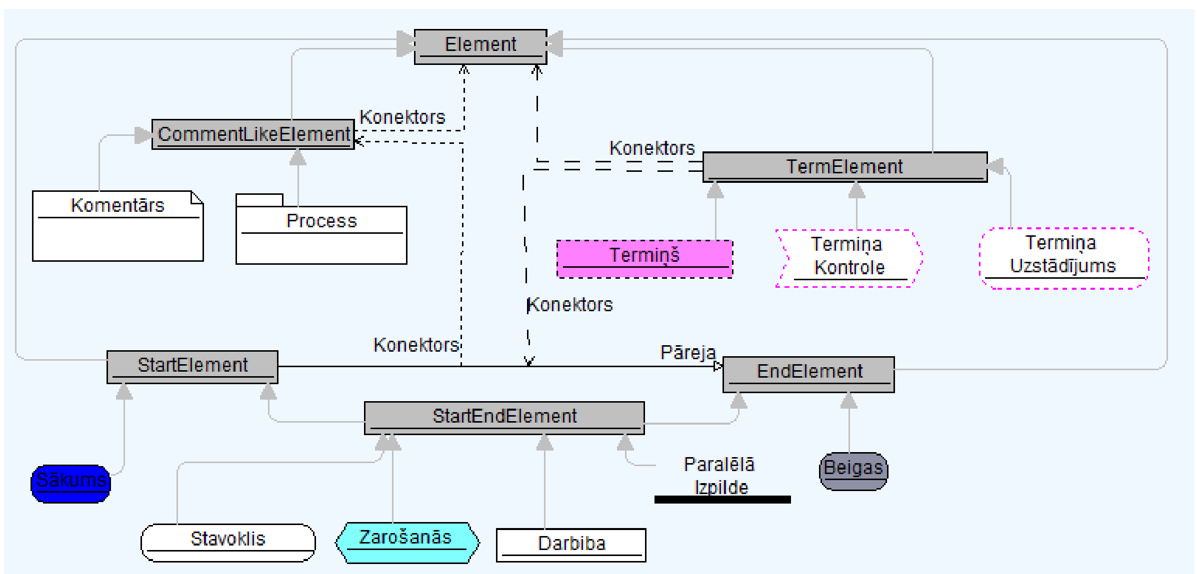
Rīku izstrādi ar konfiguratoru var sadalīt vēl divos apakšposmos. Pirmajā apakšposmā interpretators bija realizēts transformāciju valodā L0, bet otrajā - ar Lua un lQuery. Lua un lQuery ir divas būtiskas priekšrocības salīdzinājumā ar L0.

Pirmkārt, Lua ir pieejamas dažādas augsta līmeņa bibliotēkas, piemēram, teksta parsēšanai (*re* [39], *lpeg* [40]), sadarbībai ar ārējām datu bāzēm, kā arī dažādas datu struktūras, piemēram, masīvi, vienkārši saraksti, asociatīvi saraksti, utt. Šādu augsta līmeņa bibliotēku un datu struktūru lietošana ļauj būtiski samazināt izstrādājamo transformāciju koda apjomu un tā sarežģītību.

Otrkārt, Lua ir interpretēta valoda, un šis apstāklis ļauj atrisināt vairākas problēmas. Pirmkārt, transformāciju koda izmaiņas un to izpildīšana neizraisa laikietilpīgo kompilācijas procesu (pat neprasa pārstartēt rīku), kā rezultātā tas mums ļauj ievērojami samazināt transformāciju izstrādes laiku. Otrkārt, tas atrisina kopēšanas problēmu starp vairākiem vienlaicīgi atvērtiem projektiem, jo kopēšanas laikā var uzģenerēt kopēšanas transformāciju, kuru pēc tam var viegli (bez kompilēšanas) izpildīt jebkurā atvērtā projektā. Treškārt, tas ļauj relatīvi viegli atrisināt modeļu migrēšanas problēmu, izmantojot to pašu pieeju, ko kopēšanas gadījumā, proti, uzģenerēt transformāciju, kas migrē modeļus no vienas versijas uz otru, un pēc tam bez kompilēšanas šo transformāciju izpildīt visos nepieciešamajos projektos.

5.2 BiLingva

BiLingva [41] ir biznesa procesu modelēšanas rīks, kurš ir izstrādāts projektu izvērtēšanas procesu aprakstīšanai priekš Latvijas Investīciju un attīstības aģentūras. Rīka konfigurācijas grafiskais modelis ir redzama 5.1. attēlā.

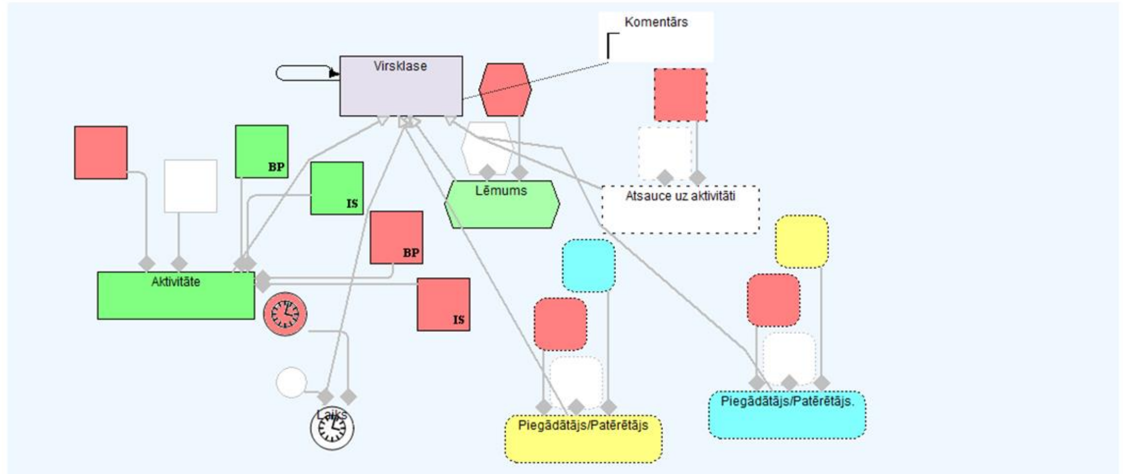


5.1. att. BiLingva konfigurācijas grafiskais modelis

Šis ir pirmais praktiski lietotais rīks, kura izstrādē tika izmantots konfigurators un interpretators, un rezultāts apliecināja šīs tehnoloģijas efektivitāti. Proti, visu rīka pamatfunkcionalitāti varēja norealizēt tikai konfigurējot (bez rīkam specifisku transformāciju izstrādes), bet papildus programmēt vajadzēja tikai skatu mehānismu, kas, izmainot elementu stilus un paslēpjot atribūtus, ļauj dažādām lietotāju grupām vienus un tos pašus modeļus rādīt atšķirīgi (skatu definēšanas mehānisms ir detalizētāki aprakstīts E. Renča promocijas darbā [42]).

5.3 ProMod

ProMod [43] ir biznesa procesu modelēšana rīks, izstrādāts procesu aprakstīšanai priekš Valsts sociālās apdrošināšanas aģentūras. Konfigurācijas grafiskais modelis vienam rīkā realizētajam diagrammas tipam ir redzama 5.2. attēlā.



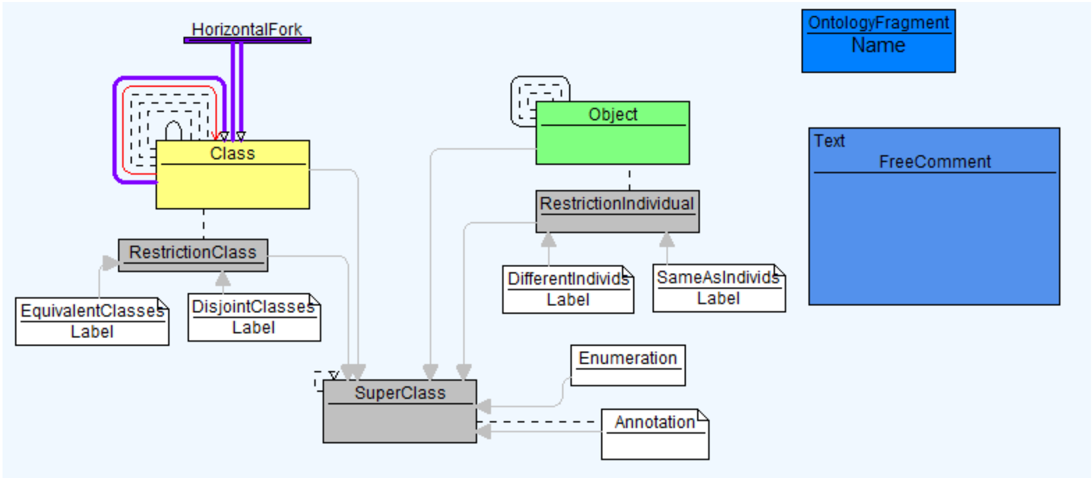
5.2. att. ProMod konfigurācijas grafiskais modelis

ProMod ir rīks ar ievērojami augstāku sarežģītību nekā BiLingva, kas, ja neskaita skatu mehānismu, ir relatīvi vienkāršs un ātri realizējams. Izstrādājot ProMod, radās vajadzība pēc vairākiem jauniem servisiem. Pirmkārt, radās nepieciešamība pēc dinamiski veidotām uznirstošajām izvēlnēm, kuras tiek radītas atkarībā no konteksta. Otrkārt, vajadzēja veikt modeļu korektību pārbaudes. Treškārt, daļa datu, kurus vajadzēja piedāvāt izvēlnēs, glabājās ārējās datu bāzēs un vajadzēja šiem datiem piekļūt (šī uzdevuma izpildi atvieglo tas, ka Lua ir pieejamas bibliotēkas sadarbībai ar ārējām datu bāzēm). Ceturtkārt, no modeļu datiem vajadzēja aizpildīt MS Word dokumentu sagataves, izmantojot MS Word API [44]. Piektkārt, vajadzēja izstrādāt daudzlietotāju režīmu.

Jāpiezīmē, ka visus šos papildus servisus varēja relatīvi viegli realizēt, pateicoties tam, ka platformā ir realizēts paplašinājuma punktu mehānisms.

5.4 OWLGrEd

OWGrEd [45, 46, 47, 48, 49, 50, 51, 52] ir uz UML klašu diagrammu valodas balstīts grafisks ontoloģiju redaktors priekš OWL [53] vizualizācijas. Rīka konfigurācijas grafiskais modelis ir redzama 5.3. attēlā.



5.3. att. OWLGrEd konfigurācijas grafiskais modelis

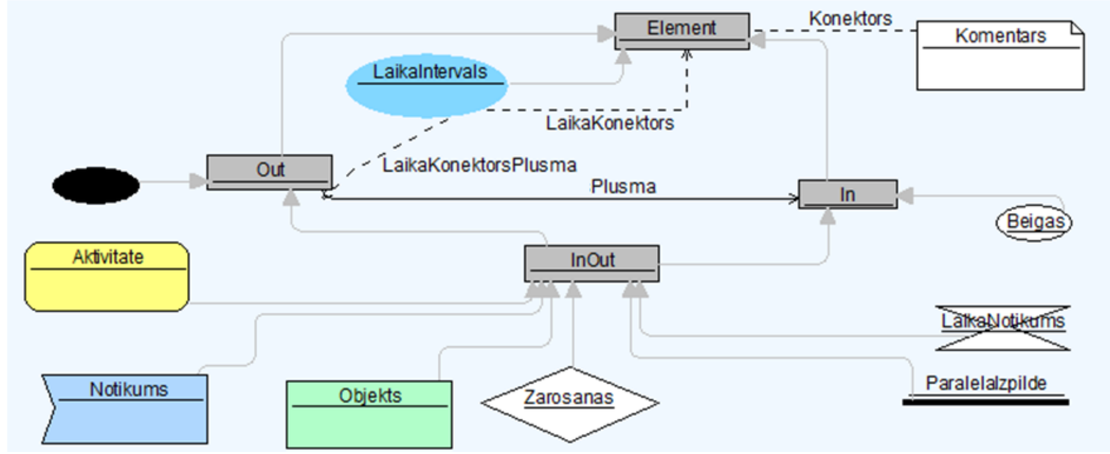
OWLGrEd, atšķirībā no iepriekš aprakstītajiem rīkiem, ir nevis biznesa procesu modelēšanas rīks, bet gan ontoloģiju [54] vizualizēšanas rīks. Sākotnēji rīks tika veidots kā populārā ontoloģiju redaktora Protege [55] spraudnis, lai vizualizētu un arī rediģētu tā ontoloģijas. Tomēr laika gaitā OWLGrEd ir attīstījies par patstāvīgu ontoloģiju redaktoru, kurā ir realizēts atbalsts ontoloģiju apmaiņai ar Protege.

OWLGrEd ir realizētas divas lietas, kuru citos rīkos nav. Pirmkārt, OWLGrEd ir realizēta iespēja atvērt ne tikai TDA platformas saglabātos projektus, bet arī kādu no plaši izplatītajiem ontoloģiju formātiem, piemēram, OWL funkcionālajā [56], XML [57] vai RDF [58] sintaksēs. Otrkārt, OWLGrEd ir realizēta iespēja no kāda modeļu datu fragmenta izveidot tā tekstuālo formu, piemēram, no ontoloģiju grafiskās formas uzģenerēt tām atbilstošo tekstuālo formu OWL funkcionālajā sintaksē. Pie tam, kā šī uzdevuma apakšuzdevums, ir jārisina problēma, ka dažu elementu atribūtu vērtības var saturēt sarežģītas loģiskās izteiksmes atbilstoši Mančesteras sintaksei [59], un, lai šīs izteiksmes varētu pārveidot atbilstoši OWL funkcionālajai sintaksei, tās nepieciešams parsēt (šī uzdevuma veikšanai būtiska priekšrocībā bija iespējai izmantot augsta līmeņa teksta apstrādes bibliotēku „re”, kura ļauj tekstu apstrādāt bezkonteksta gramatiku abstrakcijas līmenī).

Arī šī redaktora papildus funkcionalitāte ir realizēta izmantojot paplašinājuma punktu mehānismu.

5.5 LUMod

LUMod [60] ir biznesa procesu modelēšana rīks, izstrādāts procesu aprakstīšanai priekš Latvijas Universitātes. Rīka konfigurācijas grafiskais modelis ir redzama 5.4. attēlā.



5.4. att. LUMod konfigurācijas grafiskais modelis

LUMod modelēšanas valoda ir realizēta par pamatu ņemot UML aktivitāšu diagrammu valodu, bet daļa tā funkcionalitāte ir aizgūta no ProMod. LUMod nav realizēts daudzlietotāju režīms un modeļu konsistenču pārbaudes kā tas ir ProMod, bet ir realizēt dažādi citi servisi, piemēram, dinamiskās uznirstošās izvēlnes. No servisiem, kas ir realizēti specifiski priekš LUMod, var minēt aktivitāšu unikālo nosaukumu ģenerēšana atkarībā no aktivitātes atrašanās dziļuma diagrammu struktūrā, referencēto aktivitāšu sinhronizāciju ar bāzes aktivitāti, kā arī iespēja konfigurēt izvēļņu un sarakstu saturu priekš izkrītošo izvēļņu un sarakta tipa laukiem. Tādējādi sarežģītības ziņā LUMod ir vienkāršāks par ProMod, bet sarežģītāks par BiLingva.

Principiāli jauna lieta ir normatīvo aktu tekstu ģenerēšana no modeļu datiem. Ja ProMod bija realizēts, ka tiek ģenerēti MS Word dokumenti, aizpildot dokumentu sagataves ar noteiktām modeļu elementu vērtībām, tad LUMod ir iets tālāk, un tiek analizēta modeļu struktūra, proti, tiek meklēti noteikti šabloni, un pēc tam no to datu vērtībām tiek ģenerēti dabīgās valodas teikumi (arī visa šī papildus funkcionalitāte ir realizēta izmantojot paplašinājuma punktu mehānismu).

5.6 Secinājumi

Šajā nodaļā minēto rīku izstrāde pierāda, ka darbā piedāvātā rīku būves tehnoloģija ir piemērota praktiski lietojumu DSML rīku izstrādē un ka bez piedāvātās tehnoloģijas šo rīku izstrāde būtu viennozīmīgi darbietilpīgāka un neērtāka.

6 Pārskats alternatīvām rīku būves pieejām

Tā kā grafisko rīku būve nav jauna tēma, šajā nodaļā aplūkosim un salīdzināsim promocijas darbā piedāvāto rīku būves pieeju ar jau eksistējošiem risinājumiem citās grafisko rīku būves platformās un modelēšanas vidēs. Darbā ir aplūkoti šādi risinājumi - Eclipse GMF, Microsoft DSL, MetaEdit, ObeoDesigner, EuGENia Live, Graphiti, GME, RSA un Microsoft Visio.

6.1 Eclipse GMF

Eclipse GMF [61] ir ietvars, kas, apvienojot *EMF* un *GEF* [62] komponentes, nodrošina pilnu grafisku redaktoru izstrādes ciklu *Eclispe* vidē [63]. Jaunu redaktoru izstrāde *Eclispe GMF* ietvarā sastāv no vairākiem soļiem. Pirmajā solī izveido *Domain* modeli jeb definē redaktora grafiskās valodas abstrakto sintaksi, uzdodot to kā UML klašu diagrammu. Otrajā solī izveido *Graphical* modeli, ar kuru definē redaktora grafiskos elementus un to stila informāciju. Šis modelis tiek veidots ar kokveida diagrammas komponenti, kurā katram redaktora grafiskajam elementam atbilst kāda koka virsotne, bet elementus specificējošo informāciju ievada caur īpašību logiem. Trešajā solī izveido *Tooling* modeli, ar kuru definē redaktora funkcionalitāti, piemēram, redaktora paleti.

Lai gan katrs no šiem modeļiem definē kādu konkrētu redaktora sastāvdaļu, tie ir savstarpēji neatkarīgi, un tādēļ šo modeļu sasaistīšanai izveido *Mapping* modeli, ar kuru „sasien" kopā *Domain*, *Graphical* un *Tooling* modeļus. *Mapping* modelī tiek norādīts, kā viena modeļa informācija attēlojas otrā modelī, lai redaktora izpildes laikā platforma automātiski varētu nodrošināt sinhronizāciju starp šo modeļu datiem. Pēc tam visus iepriekš aprakstītos modeļus transformē par *Generation* modeli, no kura tiek uzģenerēts redaktora kods Java programmēšanas valodā. Ja nepieciešams, uzģenerēto kodu var papildināt, bet, lai atkārtoti ģenerējot kodu papildinājumi netiktu pazaudēti, jālieto speciāli šim mērķim radītas anotācijas. Pēc tam redaktoru var palaist *Eclispe* vidē.

Šī pieeja balstās uz to, ka *Domain* un *Graphical* modeļi ir atdalīti, ļaujot vienus un tos pašus datus glabāt un attēlot dažādās formās. Gadījumā, ja datus glabā atbilstoši konkrētas sintakses notācijai, tos var viegli attēlot redaktoru lietotājiem ērtā formā, bet gadījumā, ja datus glabā atbilstoši to loģiskajai struktūrai, tos ir vieglāk apstrādāt un analizēt redaktoru izstrādātājiem.

Gadījumā, ja nepieciešams izstrādāt redaktorus, kur augstas sarežģītības datu apstrāde nav nepieciešama, piemēram, dažādu datu vizualizēšanas un procesu modelēšanas valodu redaktorus, pilnīgi pietiek, ja ir definēta tikai redaktoru valodu konkrētā sintakse. Tādējādi šāda veida redaktoru izstrādē grafisko valodu abstraktās sintakses uzdošana ir lieka, un

komplektā ar atbilstību nodrošināšanu starp valodas abstrakto un konkrēto sintaksi tikai nevajadzīgi sarežģī un palēlina redaktora izstrādes procesu, jo tas gan palielina redaktora definēšanai nepieciešamo soļu skaitu, gan apgrūtina izmaiņu ieviešanu (atcerēsimies, ka tipiski redaktoru izstrāde ir iteratīva), kā arī prasa padziļinātas zināšanas par platformu.

Savukārt, ja ir nepieciešams izstrādāt redaktorus, ar kuru modeļiem ir jāveic netriviāla modeļu datu apstrāde, piemēram, ir jāizstrādā grafiska programmēšanas valoda un tās kompilators [64], tad kompilēšanai nepieciešamo datu apstrādi un analīzi ir ērtāk veikt izmantojot valodas loģisko struktūru, nekā to pašu darīt ar grafiskās notācijas simboliem.

Eclipse GMF aprakstītā pieeja ir piemērota uzdevumu klasei, kur ir nepieciešama gan modeļu grafiska reprezentācija, gan netriviāla datu apstrāde, un smagnējais izstrādes process ir cena, kas ir jāmaksā par to, ka platformā tiek nodrošināta automātiska datu sinhronizācija starp abstraktās un konkrētās sintakses modeļiem, lai redaktora izstrādātājs varētu veikt modeļu datu apstrādi atbilstoši to loģiskajai, nevis grafiskajai struktūrai.

6.2 Microsoft DSL

Microsoft DSL ir ietvars, ar kuru var izstrādāt grafiskus redaktorus *Microsoft Visual Studio* vidē [65]. Redaktoru izstrāde ir sadalīta vairākos soļos. Pirmajā solī izveido *Classes and Relationships* modeli jeb definē redaktora valodas abstrakto sintaksi ar UML klašu diagrammu valodai līdzīgu valodu. Otrajā solī definē redaktora grafiskos elementus jeb *Diagram Elements* modeli. Šo modeli uzdod ar grafisko valodu, kuras grafiskie elementi ir tādi paši kā valodai, ar kuru specificē *Classes and Relationships* modeļu konceptus, bet atšķirīgi ir to īpašību logi. Šajā solī vēl papildus specificē grafiskajiem elementiem atbilstošos paletes elementus. Trešajā solī specificē atbilstības starp *Classes and Relationships* un *Diagram Elements* modeļiem, lai redaktora izpildes laikā starp abiem modeļiem varētu nodrošināt automātisku datu sinhronizāciju. Ceturtajā solī tiek ģenerēts redaktors, bet, ja nepieciešama papildus funkcionalitāte, piemēram, validācija vai XML serializēšana, tad var izmantot iebūvētos līdzekļus un nepieciešamo funkcionalitāti norādīt caur dialogu logiem, vai arī pieprogrammēt C# programmēšanas valodā, ja ar iebūvētajiem līdzekļiem nepietiek.

Līdzīgi kā *Eclipse GMF*, arī *Microsoft DSL* ietvarā ir izmantota pieeja, ka redaktoru izstrādes laikā definē valodas abstraktās sintakses modeli un pēc tam norāda atbilstības starp abstrakto un konkrēto valodu modeļiem. Tas nozīmē, ka abas šīs pieejas ir piemērotas līdzīgām uzdevumu klasēm, bet galvenā to atšķirība ir meklējama to tehniskajā realizācijā, jo *Microsoft DSL* specifika ir tā, ka tas ir integrēts *Microsoft Visual Studio* vidē, kas savukārt ir piesaistīta Windows platformai, un tā rezultātā ar šo ietvaru izstrādātie redaktori arī ir lietojami tikai Windows vidē.

6.3 MetaEdit

Redaktoru izstrāde ar *MetaEdit* platformu ir sadalīta četros soļos. Pirmajā solī tiek definēti redaktora valodas koncepti. Šajā platformā piedāvātie koncepti atbilst *GOPPRR* modelim [66] un tie ir šādi – grafi, objekti, atribūti, porti, attiecības un lomas, tādējādi konkrēta valoda tiek definēta kā šo konceptu instances.

Otrajā solī tiek definēti ierobežojumi attiecībā uz valodas konceptiem, piemēram, platformā ir iespējams norādīt, ka vienā diagrammā nedrīkst atrasties vairāk nekā viens konkrēta tipa koncepts. Platformā ir iebūvēti dažādu tipu ierobežojumi, un tos uzstāda izvēloties konkrētos ierobežojumus no iepriekš definēta saraksta. *MetaEdit* pirmo un otro soli var izpildīt divos veidos. Viens vieds ir nepieciešamo informāciju ievadīt caur dialogu logu formām, vai arī šo informāciju ievadīt grafiski, izmantojot grafisku modelēšanas valodu.

Trešajā solī konceptiem tiek piekārtots to grafiskais attēlojums. To var izdarīt, vai nu uzzīmējot konceptam atbilstošo reprezentāciju, vai arī importējot jau esošu attēlu, kuru pēc tam uzstāda kā koncepta grafisko attēlojumu. Ceturtajā solī tiek definēti valodas ģeneratori, ar kuriem MERL valodā [67] apstaigā modeļus un izveido modeļu tekstuālās formas, piemēram, ja ar modeli apraksta kādu sistēmu, tad, apstaigājot šo sistēmu, var uzģenerēt šīs sistēmas kodu kādā programmēšanas valodā. Valodu definīcijas un ģeneratori tiek glabāti kā modeļi platformas repozitorijā, kas ļauj tālākas izmaiņas automātiski atspoguļot izveidotajos redaktora modeļos. Turklāt ģeneratorus ir iespējams integrēt ar valodas definīciju, t.i., sasaistīt tā, ka, mainot valodu, mainās arī ģenerators. Šī integrācija ļauj padarīt redaktoru definēšanu un ģeneratoru izstrādi iteratīvu un interaktīvu.

MetaEdit platformā realizētā pieeja paredz definēt tikai redaktora valodas grafiskos simbolus jeb konkrēto sintaksi, bet tās abstrakto sintaksi nedefinēt vispār. Turklāt konkrētā sintakse tiek uzdota, nevis ar UML klašu diagrammu valodu vai kādu tai līdzīgu universālu modelēšanas valodu, bet gan ar *GOPPRR* valodu, kas ir izstrādāt specifiski grafveida diagrammu valodu definēšanai. Tā rezultātā šī modelēšanas valoda ļauj redaktora izstrādātājam domāt tieši definējamās valodas terminos un padarīt jaunu redaktoru izstrādes procesu ērtu, kura laikā izstrādātājam ir intuitīvi viegli izsekot tam, kas tiek darīts, kā arī ātri veikt izmaiņas esošajos redaktoru definīcijās, nodrošinot iespēju redaktorus izstrādāt iteratīvi.

MetaEdit platforma realizē DSM filozofiju, kas paredz, ka programmatūra tiek būvēta kā grafiski modeļi, no kuriem pēc tam ģenerē kodu. Šāda pieeja ļauj celt programmatūras izstrādes abstrakcijas līmeni, un eksperimenti rāda, ka šādā veidā ir iespējams programmatūras izstrādi paātrināt pat 5-10 reizes [68]. Rezultātā *MetaEdit* realizācijā uzsvars ir likts uz programmatūras izstrādi un koda ģenerēšanu, nevis grafisku modelēšanas valodu

realizāciju. Tā sekas ir ļoti ierobežotās iespējas mainīt platformā norealizēto redaktora funkcionalitāti, vai papildus noprogrammēt redaktora specifisko funkcionalitāti.

6.4 ObeoDesigner

ObeoDesigner [69, 70, 71] ietvars ir izstrādāts izmantojot *Eclipe Modeling* tehnoloģijas un tā realizācija balstās uz *EMF*, *GEF* un *GMF* vidēm. Grafisko redaktoru izstrāde ar *ObeoDesigner* ir sadalīta trīs soļos. Pirmajā solī definē redaktora valodas abstrakto sintaksi jeb domēna metamodeli, ko uzdod ar UML klašu diagrammu.

Otrajā solī iepriekš definētajiem abstraktās sintakses konceptiem izveido grafiskās reprezentācijas, piemēram, koncepta formu, krāsu, burtu izmērus, utt., un pēc tam reprezentācijas sasaista ar domēna metamodeli. Noslēgumā, kad ir definētas konceptu grafiskās reprezentācijas, izveido redaktora paleti un tās elementu ikonas.

Redaktoru definēšanai *ObeoDesigner* tiek izmantota skatpunktu ideja un tā paredz, ka jebkura reprezentācija tiek veidota kā domēna metamodeļa skatpunkts. Tādējādi arī grafiskais redaktors ir viena veida (konkrētāk, prezentācijas) skatpunkts uz domēna metamodeli. Bez prezentācijas skatpunkta, *ObeoDesigner* ir iespējams veidot arī citus skatpunktus, kuri visi ir sasaistīti ar domēna metamodeli. Tā rezultātā jebkuras izmaiņas kādā skatpunktā tiek sinhronizētas ar domēna metamodeli un pēc tam no domēna metamodeļa tiek atjaunoti visi pārējie tam piesaistītie skatpunkti.

Trešajā solī, kad ir uzbūvēta redaktora grafiskā reprezentācija, ir iespējams definēt dažādus papildus servisus, piemēram, ģeneratorus, kas no redaktorā uzbūvētajiem modeļiem ģenerē kodu, vai validatorus, ar kuriem var pārbaudīt izveidoto modeļu korektumu.

Galvenā motivācija izstrādāt *ObeoDesigner* bija nepilnības *GMF* izmantotajā pieejā, kas, lai gan piedāvā realizēt redaktorus ar plašu funkcionalitāti, ir sarežģīta un prasa dziļas zināšanas no redaktoru izstrādātājiem. Tādēļ, lai varētu veikt redaktoru izstrādi vienkāršāk un augstākā abstrakcijas līmenī nekā tas ir ar *GMF*, *ObeoDesigner* tika uzbūvēta kā virsbūve *GMF*. Pētījumi rāda, ka šis mērķis ir sasniegts un ar *ObeoDesigner* redaktorus var izstrādāt pat 5 reizes ātrāk nekā ar *GMF* [72]. Tomēr, lai arī *ObeoDesigner* ietvarā ir samazināts ģenerēšanas soļu skaits, redaktoru izstrādes procesā ģenerēšana joprojām pastāv, un tas padara redaktoru izstrādi mazāk intuitīvu, kā arī redaktoru izstrādes laikā ir viegli kļūdīties. *ObeoDesigner* izmantotajai skatpunktu idejai ir priekšrocības, ja vieniem datiem nepieciešamas dažādas reprezentācijas, savukārt ja ir jāizstrādā redaktors, kuram ir tikai viens prezentācijas skatpunkts, tad arī šī pieeja, līdzīgi kā citas pieejas, kurās vispirms jādefinē redaktora valodas abstraktā sintakse, pēc tam konkrētā sintakse un tad vēl atbilstības starp tām, ir pārāk smagnēja.

6.5 EuGENia Live

EuGENia Live [73, 74] ir eksperimentāls, izstrādes stadijā esoša grafisko redaktoru izstrādes ietvars. Atšķirībā no citiem ietvariem *EuGENia Live* redaktori atrodas tīmeklī, kas atvieglo redaktoru piegādi un uzstādi, jo lietotājiem papildus neko nevajag lejupielādēt, instalēt un risināt operētājsistēmu saderības problēmas.

Redaktoru definēšanai ir divas komponentes – metamodelis un anotāciju valoda. Metamodelis sastāv no divām savstarpēji saistītām daļām. Viena daļa ir dinamiski mainīga atkarībā no lietotāju darbībām, un šī daļa apraksta veidu, kā redaktora modeļi un to sastāvdaļas tiek glabātas repozitorijā. Otra daļa ir statiska, t.i., tā redaktora izpildes laikā nemainās, un tās mērķis ir aprakstīt redaktora specifikāciju. Savukārt, lai uzdotu pašu redaktora specifikāciju un attiecīgi iegūtu metamodeļa statiskās daļas aizpildījumu, ir izstrādāta anotācijas valoda, ar kuru, norādot redaktora elementus un uzstādot to stilu parametrus, tiek definēts konkrēts redaktors.

Jāpiebilst, ka *EuGENia Live* ietvars ir elastīgs attiecībā uz redaktora specifikācijas mainību un izveidoto modeļu saderību, respektīvi, ja maina redaktora specifikāciju, tad veiktās izmaiņas automātiski tiek ienestas jau izstrādātajos modeļos un šie modeļi nav atkārtoti jāpārzīmē. Tādējādi šī īpašība ir īpaši ērta, ja redaktorus izstrādā iteratīvi, jo ir viegli (bez modeļu migrēšanām) integrēt specifikācijas izmaiņas esošos modeļos.

Lai gan *EuGENia Live* atšķirībā no citiem šajā nodaļā aprakstītajiem ietvariem ir tikai izstrādes stadijā un publiski ir pieejams tikai ļoti vienkāršs prototips, tas ir interesants ar to, ka tā ideoloģija ir ļoti līdzīga tai, kas ir aprakstīta šajā promocijas darbā – gan attiecībā uz metamodeļiem (apraksta redaktoru specifikāciju un modeļu datus), gan izstrādes procesu (iteratīvs, viegli maināms).

6.6 Graphiti

Graphiti [75] ir grafisku rīku būves ietvars, kas ir realizēts *Eclipse* vidē, un tas ir alternatīvs ietvars iepriekš apskatītajam *Eclipse GMF* ietvaram, proti, tas arī ir uzbūvēts kā virsbūve *EMF* un *GEF* komponentēm *Eclipse* vidē. *Graphiti* arhitektūra paredz, ka lietotājs, strādājot ar rīku, veic kādas darbības, kuras *Graphiti* ietvars klasificē kā notikumus, un pēc tam šos notikumus nosūta *Diagram Type Agent*, kura uzdevums ir veikt saņemto notikumu apstrādi. Precīzāk sakot, *Diagram Type Agent* uzdevums ir mainīt tam piesaistīto modeļu (*Domain Model*, *Pictogram Model* un *Link Model*) datus atbilstoši rīka loģikai. *Domain Model* apraksta valodas abstrakto sintaksi, *Pictogram Model* atbilst prezentācijas modelim, kas apraksta diagrammu elementu grafisko reprezentāciju, un *Link Model* apraksta saistību starp *Domain Model* datiem un to grafisko reprezentāciju. Piemēram, lai izveidotu jaunu kasti, *Diagram*

Type Agent ir jāizveido attiecīgais *Domain Model* objekts, tad šim objektam atbilstošai *Pictogram Model* objekts un pēc tam iekš *Link Model* jānodibina atbilstība starp šiem objektiem. Tādējādi, lai realizētu konkrētu rīku, mums ir jārealizē (t.i., jānoprogrammē) *Diagram Type Agent* funkcionalitāte, kas apsaimnieko šo modeļu datus.

Graphiti arhitektūra ir līdzīga TDA arhitektūrai, t.i., *Graphiti* ietvars līdzīgi kā TDA dziņi klasificē lietotāju darbības kā notikumus un pēc tam to apstrādi deleģē *Diagram Type Agent*, kas ir modeļu transformāciju analogs TDA. Tādā veidā *Graphiti* ietvarā rīku izstrādes smagums tiek pārnests uz *Diagram Type Agent* izstrādi (līdzīgi kā TDA uz modeļu transformācijām). Tomēr starp šīm divām rīku izstrādes vidēm ir viena būtiska atšķirība, proti, TDA platformā rīka valodas sintakse tiek glabāta atbilstoši vienam universālam metamodelim (t.i., rīku definēšanas metamodelim), bet *Graphiti* ietvarā priekš katra rīka ir jāveido jauns modelis (t.i., *Domain Model*). Tādējādi TDA platformā priekš visiem rīkiem var izveidot vienu universālu notikumu apstrādi (t.i., interpretatoru), kas balstās uz universālo metamodeli, savukārt *Graphiti* ietvarā šādu universālu apstrādi nevar uzbūvēt, jo šajā ietvarā katram rīkam ir savs metamodelis, un tāpēc notikumu apstrāde priekš katra rīka ir jāizstrādā specifiski. Neskatoties uz to, ka *Graphiti* ietvars tipiskiem lietojumiem ir izstrādāta funkciju bibliotēka, kas atvieglo *Diagram Type Agent* realizāciju, *Diagram Type Agent* funkcionalitāte tik un tā ir jāprogrammē, ko nevar uzskatīt par ātru un ērtu rīku izstrādi.

6.7 Generic Modeling Environment (GME)

GME [76, 77] ir platforma, ar kuru var izstrādāt DSML rīkus. Šī pieeja paredz, ka rīku grafiskā valoda tiek specificēta kā modeļu paradigma (jeb valodas metamodelis), kas satur visu informāciju par valodas semantiku, sintaksi un tās prezentāciju, proti, tā satur informāciju par to, kādi ir valodas koncepti, no kuriem tiks veidoti valodas modeļi, kādas starp konceptiem ir saistības, utt., jeb, citiem vārdiem sakot, modeļu paradigma definē likumīgo valodas modeļu kopu. Lai varētu specificēt modeļu paradigmu (t.i., valodas metamodeli), *GME* piedāvā savus modelēšanas konceptus, kuru notācija ir balstīta uz UML klašu diagrammu notāciju. Tādējādi, lai specificētu konkrēta rīka valodu, mums ir jāizveido valodas metamodelis, lietojot *GME* metamodelēšanas konceptus, savukārt, lai pievienotu metamodelim ierobežojumus, ir jālieto OCL. Pēc tam, kad ir izveidots valodas metamodelis, platformā šo metamodeli ir jāpiereģistrē kā jaunu paradigmu, tādējādi, pasakot, ka redaktora specificēšana ir pabeigta. Gadījumā, ja redaktorā ir nepieciešami kādi specifiski servisi, tad tos var pievienot, papildus noprogrammējot (nepieciešama apjomīga C++ programmēšana).

GME piedāvātā rīku būves pieeja paredz, ka rīka grafiskā valoda tiek specificēta ar universālu metamodeli, kas ir līdzīgi darba 2. nodaļā aprakstītajām rīku specificēšanas

metodēm. Tādējādi šajā platformā rīku valodas specifikācija nenotiek augstā abstrakcijas līmenī (rīku izstrādātājiem ir jāzina platformas metamodelēšanas valodas un OCL), un tā rezultātā rīku izstrāde nav vienkārša un ātra.

6.8 IBM Rational Software Architect (RSA)

RSA [78] ir grafisko rīku modelēšanas platforma, kas ir izstrādāta *Eclipse* vidē, izmantojot *GEF*, *EMF* un *GMF* komponentes. Šī platforma paredz, ka rīku grafiskās valodas metamodelis tiek specificēts kā UML profils, proti, *RSA* piedāvā bāzes meta-klases, kuras stereotipizējot var iegūtu valodas profilu. Šis profils apraksta valodas abstrakto sintaksi, bet, lai specificētu valodas konkrēto sintaksi, katrai stereotipa klasei ir jānorāda tā individuālā stila informācija. Gadījumā, ja nepieciešams, profilam var pievienot ierobežojumus OCL vai Java programmēšanas valodā. Kad tas ir izdarīts, ar profilu esam ieguvuši jauno redaktoru kā papildinājumu UML klašu diagrammu redaktoram.

Lai gan platforma piedāvā vēl dažādus servisus, kopējais secinājums ir, ka šī pieeja neļauj definēt rīka valodu augstā abstrakcijas līmenī, un arī profila definēšana ir smagnēja.

6.9 Microsoft Visio

Lai arī *Microsoft Visio* [79] ir zīmēšanas nevis modelēšanas rīks, tas bieži tiek izmantots dažādu grafisku diagrammu zīmēšanai. Tā galvenās priekšrocībās ir plašais grafisko simbolu klāsts, iespēja ērti zīmēt un izvietot simbolus, mainīt to stilus un, ja nepieciešams, zīmēt diagrammas atbilstoši kādam no esošajiem diagrammu šabloniem. Turklāt *Microsoft Visio* diagrammas ir iespējams padarīt arī uz datiem bāzētas sasaistot tās ar ārējiem datu avotiem, piemēram, *Microsoft Excel* vai *SQL Server*, un pēc tam, piedefinējot dažādus nosacījumus, mainīt diagrammu stilus atkarībā no datu vērtībām.

Tā kā daudzos gadījumos *Microsoft Visio* plašās diagrammu zīmēšanas iespējas un ērtības ir pilnīgi pietiekošas, lai izveidotu profesionāla izskata diagrammas, tad tā lietošana dažādu vienkāršu modelēšanas uzdevumu veikšanai ir ļoti ērta un populāra izvēle. Tomēr sarežģītākos gadījumos ar „vienkāršu zīmēšanu" vairs nepietiek, un modeļiem ir nepieciešami vēl papildus servisi, kurus *Microsoft Visio* neatbalsta, piemēram, modeļu korektuma pārbaudes balstoties uz modeļa valodas specifiskajiem likumiem un ierobežojumiem, specializēta interakcija ar lietotāju (modeļa valodas simboliem specializēti dialogu logi, specifiskas uznirstošās izvēlnes un īsinājuma taustiņu kombinācijas, utt.), diagrammu elementu detalizācija ar diagrammām, un pēc vajadzības dinamiska šīs sasaistes mainīšana, kā arī, vēl specifiskāku prasību gadījumā, šo prasību noprogrammēšana.

6.10 Secinājumi

Šajā nodaļā ir apskatītas un analizētas dažādas platformas DSML rīku būvēšanai. Lielākajā daļā aplūkoto platformu rīku grafiskās valodas vispirms ir jāspecificē ar valodas abstraktās sintakses modeli, to uzdodot kā UML klašu diagrammu, UML klašu diagrammu profilu vai ar valodu, kuras notācija ir līdzīga UML klašu diagrammu notācijai. Kad ir izveidots valodas abstraktās sintakses modelis, tiek specificēta valodas konkrētā sintakse. Tipiski tas nozīmē izveidot valodas konkrētās sintakses modeli un pēc tam nodibināt atbilstības starp abstraktās sintakses un konkrētās sintakses modeļiem. Ja izstrādājamajos rīkos nepieciešama papildus funkcionalitāte, tad to ir iespējams noprogrammēt, bet tas lielākoties nav vienkārši, jo tas nozīmē iejaukties ģenerētā kodā vai iejaukties platformas realizācijā. Kopumā jāsecina, ka šāds rīku specificēšanas process ir garš, kā rezultātā rīku izstrāde ir sarežģīta un grūti saprotama.

Atšķirīgu rīku būves pieeju izmanto *MetaEdit*, kas paredz rīka valodu specificēt ar konkrētās sintakses modeli uzreiz, tādējādi būtiski saīsinot un vienkāršojot rīku izstrādes procesu. Galvenais secinājums – šādā veidā rīkus ir ērti specificēt, bet *MetaEdit* risinājumā ir ļoti ierobežotas iespējas papildināt rīku funkcionalitāti.

Darbā piedāvātais risinājums ļauj rīka valodu specificēt ar konkrētās sintakses modeli (t.i., konfiguratora grafisko modeli) un nepieciešamības gadījumā rīkam specifisko funkcionalitāti pievienot caur paplašinājuma punktiem, turklāt, nevis tikai pievienojot jaunas rīkjoslas pogas, uznirstošās izvēlnes, utt. (caur vienkāršajiem paplašinājuma punktiem), bet arī iejaucoties interpretatora izpildes laikā ar specifiskām transformācijām (caur saliktajiem paplašinājuma punktiem), tādējādi papildinot interpretatora standarta uzvedību. Kopējais secinājums ir, ka promocijas darbā aprakstītā rīku būves pieeja, kas apvieno gan iespēju specificēt rīku valodas ar konkrētās sintakses modeli, gan arī dinamiski papildināt rīku funkcionalitāti, ir oriģināla, kas nevienā platformā iepriekš nav realizēta.

Nobeigums

Šis promocijas darbs ir noslēdzošais posms projektam, kura ietvaros bija paredzēts izstrādāt jaunas paaudzes rīku būves platformu, un autora galvenais uzdevums bija izstrādāt konfiguratoru, ar kuru var vienkārši un ātri definēt plašu DSML rīku klasi. Lai izpildītu šo uzdevumu, par pamatu ir ņemtas platformas komponentes, kas ir tapušas projekta izstrādes laikā un kuru autori ir citi projekta dalībnieki. Vispārīgi skatoties, ar šīm komponentēm pietiktu, lai varētu būvēt DSML rīkus, proti, katram rīkam būtu jāizstrādā savs modeļu transformāciju komplekts, kas veic lietotāju notikumu apstrādi atbilstoši rīka specifikācijai. Taču tas ir darbietilpīgs process un tādēļ izvirzās jautājums, vai šo procesu nevar būtiski vienkāršot. Promocijas darbā ir izstrādāts šīs problēmas risinājums, un tas paredz sekojošas lietas:

- izstrādāt formālu DSML rīku specificēšanas valodu;
- izstrādāt šīs valodas interpretatoru kā universālo transformāciju kopumu, kas no rīka specifikācijas izveido strādājošu rīku;
- ātrākai un ērtākai specifikāciju (jeb specificēšanas valodas modeļu) uzdošanai izstrādāt konfiguratoru.

Lai izstrādātu specificēšanas valodu, darbā tika aplūkotas un salīdzinātas divas alternatīvas metodes - specializācijas un konkretizācijas metodes. Specializācijas metode, izmantojot UML stereotipu jēdzienu, ļauj precīzi definēt tās klašu diagrammas, kuras ir likumīgas DSML rīku specifikācijas. Savukārt konkretizācijas metode paredz izstrādāt universālu rīku definēšanas metamodeli, kura instances kalpo par konkrētu DSML rīku specifikācijām. Lai izvēlētos realizācijas variantu, abas pieejas tika salīdzinātas un tika secināts, ka ar konkretizācijas metodi izveidotajam rīku definēšanas metamodelim ir vieglāk realizēt modeļu migrēšanu starp dažādām rīka versijām, un tādēļ šī metode tika izvēlēta par realizācijas variantu.

Lai specifikāciju valodas modeļus pārvērstu par strādājošiem rīkiem, tika izstrādāts universāls interpretators kā universālo transformāciju kopums, kas universāli apstrādā lietotāja notikumus, balstoties uz rīka specifikāciju. Savukārt, lai varētu neaprobežoties ar fiksētu interpretatora funkcionalitāti, interpretatorā tika realizēts paplašinājuma punktu mehānisms, kas ļauj konkrētā rīkā realizēt tam specifisku papildfunkcionalitāti, izstrādājot rīkam specifiskas transformācijas. Jāpiezīmē, ka platformā ir precīzi atklāta datu iekšējā struktūra, tādējādi ļaujot transformācijām viegli piekļūt modeļu datiem un tos mainīt. Tā rezultātā ar interpretatoru var izstrādāt dažādas sarežģītības rīkus, sākot ar tik vienkāršiem kā blokshēmu redaktors, kura realizēšanai pietiek ar specifikācijas uzdošanu un papildus

transformāciju programmēšana nav nepieciešama, un beidzot ar tādiem rīkiem kā konfigurators, ar kuru var izstrādāt citus DSML rīkus.

Lai atvieglotu specifikācijas uzdošanu, tika izstrādāts minētais konfigurators, jeb rīks, kas ļauj grafiskā formā specificēt DSML rīkus, nenolaižoties līdz rīku definēšanas metamodeļu instancēm. Tā galvenā ideja ir DSML valodas konkrētās sintakses modeli uzdodot „pa tiešo", nespecificējot valodas abstraktās sintakses modeli. Tādējādi atšķirībā no lielākās daļas citu platformu, kurās DSML valodu specificēšanai uzdod gan abstraktās, gan konkrētās valodas modeļus un pēc tam starp šiem modeļiem nodibina sarežģītas atbilstības, kuras vēl DSML valodu izmaiņu gadījumā arī ir jāmaina, konfiguratorā ir jāuzdod tikai viens modelis un nav jānodibina atbilstības starp vairākiem modeļiem. Tā kā plašai DSML rīku klasei nav nepieciešams veikt sarežģītus loģiskos aprēķinus un modeļu apstrādes, tad metode, ar kuru specificē tikai konkrētās sintakses modeli, ir piemērota DSML rīku izstādei, turklāt tas samazina gan rīka izstrādes laiku, gan vienkāršo pašu izstrādes procesu.

Vēl viena konfiguratora priekšrocība ir tā, ka tas kopā ar interpretatoru veido tehnoloģiju, kas ļauj būtiski paaugstināt rīku izstrādātāju produktivitāti, jo ar šo tehnoloģiju rīku izstrādi ir iespējams veikt iteratīvi, kam ir sekojošas priekšrocības. Pirmkārt, pa vienam definēt valodas konceptus un katrā solī pārbaudīt iegūto rezultātu. Otrkārt, mainīt un darbināt transformācijas (realizētas interpretējamā valodā Lua), iztiekot bez rīka pārstartēšanas un kompilēšana soļiem. Treškārt, migrēt modeļu versijas no vecākām rīka versijām uz jaunākām.

Šī tehnoloģija ir praktiski pārbaudīta un veiksmīgi pielietota šādu rīku izstrādē – ProMod, BiLingva, LuMod un OWLGrEd.

Autora publikāciju saraksts

1. J. Barzdins, *G.* Barzdins, K. Cerans, R. Liepins and A. Sprogis. *UML Style Graphical Notation and Editor for OWL 2*, Lecture Notes in Business Information Processing, Volume 64, Springer, 102-114, 2010 (SCOPUS)

2. A. Sprogis and J. Barzdins. *Specification, Configuration and Implementation of DSL Tool*, Frontiers in Artificial Intelligence and Applications, vol. 249, IOS Press, p. 330-343, 2013 (SCOPUS)

3. K. Cerans, R. Liepins, J. Ovcinnikova and A. Sprogis. *Advanced OWL 2.0 Ontology Visualization in OWLGrEd*, Frontiers in Artificial Intelligence and Applications, vol. 249, IOS Press, pp. 41-54, 2013 (SCOPUS)

4. 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 (SCOPUS)

5. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis and A. Zarins. *Domain Specific Languages for Business Process Management: a Case Study*, Proc. of Workshop on Domain-Specific Modeling, OOPSLA 2009, Florida, USA

6. J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. *An MDE-Based Graphical Tool Building Framework*, Scientific Papers, University of Latvia, 2010. Vol. 756 Computer Science and Information Technologies, 121–138 p

7. A. Sprogis. *The Configurator in DSL Tool Building*, Scientific Papers, University of Latvia, 2010. Vol. 756 Computer Science and Information Technologies, 173–192 p

8. J. Barzdins, *G.* Barzdins, K. Cerans, R. Liepins and A. Sprogis. *OWLGrEd: a UML Style Graphical Editor for OWL*, *CEUR* Workshop Proceedings 596, pp. 23-28, 2010 (SCOPUS)

9. J. Barzdins, *G.* Barzdins, K. Cerans, R. Liepins and A. Sprogis. *OWLGrEd: a UML Style Graphical Notation and Editor for OWL 2*, *CEUR* Workshop Proceedings 614, 2010 (SCOPUS)

10. J. Barzdins, K. Cerans, R. Liepins and A. Sprogis. *Advanced ontology visualization with OWLGrEd*, publicēta OWLED 2011, OWL: Experiences and Directions, 8th International Workshop, San Francisco, California, USA, 5-6 June, 2011

11. A. Sprogis, R. Liepins, J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, E. Rencis, A. Zarins. *GRAF: a Graphical Tool Building Framework*, ECMFA 2010 Tools and Consultancy track, Paris, France, 2010

12. K. Cerans, J. Barzdins, G. Barzdins, R. Liepins, J.Ovcinnikova, S. Rikacovs and A. Sprogis. *Graphical Schema Editing for Stardog OWL/RDF Databases using OWLGrEd/S,* CEUR-WS.org, 2012

13. R. Liepins, K. Cerans and A. Sprogis. *Visualizing and Editing Ontology Fragments with OWLGrEd,* I-SEMANTICS Poster and Demo Track, 2012

Izmantotā literatūra

[1] S. Kelly, J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, Wiley, 2008, p. 448.

[2] J. Sprinkle, B. Rumpe, H. Vangheluwe, G. Karsai, *Metamodelling - State of the Art and Research Challenges*. Model-Based Engineering of Embedded Real-Time Systems 2007: 57-76

[3] UML, http://www.uml.org

[4] BPMN, http://www.bpmn.org

[5] SysML, http://www.omgsysml.org/

[6] S. Cook, G. Jones, S. Kent, and A. C. Wills, *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007

[7] S. Robert, S. Gerard, F. Terrier, F. Lagarde, *A Lightweight Approach for DomainSpecific Modeling Languages Design*, Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on, pp.155-161, 27-29 Aug. 2009.

[8] Eclipse Modeling, http://www.eclipse.org/modeling/emf/

[9] Visualization and Modeling SDK (DSL Tools), http://code.msdn.microsoft.com/windowsdesktop/Visualization-and-Modeling-313535db

[10] MetaCase - MetaEdit+ Modeler DSM Tool, http://www.metacase.com/mep/

[11] M. Brambilla, J. Cabot, M. Wimmer, *Model Driven Software Engineering in Practice*, Morgan & Claypool Publishers, 2012

[12] http://www.gradetools.com/

[13] J. Barzdins, E. Rencis, S. Kozlovics. *The Transformation-Driven Architecture*, The 8th OOPSLA Workshop on Domain-Specific Modeling, October 19 - October 20, 2008, Nashville, TN.

[14] S. Kozloviča promocijas darbs *The Transformation-Driven Architecture and its Graphical Presentation Engines*.

[15] 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 (SCOPUS)

[16] J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. *An MDE-Based Graphical Tool Building Framework*, Scientific Papers, University of Latvia, 2010. Vol. 756 Computer Science and Information Technologies, 121–138

[17] 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.

[18] K. Freivalds and P. Kikusts. *Optimum layout adjustment supporting ordering constraints in graph-like diagram drawing*. In Proceedings of the Latvian Academy of Sciences, Section B, volume 55, pages 43-51, 2001.

[19] S. Kozlovics. *A Dialog Engine Metamodel for the Transformation-Driven Architecture*. Computer Science and Information Technologies, Scientific Papers University of Latvia, Vol. 756, pp. 151-170., 2010

[20] 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.

[21] S. Kozlovics. *Calculating The Layout For Dialog Windows Specified As Models*. In Scientific Papers, University of Latvia, volume 787, 2012.

[22] R. Ierusalimschy, *Programming in Lua*. Lightning Source UK Ltd., Milton Keynes, UK, 2006.

[23] R. Liepins. *lQuery: A Model Query and Transformation Library*. Computer Science and Information Technologies, Scientific Papers University of Latvia, Vol. 770, pp. 27-46.

[24] R. Liepins. *Library for model querying: IQuery*. Proceedings of the 12th Workshop on OCL and Textual Modelling, pp. 31-36., ACM, 2012.

[25] The Implementation of Lua 5.0, http://www.lua.org/doc/jucs05.pdf

[26] G. Michaelson, *An Introduction to Functional Programming Through Lambda Calculus*, Dover Publications, 2011

[27] jQuery, http://jquery.com/

[28] JavaScipt Tutorial, http://www.w3schools.com/js/

[29] XPath Tutorial, http://www.w3schools.com/xpath/

[30] S. Si Alhir, *Guide to Applying the UML*, Springer, 2002

[31] A. Sprogis and J. Barzdins. *Specification, Configuration and Implementation of DSL Tool*, Frontiers in Artificial Intelligence and Applications, vol. 249, IOS Press, p. 330-343, 2013 (SCOPUS)

[32] OCL, http://www.omg.org/spec/OCL/

[33] Grade2 – Graphical Tool Building Framework, grade2.lumii.lv

[34] A. Sprogis, R. Liepins, J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, E. Rencis, A. Zarins. *GRAF: a Graphical Tool Building Framework*, ECMFA 2010 Tools and Consultancy track, 2010, Paris, France

[35] A. Sprogis. *The Configurator in DSL Tool Building*, Scientific Papers, University of Latvia, 2010. Vol. 756 Computer Science and Information Technologies, 173–192 p

[36] B. Efron, R.J. Tibshirani, „An Introduction to the Bootstrap”, Chapman & Hall/CRC, 1994, 436 p.

[37] The Lx transformation language set home page, http://lx.mii.lu.lv/

[38] 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

[39] LPeg.re - Regex syntax for LPEG, http://www.inf.puc-rio.br/~roberto/lpeg/re.html

[40] LPeg Parsing Expression Grammars For Lua, http://www.inf.puc-rio.br/~roberto/lpeg/

[41] G. Karnitis, J. Bicevskis, J. Cerina-Berzina. *Information systems development based on visual Domain Specific Language BiLingva*, Proceedings of 4th IFIP TC2 Central and Eastern European conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland

[42] E. Renča promocijas darbs *Uz modeļu transformācijām balstītu rīku būves metožu izstrāde un realizācija*.

[43] J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis and A. Zarins. *Domain Specific Languages for Business Process Management: a Case Study*, Proc. of Workshop on Domain-Specific Modeling, OOPSLA 2009, Florida, USA

[44] Apache POI - the Java API for Microsoft Documents, http://poi.apache.org/

[45] OWLGrEd – Editor for Compact UML-style OWL Graphic Notation, owlgred.lumii.lv

[46] J. Barzdins, G. Barzdins, K. Cerans, R. Liepins and A. Sprogis. *UML Style Graphical Notation and Editor for OWL 2*, Lecture Notes in Business Information Processing, Volume 64, Springer, 102-114, 2010 (SCOPUS)

[47] J. Barzdins, *G.* Barzdins, K. Cerans, R. Liepins and A. Sprogis. *OWLGrEd: a UML Style Graphical Notation and Editor for OWL 2*, *CEUR* Workshop Proceedings 614, 2010 (SCOPUS)

[48] J. Barzdins, *G.* Barzdins, K. Cerans, R. Liepins and A. Sprogis. *OWLGrEd: a UML Style Graphical Editor for OWL*, *CEUR* Workshop Proceedings 596, pp. 23-28, 2010 (SCOPUS)

[49] K. Cerans, R. Liepins, J.Ovcinnikova and A. Sprogis. *Advanced OWL 2.0 Ontology Visualization in OWLGrEd*, Frontiers in Artificial Intelligence and Applications, vol. 249, IOS Press, pp. 41-54, 2013 (SCOPUS)

[50] J. Barzdins, K. Cerans, R. Liepins and A. Sprogis. *Advanced ontology visualization with OWLGrEd*, publicēta OWLED 2011, OWL: Experiences and Directions, 8th International Workshop, San Francisco, California, USA

[51] K. Cerans, J. Barzdins, G. Barzdins, R. Liepins, J.Ovcinnikova, S. Rikacovs and A. Sprogis. *Graphical Schema Editing for Stardog OWL/RDF Databases using OWLGrEd/S,* CEUR-WS.org, 2012

[52] R. Liepins, K. Cerans and A. Sprogis. *Visualizing and Editing Ontology Fragments with OWLGrEd,* I-SEMANTICS Poster and Demo Track, 2012

[53] OWL 2 Web Ontology Language Document Overview, http://www.w3.org/TR/owl2-overview/

[54] An Introduction to Ontologies and Ontology, www.springer.com/cda/content/document/cda_downloaddocument/9780857297235-c1.pdf

[55] EngineeringThe Protege Ontology Editor and Knowledge Acquisition System, http://protege.stanford.edu/

[56] B. Motik, P. F. Patel-Schneider, B. Parsia, *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax*, OWL 2 Web Ontology Language. World Wide Web Consortium. Retrieved 18 April 2010.

[57] B. Motik, B. Parsia, P. F. Patel-Schneider, *OWL 2 Web Ontology Language XML Serialization*, OWL 2 Web Ontology Language. World Wide Web Consortium. Retrieved 18 April 2010.

[58] P. F. Patel-Schneider, B. Motik, *OWL 2 Web Ontology Language Mapping to RDF Graphs*, OWL 2 Web Ontology Language. World Wide Web Consortium. Retrieved 18 April 2010.

[59] M. Horridge, Matthew, P. F. Patel-Schneider, *OWL 2 Web Ontology Language Manchester Syntax*, W3C OWL 2 Web Ontology Language. World Wide Web Consortium. Retrieved 18 April 2010.

[60] J. Barzdins, E. Rencis, A. Sostaks. *Towards Human-Executable Business Process Modeling*. Frontiers in Artificial Intelligence and Applications, Volume 249: Databases and Information Systems VII, pp. 149-163, IOS Press, 2013

[61] Graphical Modeling Framework, http://www.eclipse.org/modeling/gmp/

[62] Graphical Editing Framework (GEF), http://www.eclipse.org/gef/

[63] Eclipse, http://www.eclipse.org/

[64] A. Kalnins, J. Barzdins, and E. Celms. *Model transformation language MOLA*, Lecture Notes in Computer Science Volume 3599, pages 62-76, 2005

[65] Microsoft Visual Studio, http://www.microsoft.com/visualstudio/eng/visual-studio-update

[66] S. Kelly, K. Lyytinen, M. Rossi. *MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment*, Proceedings of CAiSE'96, 8th Intl. Conference on Advanced Information Systems Engineering, Lecture Notes in Computer Science 1080, Springer-Verlag, pp. 1–21, 1996.

[67] Getting started with MERL, http://www.metacase.com/support/45/manuals/mwb/Mw-5_3_1.html

[68] Increase Productivity of Software Development with MetaEdit+ Domain-Specific Modeling Tool, http://www.metacase.com/increase.html

[69] Obeo Designer: Domain Specific Modeling for Software Architects, http://www.obeodesigner.com/

[70] E. Juliot, J. Benois. *How to build Eclipse DSM without being an expert developer?,* Obeo Designer Whitepaper

[71] S. Robert, S. Gerard, F. Terrier, F. Langarde, *A Lightweight Approach for Domain-Specific Modeling Languages Design*, Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on, pp , Patras, 2009

[72] A. El Kouhen, C. Dumoulin, S. Gerard, P. Boulet, *Evaluation of Modeling Tools Adaptation*,http://hal.archives-ouvertes.fr/docs/00/70/68/41/PDF/Evaluation_of_Modeling_Tools_Adaptation.pdf

[73] EuGENia Live, http://eugenialive.herokuapp.com/

[74] L.M. Rose, D.S. Kolovos, R.F. Paige. *EuGENia Live: A Flexible Graphical Modelling Tool,* Proc. 1st Extreme Modelling (XM) workshop, co-located with the International Conference on Model Driven Engineering Languages and Systems (MoDELS), 2012, Innsbruck, Austria

[75] Graphiti Home, http://www.eclipse.org/graphiti/

[76] J. Davis, *GME: the generic modeling environment*, OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2003

[77] GME: Generic Modeling Environment | Institute for Software Integrated Systems, http://www.isis.vanderbilt.edu/Projects/gme

[78] IBM Rational Software Architect, http://www.ibm.com/developerworks/rational/products/rsa/

[79] Visio Professional 2013, visio.microsoft.com

1. Pielikums

Prezentācijas dziņa un tā metamodeļa funkciju bibliotēka

current_diagram ()

Sameklē aktīvo diagrammu

Atgriež:

diagrammas objekts

Izsaukuma piemērs:

diagram = utilities.current_diagram()

create_command (command_name, attrs)

Izveido komandai atbilstošo objektu

Parameteri:

- *command_name*: komandas nosaukums
- *attrs*: atribūtu saraksts

Atgriež:

komandas objekts

Izsaukuma piemērs:

cmd_obj = utilities.create_command(”CloseDgrCmd”)

execute_cmd_obj (command)

Izpilda komandai atbilstošo objektu

Parameteri:

- *command*: komandai atbilstošais objekts

Izsaukuma piemērs:

utilities.execute_cmd_obj(cmd_obj)

execute_cmd (command_name, attrs)

Izveido un izpilda komandai atbilstošo objektu

Parameteri:

- *command_name*: komandas nosaukums
- *attrs*: atribūtu saraksts

Izsaukuma piemērs:

utilities.execute_cmd(”CloseDgrCmd”, {graphDiagram = diagram})

active_elements ()

Sameklē diagrammā iezīmētos elementus

Atgriež:

iezīmēto objektu kolekcija

Izsaukuma piemērs: element = utilities.active_elements()
navigate (elem) Navigē no elementa uz diagrammu **Parameteri:** • *elem*: elements, no kura navigēs **Izsaukuma piemērs:** utilities.navigate(element)
navigate_or_properties (elem) Ja ir iespējams, navigē no elementa uz diagrammu, ja ne, tad atver dialogu logu **Parameteri:** • *elem*: elements, no kura navigēs, vai, kuram atvērs dialogu logu **Izsaukuma piemērs:** utilities.navigate_or_properties(element)
open_diagram (diagram) Atver diagrammu **Parameteri:** • *diagram*: diagrammu, kuru atvērs **Izsaukuma piemērs:** utilities.open_diagram(diagram)
align_selected_boxes () Izlīdzina aktīvās diagrammas kastes **Izsaukuma piemērs:** utilities.align_selected_boxes()
set_elem_style (elem, list) Maina konkrētas elementa stila vērtības **Parameteri:** • *elem*: elements, kuram maina stilu • *list*: saraksts ar atribūtiem, kurus maina **Izsaukuma piemērs:** utilities.set_elem_style(element, {bkgColor = 12419151, lineColor = 9067831})
reroute (elems) Pārzīmē līnijas **Parameteri:** • *elems*: elementu kopa, kuriem ir jāpārzīmē līnijas

Izsaukuma piemērs: utilities.reroute (edge)
symbol_style () Atver stila kasti un, ja vajag nomaina, aktīvā elementa stilu **Izsaukuma piemērs:** utilities.symbol_style()
symbol_style_for_collection () Atver stila kasti un, ja vajag nomaina, visiem kolekcijas elementiem stilu **Izsaukuma piemērs:** utilities.symbol_style_for_collection()
ShowInformationBarCommand (msg) Parāda ziņojumu **Parameteri:** • *msg*: ziņojuma teksts **Izsaukuma piemērs:** utilities.ShowInformationBarCommand(”Šis ir piemērs”)
get_element_from_compartment (compart) Sameklē elementa objektu, kuram šis compartments pieder **Parameteri:** • *compart*: compartment objekts **Atgriež:** elementa objekts **Izsaukuma piemērs:** element = utilities.get_element_from_compartment(compart)
set_diagram_caption (diagram, name, is_refresh_needed) Uzstāda diagrammas nosaukumu **Parameteri:** • *diagram*: diagramma, kurai nomaina nosaukumu • *name*: nosaukums • *is_refresh_needed*: norāda, vai atbilstošā koka virsotne ir jāpārzīmē **Izsaukuma piemērs:** utilities.set_diagram_caption(diagram, ”Jauna Diagramma”, true)
get_diagram_from_element (elem) Sameklē diagrammas objektu, kurai šis elements pieder **Parameteri:**

- *elem*: Element objekts, no kura meklē diagrammu

Atgriež:

diagrammas objekts

Izsaukuma piemērs:

diagram = utilities.get_diagram_from_element(element)

get_diagram_from_compartment (compart)

Sameklē diagrammas objektu, kurai šis compartments pieder

Parameteri:

- *compart*: Compartment objekts, no kura meklē diagrammu

Atgriež:

diagrammas objekts

Izsaukuma piemērs:

diagram = utilities.get_diagram_from_compartment(compart)

get_obj_type (obj)

Sameklē objekta tipu

Parameteri:

- *obj*: objekts

Atgriež:

objekta tipa objekts

Izsaukuma piemērs:

elem_type = utilities.get_obj_type (element)

generate_create_instance_code (obj)

Uzģenerē kodu, kas izveido objektu

Parameteri:

- *obj*: objekts, kura kodu ģenerē

Atgriež:

kods, kas izveido padoto objektu

Izsaukuma piemērs:

code_creating_element = utilities.generate_create_instance_code(element)

make_obj_to_var (obj)

Izveido objekta mainīgā kodu

Parameteri:

- *obj*: objekts, kuram veido kodu

Atgriež:

objekta kods

Izsaukuma piemērs:
element_variable_name = utilities.make_obj_to_var(element)

Dialogu logu dzinēja un tā metamodeļa bibliotēka

add_form (attr_table)
Izveido formu **Parameteri:** • *attr_table*: tabula, kas satur klases D#Form atribūtu vērtības **Atgriež:** formas objekts **Izsaukuma piemērs:** form = dialog_utilities.add_form({id = "new_form", width = 500})
add_button (container, attr_table, events)
Pievieno pogu **Parameteri:** • *container*: komponente, uz kuras poga tiks izveidota • *attr_table*: tabula, kas satur klases Button atribūtu vērtības • *events*: tabula, kuras indekss ir notikuma nosaukums un vērtība ir notikumu apstrādājošās transformācijas adrese **Atgriež:** pogas objekts **Izsaukuma piemērs:** button = dialog_utilities.add_button(form, {id = "close_button", caption = "Close"}, {Click = "lua.utilities.close_form"})
get_component_by_id (id)
Meklē dialoga loga komponenti pēc id vērtības **Parameteri:** • *id*: komponentes id vērtība **Atgriež:** atrastās komponentes objekts **Izsaukuma piemērs:** component = dialog_utilities.get_component_by_id("new_form")
get_component_from_container_by_id (container, id)
Meklē dialoga loga komponenti no konkrēta vecāka pēc id vērtības **Parameteri:**

• *container*: komponente, no kuras meklē • *id*: komponentes id vērtība **Atgriež:** atrastās komponentes objekts **Izsaukuma piemērs:** component = dialog_utilities.get_component_from_container_by_id(form, ”close_button”)
get_component_from_container_by_attr_name (container, attr_name, value) Meklē dialoga loga komponenti no konkrēta vecāka pēc atribūta vērtības **Parameteri:** • *container*: komponente, no kuras meklē • *attr_name*: atribūta nosaukums • *value*: atribūta vērtība **Atgriež:** atrastās komponentes objekts **Izsaukuma piemērs:** component = dialog_utilities.get_component_from_container_by_attr_name (form, ”id”, ”close_button”)
fill_list_combo_box (box, value_table) Aizpilda ListBox vai ComboBox no tabulas **Parameteri:** • *box*: ListBox vai ComboBox objekts • *value_table*: itemu saraksts **Izsaukuma piemērs:** dialog_utilities.fill_list_combo_box(combo_box, {”item1”, ”item2”, ”item3”})
delete_event (ev) Izdzēš notikuma objektu. Ja objekts nav norādīts, tad izdzēš visus objektus **Parameters:** • *ev*: notikuma objekts **Izsaukuma piemērs:** dialog_utilities.delete_event()
close_form (form_id) Aizver formu **Parameters:** • *form_id*: formas, kuru jāaizver id vērtība

Izsaukuma piemērs:

dialog_utilities.close_form(”new_form”)

Koka dzinēja un tā metamodeļa bibliotēka

add_tree_node (parent, attr_list, is_selected)

Izveido jaunu koka virsotni

Parameteri:

- *parent*: objekts, pie kura piesaista koka jauno koka virsotni
- *attr_list*: saraksts ar koka virsotnes atribūtu nosaukumiem un to vērtībām
- *is_selected*: norāda, vai jauno koka virsotni vajag aktivizēt

Atgriež:

koka virsotnes objekts

Izsaukuma piemērs:

node = dialog_utilities.add_tree_node(parent_node, {caption = ”Jauna Virsotne”}, true)

refresh (obj)

Pārzīmē koka komponenti

Parameteri:

- *obj*: koka komponentes objekts, kuru pārzīmē

Izsaukuma piemērs:

dialog_utilities.refresh(node)

delete_tree_node (node)

Izdzēš koka virsotni

Parameteri:

- *node*: koka virsotne

Izsaukuma piemērs:

dialog_utilities.delete_tree_node(node)

select_node (node)

Aktivizē koka virsotni

Parameteri:

- *node*: koka virsotne

Izsaukuma piemērs:

dialog_utilities.select_node()

get_selected_tree_node ()

Sameklē iezīmēto koka virsotni

Izsaukuma piemērs:

selected_node = dialog_utilities.get_selected_tree_node()

Palīgfunkciju bibliotēka

get_class_name (obj)

Atgriež objekta klases nosaukumu

Parameteri:

- *obj*: objekts, kura klases nosaukumu meklē

Atgriež:

objekta klases nosaukums

Izsaukuma piemērs:

element_name = utilities.get_class_name (node)

execute_fn (function_path, ...)

Izpilda patvaļīgu lua funkciju

Parameteri:

- *function_path*: ceļš uz izpildāmo funkciju
- ...: paremetri, kas tiks nodoti izpildāmajai funkcijai

Izsaukuma piemērs:

utilities .execute_fn(”interpreter.Properties”, node)

is_table_empty (list)

Pārbauda, vai saraksts ir tukšs

Parameteri:

- *list*: saraksts

Izsaukuma piemērs:

is_table_empty = utilities.is_table_empty(list)

copy_objects (source_elem, target_elem)

Pārkopē vienu lQuery objektu uz otru

Parameteri:

- *source_elem*: elements, no kura kopē
- *target_elem*: elements, kurā kopē

Izsaukuma piemērs:

utilities.copy_objects(node, copy_of_node)

make_obj_copy (obj)

Uztaisa lQuery objekta klonu

Parameteri:

• *obj*: objekts, kuru klonē **Atgriež:** objekts, kas ir padotā objekta klons **Izsaukuma piemērs:** node_clone = utilities.make_obj_copy(node)
make_reverse_list (list) Atgriež saraksta elementu pretējā secībā **Parameteri:** • *list*: saraksts **Atgriež:** saraksts pretējā secībā **Izsaukuma piemērs:** reverse_list = utilities.make_reverse_list(list)