RĪGAS TEHNISKĀ UNIVERSITĀTE
Datorzinātnes un informācijas tehnoloģijas fakultāte
Lietišķo datorsistēmu institūts

Artūrs BARTUSEVIČS
Doktora studiju programmas «Datorsistēmas» doktorants

PROGRAMMATŪRAS KONFIGURĀCIJAS PĀRVALDĪBAS MODEĻVADĀMU RISINĀJUMU IZSTRĀDE UN REALIZĀCIJA

Promocijas darbs

Zinātniskais vadītājs
profesors *Dr. habil. sc. ing.*
LEONĪDS NOVICKIS

Rīga 2015

ANOTĀCIJA

Mūsdienās, attīstoties *Agile* metodoloģijai un mākoņskaitļošanas tehnoloģijām, rodas nepieciešamība pēc iespējas ātrāk piegādāt gatavu programmatūru pasūtītājam. Konfigurācijas pārvaldība ir disciplīna, kas kontrolē programmatūras evolūcijas procesu un sniedz vadlīnijas, kā no izejas koda uzbūvēt strādājošu programmatūru. Ir nepieciešamas metodoloģijas, kas ļaus samazināt konfigurācijas pārvaldības automatizācijas ieviešanas laiku un paaugstinās procesu efektivitāti.

Promocijas darba pētījuma objekts ir programmatūras izstrādes projekti un konfigurācijas pārvaldības procesi tajos. Promocijas darba mērķis ir izstrādāt modeļvadāmu pieeju, modeļus un metodes konfigurācijas pārvaldības automatizācijas ieviešanai, kas ļautu samazināt automatizācijas ieviešanas laiku, atkārtoti izmantojot uzņēmumā esošos risinājumus konfigurācijas pārvaldības automatizācijai.

Darba teorētiskajā daļā tika definēti konfigurācijas pārvaldības galvenie uzdevumi un izanalizētas esošās pieejas, un tā rezultātā tika noteiktas pieeju attīstības tendences. Turpinājumā tika izpētītas risinājumu attīstības tendences un, balstoties uz pētījumu rezultātiem, definēti esošo rīku un pieeju trūkumi.

Balstoties uz minētās analīzes rezultātu, tika izstrādāta jauna modeļvadāma pieeja konfigurācijas pārvaldības automatizācijas ieviešanai ar modeļu palīdzību. Pieejas realizācijai tika izstrādāta jauna metodoloģija konfigurācijas pārvaldības procesu automatizācijai.

Darba praktiskajā daļā tika izstrādāts prototips modeļu attēlošanai. Tika veikti eksperimenti jaunas metodoloģijas testēšanai. Eksperimentu rezultātā izdevās noteikt jaunas metodoloģijas ieguvumus, definēt atšķirības no citiem konfigurācijas pārvaldības automatizācijas ieviešanas risinājumiem un ieviešanas riskus.

Darba galvenie rezultāti ir modeļvadāma pieeja un metodoloģija, kas ļauj ieviest konfigurācijas pārvaldības automatizāciju, atkārtoti izmantojot jau esošus risinājumus atsevišķiem procesa posmiem.

Par promocijas darba galvenajiem rezultātiem tika nolasīti 11 referāti 10 starptautiskās konferencēs, tie ir atspoguļoti 12 zinātniskajos rakstos.

Darbā ir ievads, 5 nodaļas un secinājumi. Tā pamattekstā ir 238 lappuses, 55 attēli un 32 tabulas, kā arī 2 pielikumi un literatūras saraksts ar 117 nosaukumiem.

Darbs izstrādāts ar daļēju valsts pētījumu programmas «Kiberfizikālās sistēmas, ontoloģijas un biofotonika drošai & viedai pilsētai un sabiedrībai» (*SOPHIS*) atbalstu, līgums Nr. 10-4/VPP-4/11.

ANNOTATION

Nowadays during improvement of Agile, important requirement is organization of ready software delivery to customer as soon as possible. Software configuration management is a discipline that controls software evolution process and provides guidelines how to build a valid software from a source code. There are lack of methodologies allows to reduce implementation time of software configuration management process and improve quality of mentioned process.

The thesis describes software development projects and software configuration management processes in these projects. Aim of the thesis is development of approach, models and methods that allows to reduce implementation time of software configuration management process increasing reuse of existing solutions.

In the theoretical part of the thesis a set of main software configuration management tasks is defined and improvement trends are underlined. Based on last trends of approaches and tools for supporting of software configuration management, advantages and disadvantages of mentioned approaches and tools are analyzed.

Based on nowadays trends, advantages and disadvantages of approaches and tools for software configuration management, new approach is designed. The scope of new approach is design of software configuration management process by models to reduce implementation time using existing solutions and tools.

In the empirical part of the thesis new software is developed for desing of software configuration management models. There are five different experiments of implementation of new approach in software development projects. Experiments allows to detect gains of new approach, differences from other related solutions and risks of provided approach.

The thesis results are models and methods for implementation of software configuration management process using existing solutions for different tasks of process.

The main results of the thesis have been presented at 10 international conferences, as well as they are reflected in 12 scientific papers.

The thesis includes introduction, 5 chapters and conclusions. It contains 238 pages, 55 figures and 32 tables in the main text, 2 attachments and 117 titles large bibliografy.

SAĪSINĀJUMI

MTM	*Model – Transformation – Model* (angļu val.)
EAF	*Environment – Action – Framework* (angļu val.)
MDD	*Model-Driven Development* (angļu val.)
MDA	*Model-Driven Architecture* (angļu val.)
EM	*Environment Model* (angļu val.)
PIAM	*Platform Independent Action Model* (angļu val.)
PSAM	*Platform Specific Action Model* (angļu val.)
SCBM	*Source Code Branching Model* (angļu val.)
SM	*Service Model* (angļu val.)
PIEM	*Platform Independent Environment Model* (angļu val.)
CM	*Code Model* (angļu val.)
CIM	*Computing Independent Model* (angļu val.)
PIM	*Platform Independent Model* (angļu val.)
PSM	*Platform Specific Model* (angļu val.)

SATURS

IEVADS 7

1. PROGRAMMATŪRAS KONFIGURĀCIJAS PĀRVALDĪBAS IZPĒTE 19

1.1. Jēdzienu skaidrojums 19
1.2. Programmatūras konfigurācijas pārvaldības definīcija 21
1.3. Konfigurācijas vienumu identifikācija 28
1.4. Versiju kontrole, paralēla izstrāde un metriku savākšana 34
1.5. Produkta būvējumi un instalācija 46
1.6. Mūsdienīgas programmatūras konfigurācijas pārvaldības iezīmes 54
1.7. Nodaļas kopsavilkums 57

2. MODEĻVADĀMA KONFIGURĀCIJAS PĀRVALDĪBA 59

2.1. Modeļvadāmas arhitektūras vispārīgie principi 59
2.2. Konfigurācijas pārvaldības modeļvadāmie risinājumi 64
2.3. Nodaļas kopsavilkums 74

3. *MTM* PIEEJAS UN *EAF* METODOLOĢIJAS IZSTRĀDE 76

3.1. *MTM* pieejas definīcija un vispārīgs apraksts 76
3.2. *EAF* metodoloģija *MTM* pieejas realizācijai 79
3.3. *EAF* metodoloģijas izstrādes pamatojums 81
3.4. *EAF* metodoloģijas modeļu apraksts 83
3.5. Vižu modeļa meta-modeļa izstrāde 87
3.6. No platformas neatkarīga darbību meta-modeļa izstrāde 106
3.7. Platformas specifiskā darbību modeļa realizācija 113
3.8. Izejas koda zarošanas modelis 116
3.9. Servisu modelis 123
3.10. *EAF* metodoloģijas modeļu transformācijas 125
3.11. *EAF* metodoloģijas kopsavilkums 131
3.12. Nodaļas kopsavilkums 132

4. MODEĻVADĀMAS KONFIGURĀCIJAS PĀRVALDĪBAS METODOLOĢIJAS APROBĀCIJA UN TESTĒŠANA 134

4.1. Modeļvadāmas pieejas rezultātu aprobācija un testēšanas gaita 134
4.2. Eksperimentu sagatavošana un plāns 135
4.3. Konfigurācijas pārvaldības modeļu veidošanas prototips 142
4.4. *EAF* metodoloģijas modeļu veidošanas piemērs 145
4.5. Modeļvadāmas pieejas vērtēšanas kritēriji 156
4.6. Eksperimentu apraksts un rezultātu apkopojums 162
4.7. *EAF* metodoloģijas priekšrocības un trūkumi, balstoties uz eksperimentu rezultātiem 172
4.8. *EAF* metodoloģijas iteratīvas izstrādes un atkārtotu eksperimentu nepieciešamības pamatojums 175
4.9. Nodaļas kopsavilkums 176

5. *EAF* METODOLOĢIJAS UZLABOŠANA UN IEVIEŠANAS REKOMENDĀCIJAS 178

5.1. *EAF* metodoloģijas uzlabotā versija 178
5.2. Eksperimentu plāns uzlabotās *EAF* versijas testēšanai 200
5.3. *EAF* metodoloģijas uzlabotās versijas testēšana 201
5.4. *EAF* metodoloģijas trūkumu novēršanas pasākumi 206

5.5. *EAF* metodoloģijas uzlabotās versijas galvenie ieguvumi, atšķirības no citiem konfigurācijas pārvaldības risinājumiem un turpmākas uzlabošanas virzieni 210
5.6. Praktiskas rekomendācijas modeļvadāmajai konfigurācijas pārvaldības ieviešanai programmatūras izstrādes projektā 213
DARBA KOPĒJIE REZULTĀTI, SECINĀJUMI UN TURPMĀKIE PĒTĪJUMI 217
BIBLIOGRĀFISKAIS SARAKSTS 221
PIELIKUMI 234
1. pielikums 235
Projekta «*Test Solution*» vižu modelis *XML* formātā 235
2. pielikums 237
Projekta «*Test Solution*» *PIAM* modelis *XML* formātā 237

IEVADS

2009. gadā konferencē «*Velocity Conference*» tika prezentēts Džona Allspava (*John Allspaw*) un Pola Hamonda (*Paul Hammond*) referāts «10 instalācijas vienā dienā» (angļu val. *10 Deploys A Day*). Referātā bija uzsvērta problēma, ka, attīstoties spējo programmatūras izstrādes metodoloģijai (angļu val. *Agile*) un mākoņskaitļošanas tehnoloģijām, operācijas, kas sagatavo programmatūras būvējumus un laidienus, nespēj savlaicīgi piegādāt pasūtītājam gatavu produktu [CON 2015]. Minēta konference tiek uzskatīta par sākumu *DevOps* metodoloģijai, kuras mērķis ir paātrināt programmatūras būvējumu veidošanu un instalācijas, kā arī uzlabot to kvalitāti [AZO 2014]. Sākot no 2009. gada, *DevOp*s metodoloģijai veltīti pasākumi notiek regulāri. Paralēli tam strauji attīstās rīki, kas atvieglo un paātrina programmatūras būvējumu un instalācijas procesu [AZO 2014]. Viena no šādu rīku vadošajam speciālistēm Treisija Reigane (*Tracy Ragan*) savā rakstā [RAG 2014] iezīmē mūsdienīgas būvējumu un instalācijas rīku attīstības tendences. Rīkiem, kas atbalsta programmatūras būvējumus un instalācijas, jābūt modeļvadāmaiem, jo, attīstoties mākoņskaitļošanas tehnoloģijām, statiski skripti vairs nevar nodrošināt ātru un efektīvu programmatūras būvējumu un instalāciju mākoņos [RAG 2014]. Apliecinājums tam – kopš jau minētā 2009. gada programmatūras tirgū parādījās daudz rīku, kas atbalsta modeļvadāmu programmatūras būvējumu un instalāciju, piemēram, *Serena* un *Open Make* kompānijas produkti, kā arī daudzi citi [AZO 2014].

Programmatūras izstrādes uzņēmumi ne vienmēr var atļauties pāriet uz jaunākajiem programmatūras būvējumu un instalācijas rīkiem. Jaunākie rīki bieži ir komerciāli un piedāvā specifisku programmatūras projektu struktūru, lai vispār būtu iespējams rīku lietot. Tas nozīmē, ka uzņēmumam ir jāiegulda nauda un jāmaina esošā projektu struktūra. Šāda restrukturizācija bieži tiek uzskatīta par riskantu, līdz ar to pastāv tendence palikt pie veciem, taču gadiem pārbaudītiem automatizācijas risinājumiem [AIE 2010]. Turpinot lietot pārbaudītus risinājumus, uzņēmumi meklē veidus, kā uzlabot risinājumu efektivitāti. Konfigurācijas pārvaldības vadošie speciālisti [УДО 2011, AIE 2010] atzīmē, ka mūsdienās jaunie programmatūras izstrādes projekti attīstās ļoti strauji, tāpēc katrā jaunajā projektā nepieciešams pēc iespējas ātrāk ieviest automatizācijas procesus, kas sniedz kvalitatīvu atbalstu programmatūras būvējumu un instalācijas procesam.

Atgriežoties pie mūsdienīgiem rīkiem programmatūras būvējumu un instalāciju procesu atbalstam [AZO 2014], jāatzīmē, ka lielāka daļa no rīkiem koncentrējas vien uz programmatūras būvējumu un instalāciju, taču pievērš maz uzmanības citiem procesiem, kas

tieši ietekmē programmatūras būvējumu. Programmatūras konfigurācijas pārvaldība ir disciplīna, kas apskata visus procesus, kas ietekmē programmatūras būvējumu un atbilstoša būvējuma instalāciju un piegādi pasūtītājam. Kā norāda nozares vadošie speciālisti [AIE 2010, MET 2002, УДО 2011], uzbūvēt no izejas koda kvalitatīvu programmatūru ir iespējams tikai tad, ja kvalitatīvi tiek organizēti visi konfigurācijas pārvaldības procesi kopumā. Tāpēc šajā promocijas darbā pēc iespējas plašāk tiks apskatīts konfigurācijas pārvaldības jēdziens, lai identificētu pēc iespējas vairāk faktoru, kas ietekmē programmatūras būvējumus. Konfigurācijas pārvaldība promocijas darba kontekstā turpmāk tiks interpretēta kā vairāku procesu kopa, kas programmatūras izstrādes projektos īsteno tādus procesus kā izejas koda versiju kontrole, būvējumu un instalāciju pārvaldība, programmatūras piegāde pasūtītājam un citus saistītus procesus.

Analizējot mūsdienīgus konfigurācijas pārvaldības automatizācijas risinājumus un to attīstības tendences, jāatzīst ka risinājumi tiecas uz modeļvadāmas arhitektūras formātu (angļu val. *MDA – Model-Driven Architecture*). Pirmkārt, modeļvadāma pieeja, ko piedāvā *MDA*, ļauj samazināt cilvēcisku faktoru, pārejot no prasībām pie implementācijas [OSE 2011]. Otrkārt, attīstoties mākoņskaitļošanas tehnoloģijām, statiski programmatūras būvējumu skripti vairs neder, jo risinājums atrodas mākoņos un skripti nevar operēt ar absolūtām serveru adresēm un citām saistītām vērtībām [RAG 2014]. Apliecinājums tam ir fakts, ka lielāka daļa mūsdienīgu rīku, kas atbalsta konfigurācijas pārvaldības procesus, izmanto modeļvadāmas arhitektūras idejas [AZO 2014].

Tēmas aktualitāte

Konfigurācijas pārvaldības procesa būtiskākais rezultāts ir no izejas koda uzbūvēta programmatūra, kas tiek piegādāta pasūtītājam. Lai to paveiktu, konfigurācijas pārvaldības disciplīna veic programmatūras izejas koda pārvaldību un no tā uzbūvē strādājošu programmatūru. Ja kāda no šim darbībām notiek neveiksmīgi un pasūtītājs saņem nestrādājošu vai nekvalitatīvu programmatūru, attiecīgajam programmatūras izstrādes projektam zūd pievienotā vērtība. Šajā gadījumā pasūtītājam ir mazsvarīgi tas, cik daudz laika tika patērēts izstrādei un tas, ka izstrādi veica vadošie uzņēmuma izstrādātāji. Gala rezultāta nav, un tas diemžēl ir vienīgais, ko uztvers pasūtītājs. Tāpēc ir ārkārtīgi svarīgi, lai programmatūras izstrādes projektā konfigurācijas pārvaldības procesi būtu kvalitatīvi un efektīvi. Tieši tāpēc programmatūras izstrādes nozares kvalitātes standarti pieprasa sakārtotu un automatizētu konfigurācijas pārvaldību [AIE 2010]. Avotā [УДО 2011] ir minēts, ka viens no konfigurācijas pārvaldības procesa aktualitātes apliecinājumiem ir fakts, ka *CMMI* (angļu

val. *Capability Maturity Model Integration*) standartā konfigurācijas pārvaldības process ir tik pat svarīgs kā sakārtots izstrādes un testēšanas process. Konfigurācijas pārvaldība ir aprakstīta arī svarīgos kvalitātes standartos: *IEEE Std 12207-2008* un *ISO 9001:2000* [IEE 2015, ISO 2015]. Pēdējais ir izplatīts arī Latvijā, un tas liecina, ka uzņēmumi seko līdzi, lai konfigurācijas pārvaldības procesi no standartu viedokļa tiktu realizēti pietiekamā līmenī.

Konfigurācijas pārvaldības problēmas ir aktuālas tāpēc, ka, salīdzinot ar citām nozarēm, programmatūrā veikt izmaiņas ir salīdzinoši viegli. Piemēram, atvērot izejas koda failu, izdzēšot vienu rindu un saglabājot izmaiņas, jau ir iegūstama jauna programmatūras versija. Taču, piemēram, lai nomainītu detaļu datoram, ir nepieciešams relatīvi vairāk laika un arī izmaiņas pamanīt ir salīdzinoši vieglāk. Ņemot vērā, ka veikt izmaiņas programmatūrā ir salīdzinoši viegli, ir nepieciešama kvalitatīva izmaiņu pārvaldība, lai vēlāk varētu uzbūvēt programmatūru no izejas koda.

Mūsdienās programmatūras izstrādes uzņēmumiem ir nepieciešami risinājumi, kas būtu spējīgi:

- palielināt konfigurācijas pārvaldības procesu efektivitāti, samazinot automatizācijas ieviešanas laiku. Jaunajos programmatūras izstrādes projektos iespēju robežās atkārtoti jāizmanto esošu projektu automatizācijas risinājumi, no nulles izstrādājot tikai projekta specifiskus konfigurācijas pārvaldības uzdevumu automatizācijas;
- uzkrāt pieredzi, lietojot konfigurācijas pārvaldības automatizācijas risinājumus dažādos programmatūras izstrādes projektos. Ar katru jaunu projektu automatizācijas risinājumiem vajadzētu kļūt arvien efektīvākiem, jo ir apzināti veiksmīgi un neveiksmīgi lietošanas gadījumi iepriekšējos projektos un ir veikti atbilstoši uzlabojumi;
- kompleksi automatizēt visus konfigurācijas pārvaldības uzdevumus, ne tikai būvējumu un instalāciju veidošanu.

Problēmas nostādne

21. gadsimtā, kad strauji attīstās *Agile* programmatūras izstrādes pieeja un tiek izstrādātas apjomīgas un sarežģītas programmatūras, bieži jauna projekta sākums līdzinās sprādzienam. Jau pēc dažam dienām programmatūras pasūtītājs grib saņemt pirmo tās versiju. Tikmēr formāls un automatizēts process, kas uzbūvē programmatūru, vēl nav gatavs. Rodas tā saucamais «meistara faktors», kad konkrēts speciālists prot sagatavot programmatūras laidienu no lokālas darbstacijas, izmantojot vien savas praktiskas iemaņas. Šāda situācija

vēlāk izraisa neparedzētas kļūdas konfigurācijas pārvaldības procesā, kā arī process kļūst ļoti atkarīgs no konkrētā cilvēka kompetences.

Mūsdienās trūkst zinātniski pamatotu pieeju konfigurācijas pārvaldības procesu automatizācijas ieviešanai, kas izmantotu formālu un stingri definētu ceļu no procesa prasībām līdz implementācijai. Papildus tam implementācijas stadijā jāprot atkārtoti izmantot jau esošās implementācijas atsevišķām procesa daļām. Tas varētu paātrināt konfigurācijas pārvaldības procesu automatizācijas ieviešanas laiku, jo no nulles vajadzētu izstrādāt tikai konkrētā projekta specifiskās daļas nevis pilno automatizācijas implementāciju.

Zinātnisku pieeju trūkums ir jūtams ne tikai konfigurācijas pārvaldības disciplīnā, bet arī visā programmatūras izstrādes nozarē. Izstrādājot programmatūru, daudz laika tiek patērēts manuālai koda rakstīšanai, taču beigās izstrādātais kods neatbilst sākotnējām prasībām. Idejas programmatūras izstrādes nozares zinātniskai stiprināšanai sniedz modeļvadāma arhitektūra (angļu val. *Model-Driven Architecture*). Viens no galvenajiem izaicinājumiem ir nodrošināt formālu un automatizētu saikni starp prasībām un izejas kodu, kas varētu būt izpildāms dažādās platformās. Šajā virzienā ir paveikts liels darbs ne tikai pasaulē, bet arī Latvijā. Pētot Rīgas Tehniskās universitātes zinātnieku darbus [ASN 2010, OSI 2010, OSI 2008, NIK 2008, NIK 2009], promocijas darba autoram radās idejas, kā ar modeļvadāmas arhitektūras formātu varētu uzlabot efektivitāti arī konfigurācijas pārvaldības automatizācijas ieviešanai.

Promocijas darba galvenais izaicinājums ir radīt modeļvadāmu pieeju un metodoloģiju, kas varētu atrisināt minētas problēmas konfigurācijas pārvaldībā, palielinot esošo automatizācijas risinājumu efektivitāti un sniedzot rekomendācijas jaunu risinājumu efektīvai ieviešanai.

Promocijas darba mērķis

Promocijas darba mērķis ir izstrādāt modeļvadāmu pieeju un metodoloģiju konfigurācijas pārvaldības procesu automatizācijas ieviešanai, kas ļauj samazināt automatizācijas ieviešanas laiku un uzlabot automatizācijas kvalitāti.

Modeļvadāma pieeja konfigurācijas pārvaldības procesu automatizācijas ieviešanai parāda, kā ar modeļu palīdzību var automātiski iegūt izejas kodu procesu automatizācijai. Modeļi atbilst *MDA* (angļu val. *Model-Driven Architecture*) formātam. Pieeja definē katra konfigurācijas pārvaldības modeļa mērķi, galvenos uzdevumus un darbības principus. Pieejas realizācijai tika izstrādāta jauna metodoloģija, kas realizē pieejas modeļus. Saistībā ar šo metodoloģiju tika izstrādāti jauni modeļi un metodes, kas, realizējot piedāvātās

modeļvadāmas pieejas principus, ļauj automātiski iegūt izejas kodu konfigurācijas pārvaldības automatizācijai.

Promocijas darba kontekstā konfigurācijas pārvaldības automatizācijas kvalitātes rādītājs ir kļūdainu programmatūras būvējumu skaits. Programmatūras būvējums ir galvenais rezultāts konfigurācijas pārvaldības automatizācijas procesā, līdz ar to pieņem, ka mazāks kļūdainu būvējumu skaits atbilst kvalitatīvākam automatizācijas procesam.

Darba uzdevumi

Promocijas darba mērķa sasniegšanai ir izvirzīti šādi uzdevumi:

- izpētīt esošos risinājumus konfigurācijas pārvaldības procesu automatizācijas ieviešanai, apzināties galvenās problēmas un risinājumu attīstības tendences;
- identificēt galvenos ieguvumus un trūkumus jaunākajos konfigurācijas pārvaldības automatizācijas risinājumos, kas atbilst mūsdienīgām attīstības tendencēm;
- izstrādāt pieeju, metodoloģiju, modeļus un metodes konfigurācijas pārvaldības procesu automatizācijai. Pieejai jābūt orientētai uz automatizācijas ieviešanas laika samazināšanu, atkārtoti izmantojot uzņēmumā esošus automatizācijas risinājumus;
- izstrādāt programmatūras prototipu piedāvātu modeļu attēlošanas automatizācijai;
- izstrādāt kritērijus piedāvātas pieejas novērtēšanai;
- ieviest konfigurācijas pārvaldības procesu automatizāciju programmatūras izstrādes projektos un pēc izstrādātiem kritērijiem noteikt izstrādātas pieejas ieguvumus un trūkumus;
- balstoties uz eksperimentu rezultātiem, noteikt pieejas ieguvumus, ierobežojumus, ieviešanas riskus, kā arī sniegt rekomendācijas piedāvātās pieejas un modeļu ieviešanai programmatūras izstrādes projektā;
- definēt piedāvātās pieejas turpmākus attīstības un uzlabošanas virzienus.

Pētījuma objekts un priekšmets

Pētījuma objekts ir programmatūras izstrādes un uzturēšanas projekti.

Pētījuma priekšmets ir programmatūras konfigurācijas pārvaldības procesi šajos projektos.

Darba hipotēzes

Konfigurācijas pārvaldības pētījums ir balstīts uz faktu, ka, attīstoties *Agile* programmatūras izstrādes metodoloģijai, programmatūras izstrādes projekts iesākas ļoti strauji salīdzinājumā, piemēram, ar ūdenskrituma metodoloģiju. Šis fakts izraisa nepieciešamību pēc iespējas ātrāk ieviest automatizāciju konfigurācijas pārvaldības procesiem, lai pasūtītājs varētu pēc iespējas ātrāk saņemt programmatūras pirmās versijas.

Izstrādājot jaunu pieeju un modeļus konfigurācijas pārvaldības automatizācijai, tika izvirzītas šādas hipotēzes:

- lai samazinātu konfigurācijas pārvaldības automatizācijas ieviešanas laiku, var atkārtoti izmantot jau esošus automatizācijas risinājumus, kas funkcionē citos programmatūras izstrādes projektos;
- jo ilgāk konfigurācijas pārvaldības automatizācijas risinājumi tiek lietoti dažādos projektos, jo efektīvāk var tos lietot atkārtoti, ieviešot konfigurācijas pārvaldības procesu automatizāciju jaunajā programmatūras izstrādes projektā.

Pirmā hipotēze balstās uz faktu, ka pielāgot un konfigurēt jau esošus risinājumus ir ātrāk, nekā izstrādāt no nulles pilnīgi jaunus. Jebkura izstrāde prasa laiku gan izstrādei, gan risinājuma testēšanai. Izmantojot risinājumu atkārtoti, šis laiks ir mazāks, jo izstrāde un risinājuma pilnīga pārtestēšana nav jāveic atkārtoti. Ja konfigurācijas pārvaldības automatizācijas ieviešanai atkārtoti izmanto jau esošus risinājumus, no nulles ir jāizstrādā tikai tās procesa daļas, kas ir specifiskas konkrētam projektam.

Otrā hipotēze balstās uz faktu, ka programmatūrā pilnīgi visas kļūdas testēšanas posmā nevar atklāt. Ir kļūdas, ko ir iespējams atklāt tikai programmatūras reālas ekspluatācijas laikā. Risinājumi, kas automatizē konfigurācijas pārvaldību, nav izņēmums. Tāpēc, jo ilgāk risinājums tiek lietots konfigurācijas pārvaldības procesos, jo vairāk kļūdu un nepilnību var atklāt, un padarīt to risinājumu stabilāku. Tāpēc konfigurācijas pārvaldības automatizācijas risinājumu atkārtotas izmantošanas efektivitāte būs atkarīga no tā, cik ilgi tiek lietots konkrētais risinājums.

Pētījuma metodes

Pētījumā tika izmantotas šādas metodes:

- literatūras analīze – esošo pieeju, metodoloģiju un rīku izpētei, lai identificētu to ieguvumus, trūkumus un attīstības tendences;
- modelēšana un metamodelēšana;
- modeļu transformācijas;
- eksperimentu plānošana un organizācija.

Zinātniskais jaunieguvums

Promocijas darbam ir šāds zinātniskais jaunieguvums:

- izstrādāta jauna pieeja *MTM* (Modelis – Transformācija – Modelis) konfigurācijas pārvaldības procesu automatizācijas ieviešanai ar modeļu palīdzību, atkārtoti lietojot jau esošus automatizācijas risinājumus;
- izstrādāta jauna metodoloģija *EAF* (Vide – Darbība – Ietvars, angļu val. *Environment – Action – Framework*), kas implementē jaunas *MTM* pieejas principus un definē soļus konfigurācijas pārvaldības automatizācijas ieviešanai;
- izstrādāti modeļi konfigurācijas pārvaldības procesu attēlošanai *EAF* metodoloģijas ietvaros;
- izstrādāta metode konfigurācijas pārvaldības esošu risinājumu glabāšanai.

Teorētiskā vērtība

Promocijas darba teorētiska vērtība ir šāda:

- izanalizētas konfigurācijas pārvaldības definīcijas un definēti konfigurācijas pārvaldības galvenie uzdevumi;
- balstoties uz literatūras analīzi par konfigurācijas pārvaldības uzdevumiem, definēta konfigurācijas pārvaldības procesu automatizācija;
- izanalizēti esoši risinājumi konfigurācijas pārvaldības automatizācijai un apkopotas risinājumu attīstības tendences;
- izstrādāta jauna pieeja, metodoloģija, modeļi un metode konfigurācijas pārvaldības automatizācijas ieviešanai, balstoties uz *MDA* formātu;

- izmantojot *MetaEdit+* rīku, izstrādāta modelēšanas valoda, kas ļauj implementēt jaunas *MTM* pieejas elementus, definējot modeļus, transformācijas un papildu elementus pieejas implementācijai;
- izdevās noskaidrot, ka modeļvadāma pieeja konfigurācijas pārvaldības procesu ieviešanai palīdz samazināt cilvēciska faktora risku, pārejot no procesa automatizācijas prasībām uz implementāciju.

Praktiskā nozīmība

Promocijas darbam ir šāda praktiskā nozīmība:

- izstrādāts eksperimentālais programmatūras prototips, kas automatizē *EAF* metodoloģijas modeļu ģenerēšanu un transformāciju;
- izveidota kompetences grupa 17 cilvēku sastāvā *EAF* metodoloģijas praktiskas testēšanas aktivitātēm. Kompetences grupā piedalījās vecākie un vadošie programmētāji, kas ikdienas darbā saskaras ar konfigurācijas pārvaldības procesiem strādājot uzņēmumā SIA «*Tieto Latvia*»;
- izstrādāti kritēriji *EAF* metodoloģijas novērtēšanai, sniegts skaidrojums, kā var aprēķināt kritēriju rādītājus;
- veikti eksperimenti, ieviešot konfigurācijas pārvaldības automatizāciju piecos programmatūras izstrādes un uzturēšanas projektos. Balstoties uz eksperimentu rezultātiem, definēti *EAF* metodoloģijas praktiskie ieguvumi, atšķirības no citiem konfigurācijas pārvaldības automatizācijas risinājumiem, metodoloģijas ieviešanas riski;
- izstrādāta praktisku rekomendāciju kopa, kā var ieviest konfigurācijas pārvaldības automatizāciju pēc jaunās *EAF* metodoloģijas.

Promocijas darba praktiskus rezultātus var izmantot programmatūras izstrādes uzņēmumi, kas vēlas uzlabot konfigurācijas pārvaldības automatizācijas risinājumu efektivitāti, samazināt automatizācijas ieviešanas laiku jaunajos projektos.

Promocijas darba praktiskās daļas nobeigumā ir uzskaitīti iespējami *EAF* metodoloģijas uzlabojumi. Līdz ar to konfigurācijas pārvaldības modeļus var papildināt un veikt atkārtotus eksperimentus, lai iegūtu rādītājus par metodoloģijas ieguvumiem.

Darba aprobācija

Par promocijas darba rezultātiem tika ziņots 10 starptautiskās konferencēs Latvijā, Itālijā, Turcijā, Francijā un Austrijā:

- 2011. g. 13. oktobris. RTU 52. Starptautiskā zinātniskā konference, Rīga, Latvija.
- 2012. g. 12. oktobris. RTU 53. Starptautiskā zinātniskā konference, Rīga, Latvija.
- 2013. g. 17. oktobris. RTU 54. Starptautiskā zinātniskā konference, Rīga, Latvija.
- 2014. g. 14. oktobris. RTU 55. Starptautiskā zinātniskā konference, Rīga, Latvija.
- 2012. g. 27. aprīlis. LLU Applied Information and Communication Tehnology 2012, Jelgava, Latvija.
- 2013. g. 27. aprīlis. LLU Applied Information and Communication Tehnology 2013, Jelgava, Latvija.
- 2014. g. 22.–24. novembris. 3rd International Conference on Systems, Communications, Computers and Applications (CSCCA"14), Florence, Itālija.
- 2014. g. 15.–17. decembris. 13th International Conference on Telecommunications and Informatics TELE-INFO'14, Stambula, Turcija.
- 2015. g. 9.–11. februāris. 3rd International Conference on Model-Driven Engineering and Software Development MODELSWARD 2015, Anžē, Francija.
- 2015. g. 15.–17. martā. International Conference on Applied Physics, Simulation and Computers, Vīne, Austrija.

Saistībā ar promocijas darbu veikto pētījumu rezultāti ir atspoguļoti šādās publikācijās:

1. Bartusevics A., Kotovs V., Novickis L. A Method for Effective Reuse-Oriented Software Release Configuration and Its Application in Insurance Area. In: Scientific Journal of Riga Technical University. Information Tehnology and Management Science, 15th series, RTU Publishing House, 2012, Riga, Latvia, pp. 111–115. (Indeksēts: *EBSCO, VINITI, Google Scholar*)

2. Bartusevics A., Kotovs V. Towards the effective reuse-oriented release configuration process. In: Proceedings of the 5th International Scientific Conference «Applied Information and Communication Tehnologies», 2012, Jelgava, Latvia, pp. 99–103. (Indeksēts: *EBSCO, VINITI*)
3. Bartusevics A., A Methodology for Model-Driven Software Configuration Management Implementation and Support. In: Proceedings of the 6-th International Scientific Conference «Applied Information and Communication Tehnologies», 2013, Jelgava, Latvia, pp. 252–258. (Indeksēts: *EBSCO, VINITI*)
4. Bartusevičs, A., Novickis, L., Bluemel, E. Intellectual Model-Based Configuration Management Conception. In: Scientific Journal of Riga Technical University. Applied Computer Systems. 2014./15, pp. 22.–27. ISSN 2255-8683. e-ISSN 2255-8691. (Indeksēts: *EBSCO, VINITI, Google Scholar*)
5. Bartusevičs, A., Novickis, L., Model-Driven Software Configuration management and Environment Model. In: Recent Advances in Electrical and Electronic Engineering. In: Proceedings of the 3rd International Conference on Systems, Communications, Computers and Applications (CSCCA"14), Itālija, Florence, 22.-24. novembris, 2014. Italy: WSEAS Press, 2014, pp. 132.–140. ISBN 978-960-474-399-5. ISSN 1790-5117. (Tiks indeksēts: *SCOPUS*)
6. Bartusevičs, A., Novickis, L., Lesovskis, A. Model-Driven Software Configuration Management and Semantic Web in Applied Software Development. In: Proceedings of the 13th International Conference on Telecommunications and Informatics (TELE-INFO '14), IIstanbul, Turkey December 15–17, 2014, pp. 108.–116. (Tiks indeksēts: *SCOPUS*)
7. Bartusevičs, A., Novickis, L. Models for Implementation of Software Configuration Management. No: Procedia Computer Science. Valmiera, Latvia: 2014, 3.–10. lpp. (Tiks indeksēts: *SCOPUS*)
8. Bartusevičs, A., Novickis, L., Leye, S. Implementation of Software Configuration Management Process by Models: Practical Experiments and Learned Lessons. In: Scientific Journal of Riga Technical University. Applied Computer Systems. Nr. 16, 2014, RTU Press, pp. 26.–32. ISSN 2255-8683. e-ISSN 2255-8691. (Indeksēts: *EBSCO, VINITI, Google Scholar*)

9. Bartusevičs, A., Novickis, L. Model-based Approach for Implementation of Software Configuration Management Process. No: MODELSWARD 2015: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, Francija, Angers, 9.–11. februāris, 2015. Lisbon: SciTePress, 2015, 177.–184.lpp. ISBN 978-989-758-083-3. (Tiks indeksēts: *SCOPUS*)
10. Bartusevičs, A., Novickis, L. Towards the Model-driven Software Configuration Management Process. In: Scientific Journal of Riga Technical University. Information Technology and Management Science. Vol. 17, 2014, pp.32-38. ISSN 2255-9086. e-ISSN 2255-9094. (Indeksēts: *EBSCO, VINITI, Google Scholar*)
11. Bartusevičs, A., Lesovskis, A., Novickis, L. Semantic Web Technologies and Model-Driven Approach for the Development and Configuration Management of Intelligent Web-Based Systems. No: Proceedings of the 2015 International Conference on Circuits, Systems, Signal Processing, Communications and Computers, Austrija, Vienna, 15.–17. marts, 2015. Vienna: 2015, 32.–39.lpp. ISBN 978-1-61804-285-9. ISSN 1790-5117. (Tiks indeksēts: *SCOPUS*)
12. Bartusevičs, A., Novickis, L. Model-Driven Conception for Planning and Implementation of Software Configuration Management. International Journal of Computers, Volume 9, 2015, pp.62-72. ISSN 1998-4308. (Tiks indeksēts: *SCOPUS*)

Darba struktūra

Promocijas darbu veido ievads, piecas nodaļas, secinājumi, bibliogrāfiskais saraksts un pielikumi. Promocijas darba pamatteksts ir 238 lappuses, tajā ir 55 attēli un 32 tabulas. Bibliogrāfiskajā sarakstā ir 117 nosaukumu informācijas avoti.

Promocijas darba *Ievadā* tika pamatota veiktā pētījuma aktualitāte, formulēts promocijas darba mērķis un uzdevumi, izvirzītas hipotēzes, aprakstītas pētījuma metodes, aprakstīta zinātniska novitāte un iegūto rezultātu praktiskā nozīmība, kā arī atspoguļota darba aprobācija.

Darba *1. nodaļā* tika definēts programmatūras konfigurācijas pārvaldības jēdziens un konfigurācijas pārvaldības galvenie uzdevumi. Balstoties uz literatūras analīzi, tika definēta konfigurācijas pārvaldības procesu automatizācija. Tika izanalizēti esošie risinājumi

konfigurācijas pārvaldības uzdevumiem automatizācijai, definētas galvenās problēmas un mūsdienu risinājumu attīstības tendences.

Promocijas darba *2. nodaļā* tika analizētas esošās pieejas un esošie rīki konfigurācijas pārvaldības automatizācijai, kas izmanto *MDA* formātu un galvenos principus. Balstoties uz analīzes rezultātiem, tika noteikti trūkumi esošajās pieejās. Nodaļas secinājumos tika sniegtas konfigurācijas pārvaldības pieejas pazīmes, kam jābūt, lai novērstu identificētus trūkumus esošajos risinājumos.

Darba *3. nodaļā* tika aprakstīta jaunizstrādāta *MTM* pieeja konfigurācijas pārvaldības automatizācijas ieviešanai ar modeļu palīdzību. Tika piedāvāta jauna *EAF* metodoloģija *MTM* pieejas realizācijai, kuras izstrāde ir balstīta uz *MDA* formātu. Jaunas metodoloģijas implementācijai tika definēti jauni modeļi un atbilstoši meta-modeļi konfigurācijas pārvaldības procesu attēlošanai, kā arī modeļu transformācijas likumi, kas ļauj mainīt modeļu abstrakcijas līmeni. Metodoloģija piedāvā ieviest konfigurācijas pārvaldības procesu automatizāciju, izmantojot esošās automatizācijas atsevišķiem konfigurācijas pārvaldības uzdevumiem. Tika izstrādāta metode esošo konfigurācijas pārvaldības automatizācijas risinājumu glabāšanai.

Promocijas darba *4. nodaļā* tika veikta jaunas *EAF* metodoloģijas testēšana. Tika aprakstīti metodoloģijas vērtēšanas kritēriji un teorētisko rezultātu aprobācijas rezultāti. Metodoloģijas testēšanas gaitā konfigurācijas pārvaldība tika ieviesta piecos programmatūras izstrādes projektos. Eksperimentu rezultātā tika noteikti metodoloģijas ieguvumi un trūkumi. Analizējot iegūtus ieguvumus, trūkumus, recenzentu atsauksmes, kas tika iegūtas, publicējot metodoloģijas teorētiskus pamatus zinātnisku konferenču krājumos, tika konstatēts, ka esošos ieguvumus var palielināt, bet trūkumu skaitu samazināt, veicot uzlabojumus metodoloģijā.

Darba *5. nodaļā* tika aprakstīta *EAF* metodoloģijas uzlabotās versijas izstrāde. Izstrādes galvenais mērķis bija novērst eksperimentu rezultātā atklātus trūkumus un ņemt vērā piezīmes, ko izteica zinātnisku rakstu recenzenti, kas iepazinās ar *EAF* metodoloģiju. Veicot atkārtotus eksperimentus, izdevās parādīt, kā trūkumi tika novērsti un ieguvumi palielinājās. Nodaļas beigās ir aprakstīti trūkumu novēršanas pasākumi. Balstoties uz eksperimentu pirmās un otrās kārtas rezultātu salīdzinājumu, ir noteikti metodoloģijas galvenie ieguvumi, atšķirības no citām pieejām konfigurācijas pārvaldības automatizācijas ieviešanai, kā arī noteikti metodoloģijas ieviešanas riski un definēti turpmākie attīstības virzieni.

Promocijas darba *noslēguma daļā* tika izklāstīti promocijas darba galvenie rezultāti, pamatota mērķa un uzdevumu izpilde, hipotēžu pierādījums, kā arī tika uzskaitīti iespējamie turpmāko pētījumu virzieni.

1. PROGRAMMATŪRAS KONFIGURĀCIJAS PĀRVALDĪBAS IZPĒTE

1.1. Jēdzienu skaidrojums

Šīs apakšnodaļas mērķis ir definēt jēdzienus, kas tiks plaši lietoti šajā promocijas darbā un kuriem var būt vairākas nozīmes atkarībā no konteksta. Minēti jēdzieni ir paskaidroti tabulā 1.1.

1.1. tabula

Jēdzienu tabula

Jēdziens	Skaidrojums
Konfigurācijas pārvaldība	Šis jēdziens pats pa sevi ir ļoti plašs tādā ziņā, ka var būt attiecināms uz vairākām nozarēm. Šajā promocijas darbā konfigurācijas pārvaldība ir attiecināma uz informācijas tehnoloģijas nozari. Savukārt, ja runa ir par informācijas tehnoloģijas nozari, tad izšķir divu veidu konfigurācijas pārvaldības. Pirmā ir iekārtu (angļu val. *hardware*) konfigurācijas pārvaldība, otrā ir programmatūras (angļu val. *software*) konfigurācijas pārvaldība. Šajā darbā tiks apskatīta tikai programmatūras konfigurācijas pārvaldība.
Projekts	Projekts ir process, ko veido koordinētu un kontrolētu norišu kopums, kuram ir noteikti sākuma un beigu termiņi un īpašām prasībām atbilstošais mērķis jāsasniedz, iekļaujoties atvēlētajos laika, izmaksu un resursu ietvaros. Promocijas darba kontekstā minēto norišu kopumu veido programmatūras izstrādes aktivitātes, tādas kā prasību definēšana, programmatūras projektēšana, programmatūras izstrāde, programmatūras testēšana uc. Minētas aktivitātes notiek secīgi un katra nākama aktivitāte izmanto iepriekšējas aktivitātes rezultātus.

	Visu aktivitāšu galvenais mērķis ir radīt unikālu informācijas tehnoloģiju produktu vai pakalpojumu (lietojumprogrammatrūra, tīmekļa serviss utt.).
Konfigurācijas pārvaldības process	Process, kas kontrolē programmatūras evolūcijas procesu projektā, nosakot kārtību kādā tiek definētas programmatūras vienības, ka arī nodrošina minēto vienību versiju kontroli, statusu uzskaiti un reglamentē vienību iekļaušanu produktā.
Produkts	Projekta galvenais nodevums. Produkts tiek izstrādāts vai uzturēts projekta gaitā. Šajā darbā ar produktu jāsaprot izstrādājamu vai uzturamu lietišķu programmatūru, tīmekļa servisu, uzņēmumu resursu plānošanas datorizētu sistēmu un līdzīgus informācijas tehnoloģiju risinājumus.
Pieeja	Pieeja ir elementu un darbību vizuāls attēlojums, kas parāda kādas darbības ar elementiem ir jāveic, lai sasniegtu noteiktu mērķi. Promocijas darba kontekstā pieejas mērķis ir iegūt izejas kodu konfigurācijas pārvaldības procesa automatizācijai. Elementi ir konfigurācijas pārvaldības serveris, kurā tiks izpildīts minētais izejas kods, un modeļi, ar kuru palīdzību izejas kods tiks ģenerēts. Darbības parāda kā izejas koda veidošanas procesu.
Metode	Metode ir sistematizēts paņēmienu kopums, kas nepieciešams, lai izpildītu kādu zināmu uzdevumu vai sasniegtu konkrētu mērķi. Promocijas darbā minētais uzdevums ir glabāt un pārvaldīt atkārtoti izpildāmu izejas kodu konfigurācijas pārvaldības darbību automatizācijai. Līdz ar to, paņēmienu kopums ietver sevī datubāzes entīšu – relāciju (*ER*) diagrammas definīciju un vadlīnijas kā pārvaldīt minēto datubāzi (pievienot,

	modificēt, dzēst atkārtoti pielietojamas izejas koda vienības).
Modelis	Konfigurācijas pārvaldības procesa shematiskais attēlojums noteiktā abstrakcijas līmenī. Modelī tiek attēloti konfigurācijas pārvaldības procesā iesaistītie elementi, kā arī saiknes starp elementiem.
Vide	Infrastruktūras kopa (datubāzes serveri, lietojumu serveri, ugunsmūri, klāsteri utt.), kas ir nepieciešama programmatūras produkta palaišanai noteiktiem vienveidīgiem projekta mērķiem, piemēram, programmatūras izstrādei, testēšanai, ekspluatācijai. Parasti vienai videi ir īss nosaukums, kas atspoguļo vides mērķus, piemēram, *DEV* – izstrādes vide, *TEST* – testa vide, *PROD* – ekspluatācijas jeb produkcijas vide.
Izmaiņu plūsma	Programmatūrā veiktas izmaiņas (izmaiņas, kļūdu labojumi, dokumentācijas papildinājumi utt.), kas tiek pārnestas starp divām dažādām projekta vidēm konfigurācijas pārvaldības procesā.

1.2. Programmatūras konfigurācijas pārvaldības definīcija

Programmatūras konfigurācijas pārvaldības vēsture ir pietiekami gara, un tās pirmsākumi meklējami tālajā 1960. gadā [HIS 2014]. Kopš tā laika konfigurācijas pārvaldība nepārtraukti attīstās, un to ietekmē daudzi faktori: programmatūras izstrādes nozares attīstības tendences [BIL 2014, AZO 2014, TRE 2014, WHA 2014], programmatūras izstrādes metodoloģiju attīstības vēsture [JIA 2009], programmatūras izstrādes nozares kvalitātes standarti [GAL 2009, ABO 2014, BAM 1995]. Pētot konfigurācijas pārvaldības definīciju, mērķis ir noteikt pēc iespējas pilnīgāku definīciju, kas atbilst mūsdienīgām attīstības tendencēm programmatūras izstrādes nozarē.

Literatūras analīzes rezultātā [AIE 2010, BER 2003, DEP 2010, PAU 2007, MET 2002, KAN 2005, CON 2002, GLO 2012, BRU 2004, DAR 2001, WES 2005, MEL 2006,

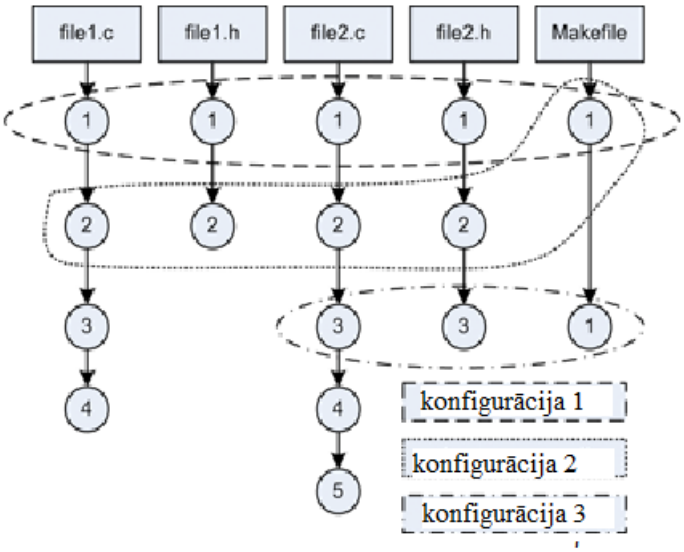
BEL 2005, VAC 2006, WIK 2013, ОРЛ 2011, ЛАП 2004, УДО 2011, ЗАМ 2008, IEE 2015, ISO 2015] tika atrasts vairāk nekā 20 dažādu definīciju, kas izskaidro konfigurācijas pārvaldības jēdzienu. Atrastajām definīcijām tika apvienotas kopējās daļas, un rezultātā tika iegūti šādi piemēri konfigurācijas pārvaldības definīcijām:

- konfigurācijas pārvaldība ir uzdevums pārvaldīt un kontrolēt izmaiņas programmatūrā. Konfigurācijas pārvaldības praktiskā puse ietver revīziju kontroli un bāzes līniju uzturēšanu. Šajā pašā avotā [SOF 2014] konfigurācijas pārvaldību definē kā programmatūras evolūcijas procesa kontroli;
- konfigurācijas pārvaldība ir produkta izmaiņu aktivitātes kontrole. Galvenokārt konfigurācijas pārvaldība ir vērsta uz rīku pārvaldi, taču galvenais konfigurācijas pārvaldības uzdevums ir palīdzēt programmatūras izstrādātājiem pareizi kontrolēt un pārvaldīt viņu darbu, lai viņiem būtu pārliecība, ka darbs nepazudīs un netiks nejauši nograuts [VAC 2006];
- konfigurācijas pārvaldība ir disciplīna, kas apraksta rīkus un metodoloģijas, kā kontrolēt izmaiņas programmatūras vienumos [BEL 2005];
- konfigurācijas pārvaldība ir labā prakse, kas palīdz programmatūras izstrādātājiem un projektu vadītājiem labāk definēt potenciālas problēmas projektā, pārvaldīt izmaiņas un sekot projekta attīstības progresam. Konfigurācijas pārvaldība identificē produkta sastāvdaļas un nosaka, kā pareizi kontrolēt izmaiņas tajās [MET 2002];
- konfigurācijas pārvaldība ir apakšprocesu virkne programminženierijā, kas nosaka produkta bāzes līniju veidošanu un uzturēšanu. Konfigurācijas pārvaldība piedāvā tehnikas un metodes, kā identificēt un kontrolēt izmaiņas izstrādājamā produktā [BRU 2004];
- konfigurācijas pārvaldība ir metodoloģija, kas nosaka programmatūras problēmu un izmaiņu pārvaldību. Metodoloģija nosaka vadlīnijas, kā kontrolēt izmaiņas produkta arhitektūrā un produkta dažādas versijas, administrēt produkta sastāvdaļas, nosakot drošības politiku, veikt produkta kvalitātes pārskatus [KAN 2005].

Apkopojot minētās definīcijas, ir redzams, ka nav vienota viedokļa par to, kas ir konfigurācijas pārvaldība. Daži speciālisti uzskata to par metodoloģiju, daži par procesu vai procesu virkni, savukārt citi runā par konfigurācijas pārvaldību kā par atsevišķu disciplīnu. Ir speciālistu grupa, kas neuzskata konfigurācijas pārvaldību ne par disciplīnu, ne par metodoloģiju, bet uztver to kā realizēto labo praksi, kas projektā var būt un var nebūt. Darba autors neapskatīs detalizēti definīcijas, kas ir saistītas ar labo praksi, jo uzskata, ka no

zinātniskā viedokļa to ir grūti definēt. Tātad – paliek metodoloģija, procesi un disciplīna. Lai gan dažādās definīcijās var sastapt visus trīs minētos lietvārdus [BRU 2004, BEL 2005, KAN 2005], gandrīz visās definīcijās [AIE 2010, BER 2003, DEP 2010, PAU 2007, MET 2002, KAN 2005, CON 2002, GLO 2012, BRU 2004, DAR 2001, WES 2005, MEL 2006, BEL 2005, VAC 2006, WIK 2013, ОРЛ 2011, ЛАП 2004, УДО 2011, ЗАМ 2008], ir minēts atslēgvārds «kontrole». Tieši kontrole ir galvenais definīcijas atslēgvārds, spriežot pēc literatūras analīzes. Lai kā tiktu uztverta konfigurācijas pārvaldība – vai kā metodoloģija, vai kā process, svarīgi ir tas, ka galvenā funkcija ir kontrole. Definējot konfigurācijas pārvaldību, ir jāsaprot, ko tieši tā kontrolē un kādus uzdevumus pilda, īstenojot šo kontroli.

Kā jau tika minēts, galvenais atslēgvārds konfigurācijas pārvaldības definīcijā ir «kontrole». Vairākās definīcijās [AIE 2010, УДО 2011, MET 2002, CON 2002] ir rakstīts, ka tiek kontrolētas izmaiņas izstrādājamā produktā un šā produkta versijas. Programmatūras izstrādes nozarē izstrādājamam produktam parasti ir vairākas daļs jeb vienumi. Vienumi var būt izejas koda faili, dokumentācijas faili, dažādi palīginstrumenti. Izstrādājamam produktam ir vairāki vienumi, kas laika gaitā mainās. Tiklīdz ir mainījies kāds vienums, rodas konkrēta vienuma jaunā versija. Iekļaujot šo versiju produktā, pašam produktam arī rodas jauna versija. Konfigurācijas pārvaldības kontekstā produkta vienumus sauc arī par konfigurācijas vienumiem (angļu val. *Configuration Items*) [AIE 2010, MET 2002, ЛАП 2004]. Ja saliek kopā konfigurācijas vienumu noteiktās versijas, iegūst noteiktu konfigurāciju. Ar versiju jāsaprot konkrēta vienuma stāvokli noteiktā laika momentā. Šo stāvokli nepieciešamības gadījumā vienmēr jāprot atjaunot, neatkarīgi no pārējām izmaiņām, kas tika veiktas šajā konfigurācijas vienumā [УДО 2011]. Līdz ar to konfigurācijas pārvaldība kontrolē un pārvalda izstrādājama produkta konfigurācijas vienumu versijas un procesu, kura rezultātā no vienumu versijām rodas dažādas produkta konfigurācijas [AIE 2010, УДО 2011, MET 2002]. Attēlā 1.1. [УДО 2011] redzami produkta konfigurācijas vienumi, to dažādas versijas un dažādas produkta konfigurācijas.



1.1. att. Konfigurācijas vienumi un to versijas

Konfigurācijas pārvaldība funkcionē visa projekta dzīves cikla garumā neatkarīgi no programmatūras izstrādes metodoloģijas [AIE 2010, УДО 2011]. Tiklīdz sāk parādīties konfigurācijas vienumi, piemēram, dokumentācija, prasību specifikācija vai izejas koda faili, sāk darboties konfigurācijas pārvaldība. Nepieciešams identificēt konfigurācijas vienumus un nodrošināt tiem versiju kontroli. Ar kontroli ir jāsaprot to, ka vienumā var veikt izmaiņas, bet izmaiņu vēsture saglabājas tā, ka jebkurā laikā var atjaunot jebkuru versiju un iekļaut to izstrādājamā produktā. Ar laiku izstrādājamā produktā parādās arvien vairāk vienumu un rodas vienumu grupa, kuros izmaiņas veic dažādi cilvēki, iespējams pat – dažādas izstrādātāju komandas. Šajā gadījumā konfigurācijas pārvaldība sniedz rekomendācijas, kā organizēt izmaiņu veikšanu, lai nevienas izmaiņas nepazustu un lai vienas izmaiņas nebūtu pretrunā ar citām. Ja no konfigurācijas vienumiem ir iespējams sakomplektēt produkta strādājošu versiju un nodot to pasūtītājam, arī šajā gadījumā konfigurācijas pārvaldība sniedz vadlīnijas, noteikumus un rekomendācijas, kā to izdarīt. Ja ir nepieciešamība atgriezties uz iepriekšējo produkta versiju, arī šeit palīgā nāk konfigurācijas pārvaldība. Ja projekta gaitā rodas nepieciešamība apkopot metrikas par veiktām izmaiņām vienumos, konfigurācijas pārvaldība sniedz metodes un rīkus, kā to īstenot [УДО 2011].

Analizējot dažādas konfigurācijas pārvaldības definīcijas, nācās secināt, ka, lai veiktu produkta konfigurācijas vienumu identifikāciju un kontroli visa projekta dzīves cikla laikā, nepieciešams atrisināt konkrētus uzdevumus, kas kopumā ļaus sasniegt kontroles un pārvaldības mērķi [AIE 2010, BER 2003, DEP 2010, PAU 2007, MET 2002, KAN 2005, CON 2002, GLO 2012, BRU 2004, DAR 2001, WES 2005, MEL 2006, BEL 2005, VAC

2006, WIK 2013, ОРЛ 2011, ЛАП 2004, УДО 2011, ЗАМ 2008]. Līdzīgi kā ar pašu definīciju, arī par konfigurācijas pārvaldības uzdevumiem katrā literatūras avotā tiek runāts nedaudz citādāk, taču pēc detalizētas izpētes tika identificēti šādi galvenie uzdevumi:

- konfigurācijas vienumu identifikācija;
- konfigurācijas vienumu versiju kontrole;
- gatava produkta komplektācija (laidienu vai instalācijas paku sagatavošanas process (angļu val. *building engineering*));
- gatava produkta instalācija (angļu val. *deployment*);
- paralēlas izstrādes nodrošinājums (ar konfigurācijas vienumiem vienlaikus strādā vairāki izstrādātāji; angļu val. *branching*);
- metriku savākšana par konfigurācijas vienumu izmaiņām, versijām un dažādām produkta konfigurācijām;
- konfigurācijas vienumu uzskaite un audits.

Ir sastopami avoti, kuros iedala daudz vairāk uzdevumu, kas būtu jārisina konfigurācijas pārvaldībai [SOF 2014, ЗАМ 2008, BEL 2005, MEL 2006], taču šā darba autors uzskata, ka tos uzdevumus var apskatīt kā apakšuzdevumus vienam vai vairākiem uzdevumiem, kas jau tika minēti. Tāds uzskats radās pēc tam, kad tika izlasīta grāmata [AIE 2010], kuras autors vairāk nekā 25 gadus darbojas konfigurācijas pārvaldības jomā un ir apkopojis vairākas grāmatas par konfigurācijas pārvaldību. Turpinot analizēt konfigurācijas pārvaldības būtību, tiks dots neliels apraksts katram no galvenajiem konfigurācijas pārvaldības uzdevumiem.

- *Konfigurācijas vienumu identifikācija.* Šis ir būtiskākais uzdevums, kas jāatrisina projekta sākumā. Risinot šo uzdevumu, ir jāsaprot, no kā sastāvēs topošais produkts un kādā veidā tiks identificēti jauni konfigurācijas vienumi. Ne mazāk svarīgi ir noteikt to, kurš būs atbildīgs par konfigurācijas vienumu identificēšanu. Konfigurācijas vienumu identificēšanu vēlams aprakstīt vēl pirms tam, kad parādīsies paši vienumi, tādi kā dokumentācija vai izejas kods. Identifikācijas procesā katram vienumam ir jāpiešķir unikāls identifikators un nosaukums.
- *Konfigurācijas vienumu versiju kontrole.* Brīdī, kad ir zināma konfigurācijas vienumu identificēšanas procedūra, ir jānodrošina, lai izmaiņas šajos vienumos būtu kontrolējamas, kā arī – lai saglabātos izmaiņu vēsture. Mūsdienās ir pieejamas dažādas atvērta pirmkoda un komerciālas programmatūras, kas palīdz šo uzdevumu risināt [DAR 2001, ЗАМ 2008, AIE 2010]. Šīs programmatūras sauc par versiju

kontroles sistēmām. Kā piemēru var minēt *Subversion*, *Mercurial* un *Git*. Risinot versiju kontroles uzdevumu, nepieciešams visus identificētus konfigurācijas vienumus ievietot versiju kontroles sistēmā un definēt vadlīnijas, kas reglamentēs sankcionētu izmaiņu veikšanu un jaunu versiju tapšanu. Ar to ir jāsaprot, ka projektā jābūt skaidri zināmam, kurš un kad drīkst veikt izmaiņas, kādos konfigurācijas vienumos un kādā veidā šis process tiks dokumentēts [MEL 2006, AIE 2010].

- *Gatava produkta komplektācija.* Šo uzdevumu lielāka daļa speciālistu uztvers kā produktu kompilācijas vai būvējumu procesu [ЗАМ 2008]. Taču, spriežot pēc avotiem [AIE 2010, MEL 2006, BEL 2005], uz šo procesu vajadzētu skatīties nedaudz plašāk. Kopumā šā uzdevuma risināšanas gaitā ir jādefinē rīki, kas spēj no konfigurācijas vienumiem izveidot produkta būvējumu. Visbiežāk ar šo ir domāts process, kura ietvaros izejas kods tiek pārvērsts izpildāma modulī, ko bieži sauc arī par kompilācijas jeb būvējuma procesu [AIE 2010]. Ir jāprot ne tikai nokompilēt izejas kodu dažādiem vienumiem un izveidot izpildāmu moduli, bet arī jāprot noteikt, kādas konfigurācijas vienumu versijas jāiekļauj kompilācijas procesā.
- *Gatava produkta instalācija.* Risinot šo uzdevumu, konfigurācijas pārvaldība nosaka, kādā veidā tiek veikta uzkompilēta produkta instalācija, lai to varētu lietot. Instalācijas procesā jāprot atrisināt dažāda veida problēmas, kas ir saistītas ar infrastruktūru tajā vidē, kur produkts tiks lietots. Jāatzīmē, ka instalācijas procesā jābūt iespējai atgriezties uz iepriekšējo produkta versiju, ja, piemēram, produktā tiek atklāta kāda kritiska kļūda. Instalācijas procesam jābūt dokumentētam un atkārtoti lietojamam [AIE 2010].
- *Paralēlas izstrādes nodrošinājums.* Gadījumā, kad produktu paralēli izstrādā vairākas komandas, konfigurācijas pārvaldība sniedz vadlīnijas, procedūras un rīkus, kas nodrošina iespēju veikt savā starpā nekonfliktējošas izmaiņas konfigurācijas vienumos. Risinot šo uzdevumu, izmanto funkcijas, ko sniedz versiju kontroles rīki. Versiju kontroles rīki ne tikai uztur versijas, bet arī var uzturēt paralēlas kopijas konfigurācijas vienumiem un prot apvienot kopā izmaiņas no dažādām kopijām, izvairoties no konfliktiem [DAR 2001]. Risinot šo uzdevumu, ir svarīgi ne tikai izvēlēties versiju kontroles sistēmu, kas var nodrošināt paralēlu izmaiņu veikšanu, bet arī definēt vadlīnijas: kā tas tiks darīts un kā sadalās atbildība starp izstrādātajiem par veiktajām izmaiņām. Pretējā gadījumā pat ļoti populāras un atzītas versiju kontroles sistēmas nespēs nodrošināt paralēlu izstrādi bez problēmām un neparedzētām kļūdām [AIE 2010, DAR 2001, ЗАМ 2008, WES 2005, MEL 2006].

- *Metriku savākšana.* Kā jau tika minēts, konfigurācijas pārvaldība nodarbojas ar kontroli. Taču, kā zināms, ir grūti kontrolēt to, ko nevar izmērīt. Tāpēc konfigurācijas pārvaldība sniedz vadlīnijas, kā iegūt metrikas par veiktajām izmaiņām produktā. Metriku nepieciešamību parasti nosaka konkrēta projekta vajadzības un specifika, taču, kā atzīmē vadošie konfigurācijas pārvaldības speciālisti, galvenais, lai metrikas palīdzētu izvairīties no potenciālām problēmām un sniegtu priekšstatu par produkta evolūcijas procesu [AIE 2010, MEL 2006].

Runājot par konfigurācijas pārvaldības definīciju un uzdevumiem, jāpiemin arī nozīmīgi kvalitātes standarti *IEEE Std 12207-2008* un *ISO 9001:2000* [IEE 2015, ISO 2015]. Abos standartos uzsvars ir uz dokumentētu procesu, kura rezultātā jābūt izveidotam konfigurācijas pārvaldības plānam. Šo dokumentu obligāti vajadzētu apstiprināt programmatūras pasūtītājam jau projekta sākumā. Konfigurācijas pārvaldības plāns ietver lomas un atbildības konkrētajā izstrādes projektā, lielāko akcentu liekot uz tām daļām, kas ir konfigurācijas pārvaldības pakļautībā. Papildus tam konfigurācijas plānā jābūt vismaz šādām sadaļām: programmatūras vienumu identifikācija un versiju kontrole, izmaiņu pārvaldība, programmatūras vienumu statusu uzskaite un audits [IEE 2015, ISO 2015].

Šajā apakšnodaļā definēti konfigurācijas pārvaldības uzdevumi, ko jārisina programmatūras izstrādes projektā. Konfigurācijas pārvaldība ir disciplīna, kas identificē produkta sastāvdaļas un nodrošina to versiju kontroli, produkta kompilācijas un instalācijas procesu, kā arī metrikas savākšanas un paralēlas izstrādes iespējas. Analizējot literatūru, izdevās secināt, ka galvenais uzsvars konfigurācijas pārvaldībā ir uz kontroli. Līdz ar to projekta sākumā jāprot pareizi identificēt produkta sastāvdaļas un atrisināt jau minētos konfigurācijas pārvaldības uzdevumus, kas nodrošinās produkta kontroli visa projekta dzīves cikla garumā. Nākamajā apakšnodaļā detalizēti tiks apskatīts katrs no uzdevumiem, to risināšanas paņēmieni un aktuālas problēmas.

Analizējot literatūru, izdevās noskaidrot galvenās konfigurācijas pārvaldības definīcijas un galvenos uzdevumus. Sakarā ar to, ka promocijas darba mērķis ir izstrādāt pieeju un metodoloģiju konfigurācijas pārvaldības automatizācijas ieviešanai, balstoties uz šajā nodaļā iegūto informāciju, tiks definēta konfigurācijas pārvaldības automatizācija.

Konfigurācijas pārvaldības automatizācijas risinājumi – programmatūra, kas realizē šajā nodaļā definētos konfigurācijas pārvaldības uzdevumus, minimizējot cilvēka iejaukšanos. Galvenokārt, automatizācija ir vērsta uz versiju kontroli, izejas koda pārvaldību, programmatūras būvējumu veidošanu, programmatūras instalāciju.

Līdz ar to formulējums «izstrādāt automatizāciju konfigurācijas pārvaldībai» promocijas darba kontekstā nozīmē izstrādāt programmatūras kopu (skripti, bibliotēkas, ietvari), kas ar minimālu cilvēka iejaukšanos spēj veikt šajā nodaļā definētos konfigurācijas pārvaldības uzdevumus, galvenokārt, versiju kontroli, izejas koda pārvaldību, būvējumu un instalācijas veidošanu.

1.3. Konfigurācijas vienumu identifikācija

Literatūras analīzes rezultātā [AIE 2010, BER 2003, MET 2002, KAN 2005, CON 2002, ЛАП 2004, ЗАМ 2008, SCM 2001, OSE 2002, BRO 2002, SCH 2010, SEK 2012, SAR 2008, EIL 2006, HAG 2010, BRO 2005, JUI 2002] tika secināts, ka visus avotus, kur ir aprakstīta konfigurācijas vienumu identifikācijas pieejas, var iedalīt trijās grupās.

1. grupā ir aprakstīti tikai vispārīgie principi, kas palīdz identificēt produkta konfigurācijas vienumus. Šajos avotos ir uzsvērts, ka konfigurācijas vienumu noteikšanas process ir pilnīgi atkarīgs no cilvēka, kas veic identifikāciju, no cilvēka zināšanām un spējām pareizi interpretēt konkrēta produkta arhitektūru. Šajos avotos praktiski nekas nav teikts par rīkiem un metodēm, ko var izmantot konfigurācijas vienumu identifikācijas procesā. Pārsvarā tiek uzsvērts, ka ne tik svarīgi ir izvēlēties, kādu rīku vai metodi, bet svarīgi ir pilnībā izprast procesu, labi pārzināt produkta arhitektūru un izstrādes tehnoloģiju. Kopumā identifikācija tiek pasniegta kā process, kas pilnībā ir atkarīgs no cilvēka vai cilvēku grupas, kas veic identifikāciju [AIE 2010, BER 2003, MET 2002, SCM 2001, OSE 2002].

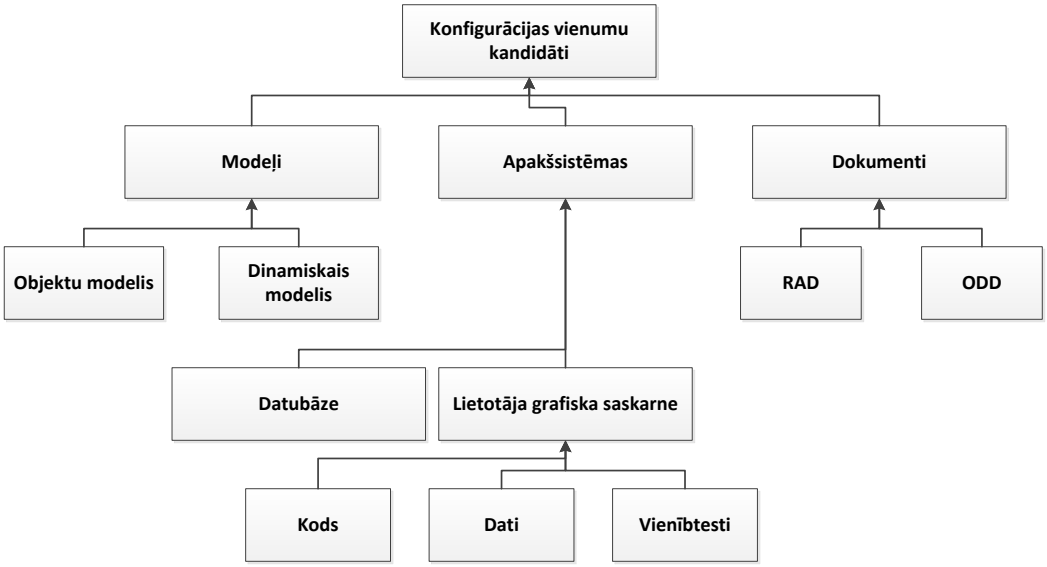
2. grupā tiek uzsvērts, ka galvenā panākumu atslēga konfigurācijas vienumu identifikācijas procesā ir izstrādājama produkta prasību definīcija un analīze. Jo precīzāk izdodas definēt produkta prasības un modelēt sistēmas arhitektūru, jo veiksmīgāks būs konfigurācijas vienumu identifikācijas process. Šāda veida publikācijās no vienas puses ir parādīts, ka konfigurācijas pārvaldība ir atkarīga no citiem projekta procesiem, šajā gadījumā – no prasību analīzes fāzes. No otrās puses – ir mēģinājumi sniegt metodes un rīkus, kas modelē sistēmas arhitektūru un tādējādi atvieglo un automatizē konfigurācijas vienumu identifikācijas procesu [SEK 2012, EIL 2006, HAG 2010, BRO 2005, JUI 2002].

3. grupā pētnieki atklāti atzīst, ka konfigurācijas vienumu identifikācijas process pats pa sevi ir pietiekami sarežģīts un daudzveidīgs, lai ieteiktu kādu universālu metodi, kas būtu piemērota jebkura veida projektiem neatkarīgi no izstrādes tehnoloģijas. Pretstatā šim apgalvojumam pētnieki uzsver, ka nepilnības konfigurācijas identifikācijā var novērst, risinot

citus konfigurācijas pārvaldības uzdevumus. Proti, identifikācijas nepilnības var labot, veicot pārdomātu un racionālu versiju kontroli. Šādas pieejas piekritēji aicina nemēģināt ideāli identificēt vienumus pašā projektā sākumā, bet izveidot korektu un efektīvu konfigurācijas pārvaldību kopumā, kas darbotos visa projekta dzīves cikla garumā. Metodoloģijas un rīki, kas ir piedāvāti šajos avotos, strādā jau ar gataviem konfigurācijas vienumiem ar mērķi optimizēt konfigurācijas vienumu saturu. [SAR 2008, SCH 2010, BRO 2002]

Turpmāk šajā nodaļā tiks analizēti avoti, kuros vai nu ir minētas metodes konfigurācijas vienumu identifikācijai, vai arī iespējas, kā var uzlabot saturu jau identificētai vienumu kopai. Analīzes mērķis ir noteikt mūsdienīgus jaunākus risinājumus konfigurācijas vienumu identifikācijā, noteikt galvenās problēmas, ar kurām saskaras pētnieki, definēt nākotnes tendences konfigurācijas vienumu identifikācijā. Detalizēti netiks apskatīti pirmās grupas avoti, jo pamatprincipi ir minēti arī otrās un trešās grupas avotos, taču no zinātniskā viedokļa daudz vairāk informācijas varētu dot metodoloģiju un to trūkumu apraksts nekā konkrētu speciālistu pieredzes apraksts.

Konfigurācijas vienumus var iegūt modelēšanas procesā. Identifikācijas procesa rezultāts ir atkarīgs no tā, cik veiksmīgs ir sistēmas projektējums, kāda ir speciālistu kvalifikācija un kādus principus izvēlas vienumu nosaukumu veidošanā [OSE 2002]. Lai šo procesu atvieglotu, [OSE 2002] autori piedāvā veikt vienkāršu sistēmas modelēšanu. Sākumā tiek uzsvērts, ka, definējot vienumu potenciālus kandidātus, jāņem vērā ne tikai izejas kods, bet arī dokumentācija, kas apraksta produktu no dažādiem aspektiem. Modelēšanas pieeja balstās uz pakāpenisku sistēmas kokveida struktūras izveidošanas. Sākotnēji tiek definēti izstrādājama produkta dokumenti, modeļi un apakšsistēmas. Nākamajā iterācijā dokumenti tiek sadalīti pa grupām, tiek definētas saiknes starp tiem. Savukārt modeļiem tiek definēti apakšmodeļi un to savstarpējās saiknes. Visbeidzot apakšsistēmām tiek definētas sastāvdaļas, piemēram, datubāzes, lietotāja grafiskās saskarnes utt. Nākamajās iterācijās struktūru paplašina ar jauniem elementiem un to savstarpējām saiknēm. Process turpinās, kamēr netiek pieņemts lēmums par to, ka katru zemāka slāņa virsotni nevar iedalīt sīkāk un ir definētas visas uz to brīdi redzamās saiknes starp elementiem. Šāda pieeja prasa vairāku speciālistu piedalīšanos. Apvienojot zināšanas par produkta prasībām, arhitektūru, izstrādes tehnoloģijām, dokumentāciju, diskusijas veidā var izveidot konfigurācijas vienumu koku. Pieeja paredz vairāku speciālistu grupveida darbu un diskusijas, kuru mērķis ir atrast kopēju priekšstatu par produkta struktūru, uzbūvi un konfigurācijas vienumiem. 1.2. attēlā ir parādīts piemērs konfigurācijas vienumu struktūrai. Šāds piemērs varētu būt tikko aprakstītās modelēšanas starprezultāts.



1.2. att. Konfigurācijas vienumi un to struktūras

Mūsdienās tikko aprakstītajam risinājumam ir parādījušies inovatīvi uzlabojumi, viens no tiem ir aprakstīts [SEK 2012] publikācijā. Ideja interpretēt produktu kā elementu koku paliek tāda pati kā avotā [OSE 2002]. Taču pastiprināta uzmanība tiek pievērsta katra elementa raksturlielumiem jeb parametriem. Ir izstrādāts algoritms, kas analizē konfigurācijas vienumu parametrus. Tiklīdz ir izveidots jauns zars, algoritms šajā zarā veic elementu analīzi un salīdzina elementu parametrus un to vērtības. Algoritma galvenais mērķis ir identificēt līdzīgus parametrus ar mērķi samazināt iespējamu konfigurāciju skaitu. Ja izdodas atrast konfigurācijas vienumus ar identiskiem parametriem, tad šādus vienumus algoritms apvieno, tādējādi samazinot kopēju vienumu skaitu. Rezultātā rodas konfigurācijas vienumi ar mazāku parametru skaitu, samazinās dažādu kombināciju skaits un kļūst vieglāk pārvaldīt dažādas konfigurācijas. Algoritmu var lietot gan elementu koka modelēšanas gaitā, gan arī tad, kad konfigurācijas vienumu koks jau ir izveidots. Avotā [SEK 2012] ir minēts, ka mūsdienās ir citi risinājumi, kas palīdz identificēt konfigurācijas vienumus, taču, lai risinājumi strādātu, ir nepieciešams definēt likumus, pēc kuriem sistēma varētu darboties. Likumu definēšanai ir vajadzīgi eksperti, un tas nozīmē, ka šāda veida risinājumi lielā mērā ir atkarīgi no konkrētu ekspertu zināšanām un pieredzes. Kā galveno priekšrocību [SEK 2012] algoritmā autori min faktu, ka parametru analīzes algoritmam nav nepieciešama jaunu likumu sastādīšana.

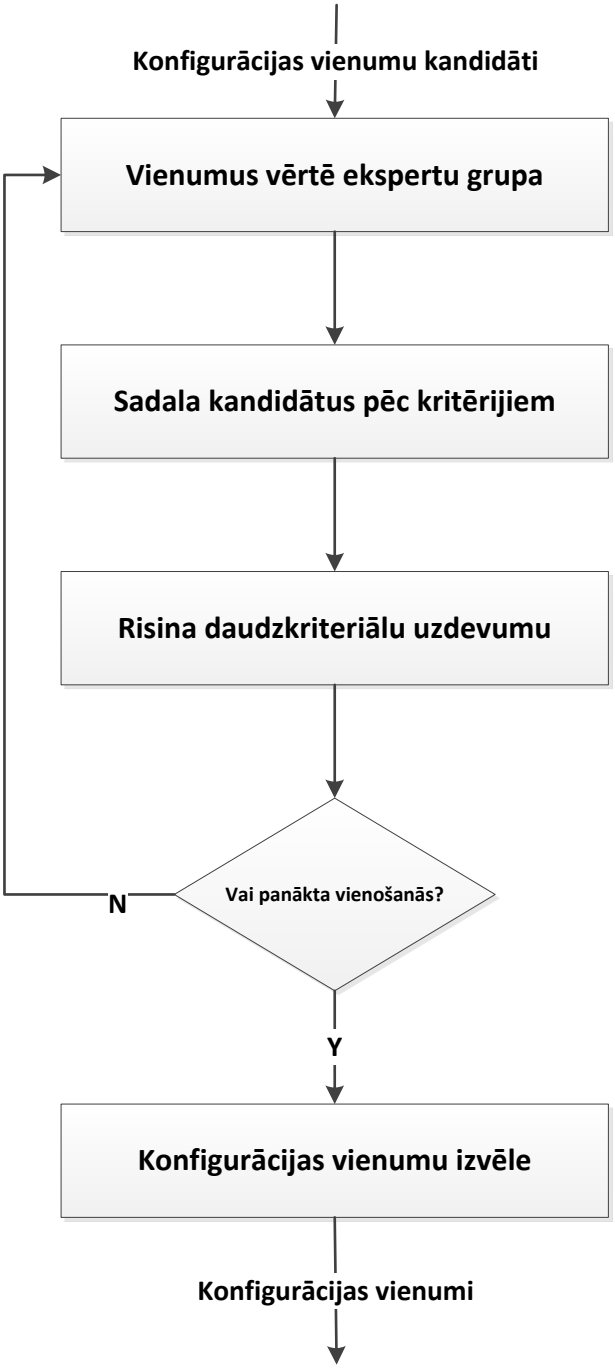
Atgriežoties pie produkta struktūras modelēšanas, avotā [SEK 2012] ir minētas konkrētas modelēšanas tehnoloģijas – *VDM* (angļu val. *The Vienna Development Method*) un *Alloy*. Minētās tehnoloģijas piedāvā modelēt programmatūras uzbūvi un to konfigurācijas

vienumus, tām ir arī rīku atbalsts. Taču visiem tiem ir viens būtisks trūkums konfigurācijas pārvaldības kontekstā, proti, nepieciešams definēt likumus, kas nosaka, kā veidojas konfigurācijas vienumi, bet to var izdarīt tikai eksperts, kuram ir zināšanas par konkrētu izstrādes tehnoloģiju un izstrādājamu produktu.

Viens no oriģināliem risinājumiem konfigurācijas vienumu identifikācijā ir modelēšanas metode, kas ir balstīta uz izplūdušās loģikas teoriju un lēmumu pieņemšanu [JUI 2002]. Publikācijā labi ir formulēta galvenā problēma konfigurācijas vienumu identificēšanas procesā. Ja definēt pārāk daudz konfigurācijas vienumu, strauji palielinās programmatūras konfigurācijas pārvaldības un kontroles izmaksas. Savukārt definējot pārāk maz konfigurācijas vienumu, pastāv risks, ka programmatūra netiks kontrolēta pietiekami kvalitatīvi. Ņemot vērā, ka kontrole ir svarīgākais uzdevums konfigurācijas pārvaldībā, var secināt, ka nekvalitatīva konfigurācijas vienumu identifikācija var izraisīt nopietnas problēmas konfigurācijas pārvaldībā kopumā.

Publikācijas [JUI 2002] galvenais mērķis ir izstrādāt metodi, kas varētu no konfigurācijas vienumu kopas izvēlēties korektus vienumus. Atlases uzdevums ir formulēts kā daudzkriteriālais lēmumu pieņemšanas uzdevums. Izplūdušu loģiku autori lieto tāpēc, ka programmatūras izstrādes sākuma posmā ir grūti definēt konfigurācijas vienumu atlases kritēriju konkrētas skaitliskās vērtības. Izvēloties vienu konkrētu kritēriju un vērtēšanas skalu, bieži ir grūti definēt konfigurācijas vienumu piederības diapazonu. Tāpēc tika pieņemts lēmums aprakstīt lēmumu pieņemšanas uzdevumu ar izplūdušās loģikas teorijas palīdzību.

Attēlā 1.3. parādīts konfigurācijas vienumu identifikācijas process, kas ir balstīts uz lēmumu pieņemšanu. Attēls ir aizgūts no [JUI 2002].



1.3. att. Konfigurācijas vienumu identifikācijas process

Metode ar izplūdušo loģiku no vienas puses šķiet labi pamatota un izmantojama. No otrās puses, patlaban metodoloģijai nav pietiekama rīku atbalsta, kas apgrūtina tās izmantošanu praksē. Vēl viens trūkums ir saistīts ar cilvēcisko faktoru. Izplūdusī loģika satura ziņā ir diezgan sarežģīta, lai to apgūtu, nepieciešams iepazīties ar lielu daudzumu teorētiskā materiāla. Pretējā gadījumā saprast [JUI 2002] algoritma darbības principus ir neiespējami,

taču industrijas prakse rāda, ka speciālistiem grūti pieņemt ko jaunu, ja nav skaidri zināmi darbības principi [AIE 2010, MET 2002].

Turpinot pētīt publikācijas par konfigurācijas vienumu identifikāciju, nācās secināt, ka ir vairākas tehnikas un metodes, kas piedāvā strādāt jau ar gatavu konfigurācijas vienumu kopu. Tas nozīmē, ka metodes koncentrējas ne tik daudz uz identificēšanas procesu, bet gan uz darbu ar jau gataviem vienumiem, ar mērķi definēt parametrus, noteikt atkarības un konfliktējušos vienumus [BRO 2005, HAG 2010, SAR 2008]. Pārsvarā šajos un saistītajos darbos konfigurācijas vienumu kopa ir definēta kā vienota glabātuve (angļu valodā ir jēdziens: *workspace*). Sistēmas, ko piedāvā pētījuma autori, *CHAMPS, FASTDash, CollabVS* pārvalda konfigurācijas vienumu glabātuvi ar mērķi identificēt konfigurācijas vienumu atkarības, konfliktus, izprast kopēju struktūru. Šādā veidā tiek piedāvāts uzlabot konfigurācijas vienumu kvalitāti un novērst kļūdas, kas tika pieļautas, sākotnēji identificējot konfigurācijas vienumus.

Apkopojot veikto pētījumu par konfigurācijas vienumu identificēšanas procesu, jāsecina, ka patlaban process ir cieši saistīts ar prasību analīzes un projektēšanas fāzi. Lai arī kādas pieejas un metodes atbalstītu konfigurācijas vienumu identificēšanas procesu, konfigurācijas vienumu kvalitāte tik un tā ir atkarīga no prasību analīzes un projektēšanas fāzes. Runājot par metodēm, ir jāsecina, ka lielāka daļa ir vērsta uz produkta struktūras modelēšanu. Ir metodes, kur svarīgāka loma ir ekspertam, kas definē konfigurācijas vienumu veidošanas likumus, ir metodes, kas pēta konfigurācijas vienumu parametrus un to saiknes. Ir metodes, kas ir piesaistītas konkrētām tehnoloģijām un to lietošana citu tehnoloģiju kontekstā būtu apgrūtinoša. Dažām pieejām vispār nav rīku atbalsta, un realizācija ir galvenais uzdevums tiem, kas grib pieeju izmantot praksē. Konfigurācijas pārvaldības vadošie speciālisti [AIE 2010, MET 2002] apgalvo, ka pārvaldību ir iespējams realizēt veiksmīgi tikai tad, ja ir atrisināti visi galvenie uzdevumi nevis tikai kāds viens no tiem. Speciālisti min faktu, ka kļūdas vienumu identifikācijā vēlāk var konstatēt un izlabot, ja kopējais konfigurācijas pārvaldības process ir kvalitatīvi organizēts. Tāpēc promocijas darba turpinājumā tiks apskatīti pārējie konfigurācijas pārvaldības uzdevumi, un pirmais no tiem būs versiju kontrole un paralēlās izstrādes nodrošinājums.

1.4. Versiju kontrole, paralēla izstrāde un metriku savākšana

Versiju kontrole ir svarīgākais uzdevums, ko risina konfigurācijas pārvaldība. Tas izriet no konfigurācijas pārvaldības definīcijas, ka konfigurācijas pārvaldība kontrolē izmaiņas produktā un uztur izmaiņu vēsturi [AIE 2010, MET 2002]. Pieņemot, ka produkts sastāv no komponentēm, kurās ik pa laikam tiek veiktas dažādas izmaiņas, versiju kontrolei jānodrošina šāda uzskaite:

- kas veica izmaiņas;
- ar kādu nolūku;
- kādas komponentes tika mainītas;
- kad tika veiktas izmaiņas;
- kas tika mainīts salīdzinājumā ar iepriekšējo versiju;
- kā radās komponentes iepriekšējā versija;
- komponentes izmaiņu vēsture, sākot no komponentes izveidošanas brīža.

Minētais ir tikai daļa no informācijas kopas, kas ir nepieciešama, lai kvalitatīvi atrisināt versiju kontroles uzdevumu [AIE 2010, MET 2002, CON 2002]. Versiju kontroles uzdevums ir nodrošināt arī paralēlas izstrādes iespējas. Šā darba kontekstā ar paralēlu izstrādi ir jāsaprot darbu ar produkta komponentēm, ko vienlaikus veic vairāki cilvēki. Ja pieņem, ka projektā eksistē vismaz viena produkta komponente, kurā veic izmaiņas vienlaikus vairāki cilvēki, tad paralēlas izstrādes uzdevums ir nodrošināt izmaiņu savstarpēju sinhronizāciju, lai varētu izsekot izmaiņu vēsturei un lai nekādas izmaiņas netiktu pazaudētas [AIE 2010, MET 2002]. Kā jau tika minēts iepriekš, viens no konfigurācijas pārvaldības uzdevumiem ir nodrošināt metriku savākšanu par izmaiņām produktā. Ņemot vērā to, ka izmaiņas uzskaita versiju kontroles sistēmas, ir jēga kopā apskatīt arī metriku savākšanas uzdevumu. Turpmāk šajā nodaļā versiju kontroles uzdevuma kontekstā tiks runāts arī par paralēlas izstrādes un metriku savākšanas uzdevumiem, jo minētie uzdevumi ir cieši saistīti un tos ir jārisina kopā konfigurācijas pārvaldības ietvaros [AIE 2010, MET 2002, CON 2002].

Runājot par versiju kontroles mūsdienīgām tendencēm, ir jāatzīmē, ka būtiskākie sasniegumi šajā jomā sākās līdz ar versiju kontroles sistēmu parādīšanos [AIE 2010]. Versiju kontroles sistēmas ir centralizētas vietas, kas glabā produkta izejas kodu, dokumentāciju, testus un cita veida produkta sastāvdaļas vai dokumentus, kas reglamentē produkta būtību. Sistēmas palīdz sinhronizēt programmētāju darbu, kas strādā vienlaikus, veicot izmaiņas viena un tajā pašā produkta vienumā. Versiju kontroles sistēma uztur visu izmaiņu vēsturi produkta izejas kodā. Jebkāda veida izmaiņas tiek definētas versiju kontroles sistēmā kā

ieraksts, kas satur visu informāciju par konkrētu izmaiņu. Katram ierakstam ir piešķirts unikāls identifikators, ko sauc arī par revīziju. Katra revīzija viennozīmīgi identificē konkrētas izmaiņas atribūtus: kas veica izmaiņas, kādas komponentes tika mainītas, kādas daļas, kad tika veiktas izmaiņas, kādas ir atšķirības, salīdzinot ar iepriekšēju versiju utt. Informācija, kas atrodas versiju kontroles sistēmā, ir ļoti svarīga projekta dzīves cikla laikā. Versiju kontroles sistēma kontrolē izejas koda failus un nepieciešamības gadījumā sniedz informāciju projekta izstrādātāju komandai. Versiju kontroles sistēma automātiski sinhronizē izmaiņas produktā, ko vienlaikus veic vairāki izstrādātāji, tādējādi atvieglojot izstrādātāju darbu [SAT 2011].

Mūsdienās versiju kontroles sistēmas var iedalīt divās grupās: centralizētās un distributīvās. Centralizētās sistēmas piedāvā tādu arhitektūru, ka ir centrālais repozitorijs un katram izstrādātājam ir sava lokāla kopija. Izstrādātājs veic izmaiņas savā kopijā un tikai tad nosūta tās uz centrālo repozitoriju. Kopš tā brīža šīm nosūtītajām izmaiņām tiek piešķirts unikālais identifikators, un izmaiņas kļūst pieejamas citiem izstrādātājiem. Centralizētu versiju kontroles sistēmu galvenais princips ir tāds, ka izmaiņu vēsturi var uzturēt tikai centrālajā repozitorijā. Lokālajā kopijā izmaiņu vēsturi uzturēt nevar, tiklīdz lokālās izmaiņas ir nosūtītas uz centrālo repozitoriju, tās kļūst pieejamas visiem. Distributīvās versiju kontroles sistēmas (pretēji centralizētajām) ļauj uzturēt lokālu izmaiņu vēsturi. Katram izstrādātājam ir sava lokālā kopija, kurā ir iespējams veikt izmaiņas. Lokālu izmaiņu vēsture ir pieejama tikai vienam konkrētam lietotājam, kurš tās ir veicis. Pārējiem lietotājiem šīs izmaiņas kļūs pieejamas tikai tad, kad lietotājs dod speciālu komandu sapludināt savas izmaiņas ar centrālo repozitoriju. Dažās jaunākajās versiju kontroles sistēmās centralizēta repozitorija jēdziens vispār neeksistē. Šāda veida sistēmās lietotājs var integrēt savas izmaiņas tikai ar kāda konkrēta lietotāja izmaiņām. Ir iespēja nosūtīt savas izmaiņas uzreiz vairākiem izstrādātājiem. Visas distributīvās versiju kontroles sistēmas ļauj sinhronizēt ar pārējiem izstrādātājiem tikai kādu noteiktu daļu no savām lokālām izmaiņām, pateicoties tam, ka lokālu izmaiņu vēsture nepārtraukti tiek uzturēta [AIE 2010, BER 2003, MET 2002].

Apkopojot avotā [STA 2008] sniegto informāciju, tiks aprakstītas populārākās versiju kontroles sistēmas.

***CVS* (*Concurrent Versions System*)** – centralizēta versiju kontroles sistēma. Centralizēta – nozīmē, ka visi kontrolējamie faili glabājas uz viena servera, bet lietotajiem ir klients, kas ļauj iegūt informāciju par failiem, versijām, iegūt faila lokālu kopiju un nosūtīt lokālās kopijas izmaiņas uz serveri, radot failā kārtējo versiju. Sistēma tiek izplatīta uz *GNU GPL* licences pamata. Patlaban sistēmas izstrāde vairs nenotiek – pēdējais laidiens bija 2008. gadā, un kopš tā laika sistēmā tiek labotas tikai kļūdas.

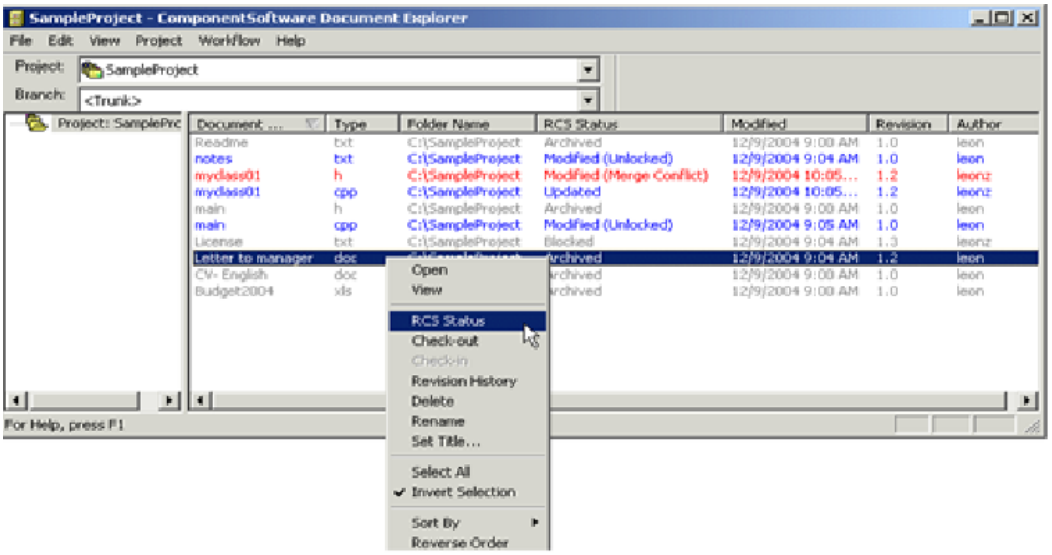
CVS izmanto servera-klienta arhitektūru. Parasti klients un serveris savā starpā savienojās caur lokālo tīklu vai interneta tīklu. Servera daļas programmnodrošinājums strādā uz *UNIX* platformas, kaut gan ir versija arī *Windows NT* serverim. Serveris glabā projekta aktuālo versiju un izmaiņu vēsturi speciālā repozitorijā, bet klients pieslēdzas pie repozitorija, lai iegūtu kādu no versijām vai radītu jauno versiju. Ar serveri var strādāt vairāki klienti vienlaikus. Kad klienti sūta rezultātus uz serveri, *CVS* pārbauda, vai izmaiņas failos nekonfliktē savā starpā. Ja izmaiņas neizdodas saplūdināt, piemēram, gadījumos, kad ir labotas faila vienas un tās pašas rindas, *CVS* ziņo lietotājiem par kļūdu un piedāvā manuāli atrisināt izmaiņu saplūdināšanas konfliktus.

CVS piedāvā salīdzināt viena faila divas dažādas versijas savā starpā, iegūt versijas informāciju: kurš versiju rādīja, kad, kādus failus modificēja un kādu komentāru par versiju ir atstājis. Lai sinhronizētu lokālo kopiju ar servera kopiju, nav nepieciešams ielādēt projektu no jauna, var izmantot atjaunināšanas funkciju, kas ļauj lejupielādēt no servera tikai izmaiņas, kuru vēl nav lokālajā kopijā.

CVS nodrošina arī paralēlu izstrādi. Ir iespēja glabāt failu vairākas kopijas, modificēt tās un pēc tam saplūdināt izmaiņas.

CVS galvenie trūkumi izpaužas pie izmaiņu pārsūtīšanas uz servera. Ja ir nepieciešams pārsūtīt uz serveri izmaiņas vairākos failos, tad serveris neatcels pārsūtīšanas transakciju, ja kādus failus neizdevās pārsūtīt. Tas dažreiz var radīt papildu problēmas, jo programmētājs veic izmaiņas vairākos failos ar domu, ka visas izmaiņas nonāks sistēmā vienlaikus.

Vēl viens sistēmas trūkums ir tas, ka sistēma neglabā informāciju par to, kā radās versija. Ja, piemēram, versija *X* ir radusies sapludināšanas ceļā no cita zara, tad pēc kāda laika, mēģinot saplūdināt failu *X* ar minēto zaru, sistēma neņems vērā, ka jāņem tikai neeksistējošās izmaiņas. Attēlā 1.4. ir redzams *CVS* sistēmas klients *Windows* platformai.



1.4. att. *CVS* klients *Windows* platformai

Vairāki programmatūras izstrādātāji pasaulē uzskata, ka *CVS* sistēma ir novecojusi un pamato šo apgalvojumu ar to, ka ir jaunākas sistēmas, kurās galvenie trūkumi ir novērsti, piemēram, sistēmas *Subversion* un *Git* [AIE 2010, STA 2008], par kurām arī tiks minēts darba turpinājumā.

Subversion – bezmaksas versiju kontroles sistēma. Sistēma ir centralizēta, tai ir klienta-servera arhitektūra. Uz servera glabājas projekta failu aktuālās versijas un informācija par iepriekšējām versijām, savukārt klients ļauj to informāciju iegūt. *Subversion* glabā informāciju ne tikai par failiem, bet arī par direktorijām, kur šie faili atrodas. *Subversion* sistēma ņem vērā pilnu ceļu līdz konkrētam failam failsistēmā, līdz ar to, ja direktoriju struktūra pamainās, *Subversion* ieraksta kārtējo projekta versiju. Līdz ar to minimālā vienība, ko var iegūt no servera lokālas kopijas, ir direktorija nevis fails.

Subversion sistēmu pirmo reizi izlaida 2004. gadā, to attīsta joprojām. Projekta mērķis bija radīt versiju kontroles sistēmu, kurai būtu visas *CVS* funkcijas un nebūtu galveno *CVS* trūkumu. Tas izstrādātājiem lielā mērā ir izdevies un patlaban *Subversion* sistēma ir ļoti populāra, to lieto tādi pazīstamie projekti kā *Apache, KDE, GCC, Free Pascal, Python, PHP, Ruby, Mono, FreeBSD, Haiku OS, AROS un MediaWiki* [STA 2008].

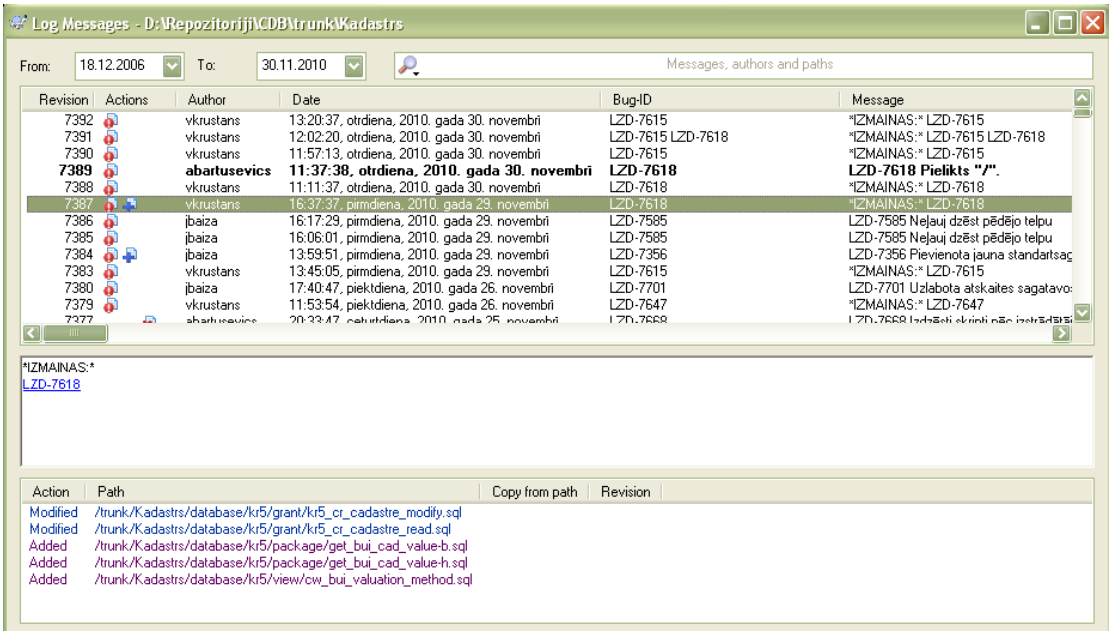
Pēc statistikas 2010. gada vasarā *Subversion* lietotāju skaits pārsniedza citu populāru sistēmu lietotāju skaitu. *Subversion* ļauj uzturēt informāciju par failu versijām pat tad, kad fails ir pārkopēts uz citu direktoriju. Sistēma ļauj glabāt gan faila, gan versijas atribūtus un veidot jaunus atribūtus, kas ir nepieciešami konkrēta projekta vajadzībām.

Subversion sistēmai labāk nekā *CVS* ir attīstīts dažādu zaru sapludināšanas mehānisms. Kad repozitorijā tiek veiktas izmaiņas, repozitorijam tiek piešķirts kārtējās versijas skaitlis. Šo skaitli sauc par revīziju. Katrai revīzijai ir tādi pamata atribūti kā revīzijas autors, datums, modificēti, pieliktie un dzēstie faili, komentārs.

Subversion sistēmai ir labi attīstīts lietotāju kontroles mehānisms, kas ļauj viegli organizēt darba kārtības ievērošanu ar failiem un direktorijām, kas glabājas repozitorijā. Uz servera atrodas skripti, kuros var definēt šos ierobežojumus. Skripti nostrādā brīdī, kad lietotājs sūta izmaiņas no savas lokālās kopijas uz serveri. Piemēram, var uzstādīt ierobežojumu, ka failu nosaukumi nedrīkst būt ar lieliem burtiem.

Viena būtiska priekšrocība *Subversion* sistēmā, salīdzinot ar *CVS*, ir izmaiņu nedalāmība. Proti, ja programmētājs mēģina nosūtīt vairākus failus uz serveri, tad kāda faila neveiksmīgas sūtīšanas gadījumā uz servera nenonāks arī citi faili.

Subversion ir vairāki klienti gan *Windows*, gan *Linux* platformām, ar grafisko lietotāju saskarni un bez tās. Pastāv arī *JAVA* bibliotēka *SVNKIT*, kas ļauj izstrādāt personalizētus klientus darbam ar *Subversion* serveri un to repozitorijiem. Bibliotēka ļauj veidot palīgprogrammas darbam ar *Subversion* repozitoriju, kas ir nepieciešamas konkrētam projektam un ko nepiedāvā standarta *Subversion* klientprogrammas. Attēlā 1.5. ir redzams viens no populārākiem klientiem darbam ar *Subversion* repozitoriju – *TortoiseSVN*.



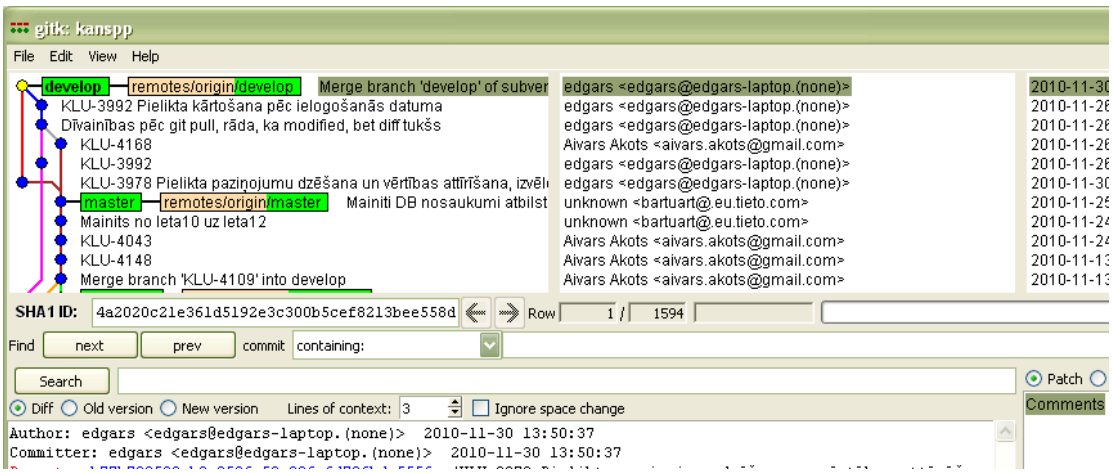
1.5. att. *Subversion* klients – *TortoiseSVN*

Git – distributīva versiju kontroles sistēma. Katram lietotājam ir sava repozitorija lokālā kopija, kuras ietvaros iespējams veikt pilnvērtīgu versiju kontroli un uz serveri sūtīt tikai vajadzīgās izmaiņas. *Git* pēc savas būtības ir programmu kopums, ko var izmantot *UNIX Shell* skriptos, un radīt pašam savu versiju kontroles sistēmu, kas vislabāk atbilst konkrētam projektam.

Git atbalsta versiju sadalīšanu un sapludināšanu, kā arī piedāvā vairākus grafiskus rīkus, kas palīdz iegūt pārskatāmu informāciju par versijām. *Git* sistēmai atšķirībā no *Subversion* revīzija glabā informāciju par to, kā šī versija radās, līdz ar to daudz vieglāka ir divu dažādu zaru sapludināšana. *Git* versiju identifikācijai izmanto nevis naturālu skaitli, kā to dara *Subversion*, bet kontrolsummu, kurā arī glabājas informācija par versijas izcelšanos. Attālināta pieeja *Git* repozitorijam ir realizēta, izmantojot *SSH* vai *HTTP* serveri.

Viens no galvenajiem *Git* trūkumiem ir tas, ka *Git* ir orientēts uz izmantošanu *Linux* vidē. Realizācijas, kas ļautu izmantot *Git* visas iespējas *Windows* vidēs, pagaidām nav. *Windows* videi populārs ir *MSYSGit* rīks, kas ļauj darboties ar *Git* sistēmu arī *Windows* lietotājiem. *Git* ir bezmaksas sistēma, ko izplata, pamatojoties uz licenci *GNU GPL*.

Attēlā 1.6. ir redzams piemērs no *Gitk* grafiskā rīka, kas ļauj analizēt izmaiņu vēsturi repozitorijā.



1.6. att. *Git* klients izmaiņu pārskatam

Mercurial – no platformas neatkarīga distributīva versiju kontroles sistēma. Atšķirībā no *Git* sistēmas tai nav centrālā repozitorija jēdziena. Katram lietotājam ir sava kopija un iespēja nosūtīt savas izmaiņas uz citu repozitorija kopiju. *Mercurial* strādā gan *Windows*, gan *Linux* platformā. Darbs ar *Mercurial* notiek konsolē, izmantojot programmu «*hg*». Ir arī grafiskās saskarnes, kas ļauj iegūt informāciju par versijām, bet veikt jebkādas

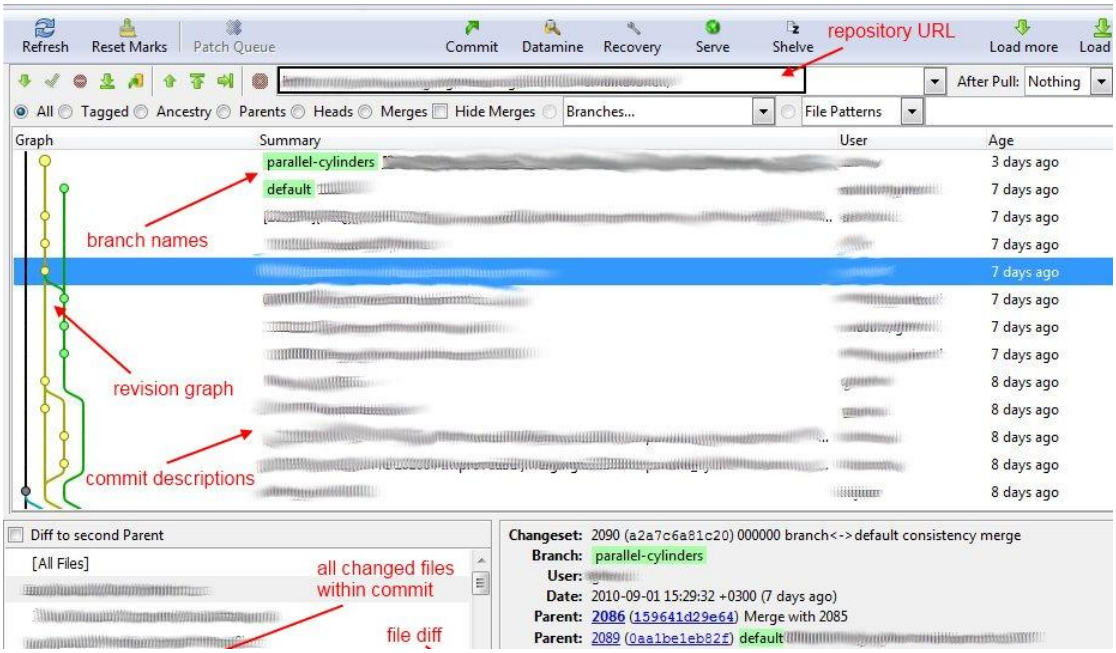
izmaiņas ir iespējams tikai konsolē. *Mercurial* ir paredzēts lieliem projektiem, kur ir milzīgi projekta repozitoriji.

Mercurial sistēmā ir ļoti labi organizēts sapludināšanas mehānisms starp dažādiem zariem. Katra revīzija glabā informāciju par sevi, no kuras vēsturiskās revīzijas tā ir radusies sapludināšanas gadījumā. Līdz ar to sistēma ir piemērota lieliem projektiem ar vairākiem paralēliem zariem.

Mercurial sistēmā ir rīks, kas ļauj konvertēt repozitorijus no citām versiju kontroles sistēmām, piemēram, *Subversion, Git, Bazaar*. Galvenais trūkums ir tāds, ka *Mercurial* visas operācijas ir jāveic no konsoles, kas varbūt nav īpaši ērti cilvēkiem, kuri ir pieraduši ikdienā strādāt ar lietotāja grafiskām saskarnēm, izmantojot dažādas programmas, un kuriem ir neērti strādāt ar *Linux* komandrindu.

Mercurial ir integrēts ar izstrādes vidēm *NetBeans, Eclipse* un *Microsoft Visual Studio*.

Attēlā 1.7. var redzēt *Mercurial* klientu, kas ļauj iegūt pārskatāmu informāciju par revīzijām.



1.7. att. *Mercurial* klients *TortoiseHg*

Lai būtu lielāks priekšstats par versiju kontroles sistēmām, tabulā 1.2. ir dota versiju kontroles sistēmu salīdzināšana, kurā ir minētas galvenās priekšrocības un galvenie trūkumi.

1.2. tabula

Versiju kontroles sistēmu salīdzinājums

Versiju kontroles sistēma	Priekšrocības	Trūkumi
CVS	• Ar vienu un to pašu koda repozitoriju vienlaikus var strādāt vairāki lietotāji. • Ļauj pārvaldīt izmaiņas gan vienam failam, gan visam repozitorijam. • Sistēmai ir ļoti daudz dažādu grafisku saskarņu. • Sistēma ir plaši izplatīta un iebūvēta gandrīz visās *Linux* veida operētājsistēmās.	• Mainot failam nosaukumu, pazūd visa iepriekšēja izmaiņu vēsture par šo failu. • Ir vāji attīstīts paralēlas izstrādes nodrošinājums, grūti sapludināt izmaiņas no viena izstrādes zara uz otro. • Bināru failu izmaiņu gadījumā saglabājas nevis izmaiņas, bet tikai faila jaunāka versija. • Operācijas ar repozitoriju prasa daudz sistēmas resursu.
Subversion	• Sistēmai ir gan grafiskās saskarnes rīki, gan arī iespēja strādāt no konsoles. • Failu un katalogu vēsture saglabājas pat pēc pārsaukšanas. • Veiksmīgi strādā gan ar teksta failiem, gan arī ar bināriem failiem. • Integrēts ar *NetBeans*, *Eclipse* un citām izstrādes vidēm. • Paralēlas izstrādes un zarošanas nodrošinājums. • Katram failam un	• Lai lokāli glabātu repozitorija kopiju, ir nepieciešams salīdzinoši daudz atmiņas, jo atribūti glabājas slēptajos (angļu val. *hidden*) failos. • Ir problēmas ar faila pārsaukšanu brīdī, ja šo pašu failu ir modificējis cits lietotājs. • Dažādu zaru sapludināšana ir vāji automatizēta, jo lietotājam pašam jāseko, kā ir radusies kāda no versijām.

	direktorijai ir atribūtu kopa, kas atvieglo versiju kontroli un administrēšanu.	
Git	• Droša revīziju salīdzināšanas un datu korektuma pārbaudes sistēma. • Labi attīstīta paralēlas izstrādes stratēģija un plašas dažādu zaru sapludināšanas iespējas. • Lokāla repozitorija esamība, kas satur visu izmaiņu vēsturi no centrālā repozitorija. Tādējādi ir iespēja kontrolēt versijas lokāli un sūtīt uz serveri tikai vajadzīgās versijas. • Augsta veiktspēja. • Daudz rīku, kas piedāvā grafisku saskarni. • Sistēma ļauj veidot savas versiju kontroles sistēmas, balstoties uz *git* komandām, ko var iestrādāt skriptos. • Universāla pieeja no tīkla, izmantojot *http*, *ftp*, *rsync*, *ssh* protokolus.	• Iespējamas divu pēc satura dažādu revīziju kontrolsummas sakritības. • Nav iespējams izsekot atsevišķu failu izmaiņām, bet tikai visa repozitorija izmaiņām. Dažreiz tas ir neērti. • Repozitorija veidošanas process ir laikietilpīgs.
Mercurial	• Ātra datu apstrāde pat lielu repozitoriju gadījumā. • Nav atkarīgs no platformas. • Labi attīstīta iespēja atrādāt ar dažādiem izstrādes	• Orientēts uz darbu konsolē. • Iespējamas divu pēc satura dažādu revīziju kontrolsummas sakritības.

	zariem, sapludināšanas mehānisms ir labi automatizēts. • Komandas ir viegli lietojamas un intuitīvi saprotamas. • Iespēja konvertēt repozitorijus uz *Mercurial* no citām sistēmām, tādām kā *CVS*, *Subversion*, *Git*.	

Apkopojot informāciju par populārākam versiju kontroles sistēmām, tika izpētītas tendences, kas raksturo versiju kontroles mūsdienīgas attīstības tendences. Analizējot avotus [KAP 2008, ALT 2008, TAR 2011, GHE 2012, HUA 2009, LAV 2011, ROS 2010, RAZ 2007, SIY 2008, BRA 2008, MUR 2008, HAT 2012, LI 2012, THA 2009], nācās secināt, ka mūsdienīgas attīstības tendences ir orientētas vairāk uz esošo sistēmu attīstību, drošības uzlabošanu un esošo algoritmu attīstību versiju kontrolē. Salīdzinājumā ar laiku, kad tika radītas jau minētas versiju kontroles sistēmas, tagad daudz mazāk uzmanības pievērš pilnīgi jaunu sistēmu izstrādei. Turpinājumā tiks sniegts neliels apkopojums izgudrojumiem versiju kontroles jomā, kas parāda mūsdienu versiju kontroles attīstības tendences.

Ņemot vērā faktu, ka mūsdienu projektos var būt ļoti apjomīgi izejas koda repozitoriji, ir virkne pētījumu, kas uzlabo repozitoriju drošību, pasargājot to no datu zudumiem. Kā piemēru var minēt darbu [KAP 2008], kurā ir aprakstīta modernizēta versiju kontroles sistēma uz *Git* bāzes. Sistēma izmanto replikācijas principus. Repozitorijs atrodas uz vairākiem serveriem, kas atrodas vienā klasterī. Starp serveriem nepārtraukti notiek repozitorija objektu replikācija. Tādējādi viena servera bojājumu gadījumā repozitorija faili nezūd, bet kļūst pieejami no cita servera. Šāda pieeja prasa ievērojami vairāk resursu, taču palielina repozitorija datu drošību. Ņemot vērā, ka datu zudumi no versijas kontroles sistēmas var radīt milzīgus zaudējumus projektam, dažreiz ir vērts ieguldīt vairāk resursu, lai paaugstinātu drošību [KAP 2008].

Versiju kontroles attīstības tendences ietekmējas arī no mūsdienīgām programmatūras izstrādes metodēm [ALT 2008]. Viena no tādām metodēm ir modeļvadāma programmatūras izstrāde, kur darbs pārsvarā notiek nevis ar izejas kodu, bet ar modeļiem. Šajā darbā aprakstītas versiju kontroles sistēmas ir orientētas uz izejas koda pārvaldību un

versiju kontroli, bet nav piemērotas modeļu versiju kontrolei. Viens no šīs problēmas risinājumiem tiek piedāvāts avotā [ALT 2008]. Pieejas pamatā ir algoritms, kas veido meta-modeli. Meta-modelis apraksta sintaksi modelim, kas varētu būt pakļauta versiju kontrolei. Metode ir paredzēta versiju kontroles sistēmām, kas kontrolē modeļus nevis izejas kodu. Pateicoties meta-modelim, ir iespēja ne tikai pārvaldīt modeļu izmaiņas, bet arī veikt modeļu dažādu versiju sapludināšanu un redzēt konfliktus līdzīgi, kā izejas koda failu gadījumā [ALT 2008]. Šobrīd trūkst rīku, kas ļautu izmantot metodoloģiju praksē. Lai gan publikācijā ir piedāvāts algoritms, ir vajadzīgs laiks un atbilstoši pētījumi, kuru rezultātā varētu parādīties rīks, kas ļautu piedāvāto metodoloģiju izmantot modeļvadāmajā izstrādē.

Mūsdienās tiek pētīta arī paralēlās izstrādes problēma. Lai gan mūsdienīgas versiju kontroles sistēmas piedāvā tehnikas un metodes paralēlai izstrādei un izmaiņu sinhronizācijai, aktīvi tiek meklētas iespējas automatizēt un padarīt uzskatāmu un saprotamu paralēlo izstrādi. Kā piemērus var minēt darbus [TAR 2011] un [GHE 2012]. Standarta iespējas versiju kontroles sistēmās ļauj uzturēt izejas koda paralēlus zarus un sinhronizēt izmaiņas starp zariem. Problēma izpaužas faktā, ka, pētot konkrētas izmaiņas, lielākoties var redzēt tikai to, no kura zara tās radušās. Taču reizēm ir nepieciešamība izprast koda attīstību ilgākā laika periodā. Minēta problēma īpaši aktuāla ir lielajos projektos, kur ir daudz paralēlu zaru un sarežģīta zaru integrācijas sistēma. Avots [TAR 2011] piedāvā algoritmu, kas analizē koda attīstību visos repozitorija zaros. Rezultātā algoritms parāda ne tikai to, no kā radās izmaiņas, bet arī visu attīstības vēsturi visos zaros. Līdz ar to rodas labāks priekšstats par koda vēsturi, kas ļauj iegūt kvalitatīvākas metrikas par produkta attīstību, un tas ir būtiski, risinot konfigurācijas pārvaldības uzdevumus. Informācija, ko ļauj iegūt [TAR 2011] algoritms, palīdz izstrādātājiem, balstoties uz koda attīstības vēsturi, labāk saplānot izmaiņas, kas būs jāveic nākotnē, kā arī ļauj projekta iesācējiem izprast zaru savstarpējas integrācijas procedūru, kas ir viens no galvenajiem priekšnosacījumiem kvalitatīvu izmaiņu veikšanai [AIE 2010, TAR 2011]. Risinājuma priekšrocība ir tā, ka to var lietot gan kopā ar versiju kontroles sistēmu, gan kādam repozitorijam neatkarīgi no versiju kontroles sistēmas, un tas ir ērti, piemēram, gadījumos, kad versiju kontroles izejas kods nav pieejams.

Diezgan līdzīga ir pieeja [GHE 2012]. Publikācijā tiek uzsvērts, cik svarīgi ir iegūt informāciju par izejas koda repozitorija attīstības vēsturi paralēlu zaru kontekstā. Pētījuma rezultātā tiek izstrādāts algoritms un bibliotēka, kuru var pieslēgt versiju kontroles sistēmai un analizēt izejas koda attīstību, iegūstot vairākus raksturlielumus, ko nepiedāvā tradicionālās versiju kontroles sistēmu iespējas.

Apkopojot informāciju par mūsdienīgām attīstības tendencēm versiju kontrolē, nācās secināt, ka pētījumus var iedalīt divos virzienos. Pirmā virziena pētījumi orientēti uz repozitoriju un to zaru detalizētu analīzi, kuras mērķis ir sniegt informāciju par produkta attīstību un palīdzēt racionāli saplānot izmaiņas nākotnē. Šā virziena pētījumi ir orientēti arī uz paralēlas izstrādes uzlabošanas iespējām. Tiek atzīmēts, ka standarta versiju kontroles sistēmu funkcionalitāte ne vienmēr sniedz pietiekamu tehnisku atbalstu, lai varētu kvalitatīvi integrēt izmaiņas starp dažādiem repozitorija zariem. Tāpēc tiek piedāvāti algoritmi un rīki, kas vai nu uzlabo esošās versiju kontroles sistēmas iespējas, vai arī piedāvā neatkarīgus rīkus, kas veic izejas koda izmaiņu analīzi, palīdzot pieņemt racionālus lēmumus un iegūt metrikas, kas raksturo produkta attīstību [KAP 2008, TAR 2011, GHE 2012, LAV 2011, RAZ 2007, SIY 2008, BRA 2008, SAT 2011, HAT 2012, LI 2012, THA 2009, GUO 2005].

Otrais pētījuma virziens, ko izdevās identificēt, pētot versiju kontroli, ir saistīts ar sekošanu mūsdienīgām un inovatīvām programmatūras izstrādes pieejām, proti, modeļvadāmai izstrādei. Pētījumos galvenokārt tiek raksturota problēma, kas piemīt klasiskajām versiju kontroles sistēmām. Problēma saistīta ar to, ka versiju kontroles sistēmas ir orientētas uz darbu ar izejas kodu nevis ar modeļiem. Tāpēc tiek piedāvātas metodoloģijas un tehnikas, kas spēj pārvaldīt izmaiņas modeļos. Ir arī mēģinājumi radīt versiju kontroles sistēmas, kas sākotnēji ir domātas darbam ar modeļiem, to versiju pārvaldībai un modeļu dažādu versiju savstarpējai integrācijai [ATL 2008, HUA 2009, ROS 2010, MUR 2008, RUA 2003, EST 2013].

Paralēli tam, ka versiju kontroles sistēmas tiek uzlabotas un pielāgotas modeļvadāmajai programmatūras izstrādes pieejai, pastāv pētnieku un tehnisko speciālistu grupa, kas apgalvo, ka versiju kontroles panākuma atslēga slēpjas nevis rīku un metodoloģiju izvēlē, bet versiju kontroles principu zināšanās [AIE 2010, BER 2003, MET 2002, KAN 2005, CON 2002, УДО 2011]. Citiem vārdiem sakot, speciālisti uzsver, ka sākumā ir jāapgūst versiju kontroles vispārīgos principus, nozares labāko praksi un tikai pēc tam var plānot un organizēt versiju kontroli un paralēlu izstrādi projektā. Kad ir zināms, kā tiks veikta versiju kontrole un kas ir vajadzīgs, var atrast arī piemērotus rīkus konkrētu praktisku uzdevumu veikšanai [AIE 2010]. Nozares vadošie speciālisti [AIE 2010, BER 2003, MET 2002, KAN 2005] uzsver vienu no būtiskākajam konfigurācijas pārvaldības problēmām, ka praksē bieži vien sākumā tiek izvēlēts rīks vai metodoloģija un vēlāk versiju kontroles process tiek pielāgots konkrētam rīkam nevis projekta vajadzībām. Ir svarīgi apzināties, ka jebkurš projekts nepārtraukti mainās un attīstās, līdz ar to vajadzētu attīstīties arī versiju kontroles procesam [CON 2002, УДО 2011]. Neviens rīks vai algoritms nepalīdzēs korekti atrisināt

versiju kontroles uzdevumu, ja konfigurācijas pārvaldības speciālistam nav pamatzināšanu par šo procesu. Ir jāzina, kas ir versija, kas ir bāzes līnija, kā veidot izejas koda jaunu zaru, kurā brīdī izmaiņas var integrēt ar pamata zaru, kādas metrikas ir nepieciešams uzturēt utt. Būtisks faktors ir arī konfigurācijas elementu identifikācija, proti, cik korekti tā tika veikta [AIE 2010].

Brīdī, kad versiju kontroles uzdevums ir atrisināts, var ķerties klāt pie nākamā konfigurācijas pārvaldības uzdevuma – produkta būvējums un instalācija. Nākamajā apakšnodaļā tiks sniegts apkopojums tehnoloģijām un metodēm, kas ļauj sakomplektēt instalējamu produktu no izejas koda un uzinstalēt to.

1.5. Produkta būvējumi un instalācija

Apakšnodaļas mērķis ir definēt produkta būvējuma un instalācijas procesu, apkopot rekomendācijas, ko sniedz nozares speciālisti, noskaidrot, kādus rīkus un metodes lieto būvējumu un instalācijas realizācijai. Galvenais uzsvars šajā apakšnodaļā ir likts uz mūsdienu attīstības tendencēm, risinot produkta būvējuma un instalācijas uzdevumu. Lai turpinātu pētījumu, ir jānoskaidro, kurā virzienā attīstās rīki un metodes. Vēl viens apakšnodaļas uzdevums ir noteikt būvējumu un instalācijas disciplīnas vietu konfigurācijas pārvaldības kopējā procesā un arī mūsdienu galvenās problēmas, kas pastāv nozarē, būvējot un instalējot produktus.

Būvējuma un instalācijas procesa galvenais uzdevums ir no izejas koda failiem iegūt bināru failu vai failus, kurus būtu iespējams uzinstalēt vai integrēt kādā no vidēm un iegūt strādājošu produktu. Galvenās prasības ir procesa ātrums un iespēja prognozēt rezultātu [AIE 2010, BER 2003, MET 2002]. Būvējuma un instalācijas procesam ir vairāki nosacījumi, kas ir nepieciešami, lai procesu varētu uzskatīt par veiksmīgu. Izpētot vairākus literatūras avotus [AIE 2010, BER 2003, MET 2002, KAN 2005, CON 2002], nācās secināt, ka galvenie būvējuma un instalācijas principi ir formulēti atšķirīgi, bet kopumā nozares speciālistu vidū viedoklis par labu būvējumu un instalācijas procesu sakrīt. Autors uzskata, ka viens no veiksmīgākajiem apkopojumiem ir avotā [AIE 2010]. Pirmkārt, tas ir viens no jaunākajiem darbiem, otrkārt, darba autoram ir liela praktiskā pieredze konfigurācijas pārvaldībā, turklāt [AIE 2010] autors pats ir apkopojis vairākus avotus par produkta būvējumiem un instalācijām. Tāpēc tieši no šā literatūras avota tiek paņemts apkopojums priekšnosacījumiem, kas nodrošina veiksmīgu būvējumu un instalāciju produktam:

- būvējuma process ir saprotams un atkārtoti izpildāms;

- process ir ātrs un drošs;
- jebkura konfigurācijas vienība, kas ir nepieciešama būvējuma procesā, ir identificējama;
- atkarības izejas kodā un kompilācijas procesā ir viegli identificējamas;
- izejas kods tiek kompilēts vienu reizi un to pēc tam var uzinstalēt jebkurā vidē;
- kļūdas un neparedzētas situācijas būvējuma un instalācijas procesā tiek identificētas un pārvaldāmas iepriekš saskaņotā veidā;
- nekorekts būvējums tiek savlaicīgi identificēts un savlaicīgi tiek likvidēti cēloņi un sekas.

Runājot par būvējumu procesu, speciālisti [AIE 2010, BER 2003, MET 2002] un arī citi pētnieki konfigurācijas pārvaldības jomā atzīst, ka, neskatoties un to, ka būvējumu un instalācijas procesu mūsdienās atbalsta vairāki gatavi rīki, galvenais panākuma faktors tomēr ir sekot labākajām praksēm, kas ir apliecinājušās sevi ne vienu vien reizi. Tāpēc, pirms apkopot mūsdienu rīkus, ko izmanto būvējumu un instalācijas procesā, tiks apkopotas speciālistu [AIE 2010, BER 2003, MET 2002] rekomendācijas, ko vajadzētu ievērot, lai neatkarīgi no izvēlētiem rīkiem, būvējuma un instalācijas process būtu veiksmīgs. Galvenās rekomendācijas ir šādas:

- katram uzbūvētam un uzinstalētam produktam ir jābūt unikālai versijai. Strādājot ar produktu, ir jābūt vietai, kur var redzēt produkta versiju. Versijas identifikatoram jāsniedz informācija par to, kādā veidā tapa konkrētais produkts, proti, kādas konfigurācijas vienības un ar kādām versijām tika ņemtas no izejas koda repozitorija. Versijas identifikatoram obligāti ir jābūt absolūti unikālam, proti, nedrīkst pastāvēt teorētiska iespēja, ka identifikatori kādreiz atkārtosies vai arī sniegs neviennozīmīgu informāciju;
- vienmēr jābūt iespējai pateikt, kura izejas koda faila versija atrodas kurā vidē. Proti, ja ir izejas koda fails *A*, kuram ir versijas 1, 2, 3 un 4, tad jebkurā momentā jāprot viennozīmīgi pateikt, kurā vidē atrodas katra faila versija. Piemēram, testa vidē var būt versija 3, savukārt produkcijas jeb ekspluatācijas vidē var būt versija 1;
- būvējuma un instalācijas procesa rīkiem vai skriptiem jānodrošina vides sagatavošana kompilācijas un instalācijas procesam. Ja ir nepieciešami vides mainīgie vai noteikta veida konfigurācija instalācijas brīdī, tad rīkiem vai skriptiem tas ir jānodrošina;

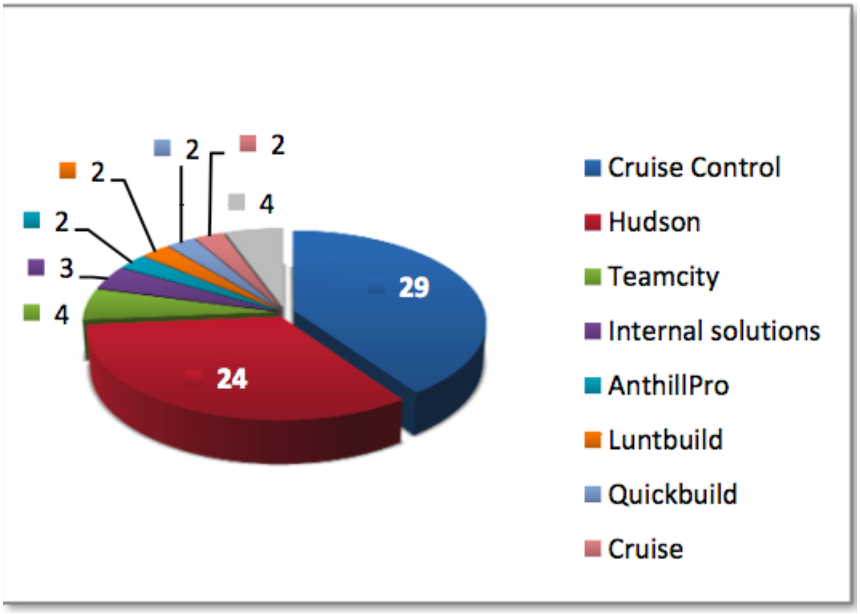
- būvējuma un instalācijas procesu nedrīkst veikt no lokālas mašīnas. Ir jāizmanto speciāli serveri, ko sauc par nepārtrauktās integrācijas serveriem, būvējumu serveriem vai dažreiz par konfigurācijas pārvaldības ietvariem (angļu val. *framework*). Izejas kods vienmēr jāņem tikai no versiju kontroles sistēmas nevis no lokālas mašīnas, kur izejas kodā var būt lokālas izmaiņas, kas vēl nav reģistrētas versiju kontroles sistēmā;
- būvējumi ir savā starpā neatkarīgi.

Lai varētu īstenot būvējuma un instalācijas procesu, mūsdienās ir daudz rīku, kas automātiski veic minēto uzdevumu, dokumentē procesu un fiksē rezultātu [AIE 2010, ALT 2010]. Šie rīki ir kompilācijas un būvējuma rīki un nepārtrauktās integrācijas serveri. Kompilācijas rīki spēj no izejas koda failiem izveidot izpildāmu bināru failu, kuru savukārt nepārtrauktās integrācijas serveris prot uzinstalēt kādā no vidēm un iegūt strādājošu produktu. Nepārtrauktās integrācijas serveriem ir vairākas funkcijas. Tie spēj pieslēgties izejas koda repozitorijiem, kas ir pakļauti kādai no versiju kontroles sistēmām, iegūt no turienes noteiktu failu noteiktas versijas, pēc tam iedarbina kādu kompilācijas rīku, lai izveidotu izpildāmo bināru failu. Kad binārais fails no izejas koda ir izveidots, nepārtrauktās integrācijas serveris sagatavo atbilstošu vidi un veic instalāciju. Paralēli nepārtrauktās integrācijas serveris dokumentē katrā soļa rezultātus un parāda tos pārskatāma veidā. Mūsdienu nepārtrauktās integrācijas serveriem ir diezgan attīstītas lietotāja grafiskās saskarnes, kas ļauj viegli konfigurēt rīku konkrēta projekta vajadzībām [AIE 2010, ALT 2010, ЛАП 2004]. Tabulā 1.3. ir uzskaitīti patlaban populārākie risinājumi nepārtrauktās integrācijas serveriem. Kā redzams, ir pieejami gan atvērta pirmkoda risinājumi, gan arī komerciālie. Savukārt attēlā 1.8. var redzēt nepārtrauktās integrācijas serveru popularitāti 2010. gadā, balstoties uz [ALT 2010] pētījumu. Kā redzams, populārākie nepārtrauktās integrācijas serveri ir *CruiseControl* un *Hudson*. Pārējiem risinājumiem ir atvēlēta salīdzinoši neliela daļa izpētīto projektu. Ar šo piemēru darba autors nekādā gadījumā nevēlas pateikt, ka minēti risinājumi ir labāki par citiem, jo, balstoties gan uz savu pieredzi, gan uz pētījumiem [AIE 2010, MET 2002], nācās secināt, ka panākuma galvenais faktors ir procesā nevis rīka izvēlē. 1.8. attēls ir dots tikai vispārīgam priekšstatam par nepārtrauktas integrācijas serveru izmantojamību.

1.3. tabula

Nepārtrauktās integrācijas serveri

Nosaukums	Ražotājs/ Projekts	Ir pieejams pirmkods	Izlaišanas datums	Avots tīmeklī
AnthillPro	*Urbancode*	–	2001	*http://www.anthillpro.com*
Bamboo	*Atlassian*	–	2007	*http://www.atlassian.com*
Continuum	*Apache project*	jā	2005	*http://continuum.apache.org*
Cruise	*ThoughtWorks*	–	2008	*http://studios.thoughtworks.com/cruise*
CruiseControl	*Sourceforge project*	jā	2001	*http://cruisecontrol.sourceforge.net*
FinalBuilder	*VSoft Technologies*	–	2001	*http://www.FinalBuilder.com*
Hudson	*java.net project*	jā	2007	*http://hudson.dev.java.net*
Lunt build	*Javaforge project*	jā	2004	*http://luntbuild.javaforge.com*
Parabuild	*Viewtier*	–	2005	*http://www.viewtier.com*
Pulse	*Zutubi*	jā	2006	*http://www.zutubi.com*
Quick build	*PMEase*	jā	2004	*http://www.pmease.com*
TeamCity	*JetBrains*	jā	2006	*http://www.jetbrains.com/teamcity*



1.8. att. Nepārtrauktās integrācijas serveru izmantošana

Kā jau tika minēts, nepārtrauktās integrācijas serveris savas darbības laikā izsauc kompilācijas jeb būvējuma rīku, kas no izejas koda failiem veido bināru izpildāmu failu, kuru iespējams uzinstalēt kādā no vidēm. Tabulā 1.4. ir dots apkopojums būvējuma rīkiem 2010. gadā [ALT 2010].

1.4. tabula

Būvējumu veidošanas rīki

Rīks	Platforma	Apraksts
Ant	*Java*	Viens no populārākiem rīkiem *JAVA* tehnoloģijas projektiem. Bāzēts uz *XML*, skriptus var laist no komandrindas un no integrētas izstrādes vides. *http://ant.apache.org/*
Maven 1	*Java*	Pārvalda *JAVA* projekta būvējumu, dokumentē rezultātus, pārvalda *atkarības no citiem projektiem.* *http://maven.apache.org/maven-1.x/*
Maven 2	*Java*	Pēc būtības paplašināts *Maven1*: pieliktas papildu iespējas, uzlabots atkarību pārvaldības mehānisms, var pieslēgt citus papildinājumus (angļu val. *plug-in*). *http://maven.apache.org/*
Gant	*Groovy/Java*	Alternatīvs rīks *Ant* skriptiem. Skripti, kas būvē projektu, tiek

		rakstīti, izmantojot *Groovy* programmēšanas valodu. *http://gant.codehaus.org/*
Nant	*.NET*	*.NET* platformas būvējumu rīks. Teorijā tas ir līdzīgs *Make* rīkam, tikai tiek novērstas rīka *Make* galvenās nepilnības. Praksē rīks ir ļoti līdzīgs *Ant* rīkam, jo skripta sintaksē arī tiek lietots *XML*. *http://nant.sourceforge.net/*
setup.py	*Python*	Standarta rīks *Pyton* projektiem. *http://pypi.python.org/pypi/setuptools*
Pip	*Python*	Modernizēts būvējumu risinājums *Pyton* projektiem, kurā ir iekļautas arī versiju pārvaldības funkcijas. *http://pip.openplans.org/*
virtualenv	*Python*	*Pyton* projektu virtualizēta būvējumu vide, kas ļauj uzturēt vienlaikus vairākus projektus un veikt neatkarīgus būvējumus. *http://pypi.python.org/pypi/virtualenv*

Neskatoties uz to, ka mūsdienās ir ļoti daudz rīku, kas sniedz tehnisku atbalstu un automatizācijas iespējas būvējuma un instalācijas procesā, joprojām pastāv problēmas. Turklāt, kā minēts avotos [AIE 2010, BER 2003, MET 2002, ALT 2010, ЛАП 2004, УДО 2011], lielākā daļa no problēmām atkārtojas atkal un atkal, parādoties jaunajam projektam. Galvenās problēmas ir šādas:

- izejas koda pārvaldības un versiju kontroles process ir nepilnīgs vai nekorekts. Katru reizi, veidojot produkta būvējumu, tiek paņemts nekvalitatīvs izejas kods, un būvējumam nav iespējams noteikt izejas koda bāzes līniju;
- būvējumu sistēmu vai skriptus veido viens cilvēks, kuram ir subjektīvs priekšstats par sistēmu un produktu kopumā. Rezultātā būvējumu un instalācijas sistēma ir saprotama tikai vienam cilvēkam, un tad, kad cilvēks pamet projektu, būvējumu sistēmu kļūst ļoti grūti vai neiespējami uzturēt, tā rezultātā visu laiku tiek risinātas vienas un tās pašas problēmas bez iespējas novērst cēloni;
- būvējumu sistēma ir pārāk sarežģīta, un brīdī, kad situācija projektā mainās, to ir grūti vai neiespējami pielāgot jaunajām prasībām. Rezultātā ir nepieciešamība izstrādāt jaunu būvējumu sistēmu, bet šādām aktivitātēm bieži nav resursu. Līdz ar to

regulāri jārisina līdzīgas problēmas ar būvējumu un instalāciju, kas izpaužas nokavētajos termiņos, nestrādājošā produktā utt.;

- būvējumu un instalācijas sistēmu sāk izstrādāt ar rīku ieviešanu nevis ar procesa un produkta apzināšanu. Speciālists, kas ievieš būvējumu sistēmu, mēģina pielāgoties kādam rīkam, ko viņš izvēlas kā palīgu, nevis meklē rīkus noteikta procesa vajadzībām. Tomēr speciālisti [AIE 2010, BER 2003, MET 2002, ALT 2010, ЛИП 2004, УДО 2011] rekomendē sākumā apzināties konfigurācijas pārvaldības procesu kopumā, produkta arhitektūru un būvējuma procesu un tikai pēc tam meklēt rīkus, kas palīdzētu procesu automatizēt.

Risinot problēmas būvējumu un instalācijas procesā, mūsdienu pētnieki izmanto dažādas pieejas. Izpētot avotus [CLE 2012, SIN 2008, BUS 2011, COM 2011, MAL 2012, JOH 2011, SHI 2010], nācās secināt, ka ir pētījumi, kas ir orientēti uz esošo rīku uzlabošanu, citi savukārt cenšas uzlabot metodoloģijas un pieejas, neiedziļinoties konkrētajos rīkos, un, visbeidzot, ir pētnieki, kas cenšas uzlabot būvējumu un instalācijas procesu, iegūstot pēc iespējas vairāk informācijas no citiem konfigurācijas pārvaldības procesiem, apskatot būvējumu kā neatņemamu sastāvdaļu no kopējā konfigurācijas pārvaldības procesa. Šajā nodaļā tiks dots neliels ieskats dažos pētījumos.

Viens no esošo rīku uzlabojumu piemēriem ir pētījums [CLE 2012]. Šajā publikācijā var iepazīties ar *CMAKE* rīka esošajām problēmām, ar ko praksē saskarās konfigurācijas pārvaldnieki. Tiek piedāvāta *CMAKE* rīka uzlabošanas un konfigurēšanas pieeja, kas pēc autoru domām novērš rakstā minētas problēmas. Promocijas darba autors [CLE 2012] pētījumā nesaskatīja *CMAKE* rīka lietošanas rekomendācijas, kas palīdzētu iegūt priekšstatu, kādās situācijas varētu lietot šo rīku. Diezgan maz bija minēts arī par vispārīgiem būvēšanas un instalācijas principiem. Konfigurācijas pieeja der tikai vienam konkrētam rīkam. Mūsdienu pētnieki [AIE 2010, BER 2003, MET 2002, ALT 2010, ЛИП 2004, УДО 2011] atzīmē līdzīgas problēmas arī citos pētījumos, kur pārāk liela uzmanība ir pievērsta konkrētam rīkam, tā uzlabojumiem, taču diezgan maz ir analizēti vispārīgie principi un prakse. Salīdzinoši maz uzmanības tiek pievērsts arī būvējumu un instalācijas procesa sadarbībai ar pārējiem konfigurācijas pārvaldības procesiem, piemēram, versiju kontrolei.

Pretstatā pētījumam [CLE 2012] publikācijā [SIN 2008] ir runāts nevis par konkrētiem rīkiem, bet gan par abstraktu modeli, kas apraksta būvējumu un instalācijas procesu. Publikācija [SIN 2008] piedāvā modeli, kas apraksta aktivitātes, kuras ir nepieciešams realizēt kvalitatīvam būvējuma un instalācijas procesam. Katrai aktivitātei ir sniegts apkopojums rīkiem, ko izmanto implementācijai un automatizācijai. Jāatzīmē, ka

piedāvātais modelis galvenokārt ir orientēts uz uzturēšanas tipa projektiem, kad produkta pirmā versija ir uzinstalēta un ir nepieciešams uzlikt jaunākas versijas ar labojumiem, ko dažreiz sauc par ielāpiem. Ir sniegtas arī rekomendācijas par rīkiem, kurus varētu izmantot realizējot kādu no modeļa aktivitātēm. Promocijas darba autors [SIN 2008] publikācijā neatrada rekomendācijas, kā nodrošināt saiti starp modeli un izmaiņām projektā. Piemēram, ja projektā mainās uzturēšanas pieejas, rīki vai tehnoloģijas, kā varētu veikt izmaiņas būvējumu un instalācijas modelī, lai modelis turpinātu atbilst projekta aktuālam stāvoklim.

Apkopojot pētījumus par būvējumiem un instalācijām, nācās secināt, ka dažos pētījumos tiek lietotas arī modelēšanas pieejas. Viens no piemēriem ir avots [BUS 2011]. Šī publikācija piedāvā uz modelēšanas balstītu metodoloģiju, kas nosaka konfigurācijas vienumu kopu, kas jāiekļauj būvējumā. Metodoloģija izmanto heiristiku un analizē visas iespējamas konfigurācijas vienumu kopas, kas potenciāli varētu būt iekļautas būvējumā. Tiek izvēlēts variants, kas sniedz iespējami labāko produkta veiktspēju. Promocijas darba autors uzskata, ka būtiskākais trūkums šim pētījumam ir gatavu rīku trūkums, kas nodrošinātu heiristiskā algoritma realizāciju kādam reālam projektam. Sīkāk izpētot publikāciju [BUS 2011], nācās secināt, ka metodoloģija sastāv no apjomīgiem aprēķiniem. Ja šo metodoloģiju gribētu izmēģināt praksē kāda projekta komanda, speciālisti varētu apšaubīt rezultātus, tāpēc, ka to iegūšanas ceļš nebūtu pietiekami skaidrs. Kā jau tika minēts šajā darbā un arī pētījumos, kas tika apskatīti, ir svarīgi, lai jaunā metodoloģija, jauns rīks vai algoritms būtu saprotams speciālistiem, kas to lietos praksē. Ja jaunā pieeja nebūs pietiekami skaidra, rezultātiem neuzticēsies. Avotā [BUS 2011] formulām nav pietiekami detalizēta apraksta, un tas var radīt grūtības līdz galam izprast metodoloģiju un pašu modelēšanas gaitu.

Modelēšanas tehnoloģijas izmanto ne tikai, lai korekti izvēlētos būvējumu saturu, bet arī lai noteiktu iespējamos riskus jeb vietas būvējuma procesā, kur potenciāli var rasties problēmas. Viens no piemēriem ir pētījums [COM 2011]. Šajā pētījumā galvenokārt ir runa par interneta mājaslapu izstrādes projektiem. Galvenā problēma, kas uzsvērta šajā avotā, ir tā, ka mājaslapas darbības pārtraukumi var radīt nopietnas neērtības lietotājiem un zaudējumus mājaslapas īpašniekiem. Līdz ar to, plānojot būvējumus un instalācijas, it īpaši situācijās, kad ir jāuzinstalē mājaslapas jauna versija, ir jāprot paredzēt visus iespējamus riskus. Metodoloģija, kas ir aprakstīta [COM 2011] pētījumā, piedāvā heiristisku algoritmu, kas spēj noteikt riskus būvējuma un instalācijas procesā, tādējādi vēl pirms būvējuma ir iespējams veikt pasākumus, kas potenciāli var novērst problēmas būvējuma un instalācijas laikā.

Cenšoties uzlabot būvējumu un instalācijas procesu, daži pētnieki pievērš pastiprinātu uzmanību ne tik pašam būvējuma procesam vai rīkam, bet produkta tehniskajai

arhitektūrai. Kā piemēru varētu minēt [MAL 2012, JOH 2011]. Šajos pētījumos arī tiek izmantotas modelēšanas tehnoloģijas, taču tiek apskatīts nevis būvējuma process, bet tiek analizēta tehniskā arhitektūra, kas varētu sniegt maksimāli labu kvalitāti produktam no būvējumu skata punkta. Tehniskā arhitektūra ir atkarīga no prasībām pret produktu. Lietojot metodoloģijas [MAL 2012, JOH 2011], svarīgi zināt rīcības plānu, kā rīkoties, ja mainās produkta arhitektūra.

Runājot par būvējumu tehnoloģiju un rīku attīstības tendencēm, jāsaka, ka ir arī pētījumi, kas koncentrējas nevis uz tehnoloģijām vai pieejām, bet analizē faktorus, kas varētu ietekmēt būvējumu un arī izstrādes procesu kopumā. Kā piemēru var minēt pētījumu [SHI 2010], kurā tiek izpētītas saites starp uzņēmumu iekšēju organizatorisku struktūru, kultūru un produkta būvējumu procesu. Citiem vārdiem sakot, pētījumā parādīts, kā uzņēmuma kultūra un organizācija var ietekmēt būvējuma procesu un kā šai ietekmei izsekot. Promocijas darba autors pievērsa uzmanību publikācijai [SHI 2010] arī tāpēc, ka tā parāda, ka kvalitatīvam būvējumu un instalācijas procesam ir svarīgi ne tikai izvēlēties rīkus un konstatēt atkarības no citiem konfigurācijas pārvaldības apakšprocesiem, bet arī noteikt ārējus faktorus, kas pastāv projektā, bet nav atkarīgi tieši no konfigurācijas pārvaldības. Jāpiebilst, ka uzņēmumu specifika ietekmē ne tikai konfigurācijas pārvaldības procesus, bet arī visu izstrādi kopumā [SHI 2010].

Apkopojot informāciju par būvējumu un instalācijas procesu, nācās atzīt, ka secinājumi ir līdzīgi, kā apskatot citus konfigurācijas pārvaldības uzdevumus. Sākumā ir nepieciešams definēt būvējumu un instalācijas procesu, faktorus, kas šo procesu ietekmē, un tikai tad izvēlēties piemērotus rīkus, kas varētu realizēt procesu tehniski. Paralēli jānodrošina, lai process būtu saprotams izstrādātāju komandai, lai nebūtu situācijas, kad būvējumu process ir pārāk sarežģīts un to saprot tikai viens cilvēks.

1.6. Mūsdienīgas programmatūras konfigurācijas pārvaldības iezīmes

Ja runa ir par mūsdienīgu konfigurācijas pārvaldību, ir svarīgi, lai tā atbilstu mūsdienīgam tendencēm programmatūras izstrādes nozarē [BIL 2014, TRE 2014, WHA 2014].

Ņemot vērā iepriekšējo apakšnodaļu informāciju, konfigurācijas pārvaldība sniedz atbildes uz šādiem jautājumiem:

- kā identificēt produkta vienumus un nodrošināt to versiju pārvaldību;

- kā uzbūvēt un uzinstalēt produktu no atbilstošiem produkta vienumiem.

Apkopojot mūsdienīgas pieejas versiju kontroles organizēšanā un izejas koda pārvaldībā, paralēli tika izpētīta programmatūras izstrādes metodoloģiju attīstības vēsture [JIA 2009] un versiju kontroles sistēmu attīstības vēsture [HIST 2014]. Analizējot trīs paaudzes versiju kontroles sistēmās [HIST 2014] un programmatūras izstrādes metodoloģiju attīstību [JIA 2009], var secināt, ka versiju kontroles sistēmas seko līdzi programmatūras izstrādes metodoloģiju jauninājumiem. No 1980. līdz 1989. gadam paradījās trešās un ceturtās paaudzes programmēšanas valodas [JIA 2009] un paralēli parādījās otrās paaudzes versiju kontroles sistēmas, kas strādā ar vairākiem izejas koda failiem (*CVS, Subversion*) [HIST 2014]. Sākot no 2000. gada, strauji sāka attīstīties *Agile* metodoloģijas programmatūras izstrādē [JIA 2009] un paralēli parādījās arī trešās paaudzes distributīvas versiju kontroles sistēmas (*Git, Mercurial, Bazaar*), kas spēj pārvaldīt liela apjoma projektus, paralēlus izejas koda zarus un dažādas izstrādātāju komandas, kas ģeogrāfiski ir sadalītas [JIA 2009]. Analizējot modeļvadāmas izstrādes pieejas priekšrocības un attīstības tendences [AZO 2008], nācās secināt, ka nākotnē versiju kontroles sistēmās jāspēj pārvaldīt ne tikai izejas kodu, bet arī modeļus. Jau šobrīd ir pieejas, kas risina šo uzdevumu [ATL 2008, HUA 2009, ROS 2010, MUR 2008, RUA 2003, EST 2013]. Apkopojot speciālistu viedokli par versiju kontroles attīstības tendencēm [CMC 2014, AIE 2010], var secināt, ka nākotnē versiju kontroles sistēmām un pieejām jābūt šādām:

- jāsniedz iespēja analizēt izejas koda vēsturi, sniedzot pēc iespējas pilnīgāku informāciju par izejas koda evolūciju;
- jāsniedz iespēja strādāt ne tikai ar izejas koda failiem, bet arī ar modeļiem, vienlaikus realizējot ne tikai failu, bet arī modeļu sapludināšanas (angļu val. *merge*) funkciju;
- jābūt interfeisiem un bibliotēkām, kas ļautu paplašināt standarta funkcijas, neiejaucoties versiju kontroles sistēmas kodā.

Runājot par mūsdienīgām tendencēm produkta būvējumu un instalācijas risinājumos, jāatzīmē, ka lielu impulsu deva 2009. gads, kas tiek uzskatīts par *DevOps* kustības sākumu [AZO 2014, RAG 2014, LES 2014, DEV 2014]. *DevOps* ir programmatūras izstrādes metodoloģija, kas ir orientēta uz sadarbību starp izstrādātājiem un sistēmu administratoriem un pārvaldniekiem. Attīstoties *Agile* metodoloģijām un pieaugot nepieciešamībai piegādāt pasūtītājiem gatavu produktu pēc iespējas biežāk, tika konstatēts, ka būvējumu un instalācijas operācijas netiek līdzi straujajām izmaiņām programmatūras kodā [AZO 2014, RAG 2014,

LES 2014, DEV 2014]. Tāpēc jaunākas pieejas produktu būvējumiem un instalācijām tiecās realizēt šādu funkcionalitāti:

- uzlabot sadarbību starp izstrādātājiem un operāciju pārvaldniekiem (tajā skaitā – arī ar konfigurācijas pārvaldniekiem);
- izveidot jaudīgus risinājumus, kas maksimāli ātri spēj būvēt produktus no izejas koda un instalēt tos paralēli uz vairākām vidēm jeb infrastruktūrām;
- atdalīt procesa loģiku no tehniskās realizācijas konkrētiem rīkiem un platformām, izvairīties no statiskiem skriptiem.

Rakstā [RAG 2014] ir skaidri dotas iezīmes mūsdienīgiem risinājumiem produktu būvējumu un instalāciju procesā. Ir pagājis laiks, kad produktu vajadzēja instalēt statiskajā vidē ar fiksētu *IP* adresi reizi dienā vai reizi nedēļā. Attīstoties mākoņskaitļošanas tehnoloģijām, būvējuma un instalācijas process kļūst abstrakts no konkrētām tehnoloģijām un platformām, tāpēc jaunākajiem risinājumiem jābūt modeļvadāmiem [WET 2012, RAG 2014]. Statiski skripti un rīki vairs nespēj tikt galā ar vairākiem desmitiem vai pat simtiem produkta būvējumu dienā, tāpēc ir nepieciešamas jaunas modeļvadāmas pieejas, kas ļaus pietiekami ātri un automātiski izstrādāt risinājumu, kas būtu neatkarīgs no konkrētām platformām.

No vienas puses, vairāki avoti [AZO 2014, RAG 2014, LES 2014, DEV 2014] norāda uz to, ka nākotnē būvējumu un instalāciju risinājumi būs modeļvadāmie un atbilstošu procesu plānošana būs vizuāla un neatkarīga no konkrētas platformas vai tehnoloģijas. No otras puses, jebkuram modeļvadāmajam risinājumam ir vairāki līmeņi. Pats pēdējais līmenis ar zemāku abstrakciju vienalga skar konkrētas tehnoloģijas un platformas [DON 2011, OSI 2011]. Parasti uzņēmumiem jau eksistē rīki un skripti, kas risina būvējumu un instalācijas uzdevumus platformu līmenī. Ieviešot jaunu modeļvadāmo risinājumu, uzņēmumi negribēs mest ārā esošos skriptus vai rīkus, kuru izstrādei bija veltīts laiks un kuri ietver vairāku pabeigtu projektu pieredzi. Drīzāk uzņēmumus interesēs tādi modeļvadāmi risinājumi, kas ļaus izmantot esošos platforma līmeņa skriptus un rīkus, tikai uzlabojot to efektivitāti un iespēju tos lietot atkārtoti [DOD 2014].

Runājot par jaunākajām tendencēm produktu būvējumu un instalāciju procesā [AZO 2014, RAG 2014, LES 2014, DEV 2014], jāatzīmē fakts, ka bieži vien izejas koda pārvaldība un versiju kontrole tiek uztverta kā pašsaprotama lieta. Šis fakts var radīt problēmas, jo tikai labi organizēta versiju kontrole un izejas koda pārvaldība ļauj organizēt kvalitatīvu būvējumu un instalācijas procesu neatkarīgi no izvēlētiem tehniskiem līdzekļiem [AIE 2010, BER 2003, MET 2002, ALT 2010, ЛАП 2004, УДО 2011].

Papildus jau minētajam, jebkuram versiju kontroles, būvējuma vai instalācijas risinājumam jāatbilst vispārīgiem programmatūras izstrādes nozares kvalitātes standartiem [BAM 1995, ABO 2014, AIE 2010].

Konfigurācijas pārvaldības risinājumam, kas atbilst mūsdienīgam tendencēm un vajadzībām programmatūras izstrādes nozarē, jābūt šādām:

- kompleksi jārisina visi galvenie konfigurācijas pārvaldības uzdevumi, kas ir definēti šajā promocijas darba nodaļā: konfigurācijas vienumu identifikācija, versiju kontrole, izejas koda pārvaldība un paralēlas izstrādes nodrošinājums, produkta būvējuma un instalācijas process, metriku savākšana;
- modeļvadāmam; risinājumam jāfunkcionē arī gadījumā, kad infrastruktūras, kur jābūvē un jāinstalē produkts, atrodas mākoņos (angļu val. *Cloud*). Procesam jābūt abstraktam ar skaidri definētu ceļu, kā, samazinot abstrakcijas līmeni, nonākt līdz konkrētam rīkam vai skriptam;
- risinājums ļauj izmantot esošus rīkus un skriptus, uzlabojot iespēju tos jaunajos projektos lietot atkārtoti;
- versiju kontrolei jāprot strādāt ne tikai ar izejas koda failiem, bet arī ar modeļiem, lai to varētu lietot arī projektos ar modeļvadāmo izstrādes pieeju;
- risinājumam jāizpilda programmatūras izstrādes nozares kvalitātes standartu prasības.

1.7. Nodaļas kopsavilkums

Nodaļā definēta konfigurācijas pārvaldība. Tas ir procesu kopums, kas identificē izstrādājama produkta vienumus un kontrolē vienumu izmaiņas līdz pat brīdim, kad tie nonāk gala produktā. Lai šo procesu īstenotu, nepieciešams atrisināt šādus uzdevumus: konfigurācijas vienumu identifikācija, versiju kontrole, izejas koda pārvaldība, produkta būvējumi un instalācijas, metriku savākšana.

Kā rāda jaunākie pētījumi, kas tika apkopoti šajā nodaļā, konfigurācijas pārvaldību vajadzētu apskatīt kompleksi, nevis atsevišķi kādu konkrētu uzdevumu, piemēram, versiju kontroli, izejas koda pārvaldību vai būvējumu. Papildus tam konfigurācijas pārvaldības procesam jāatbilst mūsdienu attīstības tendencēm programmatūras izstrādes nozarē. Pētījuma gaitā izdevās identificēt piecas svarīgas iezīmes, kas raksturo mūsdienīgu konfigurācijas pārvaldības procesu: process risina kompleksi visus uzdevumus, process ir modeļvadāms, ir iespēja izmantot esošos rīkus un skriptus jaunajā modeļvadāmajā risinājumā, versiju kontrole

strādā ne tikai ar kodu, bet arī ar modeļiem, lai varētu atbalstīt projektus ar *MDD* (angļu val. *Model-Driven Development*) pieeju, un process nav pretrunā ar kvalitātes standartiem.

Pētot programmatūras izstrādes attīstības tendences un prasības pret mūsdienīgiem konfigurācijas pārvaldības automatizācijas risinājumiem, nācās secināt, ka risinājumiem jāizvairās no statikas. Proti, 20. gadsimtā, risinot konfigurācijas pārvaldības uzdevums, tika izstrādāti statiski rīki un skripti, kas bija orientēti uz konkrētu platformu, konkrētu tehnoloģiju un salīdzinoši nelielu produkta būvējumu skaitu. 21. gadsimtā, kad strauji pieaug projektu izejas koda failu skaits, atkarību skaits no citiem projektiem, attīstās *Agile* pieejas un prakses un projekta infrastruktūras atrodas mākoņos, statiski rīki un skripti vairs neder. Līdz ar to konfigurācijas pārvaldības automatizācijas risinājumiem jābūt modeļvadāmiem ar vairākiem abstrakcijas līmeņiem. Statiski rīki un skripti darbojas tikai zemākajā abstrakcijas līmenī. Ir svarīgi, lai minēti rīki un skripti būtu sadalīti neatkarīgās daļās. Katrai daļai ir jāsaņem parametri un jāizsniedz rezultāts vai kļūdas paziņojums. Šajā gadījumā risinājumus varēs izmantot atkārtoti. Svarīgi, lai modeļvadāmie automatizācijas risinājumi platformas un tehnoloģiju līmenī ļautu izmantot esošos rīkus un skriptus, kas uzņēmumiem jau ir izstrādāti. Tas ļaus ietaupīt laiku, automatizējot konfigurācijas pārvaldību jaunajos projektos. Ja esošos rīkus un skriptus jaunajos projektos izmantot nevarēs, risinājums nebūs uzticams, un konfigurācijas pārvaldniekiem to būs grūti pieņemt.

Nākamajā promocijas darba nodaļā tiks analizēti konfigurācijas pārvaldības automatizācijas risinājumi, kas atbilst modeļvadāmas arhitektūras formātam. Katrai pieejai tiks vērtēti šādi kritēriji:

- pieejas atbilstība modeļvadāmas arhitektūras principiem;
- pieejas apgabals, risināmie konfigurācijas pārvaldības uzdevumi;
- esošo rīku vai skriptu izmantošanas iespējamība jaunajā pieejā, kā arī jaunu risinājumu radīšana, ko ir iespējams izmantot atkārtoti.

Analīzes mērķis ir definēt galvenos trūkumus un esošo pieeju neatrisinātās problēmas, lai noskaidrotu pazīmes, kurām jāpiemīt inovatīvam risinājumam, kas tiks izstrādāts promocijas darbā.

2. MODEĻVADĀMA KONFIGURĀCIJAS PĀRVALDĪBA

2.1. Modeļvadāmas arhitektūras vispārīgie principi

Modeļvadāma arhitektūra (*MDA*, angļu val. *Model-Driven Architecture*) sākotnēji tika radīta programmatūras izstrādei. Modeļvadāma programmatūras izstrāde ir sistemātiskā modeļu lietošana programmatūras izstrādes dzīves ciklā. *MDA* ir saistīta ar tādām izstrādes metodēm, kuru pamatā ir programmatūras modeļu izmantošana primāro izstrādes artefaktu un izteiksmes formu veidā, kas paredz zināšanu atspoguļošanu strukturētā veidā ar modelēšanas valodas palīdzību, ievērojot tās noteikumus. Programmatūras izstrādē modeļi bieži tiek konstruēti līdz noteiktam detalizācijas līmenim, piemēram, lai demonstrētu kopējas sistēmas koncepcijas. Balstoties uz tiem, parasti atsevišķajā izstrādes posmā notiek sistēmas programmatūras kodēšana. Ir jāatzīmē, ka arvien biežāk tiek praktizēta automātiska lietojumu koda ģenerēšana no sākuma veidotiem modeļiem, un iegūtie rezultāti variējas no pamata lietojumu karkasiem līdz izvēršamām komponentēm. Svarīga loma tādas tendences attīstībā piemīt arī vienotās modelēšanas valodai (angļu val. *UML*), kas nodrošina industrijas modelēšanas valodas standartu [DON 2011, OSI 2011].

Modeļvadāma pieeja ir viena no *OMG* (angļu val. *Object Management Group*) iniciatīvām, kas nodrošina karkasu un vadlīnijas programmatūras izstrādes procesam, kuru pamatā ir sistēmas specificēšana un strukturēšana ar modeļu palīdzību. Modeļvadāmas izstrādes dzīves cikls pēc būtības neatšķiras no tradicionāla programmatūras inženierijas kontekstā, tajā ir definējami posmi, raksturīgi vairākām izstrādes metodoloģijām. Svarīgākā atšķirība izpaužas izstrādes procesa artefaktu būtībā, kas ir formālie modeļi, saprotami datoram [DON 2011].

Viens no pamata *MDA* nolūkiem ir projektēšanas atdalīšana no arhitektūras realizācijas – sistēmas koncepciju un tehnoloģiju atkabināšana ļauj izstrādātajiem izvēlēties piemērotāko projekta vajadzībām. Atšķirībā no projektējuma, kur galvenā uzmanība tiek pievērsta sistēmas funkcionālajām prasībām, lietojuma arhitektūra nodrošina attiecīgu infrastruktūru arī sistēmas nefunkcionālo prasību realizēšanai. Citiem vārdiem sākot, *MDA* paredz atdalīt sistēmas funkcionalitāti no detaļām, kā sistēma izmanto izvēlētas platformas iespējas. *MDA* karkasa terminoloģijā tas tiek nodrošināts ar no platformas neatkarīgiem modeļiem (angļu val. *Platform Independent Model*, *PIM*) un platformas specifiskiem modeļiem (angļu val. *Platform Specific Model*, *PSM*) palīdzību [DON, 2011, SIN 2010].

PIM ir neatkarīgs no jebkuras implementēšanas tehnoloģijas un apraksta sistēmu augstākajā abstrakcijas līmenī. Parasti ar *PIM* palīdzību tiek atspoguļotas sistēmas funkcijas uzņēmuma biznesa uzdevumu realizēšanā. Nākamajos soļos *PIM* tiek transformēts vienā vai vairākos *PSM*, kas apraksta sistēmu ar kādas implementēšanas tehnoloģijas koncepcijas palīdzību. Viens no svarīgākajiem posmiem ir transformācijas process, kurā tiek izmantotas specializētas transformācijas valodas, rīki un standarti. Pēdējais transformācijas solis ir katra *PSM* pārveidošana kodā, kas atšķirībā no *PIM-PSM* transformācijas tiek uzskatīts par vieglāko uzdevumu, jo parasti *PSM* ir cieši saistīts ar implementēšanas tehnoloģijām [DON, 2011, SIN 2010].

MDA ir nākamais loģiskais solis programmatūras izstrādes procesa attīstībā, kas nodrošina augstāko izstrādātāju produktivitāti, piedāvājot līdzekļus *PIM* transformācijas automatizēšanai. Konkrētas transformācijas loģika tiek definēta vienreiz, un tā var būt izmantojama vairāku sistēmu izstrādē. *PIM* ļauj izstrādātājiem izvairīties no pārāk agras izpildāmas platformas tehniskās detalizēšanas – tehnoloģijas specifiskās detaļas tiek aprakstītas transformācijas līmenī. Arī *PSM* un koda līmenī izstrādātājam paliek mazāk darba, jo noteikts koda apjoms tiek automātiski ģenerēts no *PIM*, kas ļauj pievērst vairāk uzmanības tieši biznesa problēmu risināšanai un rezultātā ļauj iegūt labāko sistēmas funkcionalitāti atbilstoši prasībām. Protams, ar to tiek paaugstinātas prasības (pilnīgums un nepretrunīgums) sākuma līmeņa modeļiem, kas kalpo par ievadi transformācijām. Svarīgs modeļvadāmas pieejas labums – pārnesamība tiek paveikta ar no platformas neatkarīgu modeļu izmantošanu, kurus var automātiski transformēt atbilstoši platformu specifikācijām. Protams, pārnesamības pakāpe lielā mērā ir atkarīga no platformas popularitātes un nepieciešamo transformāciju eksistēšanas [OSI 2011].

Modelis *MDA* kontekstā ir sistēmas vai tās daļas apraksts valodā ar skaidri definētu formu (sintaksi) un nozīmi (semantiku), kas varētu būt automātiski interpretēta ar datora palīdzību. Neskatoties uz to, ka sistēmai var būt definēti vairāki atšķirīgi modeļi, starp tiem eksistē noteiktas saistības (piemēram, vesels-daļa, kad viens modelis definē kopējo priekšstatu, bet cits – tikai atsevišķu sistēmas daļu detalizētā veidā). Pašlaik modeļvadāmā arhitektūra parasti ir cieši saistīta ar *UML*, kas ļauj novērst nepareizu modeļu iztulkojumu, tomēr specifisko domēnu valodu (*DSL*) izmantošanā ir pieļaujama alternatīva [DON 2011, OSI 2011].

Ja modelis ir reālās pasaules parādību abstrakcija, tad meta-modelis ir vēl viena abstrakcija, kas atspoguļo modeļa īpašības. Kā spilgtākus meta-modeļu piemērus var atzīmēt *Java* valodas specifikāciju vai kopējo datu glabātuves meta-modeli. Meta-modelēšana var būt

saprasta kā tehnikas modeļu sintakses un to kopsakarību definēšanai. Meta-modelis definē struktūru, semantiku un ierobežojumus vairāku modeļu kopai. Piemēram, *UML* meta-modelis apraksta, kādā veidā *UML* modeļi tiek strukturēti, to pieļaujamos elementus un citus atribūtus [DON 2011].

Šajā sakarā ir vērts atzīmēt *OMG* piedāvāto četru-slāņu arhitektūru, kurā modelis attēlo sistēmu *M1* līmenī saskaņā ar to meta-modeli *M2* līmenī. Meta-modelis savukārt atbilst *M3* meta-meta-modelim, un *M3* atbilst sev pašam. *MOF* (angļu val. *Meta Object Facility*) ir *OMG* meta-modeļu specificēšanas standarts, kas ietver noteiktu konstrukciju kopu objektorientētas informācijas modelēšanai. Piemēram, *UML* meta-modelis ir definēts ar *MOF* palīdzību [DON 2011].

Lai arī *MOF* ļauj definēt valodas struktūras un uzvedības aspektus, tas nespecificē modeļu apspoguļošanas vai saglabāšanas standartu. Šim nolūkam *OMG* nodrošina *XML* meta-datņu apmaiņas standartu (angļu val. *XML Metadata Interchange, XMI*), kas ļauj standartizēt *MOF* sakritīgo modeļu apmaiņu un piekļuvi [DON 2011].

Ir vairāki iemesli, kāpēc meta-modelēšana ir svarīgs aspekts. Pirmkārt, tai ir nepieciešams mehānisms, kas atļautu viennozīmīgi definēt modelēšanas valodas, saprotamas transformācijas rīkiem. Otrkārt, efektīvi modeļu transformācijas noteikumi izmanto avota un mērķa modeļa meta-modeļus [DON 2011].

Modeļvadāma pieeja, kas ir bāzēta uz *UML*, valodas un citiem programminženierijas industrijas standartiem modeļu un projektējuma vizualizēšanai, glabāšanai un apmaiņai. *MDA* ļauj izveidot augstas abstrakcijas modeļus, kas nav atkarīgi no izpildīšanas platformas un kas glabājas specializētos standartizētos repozitorijos. *MDA* ietver šādas tehnoloģijas: vienota modelēšanas valoda (angļu val. *Unified Modeling Language – UML*), meta-objektu iespējas (angļu val. *Meta-Object Facilities – MOF*), *XML* meta-datu apmaiņa (angļu val. *XML Metadata Interchange – XMI*) un kopējais krātuves meta-modelis (angļu val. *Common Warehouse Metamodel – CWM*) [DON 2011, OSI 2011].

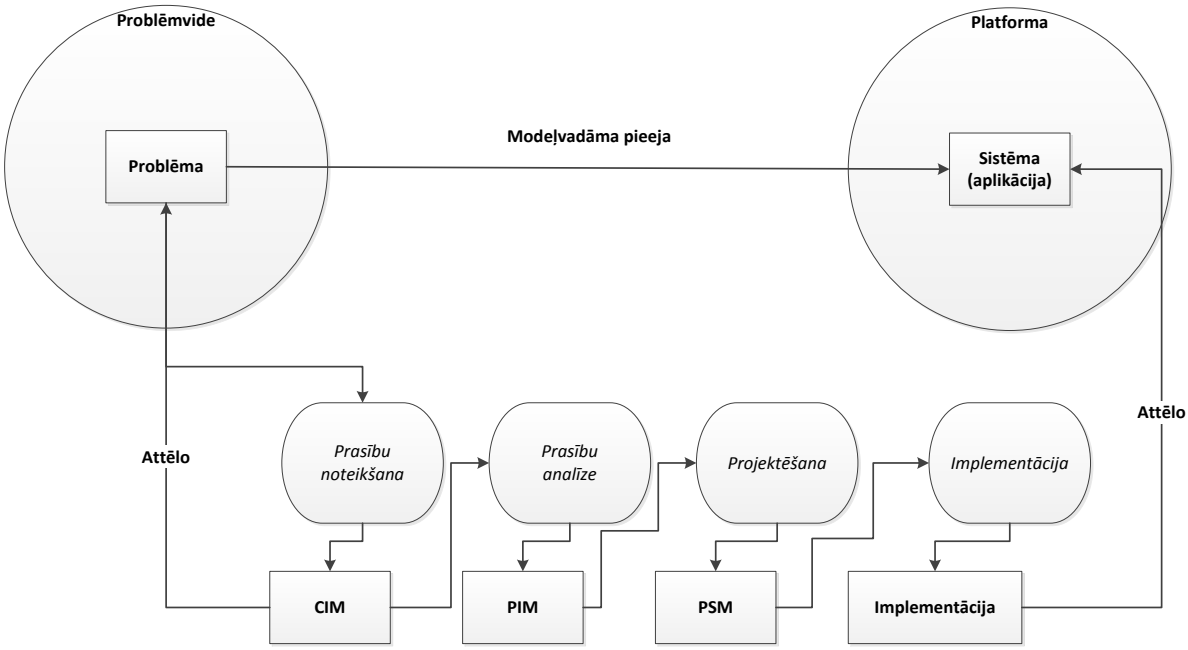
Modeļu transformācijas ir vienots sistēmas process viena modeļa konvertēšanai citā modelī, saglabājot noteikto ekvivalences saistību starp šiem modeļiem. *MDA* arhitektūras pamatā ir modelēšanas process un šo modeļu savstarpējas transformācijas. Modeli var izteikt kā *UML* diagrammu, *OCL* specifikāciju un teksta kopumu. Modeļvadāma pieeja nosaka dažādus modeļu tipus, kas var būt abstrakti (specificē sistēmas funkcionalitāti) un konkrēti, kas saistīti ar specifisku platformu, tehnoloģiju un implementāciju. Ir šādi modeļu tipi:

- *CIM* modelis (angļu val. *Computation Independent Model*) jeb no skaitļošanas neatkarīgais modelis;

- *PIM* modelis (angļu val. *Platform Independent Model*) jeb no platformas neatkarīgais modelis;
- *PSM* modelis (angļu val. *Platform Specific Model*) jeb no platformas atkarīgais modelis;
- koda modelis (angļu val. *Code Model*), retāk tiek lietots apzīmējums *ISM* modelis (angļu val. *Implementation Specific Model*).

Ir paredzēts, ka iespējams veikt transformācijas no *CIM* uz *PIM*, no *PIM* uz *PSM* un no *PSM* uz koda modeli, kā arī veikt transformācijas vienā abstrakcijas līmenī. Vienam iepriekšēja abstrakcijas līmeņa modelim var atbilst vairāki nākamā līmeņa modeļi. Piemēram, vienam *PIM* var atbilst vairāki *PSM*, kas apraksta sistēmas modeļus dažādām platformām. Transformācijas starp *MDA* modeļiem notiek ar marķēšanas (angļu val. *marking*) palīdzību: elementi avota modelī tiek marķēti un saistīti ar elementiem mērķa modelī. Modeļu transformācijas balstās uz meta-modelēšanas principiem. Meta-modelēšana ir modeļvadāmas pieejas bāzes tehnika. Pieeja ir bāzēta uz platformas modeļiem aprakstītiem ar *UML*, *OCL*, kas tiek saglabāti *MOF* repozitorijā. Meta-modelis (jeb «modeļa modelis») ir modelēšanas valodas modelis, kas definē modelēšanas valodas sintaksi un semantiku un nodrošina sadarbības iespēju starp modelēšanas procesu un transformācijas rīkiem. *MOF* ir *OMG* konsorcija standarts, kas ir paredzēts metamodeļu specificēšanai, izstrādei un vadīšanai. Tas definē valodu, kas savukārt definē modelēšanas konstrukciju kopu (jeb modelēšanas valodas sintaksi un semantiku), ko modelētājs var lietot, lai definētu un manipulētu ar sadarboties spējīgu meta-modeļu kopu. *MOF* ir starptautisks standarts. *MOF* nodrošina metame-tamodelēšanu *UML* meta-modeļiem un definē modelēšanas konstrukciju kopu, kas ļauj definēt un manipulēt ar modeļa metadatiem. Pats *MOF* ir definēts *UML* valodas notācijā un ir *UML* 2.x bāzes paplašinājums [DON 2011, OSI 2011].

Attēlā 2.1. shematiski parādīts programmatūras izstrādes process modeļvadāmas pieejas kontekstā.



2.1. att. Modeļvadāma pieeja (aizgūts no [SIN 2010])

Atgriežoties pie konfigurācijas pārvaldības, modeļvadāmajam pieejām jāsatur dažādu abstrakcijas līmeņa modeļus. Katram modelim jābūt skaidri definētam meta-modelim. Nākamās apakšnodaļas mērķis ir izpētīt esošās modeļvadāmas pieejas un rīkus konfigurācijas pārvaldības procesam un analizēt pieeju atbilstību šādiem kritērijiem:

- meta-modeļu esamība;
- modeļu abstrakcijas līmeni. Jāizpēta kādi modeļi pastāv, kāds ir abstrakcijas līmenis katram modelim;
- saiknes starp modeļiem un transformācijas. Ja risinājums satur vairākus konfigurācijas pārvaldības procesa modeļus, tad jāizpēta, kādas ir saiknes starp modeļiem un kādi ir transformācijas likumi vai arī kā tos veidot;
- rīku atbalsts. Jāizpēta, vai ir rīki, kas palīdzētu realizēt risinājumu projektā.

Šāda veida izpēte palīdzēs ne tikai identificēt esošos risinājumus, kas veido konfigurācijas pārvaldības modeļus, bet arī apzināties, cik lielā mērā tie atbilst vai neatbilst vispārīgiem *MDA* principiem. Autors uzskata, ka šāda veida analīze palīdzēs labāk apzināties esošo risinājumu priekšrocības un trūkumus un ļaus ne tikai atrisināt konfigurācijas pārvaldības problēmas, bet arī uzlabot risinājumus, kas veido konfigurācijas pārvaldības procesa modeli.

2.2. Konfigurācijas pārvaldības modeļvadāmie risinājumi

Analizējot konfigurācijas pārvaldības modeļvadāmos risinājumus, tika izpētītas pieejas un to attīstības tendences [PIN 2009, GIE 2009, BUC 2009, CAL 2012, KR 2014, FIT 2014, FUG 2014, CRA 2008], kā arī jaunākie rīki modeļvadāmas pieejas praktiskai realizācijai [OPE 2014, SER 2014, AZO 2014].

Konfigurācijas pārvaldības modeļvadāmais risinājums, kas tiek piedāvāts avotā [PIN 2009], atbilst modeļvadāmas pieejas galvenajiem principiem. Piedāvātā metodoloģijā ir minēti šādi modeļi: *CIM, PIM, PSM*. Visi pamatprincipi tiek ņemti no modeļvadāmas izstrādes pieejas, bet risinājums papildināts ar konfigurācijas pārvaldības modeļiem. Dažāda līmeņa abstrakcijas modeļi ietver produkta konfigurācijas vienumus un vienumu savstarpējās saiknes. Metodoloģija [PIN 2009] paredzēta tāda veida projektiem, kuros tiek lietota programmatūras modeļvadāma izstrādes metodoloģija. Risinājums paredz, ka konfigurācijas modeli veidos konfigurācijas vienumi un to savstarpējas saiknes, bet programmatūras izstrādes modelis definē programmatūras komponentes (klases, pakotnes, metodes utt.). Modelēšanas un modeļu transformācijas funkcionalitāti atbalsta speciāls rīks, kas ir izstrādāts [PIN 2009] darbā – *Model Driven Configuration Editor*.

Pieejai [PIN 2009] ir šādi galvenie sasniegumi:

- konfigurācijas pārvaldības un modeļvadāmas izstrādes apvienošanas koncepcija;
- produkta konfigurācijas abstrakts modelis;
- rīks, kas paredzēts konfigurācijas pārvaldības modeļu izveidei;
- instrukcijas, kā radīt un paplašināt rīku, kas atbalsta modeļvadāmo konfigurācijas pārvaldību.

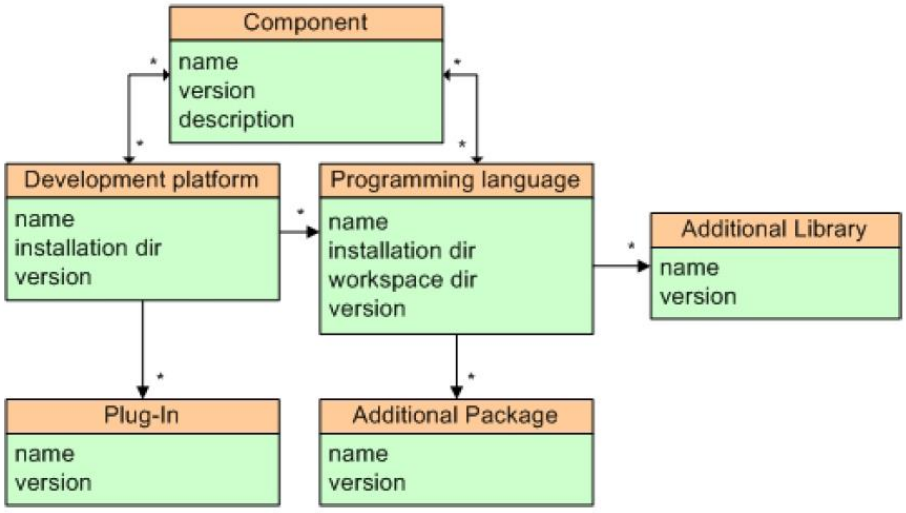
Galvenais mērķis darbā [PIN 2009] ir savienot konfigurācijas pārvaldību un programmatūras izstrādes procesus universālā modelī. Modelis apraksta problēmapgabalu abstraktā līmenī un piedāvā pamatus programmatūras izstrādes un konfigurācijas pārvaldības modeļiem. Tas tiks panākts, pateicoties transformācijām, ko definē [PIN 2009] pieeja. Programmatūras izstrādes modelis piedāvā ieejas datus sistēmas koda ģenerēšanai, bet konfigurācijas pārvaldības modelis tiks transformēts *XML* failos, kuri definē programmatūras konfigurācijas vienumus un to savstarpējas saiknes vai vienkārši kalpo kā konfigurācijas faili kādai konkrētai aplikācijai vai tehnoloģijai, kas tehniski atbalsta konfigurācijas pārvaldības procesu.

Modeļvadāma konfigurācijas pārvaldība [PIN 2009] satur modeļus, dažas meta-modeļu definīcijas un rekomendācijas, kā uzlabot konfigurācijas pārvaldības un programmatūras izstrādes sadarbību. Rezultātā tika izveidots rīks – *Model Driven Configuration Editor*. Tas ir grafiskais redaktors *Eclipse* vidē, kas izmanto *Eclipse Modeling Framework* un *Grafical Modeling Framework* kā vizuālizācijas pamatus. Transformācijām tiek izmantota *openArchitectureWare (oAW)* arhitektūra. Darbs [PIN 2009] sniedz arī detalizētas instrukcijas, kā izveidot patvaļīgu konfigurācijas pārvaldības modelēšanas rīku.

Publikācijā [PIN 2009] tiek piedāvāta konceptuāla realizācija modeļvadāmai konfigurācijas pārvaldībai. Metode orientēta uz konfigurācijas vienumu identifikāciju un attiecību noteikšanu augstākajā līmenī. Konfigurācijas vienumi varētu būt šādi:

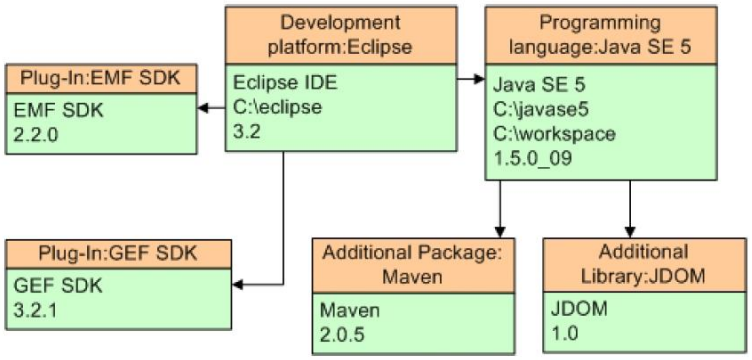
- aparātu komponentes (serveri);
- programmatūras komponentes (datubāzes, aplikācijas, operētājsistēmas);
- dokumentācija un izejas koda faili;
- organizācijas komponentes (reglamentē lietotāju pieejas tiesības utt.).

Katrai konfigurācijas vienumu grupai tiek veidots meta-modelis. Pēc tam tiek veidots *PIM* (no platformas neatkarīgais) modelis no iegūtā meta-modeļa. Vēlāk ar transformāciju palīdzību iespējams modeli transformēt *PSM* (no platformas atkarīgs) modelī un pat *XML* failos, kas apraksta konfigurācijas vienumus, struktūru un saiknes. Attēlā 2.2. var redzēt *PIM* modeļa piemēru izstrādes videi.



2.2. att. *PIM* modeļa piemērs izstrādes videi

Transformējot minēto *PIM* modeli *PSM* modelī, attēlā 2.3. var redzēt piemēru platformas atkarīgajam modelim, kas šajā gadījumā tiek veidots *Eclipse* integrētai izstrādes videi.



2.3. att. *PSM* modeļa piemērs izstrādes videi (aizgūts no [PIN 2009])

Vēlāk modeli, kas ir redzams attēlā 2.3., nepieciešamības gadījumā iespējams transformēt uz *XML* failu, kas būs saprotams konfigurācijas pārvaldības rīkam.

Modeļvadāma konfigurācijas pārvaldība bez konfigurācijas pārvaldības rīku atbalsta paliek tikai teorētiskā līmenī. Nepieciešams, lai konfigurācijas pārvaldības rīki saprastu izejas datus no modeļvadāmas konfigurācijas pārvaldības [PIN 2009]. Kā vienu no svarīgākajiem uzdevumiem modeļvadāmajā konfigurācijas pārvaldībā [PIN 2009] autors nosauc meta-modeļu definēšanu konfigurācijas pārvaldībai – modelēšanas valodu un valodas likumus. Zinātniskā darba [PIN 2009] autors uzskata, ka, lai ieviestu modeļvadāmo konfigurācijas pārvaldību projektā, nepieciešams veikt šādus uzdevumus:

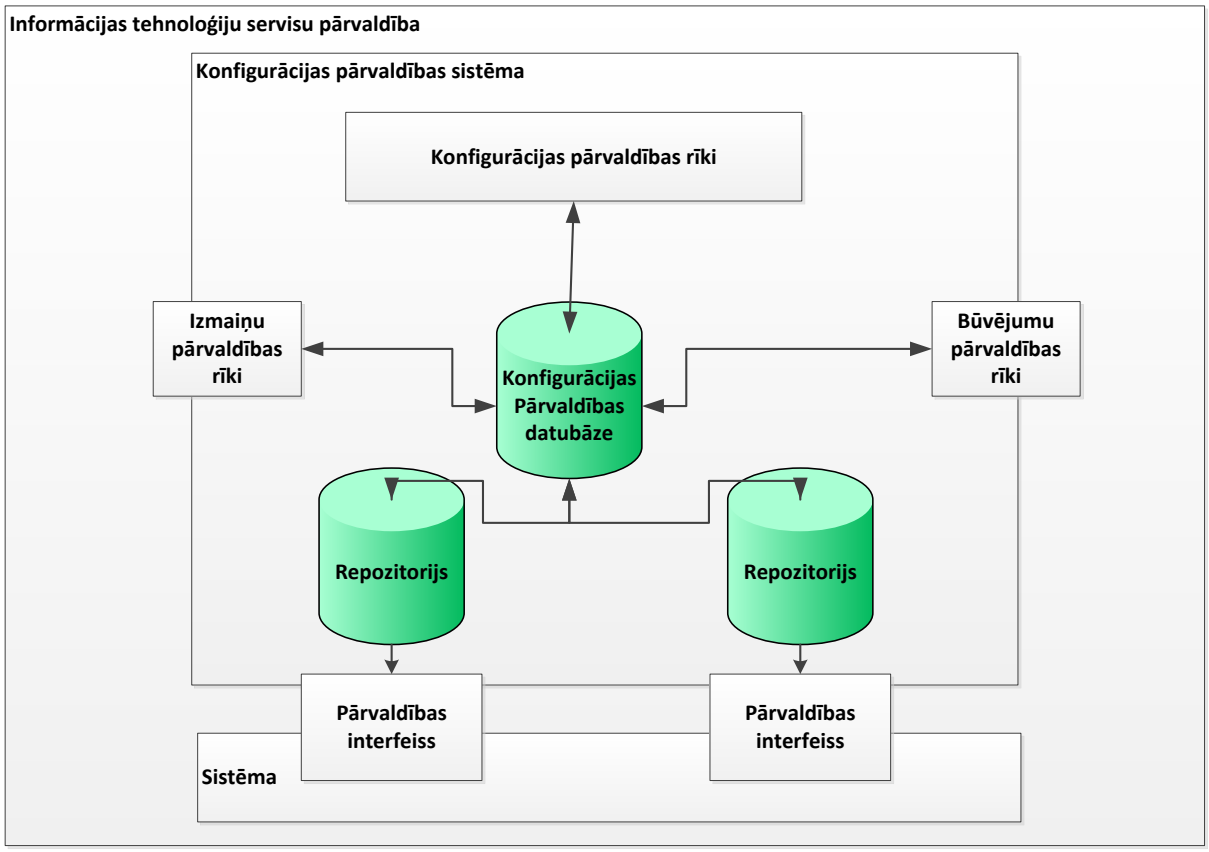
- izveidot meta-modeļus: modelēšanas valoda un tās likumi;
- izveidot konfigurācijas pārvaldības modeli. Balstoties uz iegūtajiem meta-modeļiem, būs iespējams izveidot konfigurācijas modeļus (*PIM* un *PSM*);
- ievietot modeļus konfigurācijas pārvaldības rīkos (versiju kontroles sistēmās) kā konfigurācijas vienumu definīcijas;
- definēt modeļu transformācijas.

Sakarā ar iegūto informāciju no [PIN 2009] avota, promocijas darba autors secināja:

- piedāvātais risinājums ir orientēts uz projektiem, kur izstrāde notiek pēc modeļvadāmas pieejas. Īsti nav skaidrs, vai metodoloģiju var izmantot, ja projekta komanda strādās pēc citas metodoloģijas, piemēram, pēc ūdenskrituma metodes, kur izstrādes artefakti nebūs modeļi, bet programmatūras kods;

- pieeja ir orientēta galvenokārt uz vienu no konfigurācijas pārvaldības lielajiem uzdevumiem – konfigurācijas vienumu identifikāciju;
- pieeja pilnībā atbilst modeļvadāmai pieejai – ir meta-modelis, ir priekšlikumi, kā meta-modeli veidot katram konfigurācijas vienuma veidam. Ir minēta arī *PSM* un *PIM* modeļu lietošana un transformācijas. Taču, kā jau tika minēts, metodoloģija ir orientēta vien uz konfigurācijas vienumu identifikāciju ar nosacījumu, ka projektā izmanto modeļvadāmo izstrādes metodoloģiju;

Atšķirībā no tikko apskatītā darba, kurā uzsvars bija likts uz konfigurācijas vienumu identifikāciju un atkarību noteikšanu, avotā [GIE 2009] tiek piedāvāta metodoloģija, kas apskata konfigurācijas pārvaldības procesu kopumā. Konfigurācijas pārvaldības principi tiek ņemti no *ITIL* (angļu val. *Information Technology Infrastructure Library*) un vēlāk tiek izveidoti modeļi, no kuriem savukārt varētu izveidot abstraktu konfigurācijas pārvaldības procesu un vēlāk transformēt to no platformas atkarīgajā modelī. Pieeja izmanto modeļvadāmas izstrādes galvenos principus. Uz meta-modeļiem balstīti modeļi piedāvā nepieciešamu abstrakciju, kas uzlabo konfigurācijas procesa pārvaldību, pārskatāmību un ļauj lietotājam nepieciešamības gadījumā implementēt modeli kādai noteiktai tehnoloģijai, piemēram, veicot modeļa transformāciju. Ir izveidots sistēmas prototips, kas implementē modeļvadāmo konfigurācijas pārvaldību. Konfigurācijas pārvaldības abstraktais modelis tiek izveidots *ITSM* (angļu val. *IT Service Management*) kontekstā un ir attēlots 2.4. attēlā.



2.4. att. Konfigurācijas pārvaldības abstrakts modelis

Pieeja paredz, ka eksistē šādi modeļi (sk. attēlu 2.4.):

- pārvaldības rīku (*management tools*) modeļi;
- konfigurācijas pārvaldības datubāzes (*CMDB*) modelis;
- pārvaldāma datu repozitorija (*MDR*) modelis.

Saites, kas redzamas attēlā 2.4. (*Query/Update* utt.), piedāvā realizēt kā operācijas ar modeļiem – transformācijas, apvienošana, papildināšana ar atribūtiem utt. Taču konkrētas realizācijas netiek piedāvātas. Darbā [GIE 2009] ir iekļauta implementācija piedāvātai modeļvadāmai konfigurācijas pārvaldībai. Taču risinājums ir orientēts tikai uz vienu tehnoloģiju (*JAVA*).

Runājot par kādu no konkrētiem konfigurācijas pārvaldības uzdevumiem un to modeļvadāmajiem risinājumiem, ir vērts pieminēt publikāciju [BUC 2009] un arī darbus, kas ir minēti šajā avotā. Šajās publikācijās vācu autoru grupa prezentē risinājumu modeļvadāmajai versiju kontroles sistēmai, kas varētu atbalstīt versiju kontroli ne tikai projektos, kur galvenie artefakti ir izejas kods, bet arī projektos, kur artefakti ir modeļi. Avotā [BUC 2009] tiek uzsvērta problēma, ka esošie versiju kontroles risinājumi dažreiz ir grūti pielāgojami konkrēta projekta prasībām, jo funkcionēšanas modelis ir realizēts koda veidā.

Savukārt izejas kodā ir daudz faktoru, kas sarežģī uzdevumu veikt noteiktās izmaiņas versiju kontroles sistēmā. Jaunas modeļvadāmas versiju kontroles sistēmai ir šādas iezīmes:

- sistēma bāzējas uz abstrakta modeļa, kas ir vispārīgs modelis versiju kontroles problēmapgabalam;
- modelis ir izpildāms. Izstrāde tiek atvieglota, ģenerējot sistēmas izejas kodu no modeļa;
- sistēma ir izstrādāta, lai atbalstītu zarošanu. Izpildāmais modelis sastāv no brīvi konfigurējamajām komponentēm, kuras var definēt nosakot sistēmas īpatnības;

Analizējot publikāciju [BUC 2009] tādā pašā kontekstā, kā iepriekš minētos konfigurācijas pārvaldības modeļvadāmus risinājumus, nācās secināt:

- netiek apskatīts vispārīgs konfigurācijas pārvaldības process, ir aprakstīta tikai versiju kontrole;
- nav rīku, kas tehniski varētu atbalstīt versiju kontroles modeļu veidošanu.

Viena no modeļvadāmajām konfigurācijas pārvaldības pieejām [CAL 2012] ir orientēta uz rīku savstarpēju integrāciju, cenšoties paaugstināt procesa automatizācijas līmeni. Lai uzturētu konfigurācijas pārvaldības pilnu procesu, ir vajadzīgi vairāki rīki: versiju kontroles sistēmas, problēmu pārvaldības sistēmas, būvējumu serveri, nepārtrauktās integrācijas serveri un daudzi citi. Praksē bieži ir tā, ka visi minēti rīki strādā atsevišķi viens no otra. Lai atvieglotu konfigurācijas pārvaldības procesu, tiek piedāvāta pieeja integrēt kopā visus šos rīkus. Taču, lai integrētu konfigurācijas pārvaldības dažādus rīkus, ir nepieciešams definēt katra integrējama rīka vispārīgu koncepciju [CAL 2012]. Publikācija piedāvā uzdevumu ontoloģiju konfigurācijas pārvaldības procesiem. Šī ontoloģija tiek izmantota kā konfigurācijas pārvaldības modelis, kas parāda, kādā veidā tiks integrēti dažādi konfigurācijas pārvaldības rīki. Ontoloģija galvenokārt ir balstīta uz izmaiņu kontroli, kas ir viena no galvenajām koncepcijām konfigurācijas pārvaldībā. Ontoloģija piedāvā informāciju par konfigurācijas pārvaldības apakšprocesu savstarpējām saitēm [CAL 2012].

Pēc publikācijas [CAL 2012] izpētes tika secināts:

- tiek piedāvāta uzdevuma ontoloģija, kas vispārīgi apraksta konfigurācijas pārvaldības procesu konkrētu apakšuzdevumu kontekstā. Risinājums nav atkarīgs no kādām noteiktām tehnoloģijām vai platformām;
- ontoloģija modeļvadāmas pieejas kontekstā var būt kā avots no platformas neatkarīgajam modelim, taču mulsina fakts, ka ontoloģijas izveidošanai tika izmantoti

vien konkrēti standarti (*ISO*) un rīki (*Subversion*). Tomēr trūkst apraksta, kā pašam veidot ontoloģijas elementus;

- ontoloģija pārsvarā ir paredzēta konfigurācijas pārvaldības dažādu rīku integrācijai, taču integrācijai jābūt saistītai ar procesu. Kamēr nav abstrakta procesa modeļa, par rīku instalāciju un integrāciju domāt ir pāragri [AIE 2010];
- nav rekomendāciju, kā ontoloģiju varētu izmantot, lai iegūtu, piemēram, *PSM* modeli konfigurācijas pārvaldībai, kur, ņemot vērā publikācijas kontekstu, varētu būt informācija arī par rīku savstarpēju integrāciju.

Mēģinājumi izmantot formālas mākslīga intelekta metodes rīku konfigurēšanai ir novēroti ne tikai konfigurācijas pārvaldībā. Arī programmatūras projektu vadībā ir nepieciešams efektīvi konfigurēt projekta pārvaldības sistēmas. Kā jau tika minēts promocijas darba ievadā, mūsdienās programmatūras izstrādes projekti sākās ļoti ātri. Arī no projekta vadības viedokļa ir ārkārtīgi svarīgi ātri un efektīvi nokonfigurēt pārvaldības sistēmu. Plaši šo problēmu apskata un risina RTU zinātniece Solvita Bērziša [BER 2012, BER 2011].

Tabulā 2.1. tiek dots salīdzinājums minētajiem konfigurācijas pārvaldības modeļvadāmajiem risinājumiem.

2.1. tabula

Konfigurācijas pārvaldības modeļvadāmu risinājumu salīdzinājums

Risinājuma identifikators	**Meta-modelis**	**Modeļi ar atšķirīgu abstrakcijas līmeni**	**Transformāciju risinājumi**	**Rīku atbalsts**	**Komentāri**
[PIN 2009]	+	+	+/–	+/–	Saturiskā ziņā labākais no visiem minētajiem risinājumiem, jo ir daļējs risinājums meta-modelim, rīks, kas veic modeļu transformācijas.
[GIE 2009]	+/-	+/-	–	+/–	Teorētisks risinājums, nav minētas konkrētas detaļas, kā šādu risinājumu var

					realizēt. Pieeja ir orientēta tikai uz vienu tehnoloģiju.
[BUC 2009]	–	–	–	+/–	Risinājums ir orientēts vien uz versiju kontroli nevis uz konfigurācijas pārvaldības procesu kopumā.
[CAL 2012]	+/–	+/–	+/–	+/–	Lai gan risinājums neatbilst vispārīgiem modeļvadāmas pieejas principiem, ir uzsvērta svarīga problēma, kas obligāti jāņem vērā modeļvadāmas konfigurācijas pārvaldības risinājuma izstrādē – rīku savstarpēja integrācija. Teorētiskā līmenī ir piedāvāts risinājums, kā izveidot abstraktu integrācijas modeli rīkiem, kas atbalsta konfigurācijas pārvaldības procesu.

Avots [KR 2014] piedāvā modeli, kurā apvienoti versiju kontrole un būvējumu veidošana. Tiek izveidots modelis, kura implementācija ļauj pārvaldīt produktu būvējumus pilnīgi automātiski. Modelī ir iekļauta izejas koda pārvaldība, dažādu rīku savstarpēja integrācija un atsevišķu darbību plūsma, kas secīgi jārealizē, virzoties no programmatūras izmaiņu izstrādes līdz būvējumam. Darba [KR 2014] būtiskākie sasniegumi ir šādi:

- tika izveidots versiju kontroles un būvējuma procesa abstrakts modelis, kas nav atkarīgs no kādas konkrētas tehnoloģijas;
- modelis tika realizēts praksē, izmantojot populārus rīkus konfigurācijas pārvaldības jomā. Autors pamato rīku izvēli un detalizēti apraksta eksperimentu gaitu, kas ļauj to atkārtot ar citiem rīkiem.

Analizējot [KR 2014] publikāciju, ņemot vērā konfigurācijas pārvaldības tendences, kas tika aprakstītas pirmajā promocijas darba nodaļā, konstatētas šādas nepilnības:

- no modeļvadāmas pieejas viedokļa izskatās, ka *PIM* un *PSM* modeļi ir apvienoti vienā, kas nozīmē, ka nav iespējas vienam no platformas neatkarīgam modelim iegūt vairākus platformas specifiskus modeļus. Tas var radīt grūtības, ja, piemēram, uzņēmumam būs divi projekti ar vienādu procesu darbību plūsmu, bet ar atšķirīgiem rīkiem konfigurācijas pārvaldības uzdevumu risināšanai;

- nav aprakstīts, kā vienā projekta iegūtus risinājumus atkārtoti lietot citā projektā. Kā jau tika minēts iepriekšējā promocijas darba nodaļā, mūsdienīgiem efektīviem risinājumiem jāsniedz iespēja atkārtoti izmantot jau izstrādātas komponentes (skripti, funkcijas utt.);
- pieeja ir orientēta vien uz diviem konfigurācijas pārvaldības uzdevumiem: versiju kontroli un būvējumu. Savukārt citi konfigurācijas pārvaldības uzdevumi nav apskatīti. Runājot par izejas koda pārvaldību, tiek piedāvāta tikai viena izejas koda zarošanas stratēģija. Lietotājam nav iespējams būvēt savu stratēģiju atkarībā no projekta vajadzībām.

Publikācijas [FIT 2014, FUG 2014] nepiedāvā kādu konkrētu konfigurācijas pārvaldības risinājumu, taču iezīmē galvenās tendences, apstiprinot informāciju, kas tika sniegta promocijas darba iepriekšējā nodaļā. Darbus [FIT 2014 un FUG 2014] ir vērts īpaši izcelt, tāpēc, ka autori uzsver un pamato konfigurācijas pārvaldības procesa saiknes nepieciešamību ar projekta kopēju problēmvidi. Attīstoties *Agile* metodoloģijām, ir svarīgi, lai konfigurācijas pārvaldība spētu ātri reaģēt uz katru sīku izmaiņu projektā. Pretējā gadījumā pieaug manuāla darba apjoms, gatavojot produkta laidienus, kā arī pieaug risks, ka produkta būvējumi būs nekorekti.

Galvenais secinājums pēc [FIT 2014, FUG 2014] izpētes ir nepieciešamība konfigurācijas pārvaldības modeļos paredzēt saikni ar projekta problēmvidi, lai savlaicīgi varētu veikt izmaiņas konfigurācijas pārvaldības procesā.

Pieeja, kas aprakstīta avotā [WET 2012], uzsver no platformas neatkarīga modeļa nepieciešamību konfigurācijas pārvaldības risinājumā. Publikācija parāda tendenci, ka mūsdienās arvien biežāk tiek izmantoti programmatūras risinājumi, kas atrodas mākoņos, līdz ar to procesa plānošana nedrīkst būt piesaistītai kādai konkrētai platformai, servera adresei utt. Darbā ir piedāvāta metodoloģija un rīks, kas ievieš *DevOps* [WET 2012] prakses, tajā skaitā – arī konfigurācijas pārvaldības galvenos uzdevumus projektos, kur programmatūra atrodas mākoņos. Viena no minēta darba nepilnībām slēpjas meta-modeļu definīcijā. Modeļi, kas ievieš konfigurācijas pārvaldības procesus modeļvadāmajā mākoņskaitļošanas risinājumā, ir pasniegti kā procesa ieviešanas soļi, taču elementu definīcijai netiek piešķirta pietiekama uzmanība. Līdz ar to iespējamas grūtības modeļu atkārtotā lietošanā. Kā jau tika minēts nodaļas sākumā, modeļvadāmajā risinājumā katram modelim ir jābūt definētam avotam jeb meta-modelim, kā arī transformācijas likumiem. Risinājumā [WET 2012] šīs definīcijas nav aprakstītas tādā līmenī, ka tās varētu izmantot atkārtoti jebkuros programmatūras izstrādes projektos.

Analizējot jaunākos rīkus, kas ievieš modeļvadāmu konfigurācijas pārvaldības procesu [OPE 2014, SER 2014, AZO 2014], tika apkopoti galvenie ieguvumi un trūkumi, ko konstatēja promocijas darba autors, ņemot vērā pirmajā nodaļā noteiktas konfigurācijas pārvaldības prasības. Būtiskākie sasniegumi rīkos [OPE 2014, SER 2014, AZO 2014] ir šādi:

- lielāka daļa no analizētiem rīkiem atbilst galvenajiem modeļvadāmas pieejas principiem. Rīki ļauj salīdzinoši ātri modelēt konfigurācijas pārvaldības procesu programmatūras izstrādes projektā un pēc tam implementēt to konkrētajām tehnoloģijām un platformām;
- konfigurācijas pārvaldības process ir pārskatāms un viegli konfigurējams, pateicoties intuitīvi saprotamai lietotāju grafiskai saskarnei. Konfigurācijas pārvaldnieks veido produkta būvējumu scenārijus ar peles klikšķiem, nevis rakstot milzīgus skriptus. Rīki automātiski piedāvā metrikas un statistikas savākšanu, līdz ar to par žurnālifikāciju papildus nav jādomā: visa statistika par visiem būvējumiem vizuāli labi attēlota;
- rīki pilnībā atbilst mūsdienīgām programmatūras izstrādes nozares tendencēm. Tika realizētas iespējas veidot paralēlus būvējumus, konfigurēt sistēmu, kas spēj veikt vairākus desmitus būvējumu dienā. Pārsvarā visos rīkos ir iebūvētas funkcijas, kas atbalsta būvējumu veidošanas procesus arī mākoņos. Līdz ar to vairs nav jāraksta statiski skripti katram projektam atsevišķi.

Apkopojot informāciju no avotiem [OPE 2014, SER 2014, AZO 2014], tika secināts, ka rīkiem ir arī trūkumi:

- pārsvarā visi rīki ir orientēti uz šādiem konfigurācijas pārvaldības uzdevumiem: būvējumu un instalāciju pārvaldība, produkta laidienu sagatavošana pasūtītājam un metriku savākšana. Taču reti kurš rīks pievērš uzmanību versiju kontrolei un izejas koda pārvaldībai. Savukārt bez pārdomātas versiju kontroles būvējumu un instalācijas process nevar būt kvalitatīvs [AIE 2010];
- realizējot konfigurācijas pārvaldības procesu ar rīkiem, kas minēti šajos avotos [OPE 2014, SER 2014, AZO 2014], zemāka abstrakcijas līmeņa modeļiem (skripti, projektu struktūra, kompilācijas algoritmi utt.) ir definēti konkrēti priekšnosacījumi. Ja tos neievēro, risinājums nestrādās korekti. Taču nereti uzņēmumam ir sava specifika un pieeja dažādu skriptu un projektu veidošanai un konfigurācijai, un diez vai uzņēmums būs gatavs attiekties no risinājumiem un pieejām, kas ir pārbaudīti gadiem. Piemēram, ja, ieviešot kādu no jaunajiem būvējumu un instalācijas rīkiem, visiem *JAVA*

projektiem vajadzēs pārveidot klašu un pakotņu struktūru, diez vai uzņēmums būs tam gatavs.

2.3. Nodaļas kopsavilkums

Analizējot modeļvadāmos konfigurācijas pārvaldības pieejas, izdevās identificēt pieeju attīstības tendences un trūkumus. Ņemot vērā analizētus avotus, mūsdienīgam konfigurācijas pārvaldības risinājumam jābūt šādam:

- modeļvadāmam. Attīstoties *Agile* programmatūras izstrādes metodoloģijai un mākoņskaitļošanas tehnoloģijām, noveco statiski, uz izeja koda vien balstīti risinājumi. Sāk zust statiska servera jēga, kur var palaist skriptu un iegūt rezultātu. Tāpēc ir jābūt modeļvadāmam risinājumam, kas būtu spējīgs izpildīt konfigurācijas pārvaldības uzdevumus vienlaikus uz vairākiem serveriem, par kuru fizisku atrašanas vietas nav zināmas;
- modeļvadāmajā konfigurācijas pārvaldības risinājumā jābūt vairākiem modeļiem ar dažādiem abstrakcijas līmeņiem. Samazinot procesa abstrakcijas līmeni, konfigurācijas pārvaldībai jāprot atkārtoti izmantot jau esošās risinājumu komponentes. Analizējot mūsdienīgus rīkus, kas atbalsta modeļvadāmu konfigurācijas pārvaldību, nācās secināt, ka rīki piedāvā lietot noteiktu projektu struktūru un nesniedz iespēju zemākajā abstrakcijas līmenī izmantot jau esošus risinājumus. Tāpēc modeļvadāmajam konfigurācijas pārvaldības risinājumam jābūt universālam un jāprot strādāt gan ar jau esošiem risinājumiem, gan arī veidot jaunus pēc vienotā un skaidri zināma principa. Tas ļaus ietaupīt laiku, ieviešot procesus katrā jaunajā programmatūras izstrādes projektā;
- modeļvadāmajam konfigurācijas pārvaldības risinājumam jābūt universālam un nav jāuzspiež konkrēta platforma, tehnoloģija utt. Praksē ir tā, ka, ja uzņēmumam jau ir risinājumi, ko tas ilgstoši lieto savos projektos, vieglāk būtu pieņemt risinājumu, kas ne tikai piedāvā jaunas mūsdienīgas iespējas, bet arī ļauj paaugstināt esošo risinājumu atkārtotu lietojamību;
- modeļvadāmajam konfigurācijas pārvaldības risinājumam jāspēj sniegt atbalstu visiem konfigurācijas pārvaldības uzdevumiem. Analizējot esošos modeļvadāmus rīkus, nācās atzīt, ka pārsvarā tiek atbalstīts vien produktu

būvējumu uzdevums, taču versiju kontrole un izejas koda pārvaldība tiek uztverta kā pašsaprotama. Šajā gadījumā iespējama situācija, kad programmatūras būvējumu process tiek veidots no nekorektas izejas koda bāzes, kas noved pie nopietnām problēmām un nestrādājošas programmatūras;

- risinājumam jāsniedz iespēja pēc iespējas ātrāk ieviest procesus jaunajos programmatūras izstrādes projektos, jo ilgstoša procesu ieviešana izraisa ārpus procesa darbības, kas pēc tam izraisa papildu kļūdas un izmaksas to novēršanai. Piemēram, ja process nespēj izveidot produkta būvējumu, bet pasūtītājam steidzami jāsaņem programmatūra, būvējums var tikt izveidots programmētāja datorā, kas izraisa riskus un var novest pie neparedzētām kļūdām nākotnē.

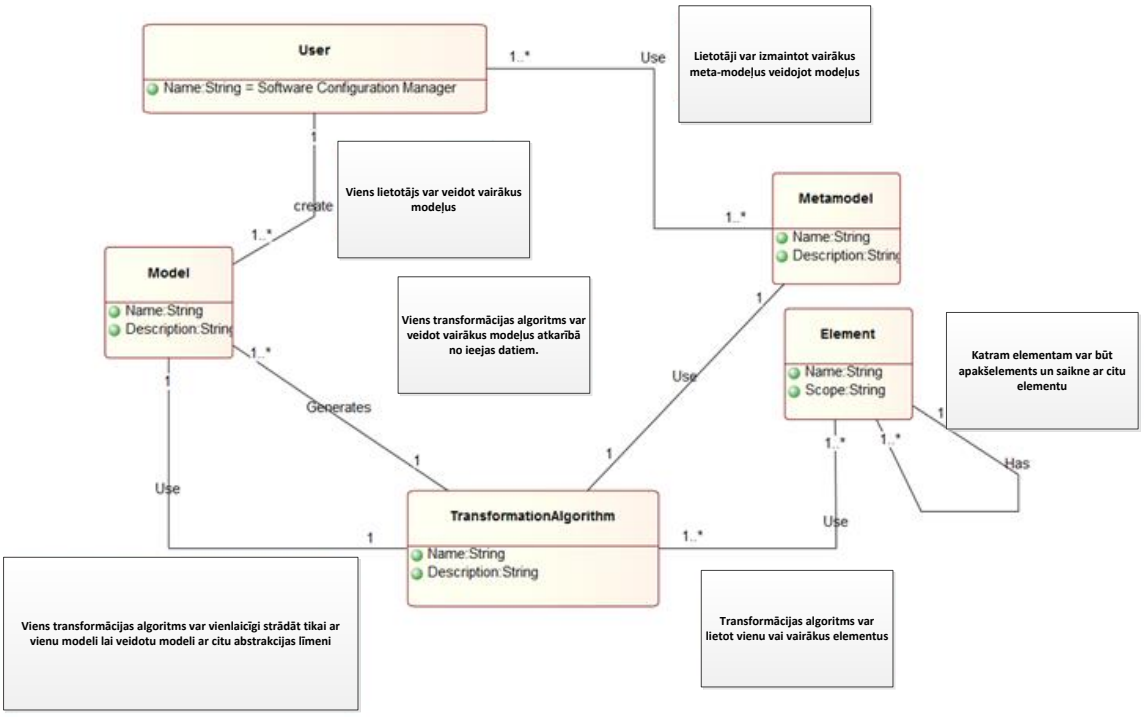
Nākamajā promocijas darba nodaļā tiks aprakstīta piedāvāta modeļvadāma pieeja konfigurācijas pārvaldības ieviešanai, kurā tika ņemti vērā trūkumi esošajos modeļvadāmajos risinājumos. Atšķirībā no citiem modeļvadāmajiem risinājumiem konfigurācijas pārvaldībā, jaunā ir orientēta uz visiem uzdevumiem, neprasa konkrētu implementācijas tehnoloģiju un sniedz iespēju implementācijā izmantot jau esošos risinājumus atkārtoti. Piedāvātā metodoloģija ir orientēta uz konfigurācijas pārvaldības procesu ieviešanas laika samazināšanu, jo jaunajos projektos maksimāli tiks izmantota iepriekšējo projektu pieredze.

3. *MTM* PIEEJAS UN *EAF* METODOLOĢIJAS IZSTRĀDE

3.1. *MTM* pieejas definīcija un vispārīgs apraksts

MTM (angļu val. *Model – Transformation – Model*) – jaunizstrādāta pieeja konfigurācijas pārvaldības procesu attēlošanai ar modeļu palīdzību. Pieeja ietver iespēju transformēt viena veida modeļus citā, mainot konfigurācijas pārvaldības procesa abstrakcijas līmeni.

MTM pieeja ir orientēta uz konfigurācijas pārvaldības ieviešanas laika samazināšanu, iespēju robežās izmantojot jau esošos risinājumus atsevišķām procesa posmiem. Attēlā 3.1. var redzēt meta-modeli jaunajai *MTM* pieejai.



3.1. att. *MTM* pieejas meta-modelis

Balstoties uz 3.1. attēlā redzamajiem elementiem, *MTM* pieejas galvenie darbības principi tiek skaidroti šādi:

- konfigurācijas pārvaldības procesu ir iespējams attēlot vairākos abstrakcijas līmeņos;

- katram abstrakcijas līmenim ir savs noteikts meta-modelis (*Metamodel*) jeb modelēšanas valoda, kas ļauj izveidot konfigurācijas pārvaldības procesu noteiktam projektam;
- konfigurācijas pārvaldības modeli var veidot lietotājs (*User*) vai transformācijas algoritms (*TransformationAlgorithm*). Lietotājs veido modeli manuāli, izmantojot atbilstošas modelēšanas valodas (*Metamodel*) elementus, savukārt transformācijas algoritms (*TransformationAlgorithm*) modeļus veido automātiski, operējot ar transformācijas likumiem, nepieciešamības gadījumā piesaistot lietotāju (*User*) un/vai papildu elementus (*Element*);
- transformācijas likumi nosaka, kā ar vienas modelēšanas valodas palīdzību (*Metamodel*) attēlot konfigurācijas pārvaldības procesu citā modelēšanas valodā. Līdz ar to likumu kreisajā pusē ir noteikts elementu stāvoklis vienai modelēšanas valodai, bet labajā pusē atbilstošais elementu stāvoklis citā meta-modeļa kontekstā;
- *MTM* realizācijā ir vismaz viens elements (*Element*), kas strukturētā veidā glabā visus esošos konfigurācijas pārvaldības risinājumus konkrētajā uzņēmumā. Šo elementu varētu nosaukt arī par glabātuvi. Katram risinājumam ir skaidri zināma konfigurācijas pārvaldības procesa daļa, kurā šis risinājums var tikt lietots. Vismaz vienā – zemākā – abstrakcijas līmeņa konfigurācijas pārvaldības modelī jābūt uzskaitītām visām tikko minētajām konfigurācijas pārvaldības procesa daļām. Lietotājam jābūt iespējai izvēlēties katrai procesa daļai atbilstošu risinājumu no glabātuves. Šo procesu varētu pārvaldīt speciāls transformācijas algoritms (*TransformationAlgorithm*). Rezultātā gatavajā modelī jābūt visām detaļām par konfigurācijas pārvaldības ieviešanas soļiem un risinājumiem.

Attēlā 3.1. ir redzami šādi *MTM* pieejas elementi:

User – modeļvadāmas konfigurācijas pārvaldības lietotājs (konfigurācijas pārvaldnieks);

Metamodel – avots konkrēta veida modelim (modelēšanas valoda, kas apraksta konfigurācijas pārvaldību noteiktā kontekstā);

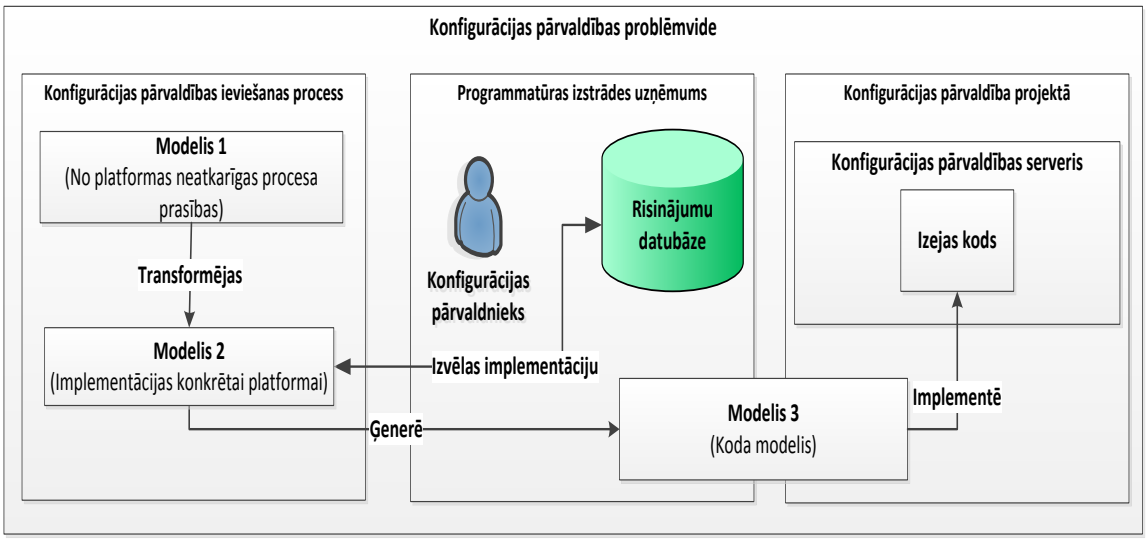
Model – konkrēts modelis, kas ir izveidots no attiecīga avota jeb meta-modeļa;

TransformationAlgorithm – transformācijas algoritms, kas pārveido viena veida modeli citā, izmantojot definētus likumus, kas operē ar atbilstošu meta-modeļu elementiem;

Element – *MTM* koncepcijas papildu elements, kas palīdz, piemēram, pārvaldīt konfigurācijas pārvaldības datubāzi vai ļauj lietotājam ievadīt informāciju brīdī, kad strādā kāds no transformācijas algoritmiem.

MTM pieejas galvenais mērķis ir parādīt, kā tiek veidots izejas kods konfigurācijas pārvaldības automatizācijai un kādi elementi ir iesaistīti koda veidošanas procesā. Kods īsteno un automatizē visus konfigurācijas pārvaldības procesa uzdevumus, un tas var būt izpildīts tikai no centralizētas vietnes: konfigurācijas pārvaldības servera. Attēlā 3.2. var redzēt, ka pieeja sastāv no trim daļām:

- **konfigurācijas pārvaldība projektā** – konfigurācijas pārvaldības serveris un izejas kods, kas tiek izpildīts no minētā servera procesu īstenošanai;
- **programmatūras izstrādes uzņēmums** – uzņēmums, kurā funkcionē *MTM* pieeja. Pieejas kontekstā svarīgākie elementi ir risinājumu datubāze un konfigurācijas pārvaldnieks. Risinājumu datubāze satur atkārtoti izpildāmus izejas koda fragmentus konkrētām konfigurācijas pārvaldības darbībām. Konfigurācijas pārvaldnieks administrē minētos koda fragmentus;
- **konfigurācijas pārvaldības ieviešanas process** – uz modeļiem balstīts process konfigurācijas pārvaldības izejas koda iegūšanai. «Modelis 1» satur vispārīgas no platformas neatkarīgas prasības konkrētam konfigurācijas pārvaldības procesam. Modelis transformējas citā modelī (Modelis 2), kas satur implementācijas detaļas konkrētai platformai. Transformācijas gaitā konfigurācijas pārvaldnieks izvēlas konkrētiem procesa posmiem atbilstošas implementācijas no risinājumu datubāzes. Tālāk no izveidotā modeļa automātiski ģenerē koda modeli – direktoriju un failu struktūru atbilstoši konkrētai platformai un programmēšanas valodai. Pieejas ieviešana noslēdzas tad, kad koda modeli implementē konfigurācijas pārvaldības serverī.



3.2. att. MTM pieejas elementi un saites

3.2. *EAF* metodoloģija *MTM* pieejas realizācijai

MTM vispārīgas konfigurācijas pārvaldības modeļvadāmas pieejas realizācijai tika izstrādāta metodoloģija *EAF* (angļu val. *Environment – Action – Framework*). *EAF* metodoloģija attēlo konfigurācijas pārvaldības procesu vairākos līmeņos, katram līmenim atbilst modelis ar noteiktu abstrakciju.

Metodoloģijas mērķis ir definēt konfigurācijas pārvaldības procesu ieviešanas soļus un sniegt iespēju jaunajos procesos izmantot jau esošos risinājumus. Risinājumu atkārtota izmantošana ļauj samazināt procesu ieviešanas laiku un mazina neparedzētu kļūdu risku, kas rodas gadījumos, kad visi risinājumi ir jāizstrādā no nulles.

EAF metodoloģijai ir šādi galvenie principi:

- *EAF* pamatuzdevums ir implementēt konfigurācijas pārvaldības procesu konkrētajā programmatūras izstrādes projektā;
- konfigurācijas pārvaldības procesa implementācijas galvenais nodevums ir konkrētā operētājsistēmā izpildāms izejas kods, kas pilnībā pārvalda konfigurācijas pārvaldības procesu, risinot visus nepieciešamus procesa uzdevumus;
- izejas kods konfigurācijas pārvaldības procesam tiek ģenerēts automātiski, lai mazinātu cilvēcisko faktoru;
- konfigurācijas pārvaldības izejas kodā nav iešūtas nekādas absolūtas vērtības (angļu val. *hardcodes*). Visas absolūtas vērtības glabājas mainīgajos un konstantēs atbilstoši konkrētai programmēšanas tehnoloģijai.

- *EAF* metodoloģijas izstrādes gaitā tika ieviesti šādi jēdzieni:
    1. **projekts** – programmatūras izstrādes projekts, kurā tiek aprakstīta konfigurācijas pārvaldība;
    2. **kompānija** – konkrēts uzņēmums, kas realizē programmatūras izstrādes projektus;
    3. **konfigurācijas pārvaldnieks (*Configuration Manager*)** – lietotājs, kas izmantojot *EAF* metodoloģiju modelē un implementē konfigurācijas pārvaldības procesu projektā;
    4. **konfigurācijas pārvaldības risinājumu glabātuve (*SCMWarehouse*)** – struktūra, kur glabājas visi konfigurācijas pārvaldības risinājumi kompānijas ietvaros;
    5. **konfigurācijas pārvaldības risinājumu glabātuves pārvaldības sistēma** – aplikācija, kas pārvalda informāciju glabātuvē *SCMWarehouse*;
    6. **platforma (*Platform*)** – konkrēta operētājsistēma, kurā tiek implementēts konfigurācijas pārvaldības procesa izejas kods;
    7. **konfigurācijas pārvaldības serveris (*SCMServer*)** – centralizēta vietne, no kuras tiek pārvaldīta konfigurācijas pārvaldības izejas koda izpilde. Serveris ir orientēts uz kādu konkrētu platformu. Par pamatu konfigurācijas pārvaldības serverim var izmantot kādu no esošajiem nepārtrauktas integrācijas serveriem, kuriem jau ir iebūvēta funkcionalitāte galveno konfigurācijas pārvaldības uzdevumu risināšanai. Piemēri: *Jenkins, Hudson, Bamboo, CruiseControl* u. c.;
    8. **vide (*Environment*)** – infrastruktūras kopa, kurā atrodas izstrādājama programmatūra (aplikāciju serveri, datubāzes, ārēju sistēmu interfeisi utt.). Katra vide ir paredzēta konkrētai aktivitātei programmatūras izstrādes dzīves ciklā, piemēram, izstrādei, testēšanai, kvalitātes akcepttestēšanai, ekspluatācijai utt. Vides nosaukums parasti ietver informāciju par nozīmi un lietošanu projektā. Piemēram, *DEV* vide – paredzēta izstrādei, *TEST* – testēšanai utt.
    9. **darbība (*Action*)** – aktivitāte konfigurācija pārvaldības procesā. Parasti aktivitāte atrisina kādu no galvenajiem konfigurācijas pārvaldības uzdevumiem, piemēram: programmatūras būvējuma veidošana, izejas koda pārvaldība, programmatūras instalācija kādā no vidēm utt.
- izejas koda ģenerēšanas process izmanto atkārtotas lietošanas principu. Veidojot izejas kodu, visur, kur vien iespējams, tiek izmantotas izejas koda vienības, kas ir lietojamas arī citos programmatūras izstrādes projektos. Lietotājam tiek piedāvāts

definēt tikai tās izejas koda vienības, kas ir specifiskas konkrētam projektam. Lai varētu realizēt atkārtotas lietošanas principu, *EAF* metodoloģija paredz, ka visi kompānijā esošie konfigurācijas pārvaldības risinājumi glabājās noteiktā struktūrā (*SCMWarehouse*) un ir pieejami līdzvērtīgi visiem kompānijas projektiem.

3.3. *EAF* metodoloģijas izstrādes pamatojums

EAF metodoloģijas izstrādes gaitā nācās vēlreiz atgriezties pie otrās nodaļas pētījuma. Analizējot izpētītos konfigurācijas pārvaldības modeļvadāmos risinājumus [PIN 2009, GIE 2009, BUC 2009, CAL 2012] un to trūkumus, nācās ņemt vērā, ka modeļvadāmajā risinājumā jābūt dažāda līmeņa abstrakcijas modeļiem un skaidri definētiem likumiem, kā pāriet no viena abstrakcijas līmeņa modeli uz citu. Tāpēc pirmais uzdevums metodoloģijas izstrādes gaitā ir saprast, kādi modeļi ir nepieciešami, lai atbalstītu konfigurācijas pārvaldības procesu. Risinot šo uzdevumu, tika apvienotas *MDA* [SIN 2010] galvenās idejas un šā darba pētījums par konfigurācijas pārvaldību [AIE 2010, BER 2003, DEP 2010, PAU 2007, MET 2002, KAN 2005, CON 2002, GLO 2012, BRU 2004, DAR 2001, WES 2005, MEL 2006, BEL 2005, VAC 2006, WIK 2013, ОРЛ 2011, ЛАП 2004, УДО 2011, ЗАМ 2008].

MDA [SIN 2010] definē trīs modeļus:

- *CIM* – no skaitļošanas neatkarīgs modelis;
- *PIM* – no platformas neatkarīgs modelis;
- *PSM* – platformas specifiskais modelis.

Visaugstākais abstrakcijas līmenis ir *CIM* modelim. Pētot konfigurācijas pārvaldības definīcijas, tika meklēts tas, kas varētu veidot konfigurācijas pārvaldības modeli, kam būtu augtākais iespējamais abstrakcijas līmenis. Apkopojot [AIE 2010, BER 2003, DEP 2010, PAU 2007, MET 2002, KAN 2005, CON 2002, GLO 2012, BRU 2004, DAR 2001, WES 2005, MEL 2006, BEL 2005, VAC 2006, WIK 2013, ОРЛ 2011, ЛАП 2004, УДО 2011, ЗАМ 2008] definīcijas, izdevās iegūt atziņu, ka konfigurācijas pārvaldība savā visaugstākajā abstrakcijas līmenī atbild uz šādu jautājumu: «Kā pārnest programmatūrā veiktas noteiktās izmaiņas no vienas instances uz citu vajadzīgajā brīdī?». Galvenais uzsvars šajā jautājumā tiek likts uz izmaiņām, kas veiktas izstrādājamā produktā, un spēju tās pārnest no vienas instances uz otru, jo parasti programmatūras izstrādes projektā tiek izmantotas vairākas instances [AIE 2010]. Konfigurācijas pārvaldības definīciju pētījuma gaitā izdevās attiecināt

galvenos uzdevumus pret konkrētām, augstākminētajām, jautājuma daļām [AIE 2010, BER 2003, DEP 2010]. Apkopojums ir redzams tabulā 3.1.

3.1. tabula

Konfigurācijas pārvaldības uzdevumi un atbilstošie jautājumi

Uzdevums	Atbilstošā jautājuma daļa
Konfigurācijas identifikācija	Ko pārnest?
Konfigurācijas kontrole	Kādas versijas pārnest?
Kompilācijas un būvējumu pārvaldība	Kā aktivizēt pārnestās izmaiņas citā instancē?
Konfigurācijas statusu uzskaite	Kā noteikt vajadzīgo brīdi, kad nepieciešams pārnest konfigurāciju?

Veidojot konfigurācijas pārvaldības modeli ar augstāku abstrakcijas līmeni, jāapzinās, kādas instances eksistē patlaban un kādu ceļu noiet programmatūras izstrādes izmaiņas, sākot ar izstrādi un beidzot ar ekspluatāciju, kā arī – kādas izstrādājamā produkta versijas tiek veidotas. Šāda veida modelis saskaņā ar izstrādāto metodoloģiju tiek nosaukts par vižu modeli, turpmāk tekstā tiks apzīmēts ar *EM* (angļu val. *Environment Model*). Savukārt katram modelim ir jābūt meta-modelim, tāpēc tika ieviests elements «Vižu modeļa meta-modelis», kas tiek apzīmēts ar *Environment meta-model*. Sākotnējā stadijā konfigurācijas pārvaldniekam, kas plāno procesu, tiek piedāvāts izveidot vižu modeli, izmantojot atbilstošā meta-modeļa elementus.

Brīdī, kad ir iegūts konfigurācijas pārvaldības procesa vižu modelis un apzināti riski, ir jāveido modelis ar zemāku abstrakcijas līmeni. Taču pirms tam ir jāsaprot, kāds tieši varētu būt nākamais modelis. Modeļa būtību palīdzēja definēt atziņa no avota [CAL 2012], kurā ir teikts, ka, neskatoties uz to, kādā veidā tiek plānots konfigurācijas pārvaldības process, beigās to tāpat atbalsta dažādi rīki, kas veic noteiktas darbības, lai realizētu galvenos konfigurācijas pārvaldības uzdevumus. Vižu modelis parāda to, kādu ceļu noiet veiktās programmatūras izmaiņas. Ņemot vērā [CAL 2012] minēto atziņu, būtu loģiski šajā stadijā saprast, kāda veida darbības būtu jāveic, lai realizētu vižu modelī aprakstītu ceļu atbilstoši konfigurācijas pārvaldības principiem [AIE 2010, BER 2003, DEP 2010]. Līdz ar to metodoloģijā tiek ieviests elements «No platformas neatkarīgs darbību modelis», kas turpmāk tiks apzīmēts ar *PIAM* (angļu val. *Platform Independent Action Model*). Arī šim modelim ir konkrēts avots jeb meta-modelis «No platformas neatkarīga darbību modeļa avots», to apzīmē ar *PIAM meta-model*.

Lai iegūtu *PIAM* modeli no vižu modeļa, *EAF* metodoloģijā tika ieviests elements «E->P». Šis elements satur transformācijas likumus, kas operē ar *EM* un *PIAM* meta-modeļu elementiem un automātiski iegūst *PIAM* modeli no gatava *EM* modeļa.

PIAM modelī ir uzskaitītas visas darbības, kas nepieciešamas, lai pārnestu programmatūras izmaiņas starp vidēm, kas ir definētas *EM* modelī. Darbības šajā modelī ir abstraktas. Informāciju par darbību implementāciju satur nākamais modelis *EAF* ietvaros: platformas specifiskais darbību modelis. Turpmāk modelis tiks apzīmēts ar *PSAM* (angļu val. *Platform Specific Action Model*). Šis modelis ir *PIAM* modeļa paplašinātais variants, kurā ir norādītas realizācijas detaļas katrai darbībai: platforma, konkrēti rīki, skripti, instalācijas instrukcijas utt.

3.4. *EAF* metodoloģijas modeļu apraksts

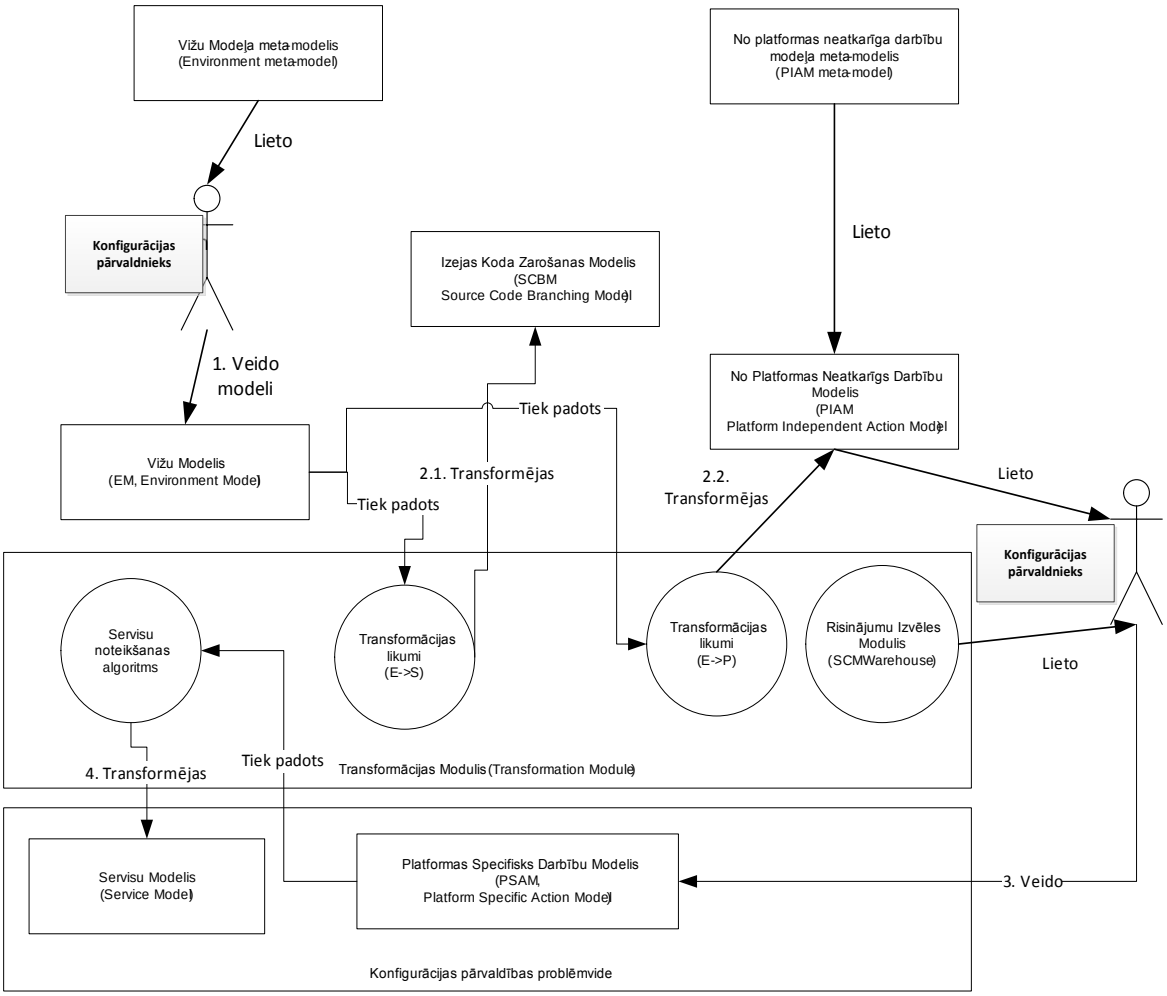
EAF metodoloģijā ir šādi elementus:

- **vižu modeļa meta-modelis** – modelēšanas valoda vižu modeļa veidošanai;
- **vižu modelis (*EM*)** – konfigurācijas pārvaldības procesa modelis, kas attēlo visas konkrēta projekta vides, starp kurām notiek programmatūras izmaiņu pārnešana;
- **izejas koda zarošanas modelis (*SCBM*)** – modelis, kas ilustrē programmatūras izejas koda pārvaldības likumus atkarībā no vižu modeļa, parāda kādi izejas koda zari atbilst kādām vidēm un kādā veidā notiek izejas koda izmaiņu pārnešana (angļu val. *merge*) starp dažādiem zariem;
- **no platformas neatkarīga darbību modeļa meta-modelis** – modelēšanas valoda *PIAM* modeļa veidošanai;
- **no platformas neatkarīgs darbību modelis (*PIAM*)** – modelis, kas parāda, kādas darbības ir nepieciešams veikt, lai pārnestu programmatūras izmaiņas starp instancēm vižu modelī. Šajā modelī darbības nesatur nekādas implementācijas detaļas un nav atkarīgas no jebkādām platformām;
- **platformas specifiskais darbību modelis (*PSAM*)** – paplašinātais variants *PIAM* modelim. Atšķirībā no *PIAM* šis modelis satur visu informāciju par darbību implementāciju: platformu, konkrētus rīkus, skriptus, instrukcijas;
- **servisu modelis (*Service Model*)** – modelis, kas attēlo rīku savstarpēju integrāciju. Modelī ir rīku pāri. Katram rīkam, kas atrodas konkrētajā pārī, ir funkciju vai metožu kopa, ko var izsaukt par otro pāra rīku. Servisu modelis ir vajadzīgs dažādu rīku

integrācijas aprakstam. *PSAM* modelī var redzēt, kādi rīki ir nepieciešami konfigurācijas pārvaldības darbību analīzei, savukārt servisu modelis parāda, kā rīki savā starpā sadarbojas (integrējas), lai varētu uzturēt pilnvērtīgu konfigurācijas pārvaldības darbību plūsmu;

- **servisu noteikšanas algoritms** – algoritms, kas atkarībā no rīkiem *PSAM* modelī nosaka rīku pārus jeb servisus. *PSAM* modeļa implementācijas laikā konfigurācijas pārvaldniekam sākumā jārealizē servisi, ko noteiks servisu noteikšanas algoritms;
- **transformācijas likumi «E->S» –** likumu kopa, kas operē ar vižu modeli un sagatavo atbilstošu izejas koda zarošanas modeli;
- **transformācijas likumi «E->P» –** likumu kopa, kas operē ar vižu modeli un sagatavo atbilstošu *PIAM* modeli;
- **risinājumu izvēles modulis (*SCMWarehouse*) –** glabātuve, kur atrodas visi kompānijā esoši konfigurācijas pārvaldības risinājumi.

Attēlā 3.3. var redzēt *EAF* metodoloģijas vispārīgo shēmu. Ar bultiņām ir apzīmētas darbības un metodoloģijas galvenie soļi. «*Configuration Manager*» ir lietotājs, kas ievieš kompānijā konfigurācijas pārvaldības procesus un veido dažādus modeļus saistībā ar šo metodoloģiju.



3.3. att. *EAF* metodoloģijas vispārīga shēma

EAF metodoloģijā ir četri galvenie soļi. Paredzēts, ka kompānijā ir izstrādāts risinājumu izvēles modulis, kas strukturētā veidā glabā visus kompānijas risinājumus konfigurācijas pārvaldības darbībām. Konfigurācijas pārvaldnieks veido vižu modeli konkrētam programmatūras izstrādes projektam, pēc tam nostrādā transformācijas likumi «E->S» un «E->P». Rezultātā tiek iegūti *SCBM* un *PIAM* modeļi. Šajā brīdī lietotājs zina, kādi izejas koda repozitorija zari ir jāveido, lai uzturētu izejas koda bāzi katrai videi no vižu modeļa. Ir zināmas arī konfigurācijas pārvaldības darbības, kas nepieciešamas, lai realizētu visas programmatūras izmaiņu plūsmas starp vidēm. Redzot darbības *PIAM* modelī, konfigurācijas pārvaldnieks no risinājumu izvēles moduļa izvēlas vienu konkrētu risinājumu katrai darbībai. Rezultātā *PIAM* modelis paplašinās līdz *PSAM* modelim, kas satur informāciju par platformām, rīkiem, skriptiem utt. Tālāk *PSAM* modeli apstrādā servisu noteikšanas algoritms, kas nosaka rīku pārus integrācijai. Visbeidzot – gan rīku integrācija,

gan *PSAM* modelis tiek implementēti projekta konfigurācijas pārvaldības problēmvidē. Tabulā 3.2. var redzēt *EAF* metodoloģijas soļu detalizētu aprakstu.

3.2. tabula

***EAF* metodoloģijas soļu apraksts**

Solis, nosaukums	Apraksts
1.Veido modeli	Konfigurācijas pārvaldnieks veido vižu modeli konkrētam projektam, izmantojot vižu modeļa meta-modeli. Rezultātā izveidojas gatavs vižu modelis konkrētam programmatūras izstrādes projektam.
2.1.Transformējas	Gatavs vižu modelis tiek padots «E->S» transformācijas likumiem, un tiek izveidots izejas koda zarošanas modelis.
2.2.Transformējas	Gatavs vižu modelis tiek padots «E->P» transformācijas likumiem, un tiek izveidots no platformas neatkarīgs darbību modelis. Modelī ir redzamas visas nepieciešamas darbības, kas ir jāveic vižu modeļa implementācijai.
3.Veido	Konfigurācijas pārvaldnieks, izmantojot *PIAM* modeli un risinājumu izvēles moduli, izvēlas platformu un konkrētu risinājumu katrai *PIAM* modeļa darbībai. Rezultātā izveidojas *PSAM* modelis, kas satur visu nepieciešamu informāciju par konfigurācijas pārvaldības darbību implementāciju.
4.Transformējas	Gatavs *PSAM* modelis tiek padots servisu noteikšanas algoritmam, kas nosaka nepieciešamus servisus jeb rīku pārus, kam jābūt integrētiem, lai implementētu konkrētu *PSAM* modeli.

3.5. Vižu modeļa meta-modeļa izstrāde

Sākot vižu modeļa modelēšanas valodas izstrādi, tika izpētīta literatūra par meta-modeļu un domēnu specifisku valodu izstrādes pamatprincipiem [PAI 1999, MER 2005, KEL 2008, WIL 2004, PFA 1997, GRO 2007, KAR 2009, WIL 2003, TOL 2005]. Pētījuma mērķis bija saprast galvenos principus, veidojot jaunu meta-modeli vai domēna valodu kāda noteikta problēmapgabala modelēšanai.

Pētījumā [KAR 2009], ar kuru promocijas darba autors iepazinās vispirms, uzreiz tika uzsvērta problēma, ka mūsdienās pastāv ļoti daudz rīku un programmatūru, kas palīdz veidot jaunās modelēšanas jeb domēna specifiskās valodas, taču rīki piedāvā vien tehnisku atbalstu, nesniedzot konkrētas rekomendācijas, kādai jābūt valodai un kādi pamatprincipi ir jāievēro. Līdz ar to turpmākajā pētījumā tika nolemts nepievērst pārāk lielu uzmanību kādai konkrētai valodai, bet atrast pēc iespējas vairāk valodas veidošanas vispārīgos principus, izplatītu kļūdu aprakstu un arī praktisku padomu. Apkopojot informāciju avotos [PAI 1999, MER 2005, KEL 2008, WIL 2004, PFA 1997], izdevās atrast rekomendācijas modelēšanas valodas izstrādei. Pirmkārt, jebkuras jaunās valodas izstrādes procesā jābūt šādiem posmiem [PFA 1997]:

- *valodas mērķis.* Pirmais valodas izstrādes posms, kurā izstrādātājam jāatbild uz jautājumiem, ko spēs modelēt jauna valoda, kādai videi tā ir paredzēta, kurš veidos modeļus, kurš būs tas, kas modeļus lietos, kurš nodarbosies ar modeļu validāciju utt. Līdzīgas atziņas ir atrodamas arī citās publikācijās [MER 2005, KEL 2008, WIL 2004]. Galvenā doma ir: valodas izstrādei nav jēgas, ja nav pilnīgi skaidrs konkrēts mērķis un lietojums;
- *valodas saturs.* Šajā posmā ir jādefinē valodas elementi. To var izdarīt, ja ir zināmi valodas mērķi, jo katrā mērķa formulējumā ir lietvārdi, kas var tieši vai netieši norādīt uz to, kādi elementi ir nepieciešami. Piemēram, ja konfigurācijas pārvaldības vižu modeļa mērķis ir parādīt konfigurācijas vienumu plūsmu pa instancēm noteiktā laika momentā, tad visticamāk parādīsies šādi koncepti: instance, plūsma, konfigurācijas vienums, laika moments jeb notikums utt. Vairākās publikācijās par domēna valodas izstrādēm [PAI 1999, MER 2005, KEL 2008, WIL 2004, PFA 1997] ir atzīmēts, ka šī posma beigās svarīgi pārbaudīt, vai koncepti tiešām ļauj sasniegt visus valodas mērķus;

- *valodas konkrēta sintakse.* Šajā posmā ir jāizstrādā valodas sintakse, kas būs pieejama valodas lietotājiem modelēšanas gaitā. Publikācijā [GRO 2007] ir sniegtas vairākas rekomendācijas par to, kā veidot lietotājam pieejamu sintaksi, tajā skaitā – elementus, operācijas ar tiem utt. Viena no svarīgākajām rekomendācijām šajā publikācijā un dažādos citos avotos pauž viedokli, ka par pamatu jaunai valodai jāizmanto kāda esoša notācija, nepieciešamības gadījumā to var papildināt ar jauniem elementiem vai operācijām ar tiem. Izmantojot esošu notāciju, ir iespējams kādam elementam mainīt nosaukumu. Vairāki modelēšanas valodas izstrādātāji un pētnieki [PAI 1999, MER 2005, KEL 2008, WIL 2004, PFA 1997, GRO 2007, KAR 2009, WIL 2003, TOL 2005] neiesaka šajā izstrādes stadijā veidot pilnīgi jaunu notāciju, ja vien tam nav stingra pamatojuma. Kā galveno argumentu pētnieki min faktu, ka katrā problēmapgabalā visticamāk jau ir kādas esošās notācijas, elementi ko lieto problēmapgabala eksperti. Gadījumā, ja tiek ieviesta pilnīgi jauna notācija, kurā ekspertiem labi pazīstami elementi tiks apzīmēti citādāk, tiks apgrūtināta jaunās valodas uztvere. Vēl viena būtiska rekomendācija [GRO 2007, KAR 2009] ir: viennozīmīgi noteikt, vai jaunā valoda būs grafiskā vai teksta valoda. Autori neiesaka izmantot kombinētu variantu, kas varētu radīt pretrunas un apgrūtināt valodas uztveri. Starp visām rekomendācijām vēl ir uzsvērtas: komentāru iespējas, nosaukumu veidošanas likumu esamība (angļu val. *naming conventions*) utt.;
- *valodas abstraktā sintakse.* Valodas izstrādes noslēdzošais posms. Šajā posmā galvenais uzdevums ir radīt iespēju iepriekšēja posmā izstrādātu sintaksi apstrādāt ar datora palīdzību. Risinājumam iespēju robežās jābūt neatkarīgam no platformas un tādā formātā, kuru būtu iespējams apstrādāt ar dažādu tehnoloģiju palīdzību [GRO 2007, KAR 2009].

a. Vižu modeļa meta-modeļa mērķis

Meta-modelim jāveido vižu modeļi, kuru mērķis ir parādīt visas programmatūras izmaiņu plūsmas pa vidēm. Katra plūsma, pa kuru programmatūras vienumu versijas nonāk no vienas vides uz otru, pieder konkrētam notikumam. Vienam notikumam var piederēt vairākas plūsmas. Piemēram, ja notikums ir «Pārnest konfigurāciju no vides *DEV* uz instanci *TEST*», tad – iespējams – šajā notikumā būs iesaistītas divas plūsmas. Vienā plūsmā konfigurācija tiks pārnesta uz instanci *TEST1*, lai pārliecinātos, ka konfigurācija ir korekta, bet nākamajā plūsmā tā pati konfigurācija tiks pārnesta uz *TEST* instanci. Līdz ar to modelim vajadzētu parādīt visus notikumus, kas ir iesaistīti konfigurācijas pārnešanā starp dažādām

instancēm, visas notikumu plūsmas, to secību un visas instances, kas glabā jebkāda veida programmatūras konfigurāciju. Vižu modeli veidos konfigurācijas pārvaldnieks, kuram būs iespēja pievienot, mainīt un dzēst notikumus, plūsmas un vides.

b. Vižu modeļa meta-modeļa vispārīga struktūra un darbības principi

Meta-modelim ir šādi elementi:

- grafisku elementu kopa, kas ir pieejama lietotājam vižu modeļa veidošanai;
- modeļa elementu *XML* interpretācija un elementu hierarhija;
- algoritms, kas pārveido vižu modeli no grafiskā uz *XML* formātu;
- vižu modeļa elementu klases un kompilācijas algoritms.

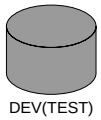
Vižu modeļa veidošana notiek šādi. Lietotājam ir pieejami grafiski elementi, no kuriem viņš veido vižu modeli konkrētam projektam. Pēc tam, kad lietotājs uzskata, ka modeļa veidošana ir pabeigta, nostrādā algoritms, kas pārveido grafisku modeli *XML* formātā. Ņemot vērā zināmu modeļa elementu notāciju un elementu hierarhiju, pārveidošanas algoritms pārmeklē katru vizuālu elementu un pārveido to *XML* formātā. Atbilstoši elementu hierarhijai tiek sastādīts *XML* dokuments. Meta-modelim ir speciālas klases, kas realizē vižu modeli. Katram elementam ir noteikta klase, kas satur vismaz elementu pievienošanas un iegūšanas metodes. Pievienošanas metodēs ir realizētas pārbaudes, kas ir nepieciešamas, lai pārbaudītu, vai izveidotais modelis ir korekts. Kompilācijas algoritms analizē iepriekšējā solī iegūtu *XML* dokumentu un veido elementu klašu objektus, izsaucot speciālas «*add*» metodes. Ja kādā solī pievienošana neizdodas, piemēram, modelis satur divas instances ar vienādu nosaukumu, kas nav pieļaujams, tad kompilācijas algoritms izveido kļūdas paziņojumu un piedāvā lietotājam labot modeļa grafisku variantu, vēlreiz veikt pārveidošanu *XML* formātā un modeļa kompilāciju. Šajā brīdī visi iepriekšēji modeļa klašu objekti tiek dzēsti, un pēc modeļa rediģēšanas kompilācija tiek uzsākta no nulles. Veiksmīgas kompilācijas gadījumā datora atmiņā glabājas vižu modelis, kas ir gatavs nodošanai transformācijas likumiem.

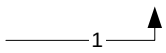
c. Meta-modeļa grafiski elementi un modeļu piemēri

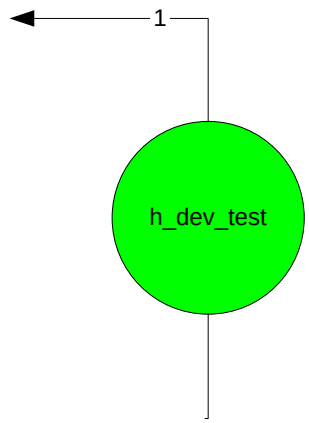
Tabulā 3.3. ir dots apraksts visiem vižu modeļa meta-modeļa elementiem. Katram elementam ir noteikti atribūti, kuriem jābūt aizpildītiem modelēšanas gaitā, citādāk pārveidošanas algoritms nevarēs izveidot *XML* dokumentu.

3.3. tabula

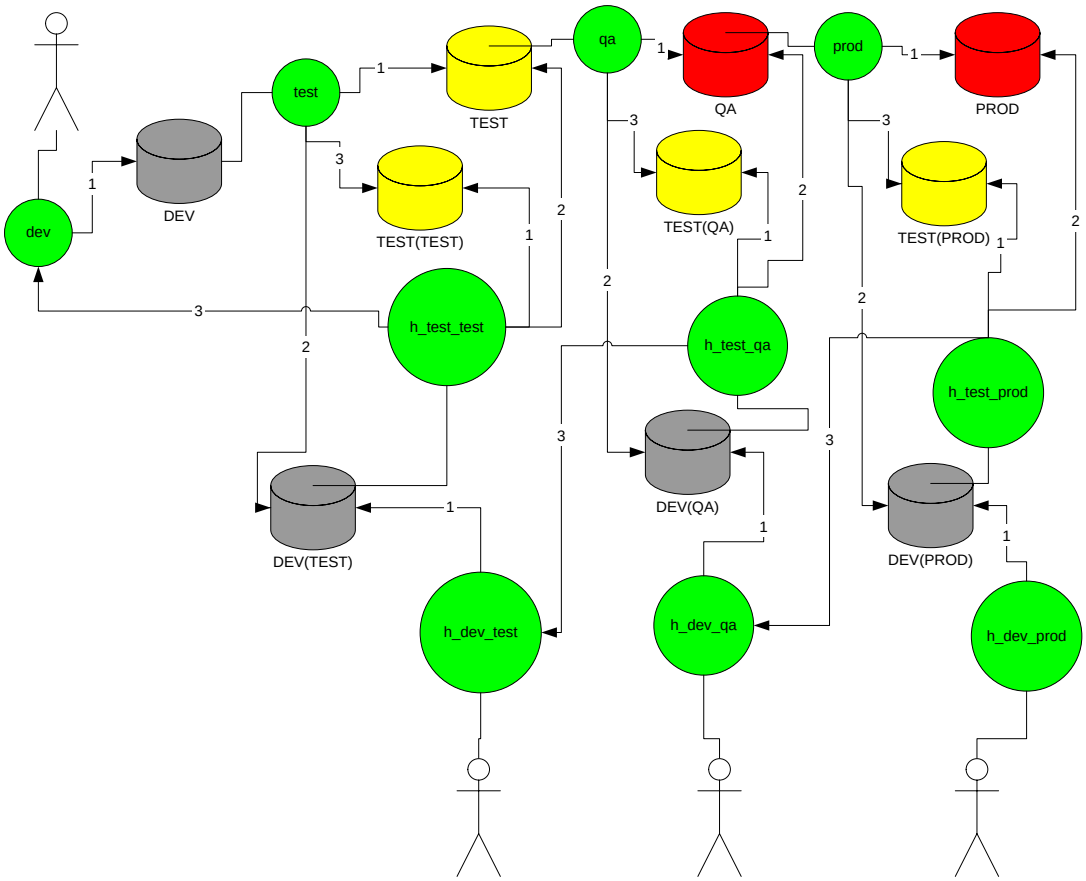
Vižu modeļa elementi un to apraksts

Nosaukums/Izskats	Atribūti un to apraksts
Aktieris (*Actor*) 	Programmatūras izstrādātājs, kas veic izmaiņas programmatūras vienumos. Aktierim ir šādi atribūti: **Nosaukums (*Name*)** **Apraksts (*Description*)** Veidojot jaunu aktieri vižu modelī, lietotājam uzreiz tiek piedāvāts aizpildīt šos atribūtus.
Vide (*Environment*) 	Instance jeb vide, kurā atrodas izstrādājama programmatūra. Šim elementam ir šādi atribūti: **Nosaukums (*Name*)** – vides nosaukums. Nosaukumam jābūt unikālam, vienā modelī nedrīkst būt vairāk par vienu vidi ar vienādiem nosaukumiem **Apraksts (*Description*)** **Pazīme, kas nosaka, vai vidi uztur pasūtītājs (*CustumerSupportFlag*)** – pazīme, kas nosaka vai konkrētu instanci uztur pasūtītājs vai izpildītājs. Ja vidi uztur pasūtītājs, tad izstrādātāju komandai nav pieejas, lai mainītu programmatūras konfigurāciju. Šajā gadījumā, lai nomainītu konfigurāciju, izstrādātājs sagatavo un nodod pasūtītājam laidienu, kuru izpildot, ir iespējams ieviest konkrētas izmaiņas vidē **Izstrādes vides pazīme (*DevelopmentFlag*)** – pazīme, kas nosaka, vai tā ir vai nav izstrādes vide. Izstrādes vidē aktieris (izstrādātājs) drīkst veikt izmaiņas brīvā veidā, visās citās vidēs izmaiņas nonāk tikai ar laidieniem **Oriģinālas vides pazīme.** Vižu modelī ir divi vižu veidi. Pirmais veids ir oriģinālas vides, ko lieto projekta konkrētiem mērķiem, tādiem kā izstrāde, testēšana, akcepttestēšana, ekspluatācija. Otrais veids ir vižu kopijas jeb neoriģinālās vides. Katra no šīm vidēm pēc būtības ir kādas oriģinālās vides kopija. Neoriģinālās vides jeb

	kopijas ir vajadzīgas, lai notestētu programmatūras izmaiņu pārnešanu starp vidēm. Praksē tas izpaužas, piemēram, tad, kad uz ekspluatācijas vides jāuzliek kārtējais laidiens ar izmaiņām programmatūrā, taču pirms tam to pašu laidienu instalē uz ekspluatācijas precīzas kopijas, lai pārliecinātos, ka laidiens ir kvalitatīvs un pēc instalācijas īstajā ekspluatācijas vidē, negaidītas problēmas nerādīsies. **Oriģinālas vides nosaukums (*OriginalEnvironmentName*)** – ja vide nav oriģināla, šeit norāda atbilstošas oriģinālas vides nosaukumu
Konfigurācijas vienumu plūsma (*ConfigurationItemFlow*) 	Apzīmē ceļu, pa kuru programmatūras izmaiņas nonāk no vienas vides uz otru vai no aktiera uz vidi, gadījumā, ja notiek izstrāde. Atribūti: **Nosaukums (*Name*)** – teksts, kas sniedz nelielu aprakstu par to, no kuras uz kuru vidi nonāk konfigurācijas vienumi **Kārtas numurs (*Sequence*)**. Katra plūsma pieder kādam konkrētam notikumam. Vienam notikumam var būt vairākas plūsmas. Atribūts parāda plūsmas kārtas numuru notikuma ietvaros **Avots (*Source*)** – vide vai aktieris, no kura nonāk programmatūras izmaiņas **Mērķa vide (*Goal*)** – vide, kur nonāk programmatūras izmaiņas **Apraksts (*Description*)** – papildu informācija par konkrētu plūsmu
Notikums (*Event*)	Apzīmē notikumu, kurā programmatūras izmaiņas nonāk no vienas vides uz citām. Notikumam ir unikāls nosaukums, apraksts un vismaz viena konfigurācijas plūsma. Tādējādi viena notikuma konfigurācijas vienumi var nonākt no vienas vides uz citām vidēm. Visām viena notikuma plūsmām ir vienāda avota vide, savukārt mērķa

	vides atšķiras. Viena konfigurācijas plūsma var piederēt tikai vienam notikumam. Notikuma atribūti ir šādi: **Nosaukums (*Name*)** **Plūsmas (*ConfigurationItemFlows*)** – norādes uz eksistējošām konfigurācijas vienumu plūsmām (*ConfigurationItemFlow*) **Apraksts (*Description*)** – papildu informācija par konkrēto notikumu **Tiek pārnestas visas pieejamas izmaiņas (*AllChangesMoveFlag*)** – atribūts, kas norāda, vai tiek pārnestas visas avota vidē pieejamas izmaiņas, vai tikai kādas noteiktas

Attēlā 3.4. ir redzams vižu modelis ar trim paralēlām izstrādes plūsmām. Tas nozīmē, ka pie šāda modeļa ir iespēja izstrādāt un testēt ne tikai produkta jaunu versiju, bet arī mainīt versijas, kas atrodas testa vidē, pasūtītāja akcepttesta vidē un ekspluatācijas jeb produkcijas vidē.



3.4. att. Vižu modelis ar trim izstrādes plūsmām

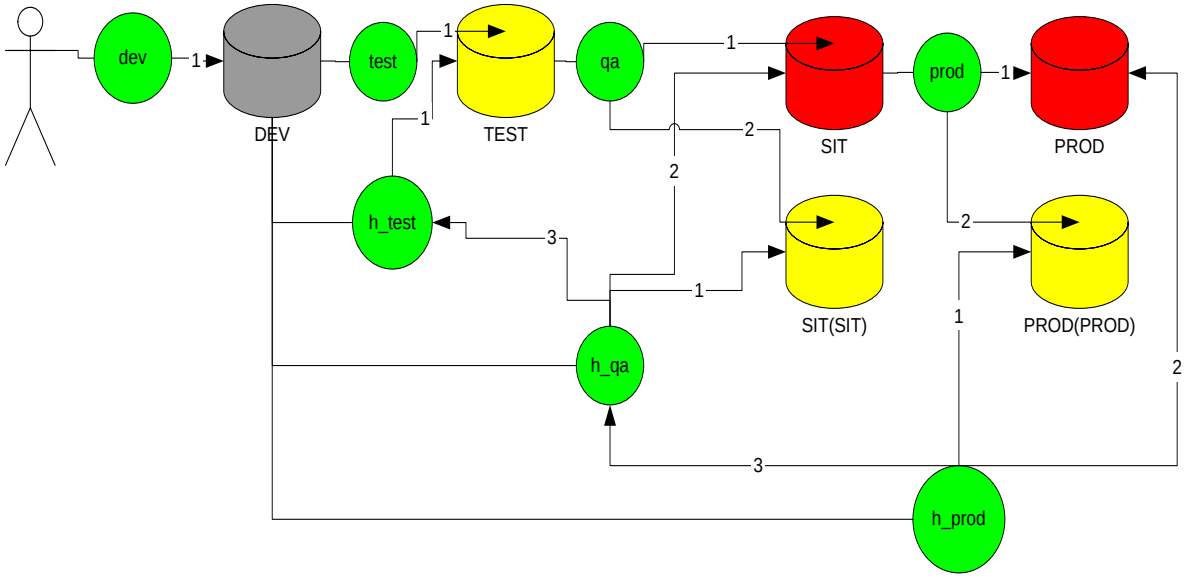
Modelī, kas redzams 3.3. attēlā, parādīta izstrādes galvenā plūsma *DEV -> TEST -> QA -> PROD*. Pa šo plūsmu virzās izstrādājama produkta jaunās versijas. Līdz ar to katrā no vidēm *TEST*, *QA* un *PROD* teorētiski var būt dažādas versijas. Piemēram, ekspluatācijas vidē *PROD* atrodas produkta versija 1.0., savukārt akcepttesta vidē *QA* pasūtītājs jau testē versiju 2.0. Izstrādātāja testa vidē *TEST* notiek 3.0. versijas testēšana un papildināšana. Projektā, kas ir aprakstīts minētajā vižu modelī, jaunā versija iznāk reizi nedēļā. Ja, piemēram, pasūtītājs konstatē kļūdu *PROD* vidē, to salabos produkta versijā 3.0., kas labākajā gadījumā tikai pēc divām nedēļām nonāks *PROD* vidē. Tas ir saistīts ar dažādiem faktoriem. Pirmkārt, pasūtītājs visticamāk negribēs pieņemt akcepttesta vidē versiju 3.0., kamēr līdz galam netiks notestēta versija 2.0., lai izvairītos no neparedzētiem gadījumiem. Otrkārt, ir vajadzīgs laiks, kamēr līdz galam notestēs un akceptēs 3.0. versiju gan testā, gan akcepttesta vidēs. Taču problēma *PROD* vidē var būt pārāk kritiska, kas traucē produkta normālu darbību. Pieņemams, ka šādas problēmas risinājums pasūtītājam būs vajadzīgs krietni ātrāk nekā pēc divām nedēļām. Tāpēc projektā ir ieviesta un vižu modelī atspoguļota *PROD* vides uzturēšanas plūsma: *DEV(PROD) -> TEST(PROD) -> PROD. DEV(PROD)* vide programmatūras konfigurācijas ziņā ir vienāda

ar *PROD* vidi, kas nozīmē, ka vidē ir produkta versija 1.0. Ja pasūtītājs atklāj problēmu ekspluatācijas vidē, izstrādātājs novērš to *DEV(PROD)* vidē. Pēc tam izmaiņas tiek pārnestas uz vidi *TEST(PROD)*, kur tiek pārbaudīts, vai tās atrisina pieteiktu problēmu versijā 1.0. Vide *TEST(PROD)* konfigurācijas ziņā arī ir vienāda ar *PROD* vidi. Ja tests ir bijis veiksmīgs, tad tās pašas izmaiņas pārnes arī *PROD* vidē. Taču ar to nepietiek. Ir tā, ka problēma ir atrisināta versijā 1.0., taču *QA* vidē šajā pašā brīdī tiek testēta 2.0. versija. Tāpēc modelī ir redzams, ka viens no notikuma «*h_test_prod*» soļiem jeb plūsmām ir izsaukt notikumu «*h_test_qa*». Praksē tas nozīmē, ka izstrādātājam vajadzētu šo pašu problēmu atrisināt arī produkta versijā 2.0. Pretējā gadījumā, pārnesot *PROD* vidē versiju 2.0., iespējams tiks nograuts labojums versijai 1.0. Tas pats attiecas arī uz *TEST* vidi un versiju 3.0., jo tur arī ir jānodrošina atbilstošās izmaiņas. Visbeidzot izmaiņas jāveic arī *DEV* vidē, kur top jauna, potenciāli 4.0. produkta versija. Šāds vižu modelis un notikumi tajā vienmēr seko galvenajai plūsmai *DEV -> TEST -> QA -> PROD*. Ja kādā no vidēm notiek labojums, tad modelī ir paredzēti notikumi, kas šādus pašus vai līdzvērtīgus labojumus ievieš arī visās iepriekšējās vidēs. No konfigurācijas pārvaldības risku viedokļa [AIE 2010] šāda veida vižu modelis varētu būt tuvu ideālam, taču praksē izraisa papildu grūtības.

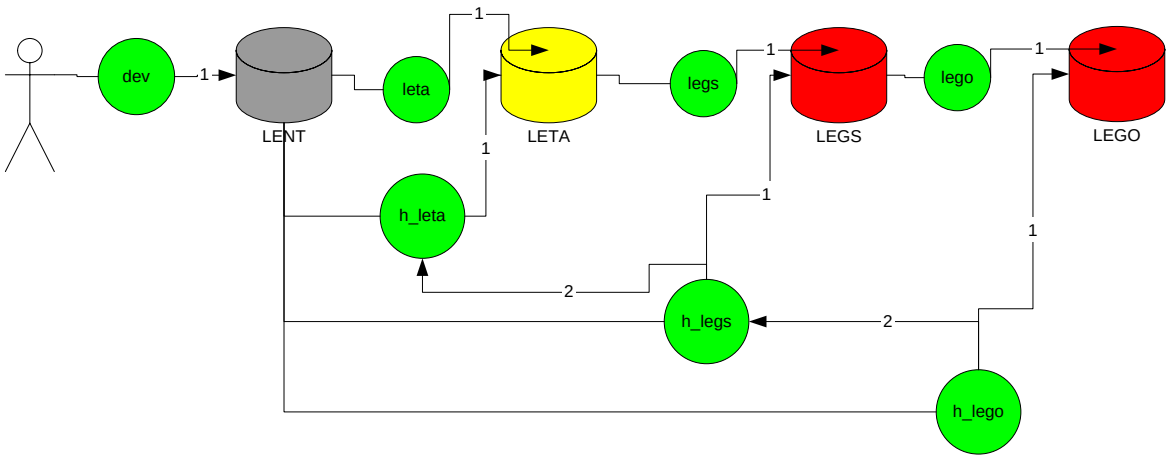
Pirmkārt, ir jānovērtē, cik daudz datorresursu (operatīva atmiņa, cietie diski utt.) ir nepieciešams vienas instances uzturēšanai. Vižu modelī, kas ir redzams attēlā 3.4., ir 10 instances. Praksē var būt tā, ka nepietiek resursu tik daudzu paralēlu vižu uzturēšanai.

Otrkārt, ir jānovērtē sistēmas stabilitāte gadījumā, kad projektā eksistē ekspluatācijas vide. Atkarībā no tā, cik bieži tiek atklātas kritiskās kļūdas ekspluatācijas laikā, kuru risinājums nevar gaidīt nākamo produkta versiju, jāpieņem lēmums, vai ir nepieciešama izstrādes plūsma *DEV(PROD) -> TEST(PROD) -> PROD*.

Treškārt, plūsmas *DEV(PROD) -> TEST(PROD) -> PROD, DEV(QA) -> TEST(QA) -> QA* un *DEV(TEST) -> TEST(TEST) -> TEST* var vienkāršot. Pieņemsim, var likvidēt vides *TEST(TEST), TEST(QA)* un *TEST(PROD)*, izmaiņas pa taisno liekot atbilstošajās oriģinālās vidēs. Šajā gadījumā strauji pieaug risks, ka izmaiņas netiks kvalitatīvi notestētas vai netiks pareizi pārnestas uz oriģinālu vidi. Vižu modeļa izveidošanas laikā tiek atspoguļoti vispārīgi riski konfigurācijas pārvaldības jomā [AIE 2010]. Sakarā ar to, ka vižu modelim nav nekādas tehnoloģiju specifiskās detaļas, ir iespēja koncentrēties tikai vispārīgam procesam un izvērtēt risku vēl pirms tam, kad sāksies jebkādi darbi ar produkta izstrādi vai uzturēšanu. Attēlos 3.5. un 3.6. ir redzami vižu modeļi no citiem projektiem, kas ir vienkāršoti, salīdzinot ar modeli attēlā 3.4., taču tajos ir vairāk konfigurācijas pārvaldības risku.



3.5. att. Projekta «X» vižu modelis



3.6. att. Projekta «Y» vižu modelis

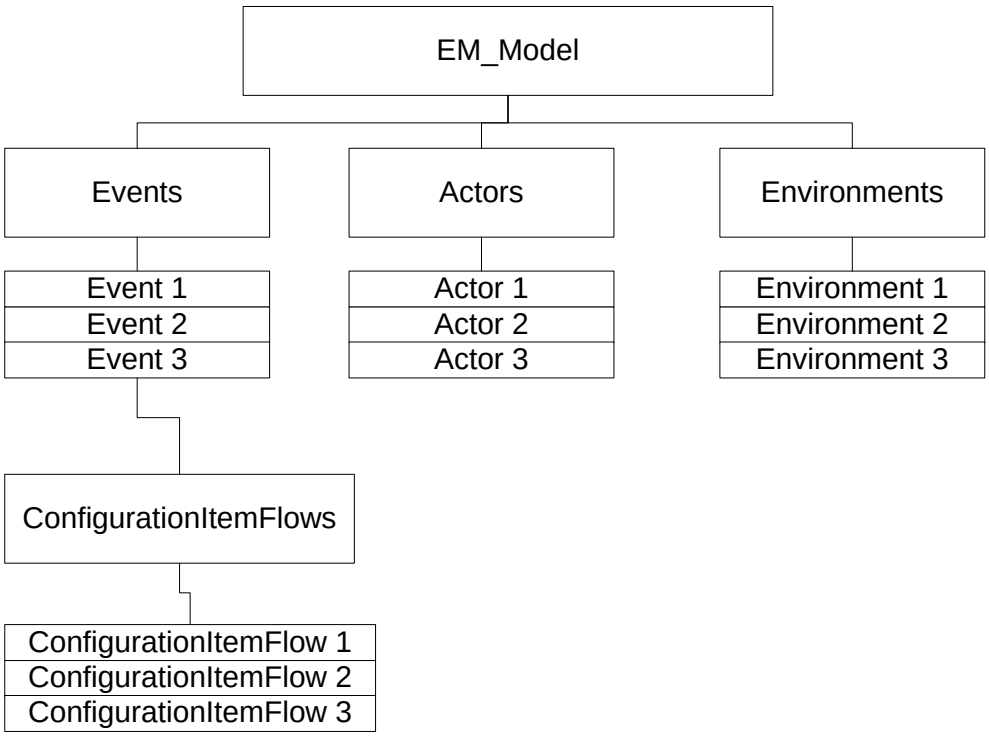
Attēlos 3.5. un 3.6. ir vienkāršoti vižu modeļi, taču ar lielākiem riskiem. Pirmkārt, abos modeļos ir tikai viena izstrādes vide (*DEV* un *LENT*). Tas nozīmē, ka izmaiņas vienmēr tiek izstrādātas tikai jaunākajā versijaā kas atrodas izstrādes stadijā. Ja kādas konkrētas izmaiņas ir nepieciešams pārnest uz citu vidi ātrāk, nekā tiks izlaista regulārā versija, izejas kodu integrē atbilstošās vides versijas izejas kodā un no integrācijas rezultāta veido produkta versijas labojumu. Rodas papildu riski, ka integrācijas laikā notiks kļūda, kā arī zūd iespēja kvalitatīvi notestēt, vai izmaiņas atrisina problēmu vidē, kur konkrētā laika brīdī ir vecāka versija nekā *DEV* vai *LENT* vidē. Modelim 3.6. ir vismazāk vižu, un tas paredz, ka visas izmaiņas tiek izstrādātas tikai vienu reizi, kas ļauj ietaupīt gan datorresursus, gan izstrādātāju laiku, taču izraisa daudz risku konfigurācijas pārvaldības kontekstā. Rodas ne tikai riski izejas koda integrācijas laikā, bet arī iespēja iegūt nestrādājošu oriģinālu vidi, jo nav nevienas vides,

kur varētu notestēt, vai no integrēta izejas koda tehniski var izveidot (uzbūvēt) strādājošu programmatūras versiju.

Vižu modelis uzskatāmi palīdz atrast resursu un konfigurācijas pārvaldības risku kompromisu. Pateicoties tam, ka modeli ir iespējams mainīt, rodas iespēja vienam un tam pašam projektam izveidot vairākus vižu modeļa variantus, novērtēt riskus un atkarībā no situācijas izvēlēties optimālu variantu konkrētajam gadījumam. Vižu modelis palīdz pasūtītājam labāk saprast, kādā vaidā izmaiņas nonāk līdz ekspluatācijai, un mazina pārpratumu iespējas. Nodarbojoties ar konfigurācijas pārvaldību projektos, kur pasūtītājs nav izpētījis vai izpratis vižu kopēju plānu jeb modeli, promocijas darba autors ir saskāries ar situāciju, kad pasūtītājs nesaprot, piemēram, kāpēc produkcijā nestrādā izmaiņas no versijas 10.0., lai gan dotā versija atrodas vēl tikai akcepttesta vidē. Neredzot vižu modeli vai plānu un neapzinoties konfigurācijas pārvaldības riskus, pasūtītājs dažreiz nesaprot, kāpēc nevar vienā acumirklī dabūt kādas noteiktās izmaiņas uz ekspluatācijas vidi bez papildu darbībām. Modelējot vides kopā ar pasūtītāju, visus šādus scenārijus ir vieglāk apspriest un uzskatāmi parādīt, kas potenciāli ļauj nonākt pie saprotama un abām pusēm pieņemama risinājuma vēl pirms tiek sākti darbi ar programmatūru.

d. Vižu meta-modeļa elementu hierarhija

Attēlā 3.7. ir parādīta vižu meta-modeļa elementu hierarhija.



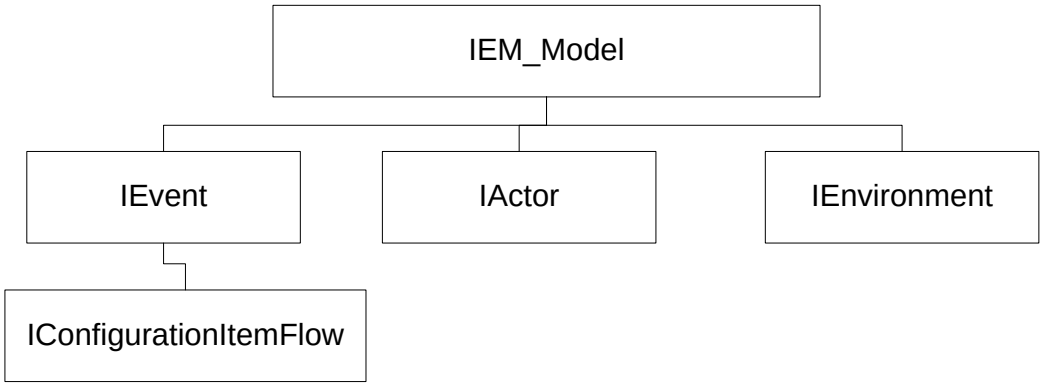
3.7. att. *EMsource* elementu hierarhija

Kā redzams attēlā 3.7., vižu modeli implementē elements *EM_Model*. Katram modelim nākamajā hierarhijas līmenī ir šādi elementi:

- *Events* – notikumi. Ietver visus notikumus (*Event*), kas realizē programmatūras izmaiņu pārnešanu starp vidēm;
- *Actors* – aktieri. Elements ietver visus aktierus, kas ir definēti konkrētajā vižu modelī;
- *Environments* – vides. Elements ietver visas vides, kas ir konkrētajā vižu modelī. Katram notikumam (*Event*) ir vismaz viena plūsma (*ConfigurationItemFlow*).

e. <u>Vižu modeļa (*EM*) kompilācija</u>

Brīdī, kad vižu modelis ir pārveidots *XML* formātā un ir gatavs datorapstrādei, svarīgi pārliecināties, vai modelī nav kļūdu, kas liegtu tālāk korekti veikt transformācijas cita veida modeļos. Šim nolūkam ir izstrādāta vižu modeļa kompilācija. Kompilācijas risinājums paredz no *XML* dokumenta, kas atbilst izveidotajam vižu modelim, izveidot analoģisku objektu struktūru ar tādu pašu hierarhiju, taču struktūras veidošanas laikā obligāti pārbaudīt modeļa elementu stāvokļa korektumu, piemēram, vai nav divas vides ar vienādiem nosaukumiem, vai viena notikuma plūsmas ir pareizi numurētas, vai izstrādātājs veic izmaiņas tikai izstrādei paredzētajā vidē utt. Šādas un līdzīgas pārbaudes arī tiks uzskatītas par modeļa kompilāciju. Lai varētu sistematizēt vižu modeļa kompilācijas likumus, tiek izstrādāta speciāla interfeisu hierarhija. Katram vižu modeļa elementam (*Actor, Event, Environment, ConfigurationItemFlow, EM_Model*) atbilst interfeiss ar līdzīgu nosaukumu, taču ar «I» burtu priekšā (*IActor, IEvent, IEnvironment, IConfigurationItemFlow, IEM_Model*). Interfeisu hierarhija ir redzama attēlā 3.8.



3.8. att. Vižu meta-modeļa elementu interfeisi un to hierarhija

Interfeisos *IEM_Model* un *IEvent* ir definētas speciālas metodes, kurās jābūt realizētai vižu modeļa kompilācijai. Ja metode beidzās veiksmīgi, tas nozīmē, ka konkrētais elements ir korekts. Realizējot vižu modeļa objektu struktūru, kurā tiks ielikta informācija no atbilstoša modeļa *XML* dokumenta, par pamatu obligāti jāņem 3.8. attēlā redzami interfeisi. Visus jaunos elementus var pievienot, tikai izmantojot interfeisos norādītās speciālās metodes. Tabulā 3.4. ir dots interfeisu apraksts. Metodēs, kurās notiek vižu modeļa elementu korektuma pārbaude jeb kompilācija, ir aprakstīti likumi, kuriem jābūt realizētiem, lai varētu īstenot modeļa kompilāciju.

3.4. tabula

Vižu meta-modeļa interfeisi un kompilācijas likumi

Interfeiss	**Metode**	**Metodes apraksts, kompilācijas likumi**
IEM_Model	***void setModelName****(String)*	Metode, kas definē modeļa nosaukumu. Modeļa nosaukums tiek padots kā teksta parametrs, nekādas papildu pārbaudes šeit nav vajadzīgas.
	void setModelDescription*(String)*	Metode, kas definē modeļa aprakstu. Apraksts var būt jebkāds brīvais teksts, tāpēc nekādas papildu pārbaudes šeit nav nepieciešamas.
	boolean createNewEvent*(String Name, String Description)*	Metode, kas izveido jaunu (tukšu) notikumu (*Event*) modelī. Metode kā parametru saņem notikuma nosaukumu un aprakstu. Tiek pārbaudīts, vai vižu modelī jau nepastāv notikums ar tādu pašu nosaukumu. Ja šāds notikums nepastāv, metode izveido jaunu objektu *Event*, piešķirot atbilstošu nosaukumu un aprakstu, un atgriež loģisku vērtību «*true*». Pretējā gadījumā, ja notikums ar šādu nosaukumu jau eksistē, metode atgriež loģisku vērtību «*false*».
	boolean	Metode, kas vižu modeļa objektu

	addModelActor*(String Name, String Description)*	struktūrā pievieno jaunu aktieri. Metode kā parametru saņem aktiera nosaukumu un aprakstu. Notiek pārbaude, vai aktieris ar šādu pašu nosaukumu jau eksistē. Ja tāda aktiera nav, metode izveido jaunu objektu *Actor* un atgriež «*true*». Pretējā gadījumā metode atgriež «*false*».
	boolean addModelEnvironment*(String Name, String Description, boolean CustomerSupportFlag, boolean DevEnvironmentFlag, boolean OriginalEnvironmentFlag, String OriginalEnvironmentName)*	Metode pievieno jaunu vidi. Kā parametru metode saņem vides nosaukumu, aprakstu, pazīmi, vai vidi uztur pasūtītājs, pazīmi, vai tā ir izstrādes vide, pazīmi, vai tā ir oriģinālā vide, oriģinālās vides nosaukumu, ja tā nav oriģinālā vide. Saņemot parametrus, metode obligāti veic šādas pārbaudes: • vai modelī neeksistē vide ar tādu pašu nosaukumu; • vai nav tā, ka izstrādes vidi uztur pasūtītājs (ja pasūtītājs uztur vidi, šajā risinājuma tas nozīmē, ka tikai viņš var mainīt tajā konfigurācijas vienumus); • ja vide nav oriģināla, vai ir padots oriģinālās vides nosaukums; • ja ir padots oriģinālās vides nosaukums, vai *OriginalEnvironmentFlag* lauka vērtība ir «*false*» un vai modelī tiešām eksistē vide ar šādu nosaukumu, un vai tā ir

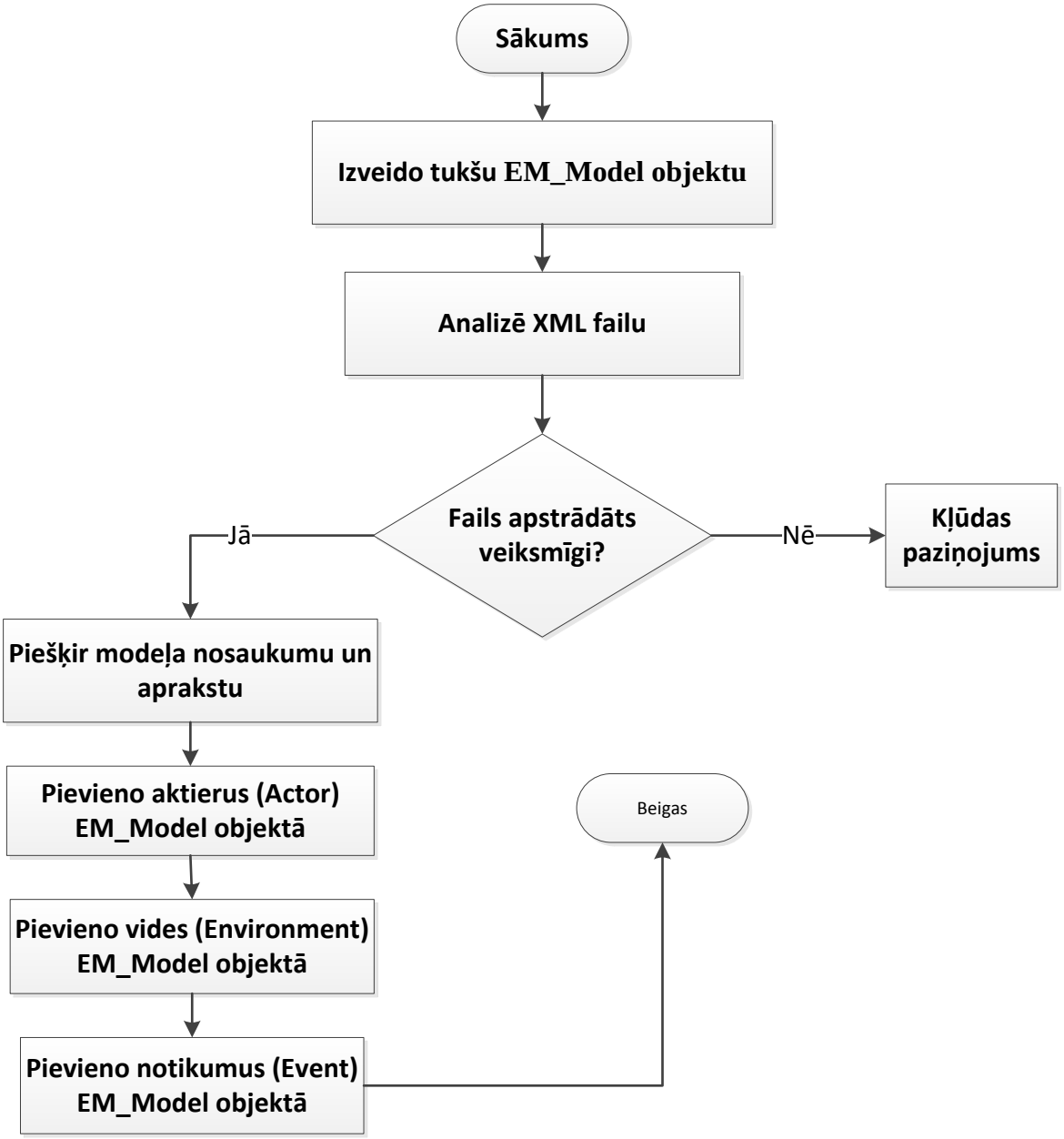
		oriģinālā vide. Ja visas pārbaudes ir bijušās veiksmīgas, metode izveido jaunu objektu *Environment* un atgriež «*true*», pretējā gadījumā metode atgriež «*false*».
	String getModelName()	Metode atgriež modeļa nosaukumu
	String getModelDescription()	Metode atgriež modeļa aprakstu teksta formātā.
IConfigurationItemFlow	***String getConfigurationItemFlowName()***	Metode atgriež konfigurācijas plūsmas nosaukumu teksta formātā.
	Integer getConfigurationItemFlowSequence()	Metode atgriež konfigurācijas plūsmas kārtas numuru kā veselu skaitli.
	String getConfigurationItemFlowSource()	Metode atgriež konfigurācijas plūsmas avotu (aktieris vai vide) teksta formātā.
	String getConfigurationItemFlowGoal()	Metode atgriež konfigurācijas plūsmas mērķi (aktieris, notikums vai vide) teksta formātā.
	String getConfigurationItemFlowDescription()	Metode atgriež konfigurācijas plūsmas aprakstu teksta formātā.
IEnvironment	***String getEnvironmentName()***	Metode atgriež vides nosaukumu teksta formātā.
	String getEnvironmentDescription()	Metode atgriež vides aprakstu teksta formātā.
	boolean getCustomerSupportFlag()	Metode atgriež pazīmi, vai šo vidi uztur pasūtītājs.
	boolean getOriginalEnvironmentFla	Metode atgriež pazīmi, vai vide ir oriģināla.

	g()	
	boolean getDevelopmentFlag()	Metode atgriež pazīmi, vai tā ir izstrādes vide.
	String getOriginalEnvironmentName()	Metode atgriež oriģinālās vides nosaukumu gadījumā, ja ir pazīme, ka dotā vide nav oriģināla.
IActor	***String getActorName()***	Metode atgriež aktiera nosaukumu teksta formātā.
	String getActorDescription()	Metode atgriež aktiera aprakstu teksta formātā.
IEvent	***boolean AddConfigurationItemFlow(*** *String Name, Integer Sequence, String Owner, String Goal, String Description, Array<Actor> Actors, Array<Environment> Environments)*	Metode pievieno notikumā *Event* jaunu konfigurācijas vienumu plūsmu. Kā parametru metode saņem plūsmas nosaukumu, kārtas numuru, avotu (*Actor* vai *Environment*), mērķi (*Environment* vai *Event*), plūsmas aprakstu, aktieru sarakstu un vižu sarakstu. Metode obligāti veic šādas pārbaudes: • vai plūsma ar šādu pašu nosaukumu jau eksistē notikumā *Event*. Ja eksistē, metode uzreiz atgriež «*false*»; • vai plūsmas kārtas numurs ir padots korekti. Ja, piemēram, notikumā *Event* vēl nav nevienas plūsmas, bet tiek padota plūsma ar kārtas numuru «*2*», metode atgriež «*false*». Ja, piemēram, šāds kārtas numurs jau eksistē, arī tad metode atgriež «*false*»; • pārbauda ienākošo parametru

		Owner. Ja notikumā *Event* jau ir citas konfigurācijas vienumu plūsmas, pārbauda, vai *Owner* parametrs jaunai plūsmai ir tāds pats, jo vienā notikumā var pārnest konfigurāciju tikai no vienas konkrētas vides vai aktiera. Gadījumā, ja notikumā *Event* vēl nav nevienas plūsmas, pārbauda, vai starp aktieriem (*Actors*) vai vidēm (*Environments*) eksistē tāds elements. Ja tāda elementa nav, metode atgriež «*false*»; • pārbauda parametru *Goal*. Vērtībai jābūt tādai, ka eksistē vai nu šāda vide, vai notikums. Ja tas ir notikums, pārbauda, vai notikums neizsauc pats sevi, ja tā ir, metode atgriež «*false*». Papildus pārbauda, vai šāds notikums vispār modelī eksistē, pretējā gadījumā atgriež «*false*». Ja *Goal* parametrā ir vide, veic papildu pārbaudes. Ja parametrā *Owner* ir kāds aktieris, bet *Goal* parametrā esošā vide nav paredzēta izstrādei, metode atgriež «*false*», jo izstrādātāji var mainīt konfigurācijas vienumus tikai izstrādes vidēs. Ja notikumā *Event* jau eksistē plūsma ar tādu pašu *Goal*

		vērtību, metode atgriež «*false*», jo vienā notikumā nevar būt divas identiskās plūsmas; • ja *Goal* vērtībā ir oriģinālā vide, pārbauda, vai šajā notikumā jau neeksistē kāda plūsma, kur *Goal* parametrā ir oriģinālā vide. Ja tā eksistē, metode atgriež «*false*», jo vienā notikumā var pārnest konfigurāciju tikai uz vienu oriģinālo vidi (nevar mainīt uzreiz gan akcepttesta, gan produkcijas vidi, jo jāsagaida, kad izmaiņas tiks notestētas akcepttesta vidē). Ja visas pārbaudes ir bijušās veiksmīgas, metode pievieno konfigurācijas vienumu plūsmu *Event* notikumam un atgriež «*true*».
	String getEventName()	Metode, kas atgriež notikuma nosaukumu teksta formātā.
	String getEventDescriotion()	Metode, kas atgriež notikuma aprakstu teksta formātā.
	void setAllChangesMoveFlag(boolean allChangesFlag)	Metode, kas uzstāda pazīmi, vai konkrētajā notikumā no avota vides tiks ņemtas visas izmaiņas vai tikai kādas noteiktas (izmaiņas noteiktos konfigurācijas vienumos).

Brīdī, kad ir realizēti visi interfeisi, kas minēti tabulā 3.4., var lietot vižu modeļa kompilācijas algoritmu. Attēlā 3.9. ir redzama kompilācijas algoritma blokshēma.



3.9. att. Vižu modeļa kompilācijas algoritma blokshēma

Aprakstot algoritmu, tiek pieņemts, ka interfeisu realizējošo klašu nosaukumi sakrīt ar interfeisu nosaukumiem, taču priekšā nesatur «*I*» burtu. Līdz ar to klases, kas realizēs 3.4. tabulā aprakstītus interfeisus, būs šādas:

- *EM_Model* – klase, kas realizē interfeisu *IEM_Model*;
- *ConfigurationItemFlow* – klase, kas realizē interfeisu *IConfigurationItemFlow*;
- *Environment* – klase, kas realizē interfeisu *IEnvironment*;
- *Actor* – klase, kas realizē interfeisu *IActor*;
- *Event* – klase, kas realizē interfeisu *IEvent*.

Kad visi interfeisi ir realizēti, kompilācijas algoritms darbojas šādi:

1. izveido objektu no klases *EM_Model*. Objekts glabās visu informāciju par vižu modeli atbilstoši modeļa elementu un interfeisu hierarhijai;
2. analizē *XML* dokumentu, kas attēlo izveidoto vižu modeli. Ja *XML* dokumentu neizdevās identificēt atbilstoši modeļu elementu struktūrai un nosaukumiem, algoritms beidz savu darbību un paziņo, ka *XML* faila formāts nav atbilstošs vižu modelim. Šāds scenārijs var būt gadījumā, kad lietotājs izmaiņas veic manuāli, vai arī nepareizi nostrādā algoritms, kas pārveido vižu modeļa elementus no grafiskā uz *XML* formātu;
3. strādā ar *XML* analīzes laikā iegūtu elementu struktūru. Izsauc *EM_Model* objekta metodes *void setModelName(String)* un *void setModelDescription(String)* un piešķir modelim nosaukumu un aprakstu;
4. strādā ar *XML* analīzes laikā iegūtu elementu struktūru. Katram elementam *Actor,* izsauc *EM_Model* objekta metodi *boolean addModelActor(String Name, String Description)*. Ja kādu aktieri nevar pievienot, paziņo par kļūdām un beidz darbību;
5. strādā ar *XML* analīzes laikā iegūtu elementu struktūru. Katram elementam *Environment* izsauc metodi *boolean addModelEnvironment(String Name, String Description, boolean CustomerSupportFlag, boolean DevEnvironmentFlag, boolean OriginalEnvironmentFlag, String OriginalEnvironmentName)*. Ja kādu vidi neizdevās pievienot, algoritms beidz savu darbību;
6. strādā ar *XML* analīzes laikā iegūtu elementu struktūru. Katram elementam «*Event*» veic darbības:
    a. izveido jaunu notikumu *Event* ar metodi *boolean createNewEvent(String Name, String Description);*
    b. nolasa konkrēta notikuma *Event* apakšelementu *ConfigurationItemFlows;*
    c. katram elementam *ConfigurationItemFlow* izsauc metodi *boolean Event.AddConfigurationItemFlow(String Name, Integer Sequence, String Owner, String Goal, String Description, Array<Actor> Actors, Array<Environment> Environments)*. Ja kaut vienu plūsmu *ConfigurationItemFlow* neizdodas pievienot, algoritms pārtrauc savu darbību;
7. algoritms beidz savu darbību un paziņo, ka vižu modelis ir veiksmīgi nokompilēts. Sistēmā glabājas aizpildīts objekts *EM_Model*, kas ir gatavs nodošanai transformācijai cita līmeņa modelī.

3.6. No platformas neatkarīga darbību meta-modeļa izstrāde

No platformas neatkarīgs darbību modelis (turpmāk tekstā *PIAM*) parāda, kādas darbības ir jāveic, lai nodrošinātu visu plūsmu implementāciju vižu modelī. Līdz ar to modeļa galvenais mērķis ir parādīt visas darbības, kas ir nepieciešamas visu plūsmu nodrošināšanai, darbību savstarpējas atkarības un darbību atribūtus. Atribūtu vērtības šajā modelī netiek aizpildītas, jo modelis nedrīkst būt atkarīgs no kādas konkrētas platformas vai tehnoloģijas.

Līdzīgi kā vižu meta-modeļa izstrādes gadījumā, arī *PIAM* meta-modelim ir šādas daļas:

- elementu uzskatījums, apraksts un hierarhija;
- modeļa elementu vizuālā interpretācija;
- modeļa elementu *XML* interpretācija.

a. *PIAM* meta-modeļa elementu uzskaitījums, apraksts un hierarhija

PIAM meta-modeļa elementi atbilst konfigurācijas pārvaldības principam, ka visu disciplīnu var sadalīt noteiktos uzdevumos [AIE 2010, BER 2003], un rekomendācijai, ka visām darbībām jānotiek centralizēti – vienā vietā, lai izpilde nebūtu atkarīga no cilvēka darbstacijas, kurš izpilda kādu no darbībām, un lai visas atbildīgas personas varētu vienādi izpildīt vienas un tās pašas darbības [AIE 2010, PAU 2007, MET 2002]. Līdz ar to *PIAM* modelī vajadzētu iekļaut elementu, kas apraksta šādu centralizētu vietu, kur notiek visas konfigurācijas pārvaldības darbības. Ņemot vērā pētījumu pirmajā promocijas darba nodaļā par pieejamiem konfigurācijas pārvaldības risinājumiem, jāsecina, ka parasti vieta, kur notiek konfigurācijas pārvaldības darbības, ir nepārtrauktas integrācijas serveri. Pirmajā promocijas darba nodaļā dots arī saraksts ar konfigurācijas pārvaldības pamatuzdevumiem. Pamatojoties uz konfigurācijas pārvaldības galvenajiem uzdevumiem, tika izstrādāts darbību saraksts, kas tiks lietots *PIAM* meta-modelī. Ņemot vērā, ka katrai tehnoloģijai, projektam un uzņēmumam, kas definē konfigurācijas pārvaldības procesus, ir dažādas specifikas un īpatnības, uzdevumi nav pārāk detalizēti un var ietvert vēl virkni apakšdarbību, kas *PIAM* modelī netiks aprakstīti. Tabulā 3.5. ir redzamas visas konfigurācijas darbības, kuras būs iespējams modelēt ar *PIAMsource* palīdzību.

3.5. tabula

***PIAM* meta-modeļa konfigurācijas pārvaldības darbības un atribūti**

Nosaukums	Apzīmējums	Apraksts
Izmaiņu izstrāde	*DEVELOPMENT*	Izmaiņu izstrādes darbība ir iekļauta *PIAM* modelī, jo konfigurācijas pārvaldība reglamentē, ka jābūt izstrādes noteikumiem [AIE 2010, MET 2002]. Modelējot šo darbību, ir iespējams nodefinēt, piemēram, noteiktas tehnoloģijas un noteiktus konfigurācijas vienumu veidošanas principus, izmaiņu saskaņošanas procedūru, rīkus, reglamentējošos dokumentus, instrukcijas utt. Vienmēr jāpatur prātā, ka, ja izstrādes process nebūs kontrolējams un pārvaldāms, turpmākos soļus konfigurācijas pārvaldības procesā būs grūti prognozēt [BER 2003].
Izmaiņu saglabāšana versiju kontroles sistēmā	*COMMIT_CHANGES*	Darbība, kas nosaka izstrādāto izmaiņu saglabāšanu centralizētā repozitorijā. Darbību reglamentē versiju kontroles uzdevums, kas tika aprakstīts promocijas darba pirmajā nodaļā. Respektīvi, visas izstrādātas izmaiņas produkta konfigurācijas vienumos jāpakļauj versiju kontrolei, lai varētu redzēt kad, kurš un kāpēc veica izmaiņas [AIE 2010]. Šī

		darbība modelē versiju kontroli un likumus, kurus jāievēro izstrādātājiem, saglabājot izmaiņas versiju kontroles sistēmā. Darbības aprakstošajos atribūto var būt informācija par rīkiem, kas automātiski nokontrolē, vai tiek ievēroti nepieciešamie likumi un principi.
Versijas bāzes līnijas sagatavošana	*PREPARE_BASELINE*	Šī darbība atkarībā no versiju kontroles sistēmas reglamentē procedūru bāzes līnijas sagatavošanai. Konfigurācijas pārvaldība reglamentē, ka jebkādām izmaiņām ir jādefinē bāzes līnija, attiecībā pret kuru tiks veiktas izmaiņas [BER 2003]. Vižu modelis paredz, ka katrai oriģinālai videi ir sava bāzes līnija – produkta izejas koda stāvoklis, kas atbilst vides konfigurācijai. Līdz ar to pirms jebkādas konfigurācijas pārnešanas no vienas vides uz otru sākumā nepieciešams atjaunot izejas kodu atbilstošai videi. Praksē šī darbība paredz izmaiņu pārnešanu starp diviem versiju kontroles sistēmas zariem, ko sauc arī par saplūdināšanu (angļu val. *merge*). Šī darbība apraksta, kā notiek konfigurācijas pārnešana starp diviem

		repozitorija zariem un sniedz aprakstu rīkiem, kas to atbalsta tehniski.
Produkta būvējums	*COMPILE_BUILD*	Darbība, kas no atbilstošā izejas koda uzbūvē produktu. Kompilācijas un būvējuma rezultātā vajadzētu izveidoties izpildāmam vienumam, kas būtu spējīgs uzinstalēt produktu vidē.
Produkta instalācija	*INSTALL_BUILD*	Darbība, kas uzinstalē uzbūvētu produktu noteiktā vidē.
Produkta piegāde pasūtītājam	*PRODUCT_DELIVERY*	Darbība, kas sagatavo un nosūta pasūtītājam uzbūvētu produktu. Šī darbība ir nepieciešama, lai produktu varētu uzinstalēt vidē, ko uztur pasūtītājs un kurai izstrādātāju komandai nav pieejas, piemēram, ekspluatācijas vidē. Darbība modelē procedūru, kas jāievēro, sagatavojot produktu nosūtīšanai. Piemēram, kopā ar produkta būvējumu jānosūta arī versijas apraksts un instrukcija, kā produktu uzinstalēt, šīs prasības reglamentē konfigurācijas pārvaldība [AIE 2010].
Informācijas saņemšana par vides atjaunošanu	*ENV_UPDATE_NOTIFICATION*	Darbība, kas nogādā izstrādātāju komandai signālu par to, ka kādā no vidēm, ko uztur pasūtītājs, ir uzinstalēta jauna produkta versija. Šajā brīdī izstrādātāju komandai vajadzētu fiksēt šo

		faktu, atbilstoši papildinot konfigurācijas vienumu informāciju versiju kontroles sistēmā, kā arī jāuzinstalē tādu pašu produkta versiju visās vidēs, kas ir kopijas atbilstošai pasūtītāja videi. Šī darbība reglamentē procedūru, kas obligāti jāizpilda, saņemot informāciju no pasūtītāja par kādas vides atjaunošanu.

Tabulā 3.6. ir redzami *PIAM* darbību atribūti un to apraksts.

3.6. tabula

***PIAM* darbību atribūti un to apraksts**

Atribūts/apzīmējums	Darbības apraksts
Platforma (*Platform*)	Specifiskās platformas nosaukums, kas ir nepieciešams darbības realizācijai. Atribūts ir nepieciešams, jo vienai un tai pašai darbībai var būt vairāki risinājumi vairākām platformām, piemēram, skripts, kas pārnes izmaiņas starp diviem repozitorija zariem, var būt realizēts gan *Linux*, gan *Windows* platformām.
Risinājuma nosaukums (*SolutionName*)	Vienai un tai pašai darbībai var būt vairāki risinājumi. Lai varētu risinājumus atšķirt un pārvaldīt, katram risinājumam tiks piešķirts unikāls nosaukums.
Nepieciešamie rīki (*NeededTools*)	Rīki, kas ir nepieciešami risinājuma implementācijai.
Risinājumu glabātuve (*LocationsOfSolutions*)	Šis atribūts paredzēts, lai glabātu jebkādu informāciju par jau izstrādātajiem risinājumiem konkrētai darbībai. Te var glabāties, piemēram, skriptu nosaukumi, speciālu programmu atrašanas vietas utt.
Apraksts (*Description*)	Papildu apraksts konkrētai darbībai projekta vai risinājuma kontekstā, kas varētu sniegt papildu informāciju konfigurācijas

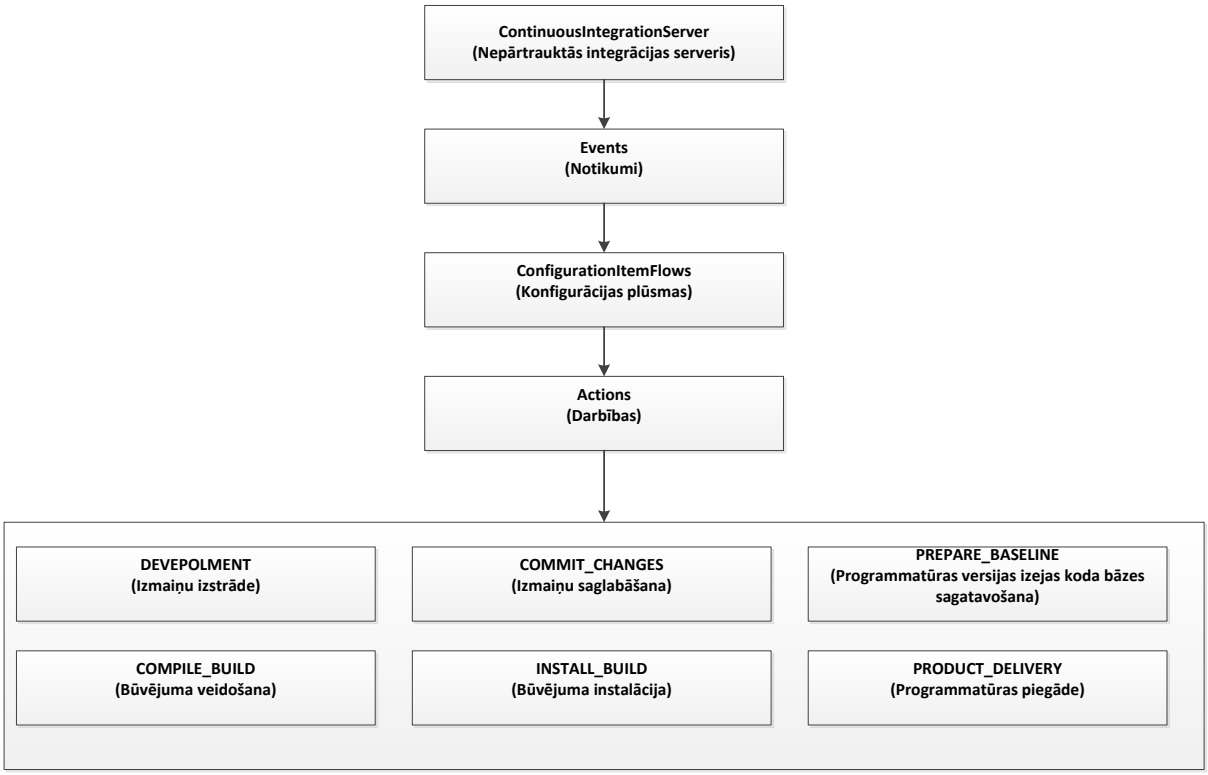
	pārvaldniekam.

Konfigurācijas pārvaldības darbībām jānotiek vienā vietā. Šāda vieta parasti ir nepārtrauktās integrācijas serveris, kas arī tiks modelēts *PIAM* modelī. Līdz ar to *PIAM* meta-modelī tiek ieviests elements «*Continuous Integration Server*» ar šādiem atribūtiem:

- platforma (*Platform*) – konkrēta platforma, kurā tiek uzinstalēts serveris;
- rīka nosaukums (*ToolName*) – servera nosaukums;
- instalācijas norādījumi (*InstallationNotes*) – norādījumi par servera instalāciju;
- risinājumu glabātuve (*LocationOfSolutions*) – gatavu risinājumu atrašanas vietas (ja tādas ir).

Vēl viens elements *PIAM* meta-modelī ir «Notikumi» (*Events*), kas ietver visus vižu modeļa notikumus. Katram notikumam ir arī visas plūsmas, kas arī tiek ņemtas no vižu modeļa. Tādējādi, iegūstot *PIAM* modeli no *EM* modeļa, galvenais uzdevums ir katra notikuma (*Event*) katrai plūsmai noteikt nepieciešamas darbības no *PIAM* meta-modeļa un attēlot šo informāciju strukturētā veidā.

PIAM modelī būs redzamas tikai konfigurācijas pārvaldības darbības, kuras ir vajadzīgas, lai implementētu visus notikumus vižu modelī, un darbību atribūti. Atribūtu vērtības tiks aizpildītas tikai *PSAM* modelī, kur tiks specificēta platforma, tehnoloģijas, rīki utt. Attēlā 3.10. ir redzama *PIAM* meta-modeļa elementu hierarhija.



3.10. att. *PIAM* meta-modeļa elementu hierarhija

b. *PIAM* elementu vizuāls attēlojums

PIAM meta-modeļa izstrādes gaitā tiek piedāvāts grafiskais attēlojums *PIAM* modelim, kas ir redzams attēlā 3.11.

ContinuousIntegrationServer			
Platform: <name>	**ToolName:** <name>	**InstallationNotes:** <notes>	**LocationsOfSolutions:** <locations>
Event: <name>		**Event:** <name>	
ConfigurationItemFlow: <name>	**Action:** <name> **Action:** <name>	**ConfigurationItemFlow:** <name>	**Action:** <name> **Action:** <name>
ConfigurationItemFlow: <name>	**Action:** <name> **Action:** <name>	**ConfigurationItemFlow:** <name>	**Action:** <name> **Action:** <name>
All Actions: Action1 Action2 Action3 .... ActionN			

3.11. att. *PIAM* modeļa grafiskais attēlojums

Attēlā 3.12. ir redzams *PIAM* modeļa elementu attēlojums *XML* formātā. Tieši šādā formātā modelis tiks pakļauts datorapstrādei, lai to varētu nodot pārveidošanai *PSAM* modelī.

```
<?xml version="1.0" encoding="UTF-8"?>
<PIAM_Model>
    <ContinuousIntegrationServer>
        <Platform="value" />
        <ToolName="value" />
        <InstallationNotes="value" />
        <LocationsOfSolutions="value" />

        <Events>
            <Event>
                <Name="value" />
                <ConfigurationItemFlows>
                    <ConfigurationItemFlow>
                        <Actions>
                            <Action  name="DEVELOPMENT/COMMIT_CHANGES/PREPARE_BASELINE/COMPILE_BUILD/INSTALL_BUILD/PRODUCT_DELIVERY/ENV_UPDATE_NOTIFICATION"/>
                        </Actions>
                    </ConfigurationItemFlow>
                </ConfigurationItemFlows>
            </Event>
        </Events>

        <Actions>
            <Action  name="DEVELOPMENT/COMMIT_CHANGES/PREPARE_BASELINE/COMPILE_BUILD/INSTALL_BUILD/PRODUCT_DELIVERY/ENV_UPDATE_NOTIFICATION">
                <PlatformName="value" />
                <SolutionName="value" />
                <NeededTools="value" />
                <LocationsOfSolutions="value" />
                <Description="value" />
                <original_environment_name="1,2,3" />
            </Action>
        </Actions>

    </ContinuousIntegrationServer>
</PIAM_Model>
```

3.12. att. *PIAM* modeļa *XML* attēlojums

3.7. Platformas specifiskā darbību modeļa realizācija

PSAM modeļa mērķis ir definēt implementāciju katrai darbībai *PIAM* modelī. *PSAM* modelim ir tās pašas komponentes kā *PIAM* modelim, tikai komponenšu atribūti jau ir aizpildīti ar konkrētām vērtībām, kas norāda uz konkrētu platformu, nepārtrauktas integrācijas servera nosaukumu, konkrētām programmām, skriptiem un citiem rīkiem, kas implementē konkrētus konfigurācijas pārvaldības uzdevumus, kas ir minēti tabulā 3.5.

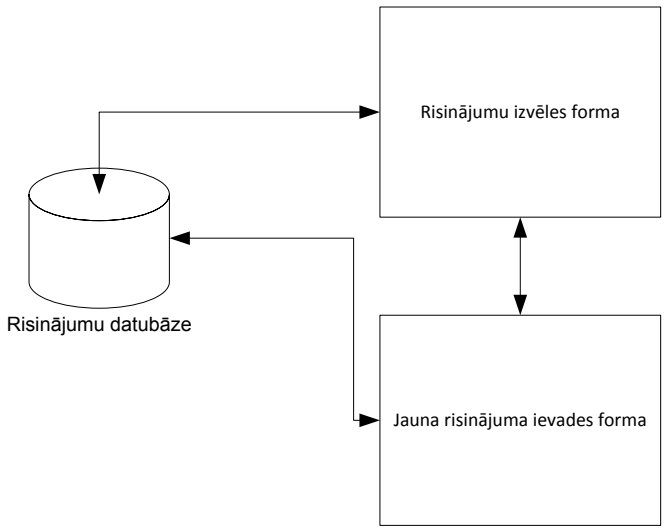
PSAM modelim ir šādi mērķi:

- glabāt informāciju par esošajiem risinājumiem katrai konfigurācijas pārvaldības darbībai no *PIAM* modeļa;
- ļaut pārskatīt risinājumus un katrai darbībai no *PIAM* modeļa ļaut izvēlēties vienu no esošajiem risinājumiem;
- ļaut ievadīt pilnīgi jaunu risinājumu jebkurai konfigurācijas pārvaldības darbībai un pēc tam attiecināt šo risinājumu pret darbību no *PIAM* modeļa;

- izveidot pārskatu, kurā tiek uzskaitītas visas darbības no *PIAM* modeļa un katrai darbībai ir definēts risinājums. Visi atribūti, kas *PIAM* modelī ir tukši, šajā pārskatā ir aizpildīti.

Pamatojot *PSAM* modeļa mērķi, tiek minēta viena no konfigurācijas pārvaldības problēmām, kas tika minēta promocijas darba pirmajā nodaļā. Ja apskata konfigurācijas pārvaldības organizāciju viena informācijas tehnoloģijas uzņēmumā, bieži var redzēt, ka konkrēti konfigurācijas pārvaldības risinājumi tiek veidoti haotiski, konkrētam specifiskam gadījumam un tie nav lietojami atkārtoti. Brīžiem rodas situācijas, kad tiek izstrādāts konfigurācijas pārvaldības risinājums, kas jau bija realizēts agrāk, bet par risinājumu nav palikusi nekāda vērā ņemama informācija. Šo problēmu varētu atrisināt, ja visi risinājumi būtu strukturēti un informācija par tiem atrastos vienā glabātuve un tiktu uzturēta pēc vienotiem principiem. *PSAM* modelis piedāvā glabāt visus risinājumus strukturētā veidā, sakārtojot tos pēc konfigurācijas pārvaldības darbībām. Konfigurācijas pārvaldības risinājumu strukturēšana pēc konfigurācijas pārvaldības darbībām paredz, ka jebkurš risinājums tiek izstrādāts neatkarīgi no citu darbību risinājumiem. Tas nozīmē, ka konkrētam risinājumam, kas, piemēram, atbalsta izejas koda saplūdināšanu (angļu val. *merge*) no viena zara uz otru, absolūti nav svarīgi, kā sapludināšanas rezultātu pēc tam izmantos produkta būvējuma skripti. Šāda pieeja atvieglo tādu risinājumu izstrādi, kurus vēlāk būs iespējams lietot atkārtoti, neatkarīgi no tā, vai kādi risinājumi tiks izmantoti citām konfigurācijas pārvaldības darbībām.

Lai īstenotu *PSAM* modeļa noteiktus mērķus, tika izstrādāts risinājumu izvēles modulis, kas ir redzams attēlā 3.13.

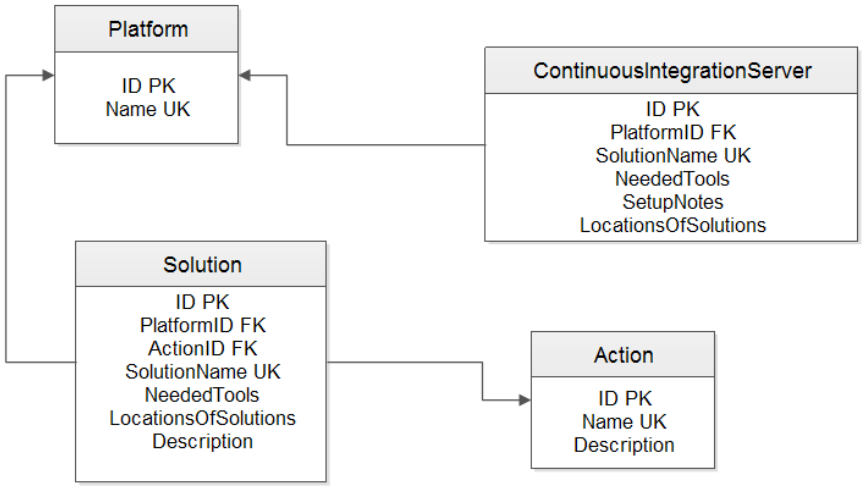


3.13. att. Risinājumu izvēles modulis

PSAMsource elementiem, kas ir redzami attēlā 3.13., ir šāda nozīme:

- **risinājumu datubāze** – konfigurācijas pārvaldības darbību risinājumu datubāze. Šajā datubāzē atrodas informācija par visiem jau realizētiem un pieejamiem risinājumiem katrai konfigurācijas pārvaldības darbībai, kas ir definēta *PIAM* meta-modelī;
- **risinājumu izvēles forma** – konfigurācijas pārvaldnieka forma, kas ļauj katrai konfigurācijas pārvaldības darbībai izvēlēties konkrētu risinājumu no datubāzes;
- **jauna risinājuma ievades forma** – lietotāja forma, kas ļauj ievadīt jaunu risinājumu kādai konkrētai konfigurācijas pārvaldības darbībai no *PIAM* modeļa. Ievadot risinājumu, tas kļūst pieejams risinājumu izvēles formā un to ir iespējams piesaistīt konkrētai konfigurācijas pārvaldības darbībai.

Risinājumu izvēles un ievades formas šajā promocijas darba nodaļā netiks aprakstītas sīkāk, jo lietotāju formas un to izstrāde var būt stipri atkarīga no konkrētas realizācijas tehnoloģijas un paņēmieniem. Taču, neatkarīgi no izvēlētās tehnoloģijas, jābūt noteiktai risinājumu datubāzes struktūrai, kas atbilstu *PIAM* meta-modelim. Attēlā 3.14. ir redzama risinājumu datubāzes *ER* diagramma, kurā tiek attēlotas minimālas prasības pret risinājumu datubāzi saskaņā ar *PIAM* meta-modeļa galvenajiem darbības principiem un mērķiem.



3.14. att. Konfigurācijas pārvaldības darbību risinājumu datubāze

PSAM modeļa veidošanas algoritms:

1. saņem *PIAM* modeli *XML* formātā;
2. nolasa elementu «*Actions*», un katru darbību ieliek risinājumu izvēles formā;

3. lietotājs strādā ar risinājumu izvēles formu. Katrai darbībai (*Action*) no risinājumu datubāzes tabulas «*Solution*» tiek atlasīti visi risinājumi, kas atbilst šai darbībai un piedāvā lietotājam izvēlēties vienu no risinājumiem. Ja lietotājs visām darbībām ir izvēlējies risinājumus, *PIAM* modeļa *XML* fails tiek papildināts ar informāciju no risinājumu datubāzes, un algoritms beidz savu darbību, atgriežot atjaunotu *XML* failu, kurā glabājas jau *PSAM* modelis;
4. ja lietotājs pieņem lēmumu, ka kādai darbība nav piemērota risinājuma vai risinājumu datubāze ir tukša, notiek darbs ar risinājumu ievades formu. Lietotājs ievada jaunu risinājumu. Pēc risinājuma ievades lietotājs atgriežas algoritma 3. solī.

3.8. Izejas koda zarošanas modelis

Izejas koda zarošanas modelis, turpmāk tekstā *SCBM* (angļu val. *Source Code Branching Model*), parāda izejas koda pārvaldības stratēģiju, ņemot vērā definētās vides vižu modelī. Kā jau tika minēts iepriekš, no konfigurācijas pārvaldības viedokļa ir svarīgi katrai videi uzturēt pārvaldāmu bāzes līniju, lai vienmēr varētu kontrolēt atbilstošās vides produkta izejas kodu.

a. *SCBM* meta-modelis

Meta-modelim ir viens elements: ***Branch***. Elements apraksta izejas koda vienu zaru un parāda izejas koda izmaiņu plūsmas sakarā ar šo zaru. Tabulā 3.7. var redzēt *Branch* elementa atribūtus un to aprakstu.

3.7. tabula

***SCMB* elementa *Branch* atribūti un to apraksts**

Atribūts/Tips	Atribūta apraksts
branchName *String*	Zara nosaukums
branchSequence *Integer*	Katram zaram ir kārtas numurs, lai būtu vieglāk pārvaldīt visus zarus. Ja, piemēram, zarā *DEV* sākās izstrāde, zara kārtas numurs ir 1. Pēc izstrādes izmaiņas tiek pārnestas testa vidē, un atbilstošam zaram *TEST* kārtas numurs jau būs 2.
mergeInput *boolean*	Pazīme, kas norāda, vai dotajā zarā ienāk izmaiņas no citiem zariem. Ja, piemēram, zara kārtas numurs (*branchSequence*) ir 1, tad šajā zarā netiek ieplūdinātas izmaiņas no citiem

	zariem un atribūta vērtība ir «*false*».
mergeOutput *boolean*	Pazīme, kas norāda, vai no dotā zara izmaiņas tiek pārnestas kādā citā zarā. Piemēram, ja zaram *PROD* ir augstāks kārtas numurs, *mergeOutput* atribūta vērtība ir «*false*». Pretējā gadījumā atribūta vērtība ir «*true*», jo izmaiņas tiek pārnestas uz citiem zariem atbilstoši procesā iesaistītām vidēm.
reverseMergeInput *boolean*	Atribūts, kas parāda, vai zarā neieplūst izmaiņas no cita zara, kuram kārtas numurs ir augstāks. Atribūts ir vajadzīgs, lai aprakstītu situāciju, kad oriģinālam vidēm ir kopijas, kurām tiek izstrādātas izmaiņas konkrētai versijai. Šajā gadījumā sākotnēji izmaiņas tiek saglabātas repozitorija zarā, kas atbilst minētajai oriģinālai videi un tikai pēc tam izmaiņas tiek ieplūdinātas zarā ar mazāko kārtas numuru. Ja konkrētajā zarā tiek ieplūdinātas izmaiņas no zara, kuram ir augstāks kārtas numurs, atribūta vērtība ir «*true*».
reverseMergeOutput *boolean*	Atribūts, kas parāda, vai no konkrētā zara izmaiņas tiek ieplūdinātas citā zarā, kuram kārtas numurs ir zemāks. Piemēram, ja oriģinālai videi *TEST* ir kopija *TEST(DEV)*, kurā tiek izstrādātas izmaiņas speciāli *TEST* vides versijai, vēlāk izmaiņas jāieplūdina *DEV* zarā. Šajā gadījumā atribūta vērtība būs «*true*».

b. *SCBM* iegūšana no *EM* modeļa

Lai būtu iespējams iegūt *SCBM* modeli no vižu modeļa, tika izstrādāti papildinājumi:

- *EM* modelī elements *Environment* tika papildināts ar vēl vienu atribūtu *processSequence*. Atribūts norāda izmaiņu galvenās plūsmas secību oriģinālajām vidēm. Piemēram, ja projektā ir četras vides: *DEV, TEST, QA* un *PROD, processSequence* atribūta vērtība būs attiecīgi 1, 2, 3 un 4;
- transformācijas algoritms «E->S» – transformācijas likumi no *EM* uz *SCBM* modeli. Izstrādājot transformācijas likumus, tika ņemtas vērā izejas koda zarošanas stratēģijas, ko apraksta nozares eksperti [BER 2003], [AIE 2010].

Tabulā 3.8. ir dotas papildu metodes un to apraksts vižu meta-modeļa elementa *EM_Model* klasē. Jaunas metodes nepieciešamas, lai varētu realizēt «E->S» transformācijas likumus.

3.8. tabula

***EM_Model* papildu metodes «E->S» transformācijas likumiem**

Metode/Semantika	Apraksts
Boolean isEnvironmentOriginal(Environment)	Nosaka, vai tā ir oriģinālā vide vai oriģinālās vides kopija.
Integer getEnvironmentSequence(Environment)	Ja vide ir oriģināla, atgriež vides kārtas numuru, ja vide nav oriģināla, atgriež nulli.
Boolean nextEnvironmentExist(Environment)	Pārmeklē visas atlikušās oriģinālās vides *EM* modelī un nosaka, vai eksistē vide ar kārtas numuru $x + 1$, kur x – dotās vides kārtas numurs.
Boolean previousEnvironmentExist(Environment)	Pārmeklē visas atlikušās oriģinālās vides *EM* modelī un nosaka, vai eksistē vide ar kārtas numuru $x - 1$, kur x – dotās vides kārtas numurs.
Boolean nextEnvironmentCopyFlag(Environment)	Atrod vidi ar kārtas numuru $x + 1$, kur x – dotās vides kārtas numurs. Nosaka, vai *EM* modelī atrastajai videi ir kopija, kas paredzēta izstrādei. Šajā gadījumā metode atgriež «*true*», pretējā gadījumā atgriež «*false*».
Boolean currentEnvironmentCopyFlag(Environment)	Atribūts, kas nosaka, vai vižu modelī eksistē izstrādes vide, kas ir dotās oriģinālas vides kopija.

Tabulā 3.9. ir redzami transformācijas likumi «E->S», kas, pateicoties jaunajām *EM_Model* klases metodēm, izveido *SCBM* modeli.

3.9. tabula

«E->S» transformācijas likumi un to apraksts

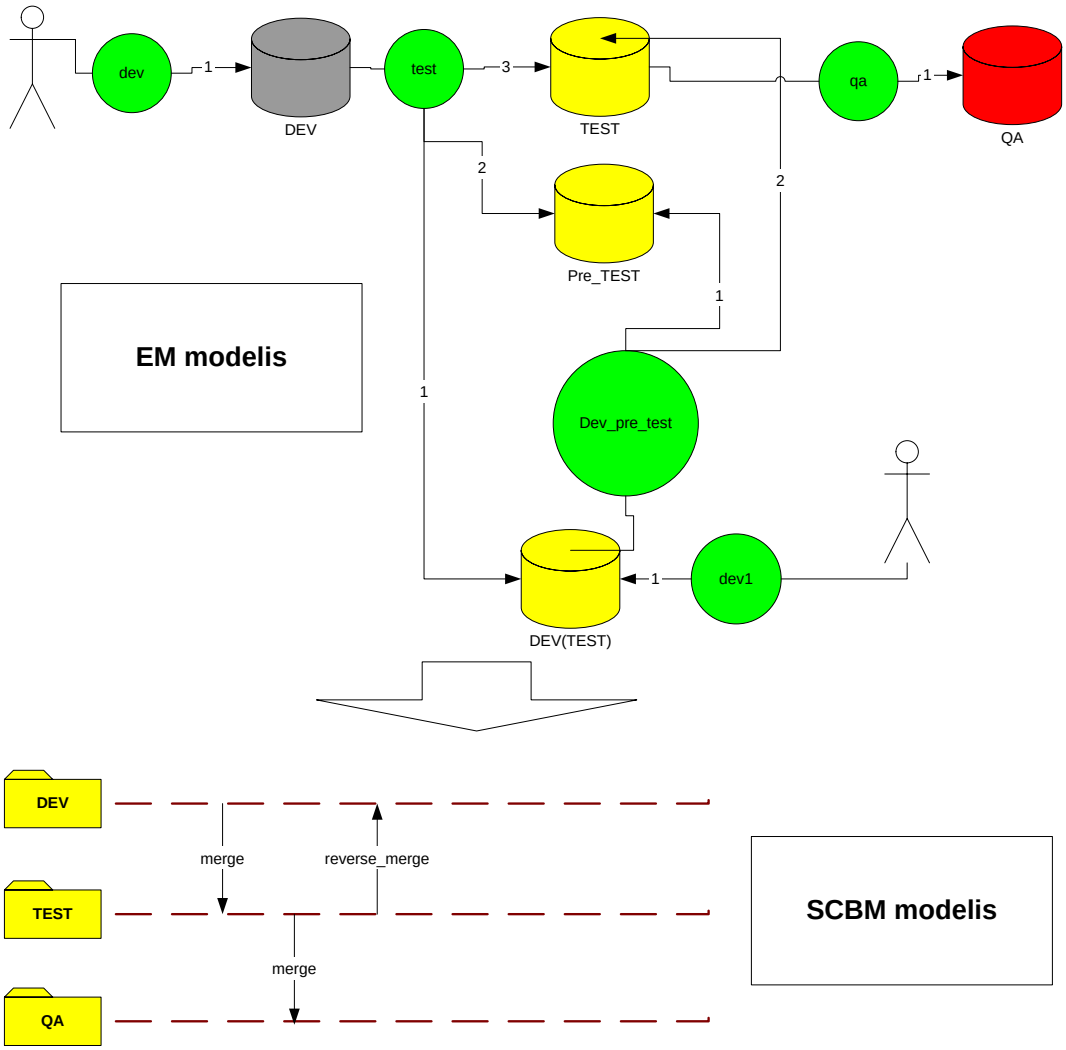
Numurs	Nosacījums (*IF* daļa)	Zaru (*Branch*) veidošana (*THEN* daļa)
1	***isEnvironmentOriginal(Environment)*** *= true* ***getEnvironmentSequence(Environment)*** *= 1* ***nextEnvironmentExist(Environment)*** *= true* ***nextEnvironmentCopyFlag(Environment)*** *= false* **Apraksts:** Oriģinālā vide, vides secība ir 1 (tipiski *DEV* vide), eksistē nākamā vide, kurai nav kopijas izmaiņu manuālai izstrādei.	***createNewBranch()*** ***branchName*** *= Environment.Name* ***branchSequence*** *= Environment.processSequence* ***mergeInput*** *= false* ***mergeOutput*** *= true* ***reverseMergeInput*** *= false* ***reverseMergeOutput*** *= false* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas tiks pārnestas uz nākamo vidi.
2	***isEnvironmentOriginal(Environment)*** *= true* ***getEnvironmentSequence(Environment)*** *= 1* ***nextEnvironmentExist(Environment)*** *= false* **Apraksts:** Gadījums, kad vižu modelī ir viena vienīga vide un citu oriģinālu vižu dotajā modelī nav. Tipiski tas ir pats projekta sākums, kad notiek produkta sākotnēja izstrāde *DEV* vidē.	***createNewBranch()*** ***branchName*** *= Environment.Name* ***branchSequence*** *= Environment.processSequence* ***mergeInput*** *= false* ***mergeOutput*** *= false* ***reverseMergeInput*** *= false* ***reverseMergeOutput*** *= false* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara nekādas izmaiņas netiek pārnestas uz kādu citu vidi, kā arī šajā zarā netiek ieplūdinātas nekādas izmaiņas.
3	***isEnvironmentOriginal(Environment)*** *= true* ***getEnvironmentSequence(Environment)*** *= 1* ***nextEnvironmentExist(Environment)*** *= true* ***nextEnvironmentCopyFlag(Environment)*** *= false* **Apraksts:** Gadījums, kad tā ir pirmā vide (*DEV*) vižu	***createNewBranch()*** ***branchName*** *= Environment.Name* ***branchSequence*** *= Environment.processSequence* ***mergeInput*** *= false* ***mergeOutput*** *= true*

	modelī, eksistē nākamā vide, kurai ir kopija, kas paredzēta izmaiņu izstrādei.	***reverseMergeInput*** = *true* ***reverseMergeOutput*** = *false* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas tiek pārnestas nākamajā zarā, kā arī no nākamā zara izstrādātās izmaiņas tiek pārnestas šajā zarā.
4	***isEnvironmentOriginal(Environment)*** = *true* ***nextEnvironmentExist(Environment)*** = *false* ***currentEnvironmentCopyFlag(Environment)*** = *false* **Apraksts:** Gadījums, kad tā ir pēdējā vide (*PROD*) vižu modelī un šai videi nav kopijas, kas būtu paredzēta izmaiņu izstrādei.	***createNewBranch()*** ***branchName*** = *Environment.Name* ***branchSequence*** = *Environment.processSequence* ***mergeInput*** = *true* ***mergeOutput*** = *false* ***reverseMergeInput*** = *false* ***reverseMergeOutput*** = *false* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas netiek pārnestas nākamajā zarā, kā arī netiek pārnestas izmaiņas uz iepriekšējo vidi.
5	***isEnvironmentOriginal(Environment)*** = *true* ***nextEnvironmentExist(Environment)*** = *false* ***currentEnvironmentCopyFlag(Environment)*** = *true* **Apraksts:** Gadījums, kad tā ir pēdējā vide (*PROD*) vižu modelī un šai videi ir kopija, kas paredzēta izmaiņu izstrādei.	***createNewBranch()*** ***branchName*** = *Environment.Name* ***branchSequence*** = *Environment.processSequence* ***mergeInput*** = *true* ***mergeOutput*** = *false* ***reverseMergeInput*** = *false* ***reverseMergeOutput*** = *true* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas netiek pārnestas nākamajā zarā, bet izstrādātās izmaiņas tiek pārnestas uz iepriekšējo vidi.

6	***isEnvironmentOriginal(Environment)*** *= true* ***nextEnvironmentExist(Environment)*** *= true* ***previousEnvironmentExist(Environment)*** *= true* ***currentEnvironmentCopyFlag(Environment)*** *= true* ***nextEnvironmentCopyFlag(Environment)*** *= false* **Apraksts:** Gadījums, kad vide ir kaut kur plūsmas vidū, tas nozīme, ka ir gan vide, no kurienes ienāk izmaiņas, gan vide, uz kurieni aiziet izmaiņas. Tipiskais piemērs ir testa vide, no kurienes parasti izmaiņas nonāk no izstrādes vides, bet pēc notestēšanas aiziet uz pasūtītāja akcepttesta vidi. Dotai videi ir kopija, kas paredzēta izmaiņu izstrādei, taču nākamajai videi šādas kopijas nav.	***createNewBranch()*** ***branchName*** *= Environment.Name* ***branchSequence*** *= Environment.processSequence* ***mergeInput*** *= true* ***mergeOutput*** *= true* ***reverseMergeInput*** *= false* ***reverseMergeOutput*** *= true* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas tiek pārnestas nākamajā zarā, arī šajā zarā tiek pārnestas izmaiņas no iepriekšējā zara. Papildus no šā zara izstrādātās izmaiņas tiek pārnestas uz iepriekšējo zaru.
7	***isEnvironmentOriginal(Environment)*** *= true* ***nextEnvironmentExist(Environment)*** *= true* ***previousEnvironmentExist(Environment)*** *= true* ***currentEnvironmentCopyFlag(Environment)*** *= true* ***nextEnvironmentCopyFlag(Environment)*** *= true* **Apraksts:** Gadījums, kad vide ir kaut kur plūsmas vidū, tas nozīme, ka ir gan vide, no kurienes ienāk izmaiņas, gan vide uz kurieni aiziet izmaiņas. Tipiskais piemērs ir testa vide, no kurienes parasti izmaiņas nonāk no izstrādes vides, bet pēc notestēšanas aiziet uz pasūtītāja akcepttesta vidi. Dotai videi ir kopija, kas paredzēta izmaiņu izstrādei, arī nākamajai videi šāda kopija ir.	***createNewBranch()*** ***branchName*** *= Environment.Name* ***branchSequence*** *= Environment.processSequence* ***mergeInput*** *= true* ***mergeOutput*** *= true* ***reverseMergeInput*** *= true* ***reverseMergeOutput*** *= true* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas tiek pārnestas nākamajā zarā, arī šajā zarā tiek pārnestas izmaiņas no iepriekšējā zara. Papildus no šā zara izstrādātas izmaiņas tiek pārnestas uz iepriekšējo zaru. Izmaiņas, kas ir izstrādātas nākamajā zarā, tiek pārnestas šajā zarā.
8	***isEnvironmentOriginal(Environment)*** *= true* ***nextEnvironmentExist(Environment)*** *= true* ***previousEnvironmentExist(Environment)*** *= true* ***currentEnvironmentCopyFlag(Environment)*** *= false*	***createNewBranch()*** ***branchName*** *= Environment.Name* ***branchSequence*** *= Environment.processSequence*

	nextEnvironmentCopyFlag(Environment) *= false* **Apraksts:** Gadījums, kad vide ir kaut kur plūsmas vidū, tas nozīme, ka ir gan vide, no kurienes ienāk izmaiņas, gan vide uz kurieni aiziet izmaiņas. Tipiskais piemērs ir testa vide, no kurienes parasti izmaiņas nonāk no izstrādes vides, bet pēc notestēšanas aiziet uz pasūtītāja akcepttesta vidi. Gan šai videi, gan arī nākamajai videi nav kopiju, kuras būtu paredzētas izmaiņu izstrādei.	***mergeInput*** *= true* ***mergeOutput*** *= true* ***reverseMergeInput*** *= false* ***reverseMergeOutput*** *= false* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas tiek pārnestas nākamajā zarā, arī šajā zarā tiek pārnestas izmaiņas no iepriekšējā zara.
9	***isEnvironmentOriginal(Environment)*** *= true* ***nextEnvironmentExist(Environment)*** *= true* ***previousEnvironmentExist(Environment)*** *= true* ***currentEnvironmentCopyFlag(Environment)*** *= false* ***nextEnvironmentCopyFlag(Environment)*** *= true* **Apraksts:** Gadījums, kad vide ir kaut kur plūsmas vidū, tas nozīme, ka ir gan vide, no kurienes ienāk izmaiņas, gan vide uz kurieni aiziet izmaiņas. Tipiskais piemērs ir testa vide, no kurienes parasti izmaiņas nonāk no izstrādes vides, bet pēc notestēšanas aiziet uz pasūtītāja akcepttesta vidi. Šai videi nav kopijas, kas paredzēta izmaiņu izstrādei, taču nākamajai videi šāda kopija ir.	***createNewBranch()*** ***branchName*** *= Environment.Name* ***branchSequence*** *= Environment.processSequence* ***mergeInput*** *= true* ***mergeOutput*** *= true* ***reverseMergeInput*** *= true* ***reverseMergeOutput*** *= false* **Apraksts:** izveido zaru, kura nosaukums sakrīt ar vides nosaukumu, un zara secība sakrīt ar vides secību. Atribūtos atzīmē, ka no šā zara izmaiņas tiek pārnestas nākamajā zarā, arī šajā zarā tiek pārnestas izmaiņas no iepriekšējā zara. Papildus no šā zara izstrādātas izmaiņas tiek pārnestas uz iepriekšējo zaru. Izmaiņas, kas ir izstrādātas nākamajā zarā, tiek pārnestas šajā zarā.

Attēlā 3.15. var redzēt vižu modeļa transformāciju *SCBM* modelī, lietojot «E->S» transformācijas likumus.



3.15.att. *SCBM* modeļa iegūšana no *EM* modeļa: vizuālais piemērs

3.9. Servisu modelis

Lai atvieglotu *PSAM* modeļa implementāciju, *EAF* metodoloģijā tika izstrādāts vēl viens papildu elements – servisu modelis, turpmāk tekstā tiks apzīmēts ar *ServiceModel*. *ServiceModel* ietver masīvu ar pāriem «*ContinuousIntegrationServer -> Tool*», kur

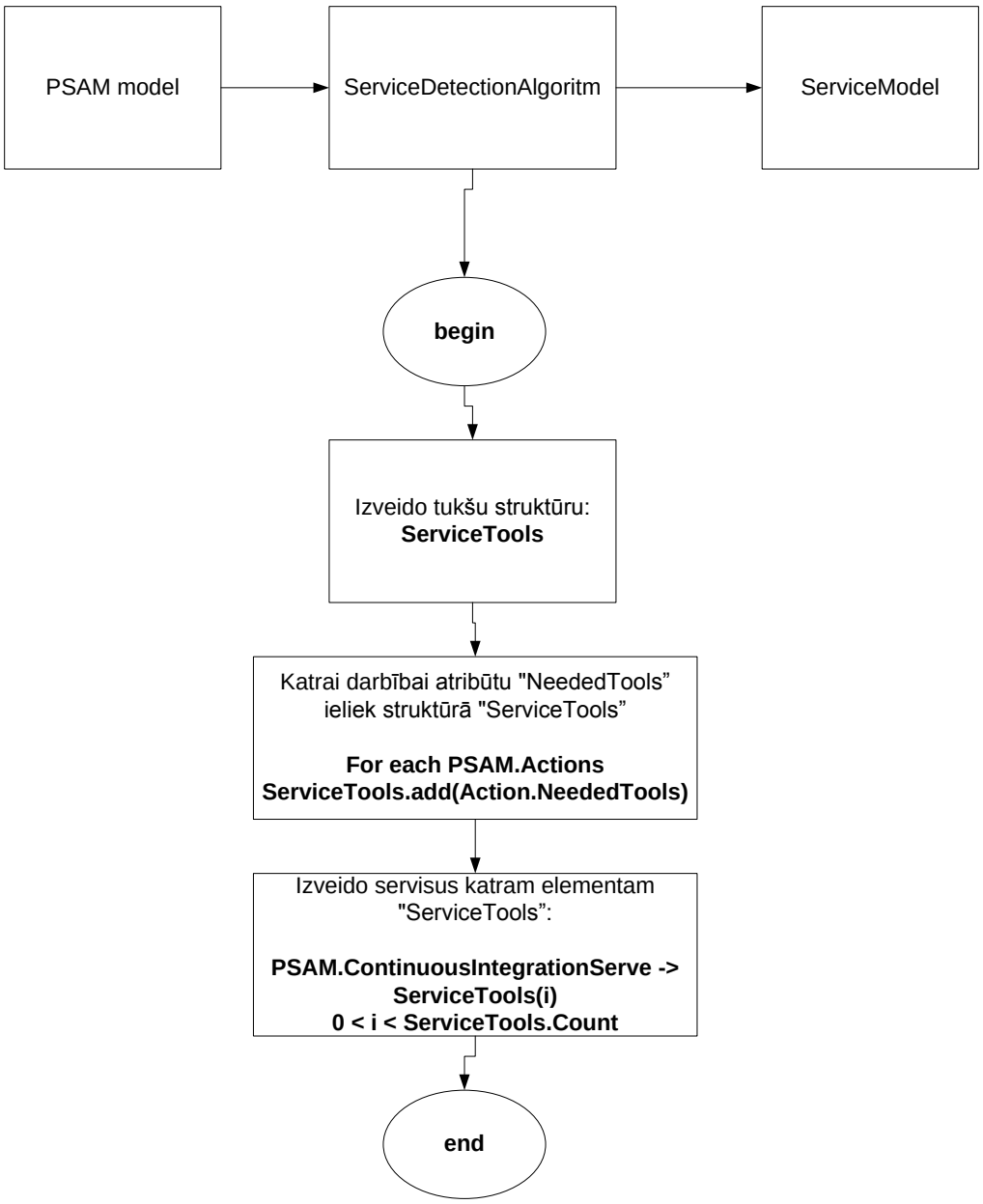
- ***ContinuousIntegrationServer*** – nepārtrauktās integrācijas serveris, kas tiek izmantots *PSAM* modelī, piemēram, *Jenkins, Hudson, Bamboo*. No šā servera vajadzēs iegūt informāciju no citiem rīkiem, kā arī veikt nepieciešamās darbības;
- ***Tool*** – rīks, kas norādīts *PSAM* darbību atribūtā *NeededTools*.

ServiceModel elementa izveidošanas algoritms:

- paņem izveidotu *PSAM* modeli;
- izveido *ServiceModel* tukšu struktūru;
- izveido tukšu struktūru *ServiceTools*;

- katrai darbībai *PSAM* modelī nolasa atribūtu *NeededTools* un ievieto to *ServiceTools* struktūrā;
- katram elementam *ServiceTools* struktūrā izveido pāri «*PSAM.ContinuousIntegrationServer -> ServiceTools(i)*» un ievieto izveidoto pāri *ServiceModel* struktūrā;
- parāda lietotājam masīvu *ServiveTools*, kur var redzēt, kādi servisi jāizstrādā.

Attēlā 3.16. var redzēt shēmu, kā veidojas jaunais elements modeļvadāmajā konfigurācijas pārvaldības risinājumā.



3.16.att. Servisu modeļa veidošana

3.10. *EAF* metodoloģijas modeļu transformācijas

Transformējot *EM* modeli *PIAM* modelī, jānosaka visas darbības, kas ir nepieciešamas vižu modeļa implementācijai. *EM* modelī sīkāka vienība, kas modelē konfigurācijas pārnešanu no vienas vides uz otru, ir konfigurācijas vienumu plūsma (*ConfigurationItemFlow*). Savukārt katra plūsma pieder konkrētam notikumam. *PIAM* modelī ir definētas konkrētas konfigurācijas pārvaldības darbības un to atribūti. Līdz ar to «E->P» transformācijas likumu mērķis ir katrai konfigurācijas plūsmai (*ConfigurationItemFlow*) vižu modelī definēt atbilstošās darbības no *PIAM* meta-modeļa.

Kā jau tika minēts, transformācijas likumi «E->P» strādās ar konfigurācijas plūsmām (*ConfigurationItemFlow*) un meklēs atbilstošās darbības *PIAM* meta-modelī. Līdz ar to transformācijas likumu realizācijai ir nepieciešams:

- *IConfigurationItemFlow* interfeisa papildinājums ar metodēm, kas ļauj noteikt konfigurācijas vienumu plūsmas atribūtu vērtības;
- likumi, kas nosaka, kādas darbības no *PIAM* meta-modeļa ir jāizvēlas atkarībā no *ConfigurationItemFlow* atribūtu vērtībām. Par pamatu likumu izstrādei tiks ņemts literatūras pētījums par konfigurācijas pārvaldību, kurā bija apkopotas rekomendācijas par to, kas ir jāievēro, pārnesot konfigurāciju no vienas vides uz otru, kādas potenciālas problēmas eksistē un kādas galvenās pārbaudes ir jāizpilda. Likumi ir sastādīti, balstoties uz rekomendācijām avotos: [AIE 2010, BER 2003, DEP 2010, PAU 2007, MET 2002];
- transformācijas algoritms. Šis algoritms saņems *EM* modeli kā *EM_Model* objektu un veiks visas nepieciešamas darbības, lai iegūtu *PIAM* modeli. Algoritms būs neatkarīgs no «E->P» transformācijas likumiem un strādās neatkarīgi no likumu daudzuma un satura.

Tabulā 3.10. var redzēt *IConfigurationItemFlow* interfeisa jaunās metodes, kas būs nepieciešamas «E->P» transformācijas likumiem, lai iegūtu informāciju par konfigurācijas plūsmas (*ConfigurationItemFlow*) atribūtiem. Katrai metodei ir skaidrojums un komentārs.

3.10. tabula

Interfeisa *IConfigurationItemFlow* papildu metodes

Metodes nosaukums un semantika	Skaidrojums
boolean ownerIsActor()	Nosaka, vai konfigurācijas plūsmas avots ir aktieris (*Actor*) vai vide (*Environment*). Ja avots ir vide, metode atgriež «*true*», pretējā gadījumā «*false*».
boolean ownerIsOriginalEnvironment()	Nosaka, vai konfigurācijas plūsmas avots ir oriģinālā vide. Ja avots ir vide un tā ir oriģinālā vide, metode atgriež «*true*», pretējā gadījumā «*false*».
boolean ownerCustomerSupportFlag()	Ja konfigurācijas plūsmas avots ir vide un to uztur pasūtītājs, metode atgriež «*true*», visos pārējos gadījumos tiek atgriezts «*false*».
boolean goalCustomerSupportFlag()	Ja konfigurācijas plūsmas mērķa vidi (*Goal*) uztur pasūtītājs, metode atgriež «*true*», pretējā gadījumā atgriež «*false*».
boolean addAction(String Action)	Pievieno darbību. Ja šāda darbība jau eksistē, atgriež «*false*». Pretējā gadījumā pievieno darbību un atgriež «*true*».
Array<String> getAllActions()	Atgriež masīvu ar visām darbībām, kas šajā brīdī eksistē konkrētajā konfigurācijas vienumu plūsmā. Metode vajadzīga nevis transformācijas likumiem, bet transformācijas algoritmam, kas beigās izvadīs *PIAM* modelī visas darbības katrai konfigurācijas plūsmai.
boolean compileActionExist()	Metode pārmeklē darbības visās konfigurācijas plūsmās, kas pieder konkrētam notikumam (*Event*). Ja kādā no konfigurācijas plūsmām jau eksistē darbība *COMPILE_BUILD*, metode atgriež «*true*», pretējā gadījumā atgriež «*false*». Šī pārbaude ir nepieciešama transformācijas likumiem.

	Piemēram, notikumā ir divas plūsmas: vienā plūsmā konfigurāciju pārnes uz testa vides kopiju, lai pārliecinātos, ka pārnešana notiks veiksmīgi un vide būs pieejama, bet citā plūsmā to pašu konfigurāciju pārnes jau uz īsto testa vidi. Konfigurācijas labā prakse šajā gadījuma paredz, ka otrajā plūsmā produkts nav jākompilē vēlreiz, bet jāinstalē tieši tādu pašu, kas bija uzkompilēts pirmajā plūsmā. Pretējā gadījumā pastāv risks, ka kompilācijas rezultāts nebūs identisks un testa vide kļūs nepieejama.

Tabulā 3.11. ir aprakstīti «E->P» transformācijas likumi.

3.11. tabula

«E->P» transformācijas likumi

Likuma kārtas numurs	**Konfigurācijas plūsmas (*ConfigurationItemFlow*) atribūtu vērtības (*IF* daļa)**	**Darbību pievienošana (*THEN* daļa), paskaidrojums**
1	*ownerIsActor() == true*	***addAction(DEVELOPMENT)*** ***addAction(COMMIT_CHANGES)*** Ja izmaiņas netiek pārnestas no kādas vides, bet tās veido manuāli, jābūt *DEVELOPMENT* darbībai, kas šā darba kontekstā nozīmē likumus, kas reglamentē produkta izmaiņu veikšanu. Izstādes beigās rezultāti obligāti jāsaglabā. Tiek izmantota darbība *COMMIT_CHANGES*, kas nozīmē, ka visām produktu vienībām jābūt pakļautam versiju kontrolei, izstrādātājam jaunas versijas jāpiefiksē atbilstoši vadlīnijām.
2	*ownerIsActor() == false*	***addAction(COMPILE_BUILD)***

	ownerIsOriginalEnvironment() == false *getConfigurationItemFlowSequence() == 1* *ownerCustomerSupportFlag() == true*	***addAction(PRODUCT_DELIVERY)*** ***addAction(ENV_UPDATE_NOTIFICATION)*** Ja nepieciešams pārnest konfigurāciju no vides kopijas uz vidi, kuru uztur pasūtītājs, un konkrētajā notikumā tā ir pirmā plūsma, nepieciešams uzkompilēt produktu, noformēt uzkompilēta produkta piegādi pasūtītājam ar visu pavadošo dokumentāciju un sagaidīt apstiprinājumu, ka pasūtītājs šo produktu ir uzinstalējis veiksmīgi, lai tālāk varētu aktivizēt citu plūsmu, kas, piemēram, uzliek tādu pašu konfigurāciju atbilstošās vides kopijā.
3	*ownerIsActor() == false* *ownerIsOriginalEnvironment() == false* *getConfigurationItemFlowSequence() == 1* *ownerCustomerSupportFlag() == false*	***addAction(COMPILE_BUILD)*** ***addAction(INSTALL_BUILD)*** Ja nepieciešams pārnest konfigurāciju no vides kopijas uz citu vidi, kuru neuztur pasūtītājs, un tā ir pirmā plūsma notikumā, nepieciešams uzkompilēt produktu un uzinstalēt to atbilstošajā vidē.
4	*ownerIsActor() == false* *ownerIsOriginalEnvironment() == false* *getConfigurationItemFlowSequence() != 1* *ownerCustomerSupportFlag() == true*	***addAction(PRODUCT_DELIVERY)*** ***addAction(ENV_UPDATE_NOTIFICATION)*** Ja nepieciešams pārnest konfigurāciju no vides kopijas uz vidi, kuru uztur pasūtītājs, un tā nav pirmā plūsma, nepieciešams izveidot produkta piegādi, par pamatu ņemot kompilāciju no iepriekšējām plūsmām, lai mazinātu riskus. Tad nepieciešams sagaidīt apstiprinājumu, ka pasūtītājs ir uzinstalējis produktu atbilstošā vidē, lai tālāk varētu aktivizēt citu plūsmu šajā pašā notikumā.

5	*ownerIsActor() == false ownerIsOriginalEnvironment() == false getConfigurationItemFlowSequence() != 1 ownerCustomerSupportFlag() == false*	***addAction(INSTALL_BUILD)*** Ja nepieciešams pārnest konfigurāciju no vides kopijas uz kādu citu vidi, ko neuztur pasūtītājs, un tā nav pirmā plūsma notikumā, nepieciešams tikai uzinstalēt produktu, par pamatu ņemot kompilāciju no iepriekšējām plūsmām šajā pašā notikumā.
6	*ownerIsActor() == false ownerIsOriginalEnvironment() == true (ownerCustomerSupportFlag() == true and goalCustomerSupportFlag() == true) == true*	***addAction(ENV_UPDATE_NOTIFICATION)*** Ja ir nepieciešams pārnest konfigurāciju starp divām oriģinālām vidēm un abas vides uztur pasūtītājs, pārnešanu pilnībā veic pasūtītājs, un par pamatu viņš ņem gatavu piegādi, kas tika instalēta avota vidē. Piemēram, ekspluatācijas vidē jāliek tieši tādas pašas izmaiņas, kas tika liktas testa vidē, pretējā gadījumā pastāv nopietni riski vides pieejamībai [AIE 2010].
7	*ownerIsActor() == false ownerIsOriginalEnvironment() == true (ownerCustomerSupportFlag() == true and goalCustomerSupportFlag() == true) == false goalCustomerSupportFlag() == false getConfigurationItemFlowSequence() == 1*	***addAction(PREPARE_BASELINE)*** ***addAction(COMPILE_BUILD)*** ***addAction(INSTALL_BUILD)*** Ja konfigurācija ir jāpārnes no oriģinālās vides, kuru neuztur pasūtītājs, uz citu vidi, kuru arī neuztur pasūtītājs, un ja šī plūsma ir pirmā notikumā, nepieciešams sagatavot vides izejas kodu, uzkompilēt produktu un uzinstalēt to.
8	*ownerIsActor() == false ownerIsOriginalEnvironment() == true (ownerCustomerSupportFlag()*	***addAction(INSTALL_BUILD)*** Ja konfigurācija ir jāpārnes no oriģinālās vides, kuru neuztur pasūtītājs, uz citu vidi, kuru arī neuztur pasūtītājs, ja tā nav pirmā

	== true and goalCustomerSupportFlag() == true) == false goalCustomerSupportFlag() == false getConfigurationItemFlowSequence() != 1 compileActionExist() == true	plūsma notikumā un ja kādā no iepriekšējām plūsmām produkts jau tika kompilēts, nepieciešams tikai uzinstalēt to, par pamatu ņemot jau esošu būvējumu.
9	*ownerIsActor() == false ownerIsOriginalEnvironment() == true (ownerCustomerSupportFlag() == true and goalCustomerSupportFlag() == true) == false goalCustomerSupportFlag() == false getConfigurationItemFlowSequence() != 1 compileActionExist() == false*	***addAction(PREPARE_BASELINE)*** ***addAction(COMPILE_BUILD)*** ***addAction(INSTALL_BUILD)*** Ja konfigurācija ir jāpārnes no oriģinālās vides, kuru neuztur pasūtītājs, uz citu vidi, kuru arī neuztur pasūtītājs, ja tā nav pirmā plūsma notikumā un ja nevienā no iepriekšējām plūsmām produkts nav kompilēts, nepieciešams sagatavot izejas kodu, uzkompilēt un uzinstalēt produktu.
10	*ownerIsActor() == false ownerIsOriginalEnvironment() == true (ownerCustomerSupportFlag() == true and goalCustomerSupportFlag() == true) == false goalCustomerSupportFlag() == true getConfigurationItemFlowSequence() == 1*	***addAction(PREPARE_BASELINE)*** ***addAction(COMPILE_BUILD)*** ***addAction(PRODUCT_DELIVERY)*** ***addAction(ENV_UPDATE_NOTIFICATION)*** Ja nepieciešams pārnest konfigurāciju no oriģinālas vides, kuru neuztur pasūtītājs, uz vidi, kuru uztur pasūtītājs, un ja tā ir pirmā plūsma notikumā, nepieciešams sagatavot vides izejas kodu, uzkompilēt produktu, sagatavot piegādi un sagaidīt apstiprinājumu, ka pasūtītājs produktu uzinstalējis, lai varētu aktivizēt citu notikumu, ja tāds ir paredzēts.

11	*ownerIsActor() == false* *ownerIsOriginalEnvironment() == true* *(ownerCustomerSupportFlag() == true and goalCustomerSupportFlag() == true) == false* *goalCustomerSupportFlag() == true* *getConfigurationItemFlowSequence() != 1* *compileActionExist() == true*	***addAction(PRODUCT_DELIVERY)*** ***addAction(ENV_UPDATE_NOTIFICATION)*** Ja nepieciešams pārnest konfigurāciju no oriģinālas vides, kuru neuztur pasūtītājs, uz vidi, kuru uztur pasūtītājs, ja tā nav pirmā plūsma notikumā un kādā no iepriekšējām plūsmām eksistē kompilācijas darbība, par pamatu jāņem jau uzkompilētais produkts un jāizveido tam piegāde, kā arī jāsagaida apstiprinājums, ka klients produktu ir uzinstalējis atbilstošajā vidē.
12	*ownerIsActor() == false* *ownerIsOriginalEnvironment() == true* *(ownerCustomerSupportFlag() == true and goalCustomerSupportFlag() == true) == false* *goalCustomerSupportFlag() == true* *getConfigurationItemFlowSequence() != 1* *compileActionExist() == false*	***addAction(PREPARE_BASELINE)*** ***addAction(COMPILE_BUILD)*** ***addAction(PRODUCT_DELIVERY)*** ***addAction(ENV_UPDATE_NOTIFICATION)*** Ja nepieciešams pārnest konfigurāciju no oriģinālas vides, kuru neuztur pasūtītājs, uz vidi, kuru uztur pasūtītājs, ja tā nav pirmā plūsma notikumā un nevienā no iepriekšējām plūsmām neeksistē kompilācijas darbība, nepieciešams sagatavot vides izejas kodu, uzkompilēt produktu, sagatavot piegādi un sagaidīt apstiprinājumu, ka pasūtītājs produktu uzinstalējis, lai varētu aktivizēt citu notikumu, ja tāds ir paredzēts.

3.11. *EAF* metodoloģijas kopsavilkums

Saistībā ar *EAF* metodoloģiju ir izstrādāti modeļi, kas attēlo konfigurācijas pārvaldības procesu programmatūras izstrādes projektā:

- vižu modelis (*EM*) – attēlo vides programmatūras izstrādes projektā;
- no platformas neatkarīgais darbību modelis (*PIAM*) – attēlo konfigurācijas pārvaldības darbības, kas ir nepieciešamas *EM* modeļa realizācijai;
- platformas specifiskais darbību modelis (*PSAM*) – paplašinātais variants *PIAM* modelim, kur darbības satur implementācijas detaļas konkrētai platformai;
- izejas koda zarošanas modelis (*SCBM*) – modelis, kas parāda programmatūras izejas koda zarošanas un pārvaldības metodi programmatūras izstrādes projektā;
- servisu modelis (*SM*) – parāda rīku savstarpējās integrācijas, kas ir nepieciešamas konfigurācijas pārvaldības darbību realizācijai.

Metodoloģijas galvenais mērķis ir iegūt izejas kodu konfigurācijas pārvaldības procesam. Tā sākas ar to, ka konfigurācijas pārvaldnieks izveido vižu modeli konkrētam projektam. Pēc tam ar transformācijas likumu «E->P» un «E->S» tika iegūti attiecīgi *PIAM* un *SCBM* modeļi. Vēlāk konfigurācijas pārvaldnieks paplašina *PIAM* modeli, no risinājumu datubāzes izvēloties implementācijas katrai konfigurācijas pārvaldības darbībai. Rezultātā tiek iegūts *PSAM* modelis, no kura vēlāk automātiski uzģenerējas *SM*, kas parāda, kādi rīki ir jāintegrē savā starpā.

EAF metodoloģijas ieviešana beidzas tad, kad konfigurācijas pārvaldības problēmvidē tiek implementēta izejas koda pārvaldības sistēma atbilstoši *SCBM* modelim, realizētas visas integrācijas, kas attēlotas *SM* modelī, un ir implementēts *PSAM* modelis.

3.12. Nodaļas kopsavilkums

Nodaļa secīgi apraksta jaunu modeļvadāmu pieeju konfigurācijas pārvaldības procesu ieviešanai un metodoloģiju, kas šo pieeju realizē. Metodoloģijas izstrādes gaitā tika izdarīti šādi secinājumi:

- jauna modeļvadāma pieeja ir orientēta uz konfigurācijas pārvaldības procesu pakāpenisku ieviešanu, pateicoties modelēšanai. Galvenā ideja ir modeļu veidošanā, pakāpeniski samazinot abstrakcijas līmeni ar transformācijas likumu palīdzību un konkrētu risinājumu izvēli no speciālas datubāzes;
- konfigurācijas pārvaldības ieviešanas metodoloģija sākotnēji ir abstrakta. Ir uzskaitīti modeļu veidi un to nozīme, balstoties uz literatūru par konfigurācijas pārvaldību. Lai

gan turpinājumā katram modelim tiek piedāvāta noteikta realizācija, teorētiski *MTM* pieeju un *EAF* metodoloģiju var realizēt arī citādāk, saglabājot definētos modeļu veidus un to mērķus;

- piedāvātā pieeja neliek obligāti lietot konfigurācijas pārvaldības procesā kādus speciālus rīkus vai tehnoloģijas, taču orientēta uz iespēju lietot jau esošus risinājumus, bet ar nosacījumu, ka tie glabāsies vienoti, centralizēti un objektorientēti. Tas nozīmē, ka viena risinājuma tehniskajai realizācijai nav nekādu darbību, kas ietekmē cita risinājuma realizāciju;
- paredzams, ka piedāvātās metodoloģijas ieviešana prasīs zināmu laiku esošo risinājumu restrukturizācijai, lai tos varētu uzglabāt tādā veidā, kādu piedāvā jaunā metodoloģija. Teorētiski ilgtermiņā projekts ietaupīs līdzekļus, ko patērē konfigurācijas pārvaldībai, taču praktiski pieeja rūpīgi jānotestē ar reāliem piemēriem, lai pārliecinātos par ieguvumiem un identificētu trūkumus. Tieši tāpēc promocijas darba nākamajā nodaļā tiks veikta jaunās modeļvadāmās pieejas testēšana. Nodaļā tiks parādīts testa piemērs, kas vēlreiz secīgi ilustrē *EAF* metodoloģijas katras komponentes lietošanu un nozīmi. Vēlāk tiks definēti kritēriji, pēc kuriem tiks vērtēts metodoloģijas ieguvums dažādos programmatūras izstrādes projektos, un nodaļas beigās, balstoties uz trūkumiem, ko identificēs eksperimentu rezultātā, tiks piedāvāti uzlabojumi, kas potenciāli var palielināt metodoloģijas ieviešanas ieguvumus.

4. MODEĻVADĀMAS KONFIGURĀCIJAS PĀRVALDĪBAS METODOLOĢIJAS APROBĀCIJA UN TESTĒŠANA

4.1. Modeļvadāmas pieejas rezultātu aprobācija un testēšanas gaita

EAF metodoloģijas aprobācija un testēšana notika šādos posmos:

- metodoloģijas pamati tika prezentēti starptautiskajās konferencēs un tika publicēti atbilstošajos konferenču rakstu krājumos [BAR 2015, BAR 2014a, BAR 2014b, BAR 2014c, BAR 2014d, BAR 2014e, BAR 2014f]. Gatavojot publikācijas, tika iegūta vērtīga rekomendācija un noderīgi recenzentu komentāri, kas palīdzēja ne tikai uzlabot metodoloģijas teorētiskus pamatus, bet arī labāk sagatavoties praktiskajiem eksperimentiem metodoloģijas testēšanas laikā;
- tika izstrādāts programmatūras prototips modeļu veidošanai un attēlošanai, lai varētu veikt praktiskus eksperimentus. Prototipu veidoja students Jānis Locāns RTU mācību kursā «Adaptīvas datu apstrādes sistēmas» kā individuāls uzdevums programmēšanā. Vēlāk izstrādātā programmatūra tika attīstīta studenta bakalaura darbā. Promocijas darba autors, balstoties uz *EAF* metodoloģijas pamatiem, sastādīja minētās programmatūras prototipa prasību specifikāciju, kā arī veica izejas koda apskati. Pēc sākotnējiem eksperimentiem promocijas darba autors attīstīja dizainu minētajam programmatūras prototipam;
- uzņēmumā SIA «*Tieto Latvia*» tika izveidota konfigurācijas pārvaldības kompetences grupa 17 cilvēku sastāvā. Grupā tika iekļauti vecākie un vadošie tehniskie speciālisti, kas ikdienā nodarbojas ar programmatūras konfigurācijas pārvaldības procesiem. Kompetences grupa tika iepazīstināta ar *EAF* metodoloģijas teorētiskiem pamatiem un darbības principiem, kā arī tika sastādīts eksperimentu plāns;
- balstoties uz praktisku eksperimentu rezultātiem un starptautisku konferenču recenzentu piezīmēm un rekomendācijām, tika veiktas šādas darbības:
    - izdarīti secinājumi par *EAF* metodoloģijas ieguvumiem, trūkumiem un ierobežojumiem;

- tika izstrādātas rekomendācijas metodoloģijas uzlabojumiem. Vēlāk *EAF* metodoloģijā tika veikta virkne uzlabojumu un papildinājumu, kā arī vēl viena eksperimentu kārta, kurā minētie uzlabojumi tika testēti. Promocijas darba nākamajā nodaļā detalizēti tiks aprakstīti visi uzlabojumi un atkārtotu eksperimentu rezultāti;
- tika izstrādātas praktiskās vadlīnijas un rekomendācijas, kā promocijas darbā izstrādāto metodoloģiju ieviest informācijas tehnoloģijas uzņēmumā, kādi būtu ierobežojumi, riski un ieguvumi;
- balstoties uz visiem eksperimentu rezultātiem, tika apkopotas *EAF* metodoloģijas atšķirības no citiem esošajiem konfigurācijas pārvaldības risinājumiem, kā arī iezīmēti ieguvumi, ko sniedz *EAF* metodoloģija.

4.2. Eksperimentu sagatavošana un plāns

Kā jau tika minēts, eksperimentiem tika izveidota kompetences grupa uzņēmumā SIA «*Tieto Latvia*». Grupā tika iekļauti vecākie un vadošie tehniskie speciālisti, kas ikdienā strādā ar konfigurācijas pārvaldības procesiem.

Eksperimentu mērķi:

- salīdzināt konfigurācijas pārvaldības automatizācijas ieviešanas laiku pēc vecajām metodēm un pēc *EAF* metodoloģijas;
- salīdzināt kļūdainu būvējumu skaitu projektos pirms un pēc *EAF* metodoloģijas ieviešanas;
- balstoties uz salīdzināšanu, noteikt, kā mainās konfigurācijas pārvaldības automatizācijas ieviešanas laiks, kļūdainu būvējumu skaits projektā, kā arī būvējumu kopējais skaits.

Eksperimentu nosacījumi:

- eksistē vismaz viens aktīvs programmatūras izstrādes projekts, kuram ir vismaz viena testa vide;
- projektā ir ieviesta programmatūras konfigurācijas pārvaldība, realizēti galvenie konfigurācijas pārvaldības uzdevumi, kas ir aprakstīti promocijas darba pirmajā nodaļā;
- konfigurācijas pārvaldības process vismaz daļēji ir automatizēts.

Eksperimentu metodika un darbības:

- eksperimentiem tika izvēlēti pieci programmatūras izstrādes un uzturēšanas projekti. Lai paaugstinātu eksperimentu ticamību, tika izvēlēti projekti ar dažādām izstrādes tehnoloģijām;
- speciālistu kompetences grupai uzņēmumā SIA «*Tieto Latvia*» tika organizētas apmācības, kurās speciālisti tika iepazīstināti ar piedāvāto metodoloģiju un modeļiem. Apmācībās tika demonstrēts arī programmatūras prototips *EAF* metodoloģijas modeļu attēlošanai;
- tika izstrādāts risinājumu izvēles modulis konfigurācijas pārvaldības risinājumu glabāšanai, kas tika aprakstīts promocijas darba iepriekšējā nodaļā. Vadošie programmētāji izstrādāja programmatūru, kas ļauj pārvaldīt automatizācijas risinājumus konfigurācijas pārvaldības uzdevumiem. Programmatūrai bija šādi elementi:
    - *Oracle* datubāze, kas realizē datubāzes *ER* diagrammu, kas redzama attēlā 3.14;
    - *Oracle ADF* forma, kas ļauj ievadīt datubāzē jaunu konfigurācijas pārvaldības automatizācijas risinājumu;
    - *Oracle ADF* forma, kas saņem gatavu *PIAM* modeli un ļauj no minētās datubāzes izvēlēties automatizācijas risinājumu katrai konfigurācijas pārvaldības darbībai. Rezultātā tika iegūts *PSAM* modelis;
- izveidotais risinājumu izvēles modulis tika papildināts ar risinājumiem, veicot šādus soļus:
    - tabulas «*ContinuousIntegrationServer*» papildināšana ar risinājumiem konfigurācijas pārvaldības serveriem. Šajā tabulā tika ievietota informācija par katru konfigurācijas pārvaldības serveri, kas tika lietots minētajos piecos programmatūras izstrādes projektos. Konfigurācijas pārvaldniekam, kas bija atbildīgs par katru konkrēto projektu, vajadzēja sagatavot instrukciju, kā instalēt konfigurācijas pārvaldības serveri, noteikt rīkus, kas ir vajadzīgi papildus, kā arī apkopot esošos risinājumus, kas ļauj atvieglot konfigurācijas pārvaldības servera sagatavošanu. Kad tas tika izdarīts, visu minēto

informāciju bija nepieciešams ievietot datubāzē, tabulā «*ContinuousIntegrationServer*». Tika aizpildīti šādi atribūti:

- ***Platform*** – platforma, kurā funkcionē konkrētais konfigurācijas pārvaldības serveris;
- ***SolutionName*** – servera unikālais nosaukums;
- ***NeededTools*** – rīku uzskaitījums, ko ir jāinstalē, lai varētu aktivizēt konfigurācijas pārvaldības serveri;
- ***SetupNotes*** – konfigurācijas pārvaldības servera detalizēta instalācijas instrukcija;
- ***LocationsOfSolutions*** – gatavu risinājumu atrašanas vietas (ja tādas ir).

o tabulas «*Solution*» aizpildīšana. Katram konfigurācijas pārvaldniekam, kas atbild par konkrētu programmatūras projektu, bija nepieciešams restrukturizēt atbilstošus automatizācijas risinājumus tādā veidā, kā to paredz *PSAM* modelis un risinājumu izvēles modulis. Veicot esošo automatizācijas risinājumu restrukturizāciju, katrs no tiem tika ielikts tabulā «*Solution*», aizpildot šādus tabulas atribūtus:

- ***Platform*** – platforma, kurā funkcionē konkrētais automatizācijas risinājums;
- ***Action*** – kuru konfigurācijas pārvaldības darbību automatizē;
- ***SolutionName*** – automatizācijas risinājuma unikālais nosaukums;
- ***NeededTools*** – rīki, kas nepieciešami risinājuma ieviešanai un lietošanai;
- ***LocationsOfSolutions*** – atkārtoti izpildāmā koda atrašanas vieta;
- ***Description*** – papildu norādījumi par risinājuma ieviešanu;

- tika izstrādāti *EAF* metodoloģijas vērtēšanas kritēriji un rādītāji, kas ir nepieciešami kritēriju aprēķinam. Nodaļas turpinājumā tiks sniegta detalizētāka informācija par rādītājiem un kritērijiem. Par katru no pieciem projektiem no SIA «*Tieto Latvia*» korporatīvas darba laika uzskaites sistēmas tika iegūti šādi dati:
  - laiks, kas tika patērēts konfigurācijas pārvaldības procesu sākotnējai ieviešanai;

    - vidējais laiks nedēļā, kas tika patērēts konfigurācijas pārvaldības procesu regulārai uzturēšanai;
    - nedēļu skaits līdz paredzētam projekta beigu datumam.

No katra projekta konfigurācijas pārvaldības datubāzes tika iegūta šāda informācija:

- programmatūras būvējumu skaits;
- programmatūras kļūdaino būvējumu skaits.

Katrā no pieciem projektiem tika veikts *EAF* metodoloģijas ieviešanas eksperiments pēc šāda plāna:

- izmantojot programmatūras prototipu, tika izveidots vižu modelis. Modelī tika iekļautas visas izstrādes, testa un ekspluatācijas vide;
- vižu modelis tika transformēts *PIAM* modelī un *SCBM* modelī;
- konfigurācijas pārvaldnieks, strādājot ar risinājumu izvēles moduli, papildināja konfigurācijas pārvaldības darbības *PIAM* modelī ar implementācijas detaļām. Rezultāta tika iegūts *PSAM* modelis;
- no *PSAM* modeļa tika iegūts *SM* modelis, kas parādīja visus rīkus, ko bija nepieciešams savstarpēji integrēt;
- konfigurācijas pārvaldnieks izstrādāja servisu modeli (*SM*) un izejas koda pārvaldības sistēmu atbilstoši *SCBM* modelim;
- *PSAM* modelis tika implementēts konfigurācijas pārvaldības serverī;
- tika fiksēts patērētais laiks, sākot ar vižu modeļa veidošanu un beidzot ar *PSAM* modeļa implementāciju;
- programmatūras konfigurācijas pārvaldības process funkcionēja pēc *EAF* metodoloģijas trīs mēnešu laikā. Pēc tam tika fiksēti šādi rādītāji:
    - vidējais laiks nedēļā, kas bija nepieciešams procesu uzturēšanai un procesa kļūdu labojumiem;
    - programmatūras būvējumu kopējais skaits;
    - programmatūras kļūdaino būvējumu skaits.
- tika organizēta sapulce, kurā kompetences grupas dalībnieki izvērtēja sākotnēji iegūtos un eksperimentu gaitā fiksētos rādītājus par patērēto laiku un būvējumu skaitu.

Tabulās 4.1. un 4.2. ir aprakstītas darbības, kas tika veiktas konfigurācijas pārvaldības automatizācijas ieviešanai pirms un pēc *EAF* metodoloģijas ieviešanas uzņēmumā SIA «*Tieto Latvia*».

4.1. tabula

Konfigurācijas pārvaldības automatizācijas darbības pirms *EAF* metodoloģijas ieviešanas

Darbība	Apraksts	Atbildīga persona
Projekta vižu plāna sastādīšana	Izmantojot kādu vizualizācijas rīku, piemēram, *MS Visio*, tika uzzīmēts projekta vižu plāns.	Konfigurācijas pārvaldnieks
Būvējuma veidošanas automatizācijas skripta izstrāde	Tika izstrādāts jauns skripts būvējuma veidošanai no izejas koda. Sakarā ar to, ka katrā projektā būvējumu veidošanas skripts bija specifisks un tajos bija arī citu konfigurācijas pārvaldības soļu automatizācija, katrā jaunajā projektā skriptu nācās izstrādāt no nulles, labākajā gadījumā no citiem projektiem ņemot tikai atsevišķus fragmentus.	Konfigurācijas pārvaldnieks Vadošais programmētājs
Programmatūras instalācijas skripta izstrāde	Tika izstrādāts jauns programmatūras instalācijas skripts, atsevišķus fragmentus ņemot no citu projektu skriptiem.	Konfigurācijas pārvaldnieks Vadošais programmētājs
Versiju kontroles automatizācijas risinājumu izstrāde	Tika izstrādāti jauni skripti versiju kontroles automatizācijai.	Konfigurācijas pārvaldnieks

Integrācija ar pieteikumu apstrādes sistēmu	Tika izstrādāti skripti, kas nodrošina konfigurācijas pārvaldības servera integrāciju ar pieteikumu apstrādes sistēmu.	Konfigurācijas pārvaldnieks
Izstrādātu risinājumu testēšana, kļūdu labošana	Tika veiktas fiktīvas izmaiņas programmatūrā, lai īstenojot galvenās konfigurācijas pārvaldības darbības, varētu notestēt izstrādātos automatizācijas risinājumus.	Konfigurācijas pārvaldnieks Vadošais programmētājs

4.2. tabula

Konfigurācijas pārvaldības automatizācijas darbības pēc *EAF* metodoloģijas ieviešanas

Darbība	Apraksts	Atbildīga persona
Vižu modeļa izstrāde	Tika izveidots vižu modelis.	Konfigurācijas pārvaldnieks
PIAM un *SCBM* modeļu iegūšana	Izmantojot transformācijas likumus, no vižu modeļa tika iegūti *PIAM* un *SCBM* modeļi. Atbilstoši *SCBM* modelim konfigurācijas pārvaldnieks izveido izejas koda zarus un apraksta minēto zaru pārvaldības stratēģiju.	Konfigurācijas pārvaldnieks
PSAM modeļa iegūšana	Izmantojot risinājumu izvēles moduli, konfigurācijas pārvaldnieks papildina *PIAM* modeļa darbības ar realizācijas	Konfigurācijas pārvaldnieks

	detaļām, rezultātā tiek iegūts *PIAM* paplašinātais variants – *PSAM* modelis.	
SM modeļa iegūšana	No *PSAM* modeļa automātiski tiek uzģenerēts *SM* (Servisu modelis), kas parāda kādiem rīkiem jābūt savstarpēji integrētiem. Šajā brīdī, nepieciešamības gadījumā, vadošais programmētājs kopā ar konfigurācijas pārvaldnieku papildina esošus integrācijas risinājumus, kas atrodas risinājumu izvēles modulī.	Konfigurācijas pārvaldnieks Vadošais programmētājs
SM un *PSAM* modeļu realizācija konfigurācijas pārvaldības serverī	Modeļu realizācija automatizācijas skriptu veidā. Atbilstošo skriptu palaišana konfigurācijas pārvaldības serverī. Tika veiktas fiktīvas izmaiņas programmatūrā, lai īstenojot galvenās konfigurācijas pārvaldības darbības, varētu notestēt izstrādātos automatizācijas risinājumus. Testēšanas gaitā galvenokārt tiek pārbaudītas tikai tās daļas, kuras tika izstrādātas speciāli konkrētajam projektam, jo pārējais izejas kods tiek paņemts no	Konfigurācijas pārvaldnieks Vadošais programmētājs

	risinājuma izvēles moduļa gatavā veidā.	

Salīdzinot tabulas 4.1. un 4.2. var redzēt, ka pirms *EAF* ieviešanas, konfigurācijas pārvaldības automatizācijas ieviešana pārsvarā tika balstīta uz jaunu skriptu izstrādi. Atkārtoti izmantot jau eksistējušo projektu risinājumus varēja tikai daļēji.

Savukārt, tabula 4.2. parāda, ka pēc *EAF* ieviešanas uzņēmumā, konfigurācijas pārvaldības automatizācijas ieviešana ir balstīta uz modeļu veidošanu. Galvenais ieguvums ir tāds, ka no nulles ir jāizstrādā tikai atsevišķas mazas daļas no izejas koda, kas automatizē procesu. Tas potenciāli aizņem mazāk laika, veicina efektīvu pieredzes apmaiņu starp projektiem un ietaupa laiku automatizācijas risinājumu testēšanā, jo tiek testētas tikai projektam specifiskās daļas.

4.3. Konfigurācijas pārvaldības modeļu veidošanas prototips

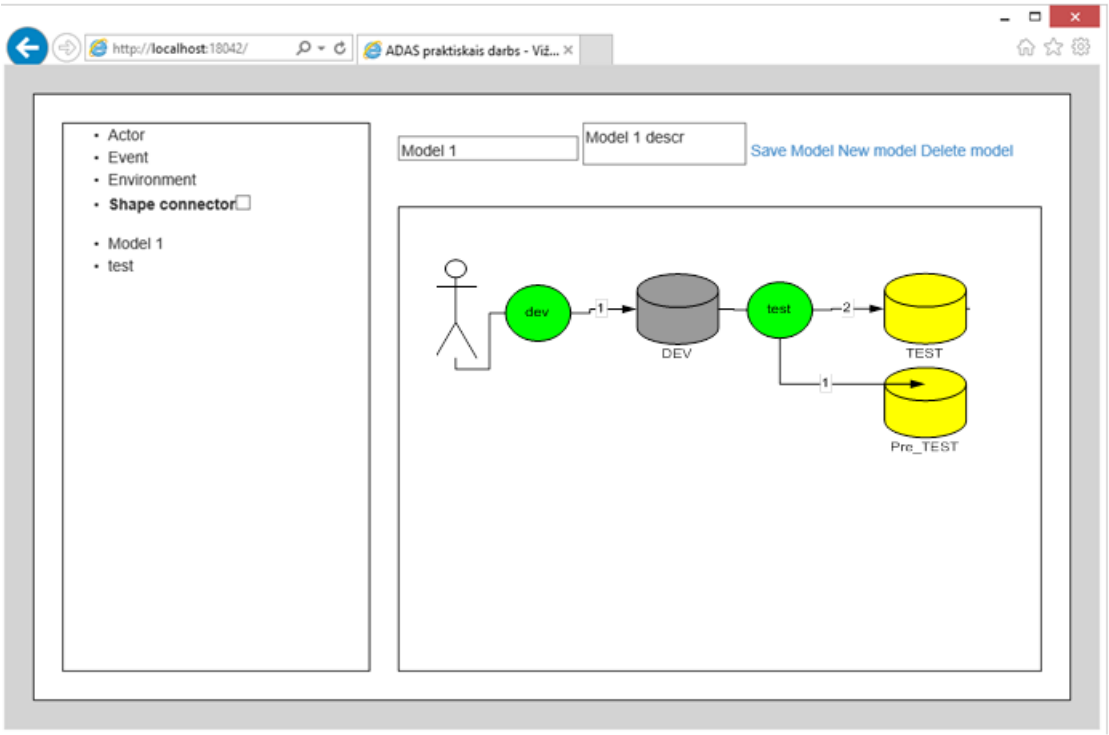
Kā jau tika minēts, programmatūru *EAF* modeļu veidošanai ir izstrādājis bakalaurants, pildot individuālu uzdevumu programmēšanā. Prototips izstrādāts, izmantojot *.NET*, *HTML5*, *CSS* un *JavaScript* tehnoloģijas. Papildus *JavaScript* tika izmantots arī *jQuery* un bibliotēka, kas ir paredzēta grafisku elementu zīmēšanai – *KineticJs*, kas atvieglo un uzlabo darbu ar *HTML5 Canvas* elementiem. Turpmāk tiek apsvērta iespēja projektu pārnest uz *AngularJs*, kas atvieglotu tā uzturēšanu. Ņemot vēra faktu, ka bakalaurantam nav lielas praktiskās pieredzes programmatūras izstrādē atbilstoši nozares standartiem, izstrādātajā prototipā ir ievērojamas dizaina nepilnības. Taču tas netraucēja veidot konfigurācijas pārvaldības modeļus eksperimentiem, kā arī neietekmēja modeļu būtību. Ar modeļu validāciju nodarbojās konfigurācijas pārvaldības kompetences grupa uzņēmumā SIA «*Tieto Latvia*». Pēc veiktajām pārbaudēm prototipa darbība tika atzīta par korektu, lai gan tika izteikti vairāki priekšlikumi uzlabojumiem.

Prototips atvieglo šādu modeļu veidošanu:

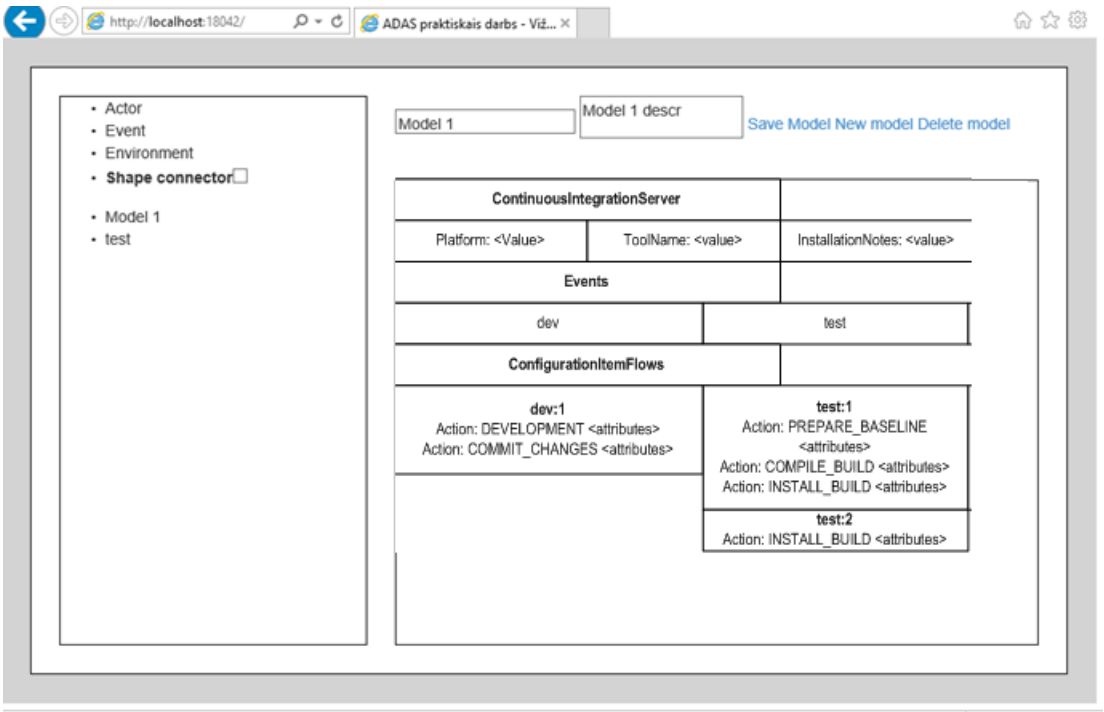
- vižu modelis;
- *PIAM* modelis;
- *PSAM* modelis.

Papildus tam prototips ļauj veikt automātisku transformāciju no *EM* uz *PIAM* modeli atbilstoši «E->P» transformācijas likumiem, kas tika aprakstīti promocijas darba iepriekšējā

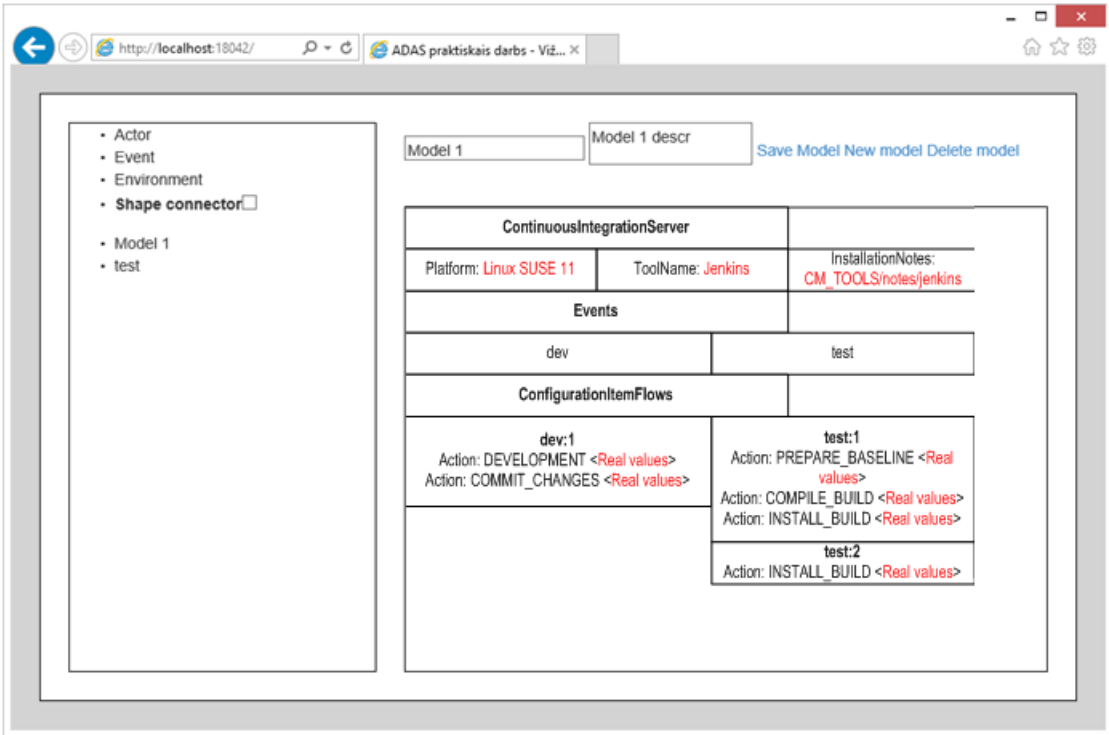
nodaļā. Attēlos 4.1.–4.3. var redzēt prototipa ekrānuzņēmumus un attiecīgi *EM*, *PIAM* un *PSAM* modeļus.



4.1. att. Vižu modeļa piemērs



4.2. att. *PIAM* modeļa piemērs



4.3.att. *PSAM* modeļa piemērs

Attēlos 4.1.–4.3. ir attēloti vienkāršoti modeļi, lai ilustrētu prototipa darbību. Vižu modelim attēlā 4.1. ir šādi elementi:

- vides *DEV, Pre_TEST* un *TEST*;
- viens aktieris jeb programmētājs, kas veic izmaiņas *DEV* vidē;
- notikumi «*dev*» un «*test*». Notikumā «*dev*» ir viena programmatūras izmaiņu plūsma, kas nodrošina izmaiņu pārnešanu no programmētāja uz *DEV* vidi (izstrāde, izmaiņu saglabāšana versiju kontroles repozitorijā). Savukārt notikumam «*test*» – divas plūmas. Pirmajā plūsmā programmatūras izmaiņas pārnes uz *Pre_TEST* vidi, kur testē, vai atbilstošais būvējums ir korekts. Ja būvējums ir uzinstalēts veiksmīgi, to instalē *TEST* vidē otrajā notikuma «*test*» plūsmā.

Pēc transformācijas likumu «E->P» izmantošanas vižu modelis tiek transformēts *PIAM* modelī, kas ir redzams attēlā 4.2. Transformācijas rezultātā notikumu plūsmām tika piešķirtas šādas darbības no *PIAM* meta-modeļa:

- **notikums «*dev*», plūsma «*1*»:** *DEVELOPMENT* – izmaiņu izstrāde atbilstoši projekta vadlīnijām, *COMMIT_CHANGES* – izmaiņu saglabāšana versiju kontroles sistēmā atbilstoši projekta vadlīnijām;

- **notikums «*test*», plūsma «*1*»:** *PREPARE_BASELINE* – sagatavo izejas kodu testa videi, *COMPILE_BUILD* – uzbūvē produktu, *INSTALL_BUILD* – uzinstalē būvējumu *Pre_TEST* vidē;
- **notikums «*test*», plūsma «*2*»:** ja iepriekšējās plūsmas darbības ir bijušās veiksmīgas, gatavu būvējumu uzinstalē *TEST* vidē.

Attēlā 4.3. ir redzams *PSAM* modelis, kurā lietotājs jau ir definējis implementāciju nepārtrauktās integrācijas serverim un minētajām *PIAM* darbībām. Attēlā var redzēt, ka konfigurācijas pārvaldības procesi tiks izpildīti *Linux SUSE 11* platformā, nepārtrauktās integrācijas serveris būs *Jenkins*.

4.4. *EAF* metodoloģijas modeļu veidošanas piemērs

Lai būtu vieglāk uztvert eksperimentu rezultātu aprakstu, tiks dots neliels testa piemērs, kurā varēs redzēt modeļu lietošanu, transformāciju un konfigurācijas pārvaldības uzdevumu risinājumu izvēles gaitu no risinājumu izvēles moduļa. Aprakstot eksperimentu rezultātus, uzsvars būs likts uz rezultātiem, ko praksē ir devuši modeļi nevis uz pašu modeļu veidošanas procesu. Tas ir tāpēc, ka vairāki ekrānuzņēmumi krietni palielinās darba apjomu un neļaus koncentrēties tieši uz rezultātiem un modeļu ieguvumiem, kas ir svarīgi promocijas darba kontekstā. Tāpēc detalizēti modeļu veidošanas process tiks parādīts tikai vienam projektam.

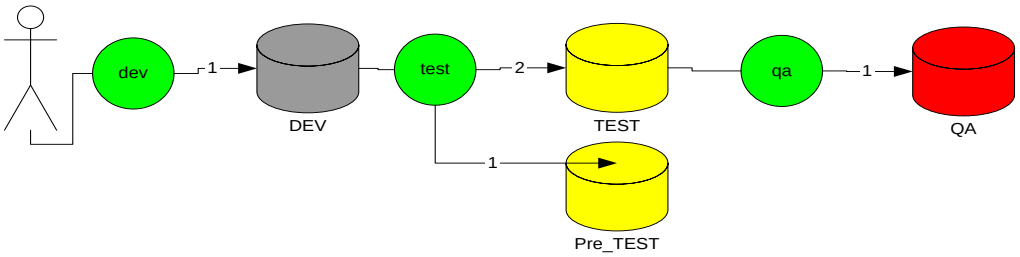
a. Testa projekta apraksts

Testa nolūkos tiek parādīts piemērs, kur tehnoloģijā *Ruby On Rails* tiek izstrādāta norēķinu sistēma kādam uzņēmumam, kas ir SIA «*Tieto Latvia*» klients. Saglabājot minētā uzņēmuma komercnoslēpumu, klienta nosaukums netiek atklāts. Paredzams, ka projektā izstrādātā sistēma būs pieejama timeklī un darbības princips būs šāds: lietotājs pieslēdzas sistēmai, iepazīstās ar saviem rēķiniem par pakalpojumiem, ko sniedz programmatūras pasūtītājs. Lietotājam ir iespēja apmaksāt rēķinu. Programmatūra nodrošina pieslēgšanos pie internetbankām, ar kurām programmatūras pasūtītājam ir noslēgti atbilstoši līgumi. Kad lietotāja maksājums ir iesniegts bankā, informācija caur tīmekļa servisu atnāk uz pamatsistēmu, un maksājums tiek attiecināts pret atbilstošo lietotāja rēķinu. Kopš šā brīža lietotājs redz to, ka viņa rēķins ir apmaksāts. Minēta programmatūra tiek izstrādāta pēc *SCRUM* izstrādes metodoloģijas. Brīdī, kad tika modelēts šis piemērs, programmatūras izstrādes projektā bija akcepttestēšanas fāze, pasūtītājs reizi nedēļā saņēma jaunu sistēmas

versiju, kurā bija visas uz tobrīd notestētās izmaiņas. Programmatūras jaunā versija tika instalēta akcepttestiem paredzētajā vidē. Ja pasūtītājs akcepttesta vidē atklāj kļūdas, tās piesaka SIA «*Tieto Latvia*» izstrādātāju komandai, un to risinājums tiek piegādāts kopā ar kādu no nākamajām programmatūras versijām. Lai mazinātu testējamā produkta pieejamības riskus, ārkārtas piegādes netiek veiktas. Projekta nosaukums ir «*Test Solution*».

b. Vižu modelis

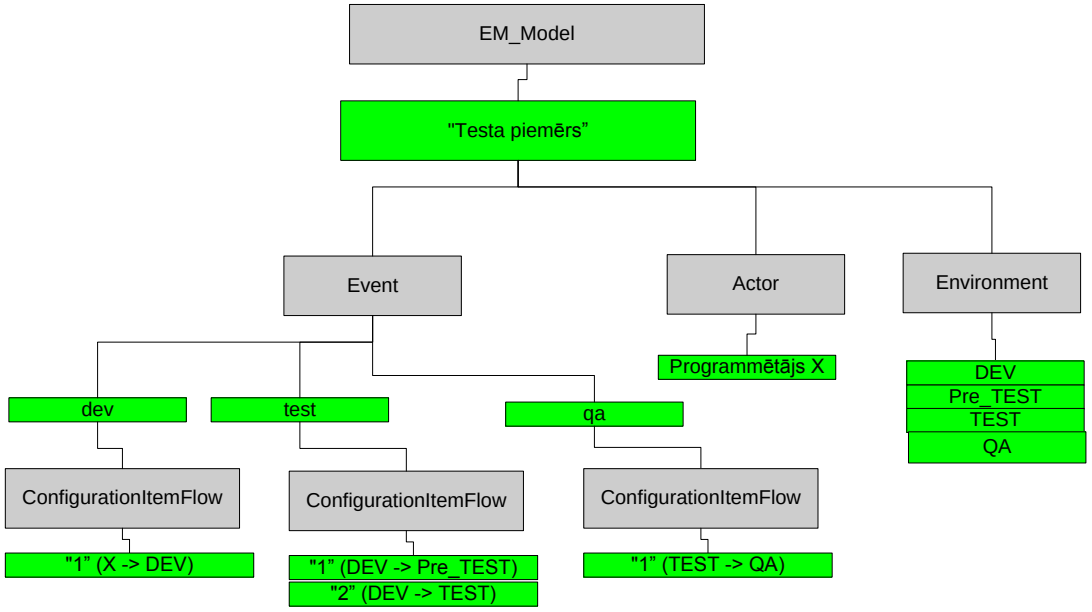
Attēlā 4.4. ir redzams tikko aprakstītā projekta vižu modelis.



4.4. att. Projekta «*Test Solution*» vižu modelis

Kā redzams, projektā «*Test Solution*» tiek lietotas četras vides: *DEV*, *Pre_TEST*, *TEST* un *QA*. *DEV* vidē programmētājs veic izmaiņas, pēc tam izmaiņas tiek uzliktas *Pre_TEST* vidē nolūkā pārbaudīt, vai tās ir korektas un nenograuj vidi. Ja šis solis ir veiksmīgs, tās pašas izmaiņas tiek uzliktas *TEST* vidē, kur notiek reālais testēšanas process. Uz *TEST* vidi vienmēr pārnes pilnīgi visas izmaiņas, kas no *DEV* vides tiek saglabātas versiju kontroles sistēmā. Reizi nedēļā visas notestētas izmaiņas no *TEST* vides tiek pārnesta uz *QA* vidi, kas ir pasūtītāja akcepttesta vide. Projekta «*Test Solution*» vižu modelis *XML* formātā tiek parādīts 1. pielikumā.

Kā redzams 1. pielikumā, vižu modelī ir trīs notikumi (*Event*), kuros konfigurācija tiek pārnesta starp norādītajām vidēm. Tālāk modelis *XML* formātā tiek apstrādāts ar universālu kompilācijas algoritmu, kas ir aprakstīts iepriekšējā nodaļā. Rezultātā tika izveidots aizpildīts *EM_Model* tipa objekts «Testa piemērs» ar visu apakšstruktūru. Attēlā 4.6. redzama klašu hierarhija, ko izveidoja vižu meta-modeļa kompilācijas algoritms. Ar pelēku krāsu ir apzīmēti klašu tipi, kas arī ir aprakstīti iepriekšējā nodaļā, ar zaļu krāsu ir apzīmētas klases. Tātad, pēc kompilācijas ir 13 objekti. Katram objektam visi atribūti ir aizpildīti, un augtākā hierarhijas līmeņa klase *EM_Model* ir gatava nodošanai transformācijas algoritmam «E->P», kuram vajadzētu izveidot *PIAM* modeli.



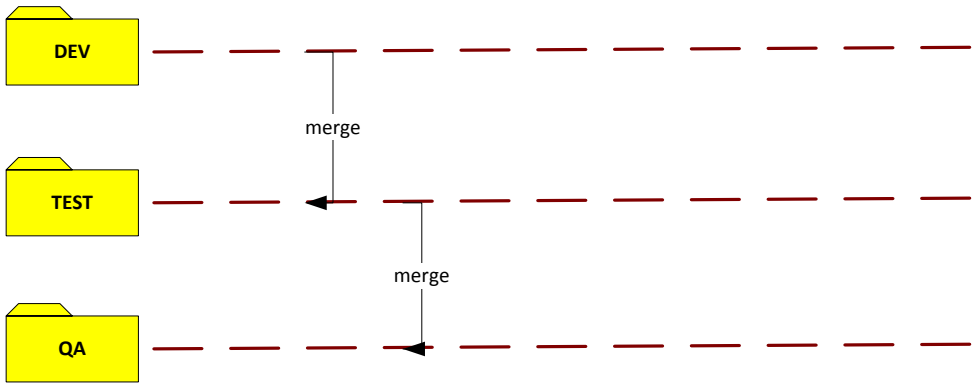
4.6. att. Projekta «*Test solution*» vižu modeļa klašu hierarhija

c. Izejas koda zarošanas modeļa iegūšana no vižu modeļa ar «E->S» algoritma palīdzību

Lietojot «E->S» transformācijas algoritmu un transformācijas likumus, kas tika aprakstīti darba iepriekšējā nodaļā, tika iegūts izejas koda zarošanas modelis ar šādiem elementiem:

- *DEV* vides zars: izejas koda zars, kurā izstrādātājs saglabā *DEV* vidē izstrādātas izmaiņas. Kad izmaiņas ir gatavas testēšanai, tās tiek sapludinātas ar citu zaru, kas atbilst TEST videi;
- *TEST* vides zars: izejas koda zars, kas atbilst *TEST* videi. Šeit nonāk izejas koda izmaiņas no *DEV* zara un tad, kad testēšanas process ir pabeigts, izmaiņas tiek sapludinātas ar zaru, kas atbilst pasūtītāja akcepttesta videi;
- *QA* vides zars: izejas koda zars, kas atbilst *QA* videi. Uz šo zaru izmaiņas nonāk no *TEST* zara, kad tās ir gatavas akcepttestēšanai.

Attēlā 4.7. var redzēt izejas koda zarošanas modeļa grafisko attēlojumu.



4.7. att. Projekta «*Test solution*» izejas koda zarošanas modelis

d. Vižu modeļa transformācija *PIAM* modelī

Transformācijas likumu «E->P» uzdevums: katrai konfigurācijas plūsmai vižu modelī noteikt konfigurācijas pārvaldības darbības no *PIAM* meta-modeļa. Tabulā 4.3. ir apkopots transformācijas rezultāts: visas konfigurācijas plūsmas no vižu modeļa, «E->P» likums, kas nostrādāja transformācijas laikā, darbības no *PIAMsource* elementa.

4.3. tabula

Projekta «Test Solution» «E->P» transformācijas likumu pielietošana

Notikums:Plūsma	**«E->P» likums, skaidrojums**	**Darbības no *PIAM* meta-modeļa**
dev:1	**Likums 1:** Ja izmaiņas netiek pārnestas no kādas vides, bet tās veido manuāli, jābūt *DEVELOPMENT* darbībai, kas šā darba kontekstā nozīmē likumus, kas reglamentē produkta izmaiņu veikšanu. Izstādes beigās rezultāti obligāti jāsaglabā. Tiek lietota darbība *COMMIT_CHANGES*, kas nozīmē, ka visām produktu vienībām jābūt pakļautām versiju kontrolei, izstrādātājam jaunas versijas jāpiefiksē atbilstoši projekta vadlīnijām.	*DEVELOPMENT* *COMMIT_CHANGES*
test:1	**Likums 7:**	*PREPARE_BASELINE*

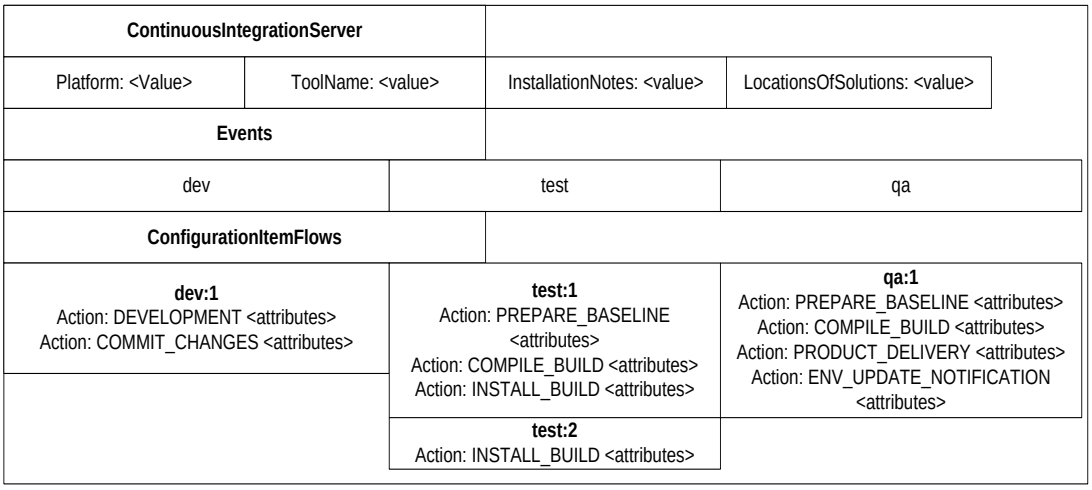
	Ja konfigurācija ir jāpārnes no oriģinālās vides, kuru neuztur pasūtītājs, uz citu vidi, kuru arī neuztur pasūtītājs, un ja šī plūsma ir pirmā notikumā, tad nepieciešams sagatavot vides izejas kodu, uzkompilēt produktu un uzinstalēt to.	*COMPILE_BUILD* *INSTALL_BUILD*
test:2	**Likums 8:** Ja konfigurācija ir jāpārnes no oriģinālās vides, kuru neuztur pasūtītājs, uz citu vidi, kuru arī neuztur pasūtītājs, ja tā nav pirmā plūsma notikumā un ja kādā no iepriekšējām plūsmām produkts jau tika kompilēts, nepieciešams tikai uzinstalēt to, par pamatu ņemot jau esošo būvējumu.	*INSTALL_BUILD*
qa:1	**Likums 10:** Ja nepieciešams pārnest konfigurāciju no oriģinālas vides, kuru neuztur pasūtītājs, uz vidi, kuru uztur pasūtītājs, un ja tā ir pirmā plūsma notikumā, nepieciešams sagatavot vides izejas kodu, uzkompilēt produktu, sagatavot piegādi un sagaidīt apstiprinājumu, ka pasūtītājs produktu uzinstalējis. Pēc tam jāaktivizē citu notikumu, ja tāds ir paredzēts.	*PREPARE_BASELINE* *COMPILE_BUILD* *PRODUCT_DELIVERY* *ENV_UPDATE_NOTIFICATION*

Transformācijas rezultātā no vižu modeļa izveidojās *PIAM* modelis projektam «Ī». 2. pielikumā var redzēt pilnu *PIAM* modeli *XML* formātā.

Kā var redzēt 2. pielikumā, vižu modeļa realizācijai ir nepieciešamas vairākas darbības no *PIAM* meta-modeļa. Šīs darbības arī veido kodolu jaunizveidotajam *PIAM* modelim:

- ***DEVELOPMENT*** – darbība, kas nodrošina izmaiņu veikšanas kontroli;
- ***COMMIT_CHANGES*** – nodrošina izmaiņu kvalitatīvu saglabāšanu versiju kontroles sistēmā;
- ***PREPARE_BASELINE*** – uztur izejas koda repozitorija zarus, pārnes noteiktās izejas koda izmaiņas no viena zara uz citu;
- ***COMPILE_BUILD*** – uzbūvē programmatūru no izejas koda. Būvējums ir paredzēts instalācijai kādā no vidēm;
- ***INSTALL_BUILD*** – instalē uzbūvētu produktu noteiktā vidē;
- ***PRODUCT_DELIVERY*** – tiek sagatavots uzbūvētais produkts (izpildāmais fails vai faili), būvējuma instalācijas instrukcija, piegādē esošu izmaiņu tehniskais un funkcionālais apraksts un citi pavadošie dokumenti, kas formāli ir noteikti projekta normatīvajos dokumentos;
- ***ENV_UPDATE_NOTIFICATION*** – darbību kopa, kas jāveic, kad pasūtītājs paziņo, ka viņam nosūtīta programmatūra ir uzinstalēta *QA* vidē. Šajā brīdī nepieciešams reģistrēt šo faktu versiju kontroles sistēmā atbilstošiem izejas koda vienumiem un to versijām.

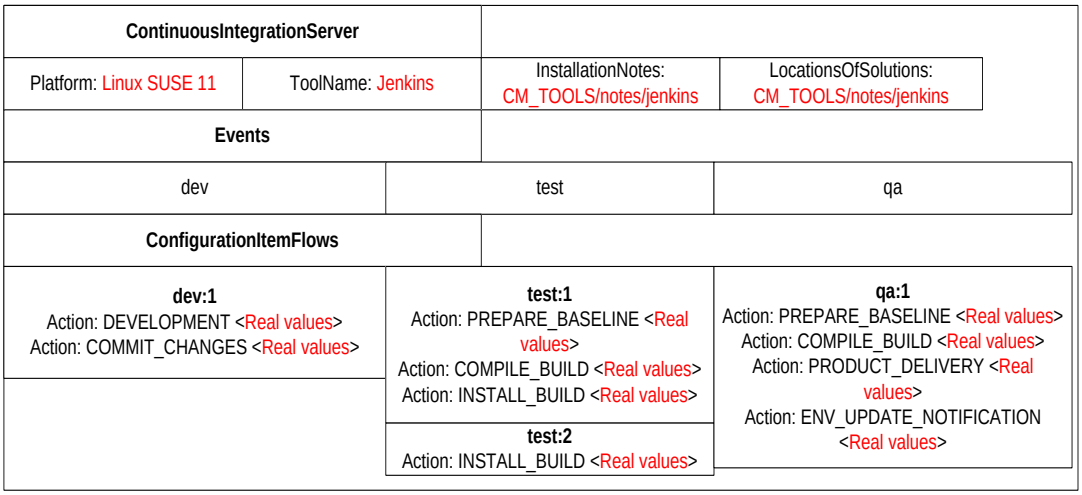
Attēlā 4.9. var redzēt *PIAM* modeļa vizuālu attēlojumu.



4.9. att. Projekta «*Test Solution*» *PIAM* modelis vizuālā formātā

Kad *PIAM* modelis ir gatavs, tas nonāk risinājumu izvēles modulī. Tur no datubāzes, kas ir definēta iepriekšēja nodaļā, katrai darbībai atbilstoši procedūrai tiek noteiktas

implementācijas detaļas. Rezultātā izveidojās *PIAM* paplašinājums jeb *PSAM* – platformas specifiskais darbību modelis. Attēlā 4.10. var redzēt «*Test Solution*» projekta *PSAM* modeli.



4.10. att. Projekta «*Test Solution*» *PSAM* modelis vizuālā formātā

Sakarā ar to, ka 4.10. attēlā nav redzamas darbību risinājumu detaļas, tās tiek parādītas tabulā 4.4.

4.4. tabula

Projekta «Test Solution» *PSAM* modeļa darbību risinājumi

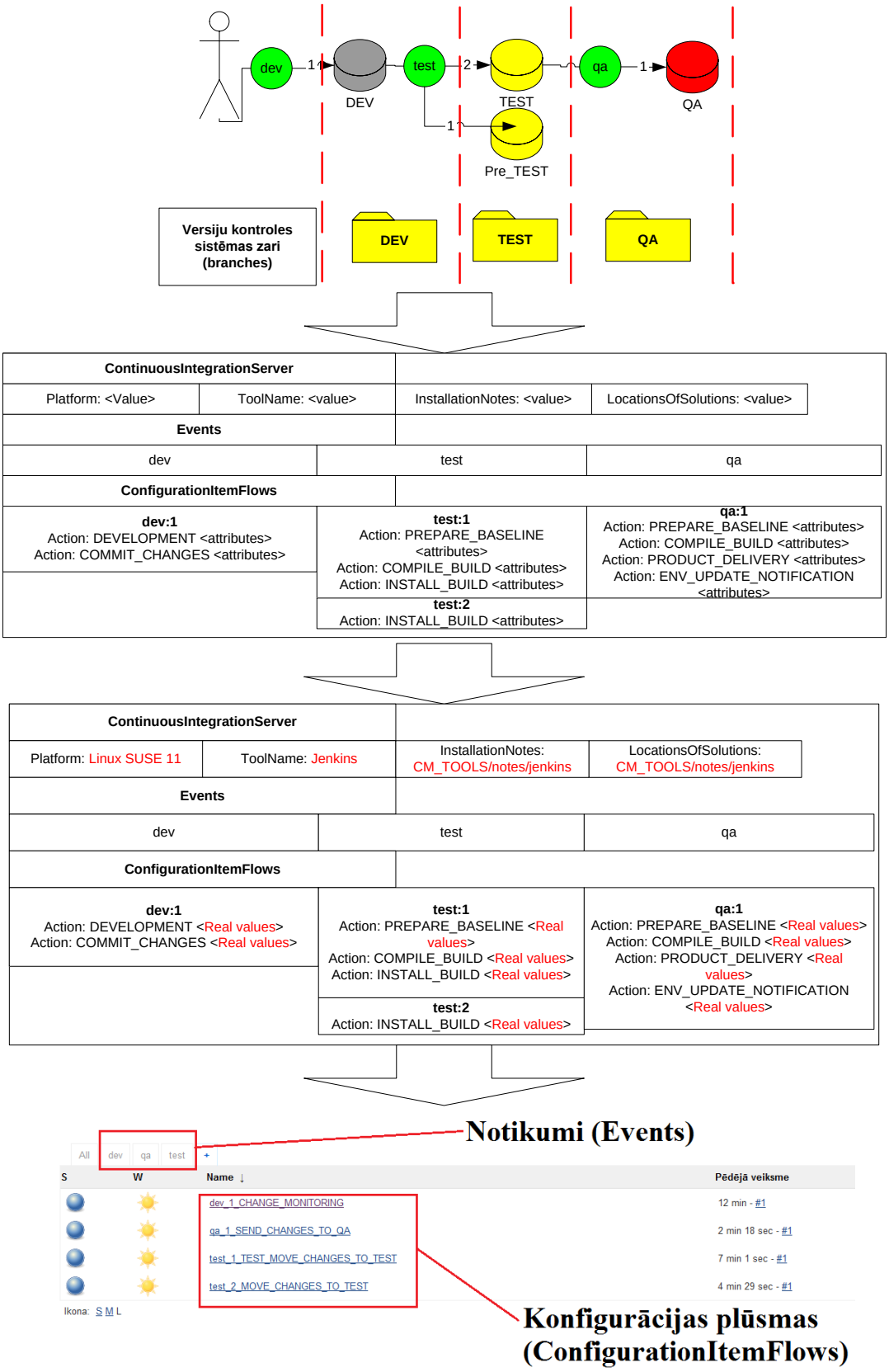
Darbība	Atribūtu vērtības, komentāri
DEVELOPMENT	***PlatformName:*** *Ruby on Rails* ***SolutionName:*** *Ruby on Rails Development Framework* ***NeededTools:*** *Ruby Development Tools 0.9.0.707021729NGT for EasyEclipse 1.2.2* ***LocationsOfSolutions:*** *Q:/Projects/Solutions/Development/RubyOnRails/Tools/DownloadSetupNotes.doc* ***Description:*** Apraksta procedūru, kā uz darbstacijas uzlikt visu nepieciešamo, lai sāktu izstrādes darbus «*Test Solution*» projektā. Ir minēti nepieciešamie rīki, to lejuplādes procedūra, norādījumi, instalācijas soļi un projekta specifisku lietu konfigurēšana.
COMMIT_CHANGES	***PlatformName:*** *Linux* ***SolutionName:*** *Subversion*

	NeededTools: *Subversion server, Subversion GNU client, Subversion Cmd client.* ***LocationsOfSolutions:*** *Q:/Projects/Solutions/VersionControlSystems/Subversion/WorkstationConfigurationGuide.doc,* *Q:/Projects/Solutions/VersionControlSystems/Subversion/SVN_Service.sh (Shell funkcijas informācijas iegūšanai, komandu izpilde no skriptiem),* *Q:/Projects/Solutions/VersionControlSystems/Subversion/hookScripts.sh (Skripti, kas kontrolē vai izmaiņas ir pareizi saglabātas),* *Q:/Projects/Solutions/VersionControlSystems/Subversion/branching_strategies.doc (Dokuments, kas apraksta SVN zaru veidošanu (branches) atkarībā no izejas koda zarošanas modeļa)* ***Description:*** Risinājums, kas nodrošina *Subversion* uzstādīšanu lokālajā darbstacijā, skripti, kas kontrolē, vai saglabātās izmaiņas ir korektas, zaru veidošanas stratēģija atkarībā no izejas koda zarošanas modeļa, *shell* funkcijas, kas nodrošina gan kvalitātes kontroles skriptu darbību, gan *SVN* komandu izsaukšanu no *Linux Shell* skriptiem.
PREPARE_BASELINE	***PlatformName:*** *Linux* ***SolutionName:*** *Subversion_Merge* ***NeededTools:*** *Subversion Cmd client* ***LocationsOfSolutions:*** *Q:/Projects/Solutions/VersionControlSystems/Subversion/SVN_FUNCTIONS.sh (Funkciju kopa, ko var izsaukt no shell skriptiem nodrošinot tādas operācijas kā svn merge, svn commit, svn revert utt.)* ***Description:*** Funkciju kopa, kas nodrošina darbu ar *Subversion* versiju kontroles sistēmu no *Linux Shell* skriptiem, lai varētu nodrošināt *SVN* komandu automatizāciju, izmantojot skriptus, ko darbinās *Jenkins* nepārtrauktās integrācijas serveris.
COMPILE_BUILD	***PlatformName:*** *Linux* ***SolutionName:*** *Capistrano Framework*

	NeededTools: *Capistrano, Ruby Gems, Ruby Gems Manager* ***LocationsOfSolutions:*** *Q:/Projects/Solutions/Compile_Build/Capistrano/capistrano_readme.doc (Dokuments, kas apraksta Capistrano Framework konfigurāciju un pielāgošanu konkrētam projektam. Šajā pašā direktorijā atrodas arī kompānijas ietvaros izstrādāti servisi, ko lieto visos Ruby projektos.)* ***Description:*** Risinājums, kas veido būvējumu *Ruby On Rails* tipa projektos.
INSTALL_BUILD	***PlatformName:*** *Linux* ***SolutionName:*** *Capistrano Framework* ***NeededTools:*** *Capistrano, Ruby Gems, Ruby Gems Manager* ***LocationsOfSolutions:*** *Q:/Projects/Solutions/Compile_Build/Capistrano/capistrano_readme.doc (Dokuments, kas apraksta Capistrano Framework konfigurāciju un pielāgošanu konkrētam projektam. Šajā pašā direktorijā atrodas arī kompānijā izstrādātie servisi, ko lieto visos Ruby projektos.)* ***Description:*** Risinājums, kas veido un instalē būvējumu *Ruby On Rails* tipa projektos, *Ruby* vižu pārvaldības mehānismi.
PRODUCT_DELIVERY	***PlatformName:*** *Linux* ***SolutionName:*** *CollectRubyDelivery* ***NeededTools:*** *Capistrano, ReleaseNoteGenerator(Custom tool), InstructionXMLGenerator (Custom tool)* ***LocationsOfSolutions:*** *Q:/Projects/Solutions/Make_Delivery/Ruby_Projects/delivery_creation_steps.doc – dokuments, kas apraksta produkta piegādes veidošanas procesu, Q:/Projects/Solutions/Make_Delivery/Ruby_Projects/Delivery_creation_functions.sh – funkciju kopa, ko var izmantot no Linux shell skriptiem, lai izveidotu Ruby tipa projekta piegādi* ***Description:*** Funkcijas, kas veido *Ruby* produkta piegādi, detalizēts apraksts un skriptu piemēri, kas atvieglo risinājuma ieviešanu

	jaunajā projektā.
ENV_UPDATE_NOTIFICATION	***PlatformName:*** *Linux* ***SolutionName:*** *Post_Delivery_Action_Ruby* ***NeededTools:*** *SVN Cmd client* ***LocationsOfSolutions:*** *Q:/Projects/Solutions/Post_Delivery_Actions/Ruby/POST_DELIVERY_FUNCTIONS.sh – funkciju kopa, kas nodrošina visas nepieciešamās darbības, kas jāveic pēc tam, kad pasūtītājs atsūta informāciju, ka jaunā programmatūras versija ir uzlikta QA vidē.* ***Description:*** Citum *Ruby* projektu funkcijas, to apraksti, piemēri, citu projektu konfigurācijas pārvaldnieku kontaktpersonas.

Gatavu *PSAM* modeli, kura detaļas ir redzamas attēlā 4.10. un tabulā 4.4., var izmantot konfigurācijas pārvaldības risinājuma realizācijai projektā «*Test Solution*». Pietiek uzinstalēt nepārtrauktās integrācijas serveri pēc norādījumiem un realizēt visas darbības, kuru realizācija ir aprakstīta atribūtā *LocationsOfSolutions*. Attēlā 4.11. var redzēt «*Test Solution*» projekta realizāciju modeļu un rīku kontekstā.



4.11. att. Projekta «*Test Solution*» modeļi un rīki

Attēlā 4.11. (virzienā no augšas uz leju) var redzēt pakāpenisku konfigurācijas pārvaldības procesa ieviešanu. Sākumā tiek veidots vižu modelis. Pēc tam, balstoties uz vižu modeli, tiek izveidota izejas koda repozitorija struktūra, kurā katrai oriģinālajai videi atbilst savs zars. Paralēli veido *PIAM* un *PSAM* modeļus. Pēc tam instalē visus rīkus un izpilda visus norādījumus, kas atrodas *PSAM* modelī *LocationsOfSolutions* atribūtos. Process ir gatavs palaišanai. 4.11. attēlā apakšējā daļā var redzēt *Jenkins* procesus, kas implementē konfigurācijas pārvaldības procesa soļus. Kā redzams, spraudņi «*dev*», «*test*» un «*qa*» atbilst notikumiem (*Events*), ko var redzēt arī *EM, PIAM* un *PSAM* modeļos. Taču konkrēti būvējumi sākās, piemēram, ar «*dev_1*», kas parāda atbilstību konkrētam notikumam un konkrētai plūsmai. Tādējādi, paskatoties uz procesa ieviešanas gaitu, var redzēt saikni starp abstraktu procesu un rīku ieviešanu.

4.5. Modeļvadāmas pieejas vērtēšanas kritēriji

Vērtējot *EAF* metodoloģijas ieviešanas ieguvumus programmatūras izstrādes un uzturēšanas projektos, ir svarīgi izmērīt šādus rādītājus:

- izmaksas konfigurācijas pārvaldības procesiem pirms un pēc jaunās metodoloģijas ieviešanas. Lai vērtējums vēlāk būtu pēc iespējas objektīvāks, šeit ir svarīgi izmērīt gan laiku, kas regulāri tiek patērēts procesu uzturēšanai, gan procesu sākotnēju ieviešanas laiku pēc veciem paņēmieniem un pēc jaunās *EAF* metodoloģijas. Lai varētu izdarīt secinājumus par ilgtermiņa ieguvumu jaunajai metodoloģijai, ir jāņem vērā arī tas, cik daudz laika ir palicis līdz konkrētā projekta beigām. Zinot šo faktu, var novērtēt laika starpību situācijā, kad procesi strādātu bez izmaiņām, un situācijā, kad tiktu ieviesta jaunā metodoloģija. Ja laiks, kas ir nepieciešams jaunās metodoloģijas, ieviešanai kopā ar laiku, kas būtu nepieciešams procesu regulārai uzturēšanai, ir lielāks par to laiku, kas būtu patērēts procesiem, neieviešot nekādas izmaiņas, tas nozīmē, ka metodoloģijas ieviešana konkrētajā projektā nav izdevīga;
- jaunās metodoloģijas ietekme uz konfigurācijas pārvaldības procesu galvenajiem nodevumiem. Konfigurācijas pārvaldībā svarīgākais nodevums ir programmatūras būvējums, ko saņem pasūtītājs un uzinstalē savās instancēs [AIE 2010], [MET 2002]. Līdz ar to, ieviešot izmaiņas procesā, jāskatās, kā

tas iespaido būvējumu organizāciju. Veicot eksperimentus, šajā promocijas darbā tika analizēts, kā mainījās būvējumu kopējais skaits projektā pirms un pēc *EAF* metodoloģijas ieviešanas, kā arī līdzīgi tika pētīts kļūdainu būvējumu skaits. Ja, piemēram, izmaksu ziņā jaunās metodoloģijas ieviešana ir izdevīga, bet kļūdainu būvējumu skaits krietni pieaug, diez vai tas nesīs lielu labumu projektam kopumā, jo pasūtītājs vairs nebūs apmierināts ar kvalitāti. Tāpēc paralēli izmaksu vērtējumiem jāpievērš nopietna uzmanība arī būvējumu kvalitātei.

Lai varētu iegūt tikko minētus rādītājus, tiks izmantoti šādi avoti:

- SIA «*Tieto Latvia*» korporatīva darba laika uzskaites sistēma. No šā avota tiks iegūti visi nepieciešami dati par projektu izmaksām. Pateicoties tam, ka visi uzņēmuma darbinieki regulāri atskaitās par padarītu, no šīs sistēmas varēs iegūt visu nepieciešamu informāciju par to, cik daudz laika tika patērēts konkrētu projektu konfigurācijas pārvaldībai pirms un pēc *EAF* metodoloģijas ieviešanas. Papildus šajā sistēmā glabājas katra projekta līgumu informācija, līdz ar to katram projektam ir zināms atlikušais laiks. Šis laiks būs nepieciešams ilgtermiņa ieguvuma aprēķināšanai;
- SIA «*Tieto Latvia*» konfigurācijas pārvaldības datubāze. Šajā datubāzē glabājas informācija par visiem būvējumiem visos projektos. Līdz ar to ir iespēja noteikt gan kopēju būvējumu skaitu noteiktā periodā, gan kļūdainos būvējumus katrā projektā.

Tabulā 4.5. ir aprakstīti rādītāji, kas tika izmērīti katram eksperimentējamam projektam.

4.5. tabula

Projektu rādītāji eksperimentu rezultātu iegūšanai

Rādītāja nosaukums, mērvienība	**Apzīmējums**	**Apraksts**
Sākotnējais konfigurācijas pārvaldības ieviešanas laiks Mērvienība: stundas	*IMPL_TIME_OLD*	Šo laiku patērēja konfigurācijas pārvaldības procesu ieviešanai projekta pašā sākumā. Ieviešanu veica pēc veciem paņēmieniem un metodēm.

Jaunais konfigurācijas pārvaldības ieviešanas laiks Mērvienība: stundas	*IMPL_TIME_NEW*	Šo laiku patērēja konfigurācijas pārvaldības procesu ieviešanai pēc jaunās *EAF* metodoloģijas.
Sākotnējais vidējais laiks konfigurācijas pārvaldības uzturēšanai Mērvienība: stundas	*AVG_H_OLD*	Šo laiku vidēji vienas nedēļas laikā tērēja konkrētam projektam konfigurācijas pārvaldības procesu regulārai uzturēšanai pirms *EAF* metodoloģijas ieviešanas.
Jaunais vidējais laiks konfigurācijas pārvaldības uzturēšanai Mērvienība: stundas	*AVG_H_NEW*	Šo laiku vidēji vienas nedēļas laikā tērēja konkrētam projektam konfigurācijas pārvaldības procesu regulārai uzturēšanai pēc jaunās *EAF* metodoloģijas ieviešanas.
Laiks līdz projektam noslēgumam Mērvienība: nedēļas	*REM_TIME*	Parāda, cik daudz nedēļu ir palicis līdz konkrēta projekta līguma beigu datumam.
Sākotnējais prognozējamais laiks konfigurācijas pārvaldības procesu uzturēšanai līdz projekta noslēgumam Mērvienība: stundas	*(REM_TIME X AVG_H_OLD)*	Šo lielumu lieto, lai prognozētu, cik daudz laika būtu nepieciešams, ja konfigurācijas pārvaldība tiktu uzturēta pēc vecām metodēm. Zinot vidējo laiku nedēļā, tas tiek sareizināts ar nedēļu skaitu līdz projekta beigām, un tā rezultātā rodas prognozējamais laiks konfigurācijas pārvaldības uzturēšanai līdz projekta beigām.
Jaunais prognozējamais laiks	*(REM_TIME X*	Šis laiks parāda, cik daudz

konfigurācijas pārvaldības procesu uzturēšanai līdz projekta noslēgumam Mērvienība: stundas	*AVG_H_NEW)* + *IMPL_TIME_NEW*	laika konfigurācijas pārvaldībai būtu nepieciešams, strādājot pēc *EAF* metodoloģijas. Šo laiku izrēķina, ņemot vērā jauno vidējo laiku konfigurācijas pārvaldības uzturēšanai un nedēļu skaitu līdz projekta beigām. Šos lielumus sareizina un pieskaita laiku, kas tika patērēts pārejai uz *EAF* metodoloģiju, jo pati pāreja jeb procesu ieviešana no jauna arī prasa laiku un resursus. Salīdzinot jauno un veco laiku konfigurācijas pārvaldības procesu uzturēšanai, var secināt, vai konkrētam projektam ir izdevīgi pāriet uz *EAF* metodoloģiju.
Sākotnējais vidējais kļūdaino būvējumu skaits Mērvienība: būvējumi	*FAILED_BUILDS_BEFORE*	Vidējais kļūdaino būvējumu skaits mēnesī pirms *EAF* metodoloģijas ieviešanas.
Jaunais vidējais kļūdaino būvējumu skaits Mērvienība: būvējumi	*FAILED_BUILDS_AFTER*	Vidējais kļūdaino būvējumu skaits mēnesī pēc *EAF* metodoloģijas ieviešanas.
Sākotnējais vidējais kopējais būvējumu skaits Mērvienība: būvējumi	*BUILD_COUNT_BEFORE*	Vidējais kopējais būvējumu skaits mēnesī pirms *EAF* metodoloģijas ieviešanas.
Jaunais vidējais kopējais būvējumu skaits	*BUILD_COUNT_AFTER*	Vidējais kopējais būvējumu skaits mēnesī pēc *EAF*

Mērvienība: būvējumi		metodoloģijas ieviešanas.

Tabulā 4.6. ir aprakstīti *EAF* metodoloģijas vērtēšanas kritēriji un to aprēķins, ņemot vērā tabulā 4.5. definētus rādītājus. Vēlāk, veicot eksperimentus, katram projektam tiks izmērīti minētie rādītāji un aprēķināti kritēriji.

4.6. tabula

Modeļvadāmas konfigurācijas pārvaldības vērtēšanas kritēriji

Kritērijs	Mērvienība	Aprēķins, apraksts
Procesu ieviešanas laika starpība	Procenti (%)	**Aprēķins:** *((IMPL_TIME_NEW/IMPL_TIME_OLD)*100)-100* **Apraksts:** Kritērijs, kas procentuāli parāda starpību starp laiku, kas tika patērēts konfigurācijas pārvaldības ieviešanai pēc vecās metodes, un eksperimentu laiku pēc jaunās *EAF* metodoloģijas. Ja vērtība ir pozitīva, tas nozīmē, ka procesu ieviešana pēc jaunās metodoloģijas aizņēma vairāk laika nekā ieviešana pēc veciem paņēmieniem. Ja vērtība ir negatīva, tas nozīmē, ka ieviešana pēc *EAF* metodoloģijas aizņem mazāk laika. **Piemērs:** Kritērija vērtība «–6» tiek interpretēta kā «Ieviešanas laiks ir samazinājies par 6 procentiem», savukārt vērtība «6» tiek interpretēta kā «Ieviešanas laiks ir palielinājies par 6 procentiem».
Regulārā uzturēšanas laika starpība	Procenti (%)	**Aprēķins:** *((AVG_H_NEW/AVG_H_OLD)*100)-100* **Apraksts:** Kritērijs, kas procentuāli parāda starpību starp laiku, kas bija nepieciešams procesu regulārai uzturēšanai pirms un pēc *EAF* metodoloģijas ieviešanas. Ja vērtība ir negatīva, tas nozīmē, ka pēc *EAF* metodoloģijas ir nepieciešams mazāk laika konfigurācijas pārvaldības uzturēšanai. Savukārt pozitīvas vērtības gadījumā nepieciešams patērēt vairāk laika procesu uzturēšanai pēc *EAF* metodoloģijas. **Piemērs:** Kritērija vērtība «–10» nozīmē, ka procesu uzturēšanas laiks ir samazinājies par 10 procentiem, ieviešot *EAF* metodoloģiju. Vērtība «10» nozīmē pretējo: uzturēšanas laiks ir palielinājies par 10 procentiem.
Kopējā	Procenti	**Aprēķins:**

uzturēšanas laika starpība	(%)	*(((REM_TIME*AVG_H_OLD)/((REM_TIME*AVG_H_NEW)+IMPL_TIME_NEW))*100)-100* **Apraksts:** Kritērijs, kas ļauj spriest par *EAF* metodoloģijas ieviešanas ilgtermiņa ieguvumu. Ņem vērā laiku nedēļās līdz projekta noslēgumam, laiku, kas bija nepieciešams *EAF* ieviešanai, un laiku, ko vidēji patērē procesu uzturēšanai pirms un pēc *EAF* ieviešanas. Procentuāli parāda starpību starp kopējo laiku, kas būtu nepieciešams procesu uzturēšanai līdz projekta beigām pēc veciem paņēmieniem, un starp kopējo laiku, kas būtu nepieciešams *EAF* ieviešanai un procesu uzturēšanai. **Piemērs:** Kritērija vērtība «–10» nozīmē, ka ieviešot *EAF* metodoloģiju, projekts patērētu par 10 procentiem mazāk laika nekā tad, ja strādātu pēc vecajām metodēm. Šajā gadījumā konkrētajam projektam pāreja uz jaunu *EAF* metodoloģiju būtu izdevīga. Tieši pretēja situācija būtu ar vērtību «10», kas nozīmētu, ka jaunās metodoloģijas ieviešana prasītu par 10 procentiem vairāk laika. Šajā gadījumā būtu izdevīgi īstenot konfigurācijas pārvaldības procesus pēc vecajām metodēm.
Kļūdaino būvējumu starpība	Procenti (%)	**Aprēķins:** *((FAILED_BUILDS_AFTER/FAILED_BUILDS_BEFORE)*100)-100* **Apraksts:** Kritērijs procentuāli parāda, kā izmainījās kļūdaino būvējumu skaits projektā pēc *EAF* metodoloģijas ieviešanas. **Piemērs:** Vērtība «–17» nozīmē, ka kļūdaino būvējumu skaits ir samazinājies par 17 procentiem, savukārt vērtība «10» nozīmē, ka kļūdaino būvējumu skaits ir pieaudzis par 10 procentiem.
Būvējumu kopējā skaita starpība	Procenti (%)	**Aprēķins:** *((BUILD_COUNT_AFTER/BUILD_COUNT_BEFORE)*100)-100* **Apraksts:** Kritērijs procentuāli parāda, kā izmainījās kopējais būvējumu skaits projektā pēc *EAF* metodoloģijas ieviešanas. **Piemērs:** Vērtība «–17» nozīmē, ka kopējais būvējumu skaits ir samazinājies par 17 procentiem, savukārt vērtība «17» nozīmē,

		ka kopējais būvējumu skaits ir pieaudzis par 17 procentiem.

4.6. Eksperimentu apraksts un rezultātu apkopojums

Saistībā ar jaunās *EAF* metodoloģijas testēšanu tika veikti eksperimenti uzņēmumā SIA «*Tieto Latvia*». Eksperimentu realizācijā piedalījās ne tikai promocijas darba autors, bet arī citi uzņēmuma SIA «*Tieto Latvia*» tehniskie speciālisti. Eksperimentu rezultātā *EAF* metodoloģija tika ieviesta piecos projektos. Lai eksperimentu laikā netiktu apdraudēti esošie projektu procesi, *EAF* ieviešana un testēšana notika uz jauniem konfigurācijas pārvaldības serveriem, kas bija neatkarīgi no tiem, kur darbojās esošie vecie procesi. Lai saglabātu konfidencialitāti, reāli projektu nosaukumi un pasūtītāji, kas ir pazīstami uzņēmumi Latvijas un Eiropas mērogā, netiks minēti. Savukārt visa informācija par izstrādes tehnoloģijām un risinājumiem, kas ietekmē konfigurācijas pārvaldību, tiks uzskaitīta šajā nodaļā.

Eksperimentiem tika izvēlēti aktīvi programmatūras izstrādes projekti. Lai eksperimentu rezultāti būtu pēc iespējas objektīvi, tika ņemti projekti, kuros izmanto dažādas izstrādes tehnoloģijas un dažādus rīkus, kas risina konkrētus konfigurācijas pārvaldības uzdevumus, piemēram, versiju kontroli vai būvējumu pārvaldību. Tabulā 4.7. var redzēt projektu aprakstu.

4.7. tabula

Programmatūras izstrādes projektu apraksts

Projekta numurs	Apraksts
1	**Vispārīgs apraksts:** Sistēma *Oracle E-Business Suite* uzturēšana, esošās funkcionalitātes papildināšana. **Izstrādes tehnoloģija:** *Oracle, Java, PL/SQL* **Versiju kontroles sistēma:** *Suversion* **Pieteikumu apstrādes sistēma:** *JIRA* **Nepārtrauktās integrācijas serveris:** *Jenkins* **Vižu modeļa apraksts:** Ir četras oriģinālās vides: *LENT, LETA, LEGS, LEGO. LENT* – izstrādes vide, *LETA* – testa vide, *LEGS* – klienta akcepttesta vide, *LEGO* – ekspluatācijas vide. Sakarā ar to, ka vienai videi ir nepieciešams ļoti daudz resursu, vižu kopiju, kur var notestēt konfigurācijas pārnešanu, nav. Starp vidēm tiek pārnestas programmatūras

	izmaiņas izvēles kārtībā.
2	**Vispārīgs apraksts:** Sistēmas *Oracle Custumer Care and Billing Utilities* ieviešana un uzturēšana **Izstrādes tehnoloģija:** *Oracle CC&B, PL/SQL, Java, COBOL* **Versiju kontroles sistēma:** *Subersion* **Pieteikumu apstrādes sistēma:** *JIRA* **Nepārtrauktās integrācijas serveris:** *Jenkins* **Vižu modeļa apraksts:** Ir četras oriģinālās vides: *Tieto DEV, Tieto TEST, LE_SIT, LE_PROD*. *Tieto DEV* tiek izmantots izstrādei, *Tieto TEST* testēšanai, *LE_SIT* vidē klients pārtestē jaunas programmatūras izmaiņas, taču *LE_PROD* ir vide, kurā notiek programmatūras ekspluatācija. Uz *Tieto TEST* vidi vienmēr tiek pārnesta visa konfigurācija no *Tieto DEV*, taču nav vides, kur notestēt pārnešanu. Savukārt konfigurācijas pārnešanai uz *LE_SIT* un *LE_PROD* ir attiecīgās kopijas *Tieto_SIT* un *Tieto_PROD*, kur notestē jaunu būvējumu.
3	**Vispārīgs apraksts:** Iekšējas korporatīvas sistēmas uzturēšana un attīstība **Izstrādes tehnoloģija:** *Ruby On Rails* **Versiju kontroles sistēma:** *Git* **Pieteikumu apstrādes sistēma:** *JIRA* **Nepārtrauktās integrācijas serveris:** *Jenkins* **Vižu modeļa apraksts:** Ir četras oriģinālas vides, kurās notiek izstrāde, testēšana, akcepttestēšana un produkta ekspluatācija. Testa, akcepttesta un produkcijas vidēm ir atbilstošās kopijas, kur ir iespējams notestēt gan konfigurācijas pārnešanu, gan labojumu, kas ir specifisks konkrētās vides versijai (*HotFix*).
4	**Vispārīgs apraksts:** Pakalpojumu administrēšanas un uzskaites sistēmas izstrāde **Izstrādes tehnoloģija:** *Oracle ADF, PL/SQL, Java* **Versiju kontroles sistēma:** *Subversion* **Pieteikumu apstrādes sistēma:** *JIRA* **Nepārtrauktās integrācijas serveris:** *Bamboo* **Vižu modeļa apraksts:** Ir trīs oriģinālās vides: izstrādei, testēšanai, akcepttestēšanai. Ierobežotu resursu dēļ nav iespējams uzturēt vižu kopijas

	un testēt konfigurācijas pārnešanu.
5	**Vispārīgs apraksts:** Tīmekļa servisu izstrāde un uzturēšana **Izstrādes tehnoloģija:** *Oracle SOA Suite 11g, PL/SQL, Oracle ADF, Java* **Versiju kontroles sistēma:** *Subversion* **Pieteikumu apstrādes sistēma:** *JIRA* **Nepārtrauktās integrācijas serveris:** *Hudson* **Vižu modeļa apraksts:** Ir četras oriģinālas vides, kurās notiek izstrāde, testēšana, akcepttestēšana un produkta ekspluatācija. Testa, akcepttesta un produkcijas vidēm ir atbilstošās kopijas, kur ir iespējams notestēt gan konfigurācijas pārnešanu, gan labojumu, kas ir specifisks konkrētās vides versijai (*HotFix*).

EAF metodoloģijas ieviešanas gaita aprakstītajos projektos bija šāda:

- metodoloģijas prezentācija projekta konfigurācijas pārvaldniekiem, programmētājiem un projekta vadītājiem, lai visi saprastu ideju, mērķus un metodoloģijas darbības principus;
- vižu modeļu izveidošana katram projektam un to saskaņošana ar projekta komandu. Vižu modeļa veidošanā tika izmantots prototips, kas tika aprakstīts iepriekšējā promocijas darba nodaļā;
- modeļu transformācija *PIAM* modelī, izveidoto *PIAM* modeļu apspriešana ar projekta komandu;
- risinājumu izvēles moduļa papildināšana ar implementācijas risinājumiem konfigurācijas pārvaldības darbībām. Šis solis bija visgrūtākais un prasīja daudz laika un diskusiju, kas turpmāk atspoguļosies arī eksperimentu rezultātos un *EAF* metodoloģijas turpmākajā attīstībā;
- risinājumu izvēle, *PSAM* modeļu veidošana un saskaņošana;
- servisu modeļu iegūšana no *PSAM,* atbilstošo servisu izstrāde un testēšana;
- *PSAM* modeļu realizācija praksē, testēšanas gadījumu sagatavošana, kļūdu labošana;
- izvēlēto rādītāju fiksēšana un definētu kritēriju aprēķini;
- iegūto rezultātu analīze un uzlabošanas iespēju noteikšana.

Tabulā 4.8. var redzēt radītājus, kas tika iegūti pēc tam, kad piecos projektos bija ieviesta *EAF* metodoloģija un kopš ieviešanas brīža bija pagājuši trīs mēneši. Šajos mēnešos

konfigurācijas pārvaldības procesi tika īstenoti pēc *EAF* metodoloģijas. Tabulā 4.9. ir redzami aprēķinātie projektu vērtēšanas kritēriji. Vērtības tika rēķinātas, izmantojot tikai 4.8. tabulā esošos datus.

4.8. tabula

Projektu rādītāji vērtēšanas kritēriju aprēķinam

Rādītāji											
Projekts	**IMPL_TIME_OLD**	**IMPL_TIME_NEW**	**AVG_H_OLD**	**AVG_H_NEW**	**REM_TIME**	**(REM_TIME X AVG_H_OLD)**	**(REM_TIME X AVG_H_NEW) + IMPL_TIME_NEW**	**FAILED_BUILDS_BEFORE**	**FAILED_BUILDS_AFTER**	**BUILD_COUNT_BEFORE**	**BUILD_COUNT_AFTER**
1	160	170	10	9	52	520	638	15	11	84	87
2	300	231	15	14	36	540	735	5	3	160	92
3	76	32	2	1,5	104	208	188	3	2	65	67
4	45	40	8	5	52	416	300	10	4	73	71
5	203	110	3	3	52	156	266	7	5	88	91

4.9. tabula

Projektu vērtēšanas kritēriji

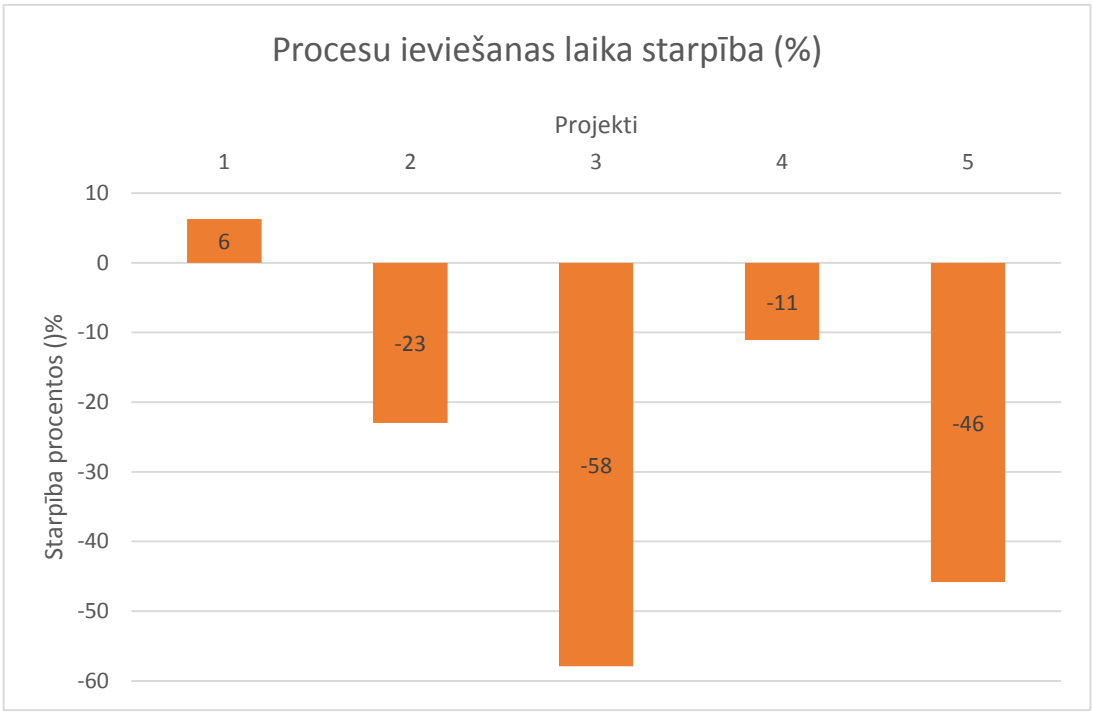
Kritēriji					
Projekts	**Procesu ieviešanas laika starpība (%)**	**Regulārā uzturēšanas laika starpība (%)**	**Kopējā uzturēšanas laika starpība (%)**	**Kļūdaino būvējumu starpība (%)**	**Būvējumu kopējā skaita starpība (%)**
1	6	–10	23	–27	4
2	–23	–7	36	–40	–43
3	–58	–25	–10	–33	3
4	–11	–38	–28	–60	–3
5	–46	0	71	–29	3

Aprēķinot rādītājus, kritērijus, izvērtās plašas diskusijas par eksperimentu rezultātiem un *EAF* metodoloģijas uzlabošanas iespējam. Eksperimentu rezultātu analīze sākumā tika organizēta katram kritērijam atsevišķi. Analīzei bija šādi posmi:

- konkrēta kritērija izvēle;
- rezultātu salīdzinājums starp projektiem. Salīdzinot rezultātus starp projektiem, tika mēģināts atrast cēloni starpībai, kā arī definēt metodoloģijas priekšrocības un trūkumus;
- tika apkopotas *EAF* metodoloģijas priekšrocības un trūkumi un definēti metodoloģijas attīstības virzieni.

a. Procesu ieviešanas laika starpība

Attēlā 4.12. ir redzams grafiks, kura horizontālajā asī norādīts projekta numurs, bet vertikālajā – ieviešanas laika starpība procentos.



4.12. att. Procesu ieviešanas laika starpības salīdzinājums starp projektiem

Sākotnēji *EAF* metodoloģijas ieviešana tika veikta projektam «1». Šajā brīdī risinājumu izvēles modulis vēl bija tukšs. Kamēr katrai *PIAM* darbībai tika izstrādāts un notestēts risinājums, pagāja diezgan daudz laika: 170 stundas. Izrādījās, ka tas ir par 10 stundām vairāk, nekā procesu ieviešanā izmantojot vecās metodes. Rezultāts izraisīja plašas diskusijas par *EAF* mnetodoloģijas efektivitāti, it īpaši par risinājumu izvēles moduļa struktūru. Speciālisti saskārās ar lielām prakstiskajām grūtībām, atbilstoši jaunajām prasībām implementējot tādas *PIAM* darbības kā *COMPILE_BUILD* un *PREPARE_BASELINE*. Šīs nodaļas beigās tiks detalizēti aprakstītas uzlabošanas idejas, kas radās pēc pirmā eksperimenta noslēguma. Kopumā tik lielu ieviešanas laiku (170 stundas) attaisnoja ar to, ka sākotnēji risinājumu izvēles modulis bija tukšs un nevienai darbībai nevarēja paņemt gatavu rezultātu.

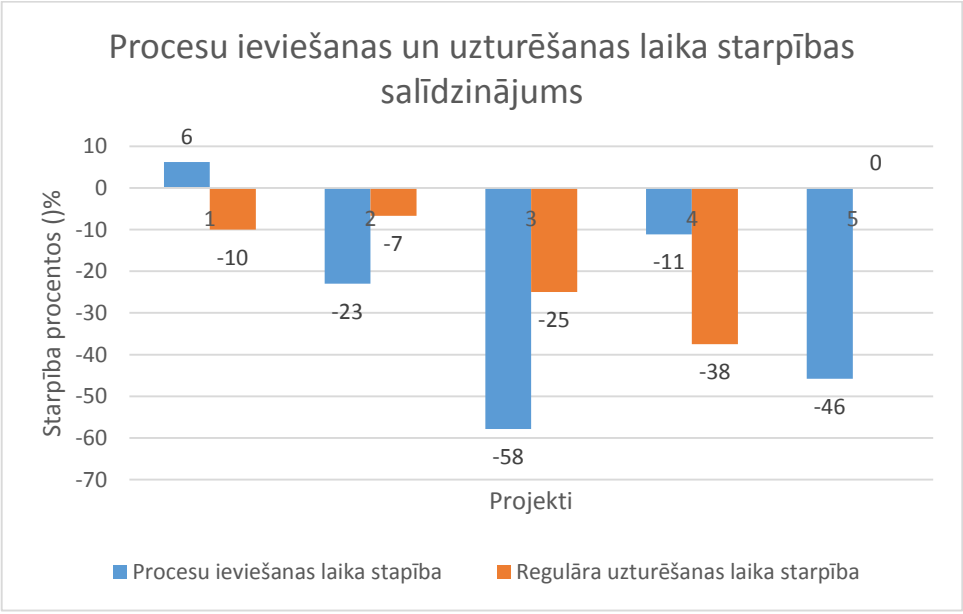
Implementējot *EAF* metodoloģiju projektos «2» un «3», risinājumu izvēles modulī jau bija implementācijas risinājumi vairākām darbībām. Tehnoloģijas bija diezgan līdzīgas, tāpēc otrajā un trešajā projektā rezultāti bija krietni labāki nekā pirmajā eksperimentā. Rezultātā otrajā un trešajā eksperimentā konfigurācijas pārvaldības procesu ieviešanas laiks samazinājās attiecīgi par 23 un 58 procentiem. Pēc minēto trīs eksperimentu pabeigšanas apstiprinājās viens no teorētiskajiem pieņēmumiem *EAF* metodoloģijas izstrādes gaitā: jo vairāk implementāciju ir risinājumu izvēles modulī, jo ātrāk varēs ieviest procesus jaunajā projektā, jo būs iespējams atkārtoti izmantot jau esošos risinājumus.

Projektā «4» atkal tika patērēts ļoti daudz laika metodoloģijas ieviešanā. Tas notika tāpēc, ka ceturtajā projektā atšķīrās tehnoloģijas, kas veica konfigurācijas pārvaldības uzdevumus. Līdz ar to ļoti daudz laika aizņēma risinājumu izvēles moduļa papildināšana. Taču atklājās vēl viena problēma. Darbībām *COMPILE_BUILD* un *INSTALL_BUILD* nācās pielikt klāt ļoti daudz papildu parametru, lai darbībām atbilstošās funkcijas varētu izsaukt vairākām dažādām tehnoloģijām. Šajā brīdī kļuva skaidrs, ka risinājumu izvēles moduļa risinājumu struktūra ir pārāk vispārīga. Praksē ir ļoti grūti uzrakstīt vienotu *Linux Shell* funkciju *compile_build*, kas varētu atkarībā no parametriem izveidot būvējumu gan *Oracle EBS*, gan *Ruby On Rails*, gan *Oracle SOA* tehnoloģijām. Pirmkārt, funkcijai ir jāiedod ļoti daudz dažādu parametru, otrkārt, funkcijas kods ir pārāk apjomīgs, ar vairākiem zarojumiem un to ir grūti uzturēt. Šajā brīdī radās vairākas idejas, kā mainīt risinājuma izvēles moduļa struktūru, lai ieviestu sīkākas vienības atkārtoti lietojamam izejas kodam.

Lai gan projektā «5» ar *EAF* metodoloģiju izdevās samazināt procesu ieviešanas laiku par 46%, idejas par risinājuma izvēles moduļa restrukturizāciju tikai vairojās. Bija skaids, ka atkārtoti lietot vienreiz uzrakstītu kodu konfigurācijas pārvaldības konkrētam posmam, ir laba koncepcija, bet koda vienībām jābūt krietni sīkākām, lai atveiglotu praktisko lietošanu jaunajos projektos.

b. Regulārā uzturēšanas laika starpība

Attēlā 4.13. var redzēt divu kritēriju salīdzinājumu: procesu ieviešanas laika starpība un regulārā uzturēšanas laika starpība.

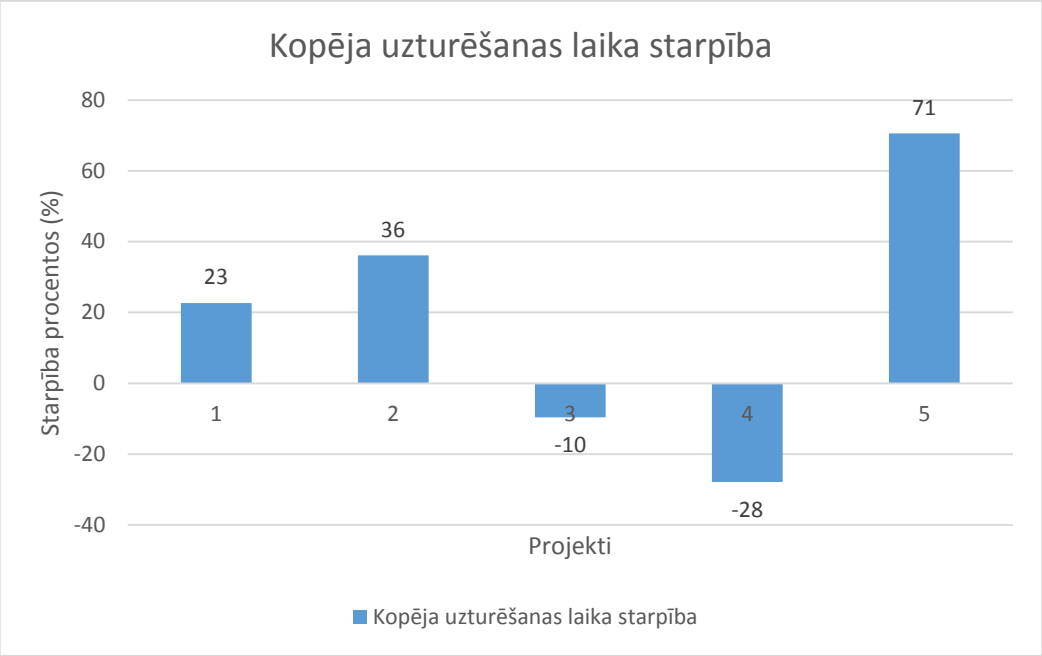


4.13. att. Procesu ieviešanas un uzturēšanas laika starpības salīdzinājums

Kā redzams attēlā 4.13., neskatoties uz to, ka projektā «1» procesu ieviešanai pēc jaunās *EAF* metodoloģijas bija nepieciešams vairāk laika (par 6 %), vēlāk, ikdienā uzturēšanai bija vajadzīgs par 10 % mazāk laika. Līdzīga tendence ir arī projektos «2», «3» un «4», kur ikdienā procesu uzturēšanai pēc *EAF* metodoloģijas ir nepieciešams attiecīgi par 7, 25 un 38 procentiem mazāk laika. Šie rezultāti palīdzēja identificēt *EAF* metodoloģijas vienu būtisku ieguvumu. Neskatoties uz to, ka procesu ieviešana prasa pārāk daudz laika ne visai veiksmīgas risinājumu izvēles moduļa struktūras dēļ, *EAF* palīdz sasniegt lielāku procesu automatizācijas līmeni. Līdz ar to ikdienā ir nepieciešams mazāk laika procesu manuālai uzturēšanai. Automatizācijas līmeni izdevās paaugstināt, pateicoties *EAF* principam, ka konfigurācijas pārvaldības process ir izejas kods, kas tiek palaists no konfigurācijas pārvaldības servera. Līdz ar to pazuda starpdarbības, ko agrāk konfigurācijas pārvaldnieks veica manuāli. Vienīgi projektā «5» neizdevās samazināt procesu ikdienas uzturēšanas laiku. Sīkāk analizējot projekta «5» specifiku, izdevās saprast, ka pirms *EAF* metodoloģijas ieviešanas procesiem jau bija maksimāli augsts automatizācijas līmenis. Galvenais secinājums, ko deva metodoloģijas vērtēšana pēc kritērija «Regulāra uzturēšanas laika starpība», ir tas, ka *EAF* metodoloģija palielina procesu automatizācijas līmeni projektos, kur pirms tam bija manuāli soļi un ne visas darbības tika izpildītas automātiski no konfigurācijas pārvaldības servera. Savukārt, kā parādīja piektā eksperimenta rezultāti, projektos ar augstu automatizācijas līmeni *EAF* metodoloģijas ieviešana to īpaši neietekmē.

c. Kopējā uzturēšanas laika starpība

Vērtējot laiku, kas būtu nepieciešams konfigurācijas pārvaldībai pēc vecās un pēc jaunās metodoloģijas, ņemot vērā atlikušu laiku līdz katra projekta beigām, būtībā tika iegūts prognozētais ilgtermiņa ieguvums no *EAF* metodoloģijas. Rezultāti lika nopietni aizdomāties par vairākiem metodoloģijas uzlabojumiem, jo tikai projektos «3» un «4» metodoloģijas ieviešana būtu izdevīga, ņemot vērā to, cik ilgi vēl pastāvēs konkrētie projekti. Attēlā 4.14. ir redzams kopējā uzturēšanas laika starpības salīdzinājums starp projektiem.



4.14. att. Kopējā uzturēšanas laika starpība

Skatoties uz 4.14. attēlā redzamo grafiku, jāsecina, ka projektā «5» *EAF* metodoloģijas ieviešanai būtu vislielākie zaudējumi. Pētot sīkāk gan 4.14. grafiku, gan arī citus projektu rādītājus, izdevās secināt, ka *EAF* metodoloģijas ieviešana varētu būt izdevīga projektos, kur līdz projekta beigām vēl ir daudz laika un kur pirms tam bija zems automatizācijas līmenis (daudz manuālu darbību). Šajā gadījumā metodoloģija palielina automatizācijas līmeni, un ikdienā procesu uzturēšanai nepieciešams mazāk laika. Rezultātā projektā ilgtermiņā ietaupa procesu uzturēšanas līdzekļus. Savukārt, ja automatizācijas līmenis vēl pirms *EAF* metodoloģijas ieviešanas bija samērā augsts un ikdienas uzturēšanai nevajadzēja tērēt pārāk daudz laika (projekts «5»), tad *EAF* metodoloģijas ieviešana nav izdevīga. Šajā gadījumā, jo mazāk laika paliek līdz projekta beigām, jo mazāk izdevīga ir *EAF* metodoloģijas ieviešana.

Palielināt *EAF* metodoloģijas ilgtermiņa ieguvumu var, samazinot ieviešanas laiku un iespēju robežās palielinot automatizācijas līmeni, jo projekta beigu datumu konfigurācijas pārvaldības procesi nevar tieši ietekmēt. Rezultātā, strādājot pie *EAF* metodoloģijas pilnveidošanas, galvenais akcents būtu jāliek uz ieviešanas laika samazināšanu un risinājumu izvēles moduļa restrukturizāciju.

d. Būvējumu skaita izmaiņu analīze

Kā jau tika minēts iepriekš, vērtējot jebkādas restrukturizācijas konfigurācijas pārvaldības procesos, svarīgi analizēt, kā mainījās kopējais būvējumu skaits un kļūdaino

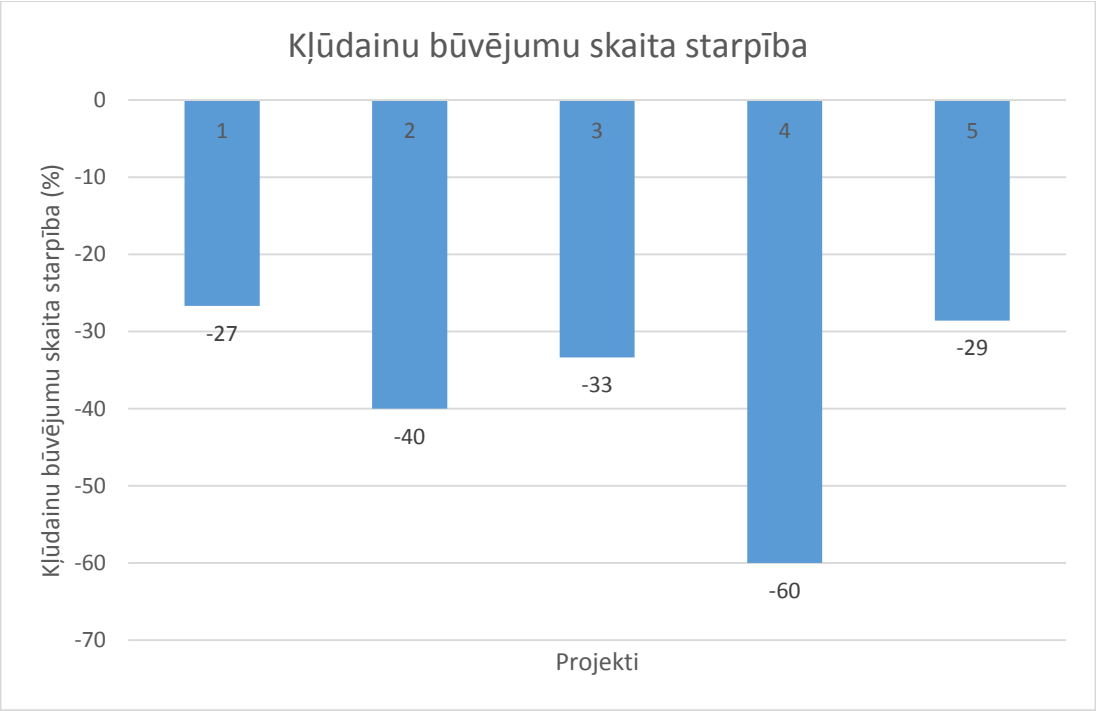
būvējumu skaits projektā. Tas ļauj objektīvāk paskatīties uz ieguvumu rādītājiem, jo ir dažādi faktori, kas to var ietekmēt. Piemēram, ja ieguvumu rezultāti rāda, ka par 25 procentiem ir samazinājies procesu uzturēšanas laiks, bet tajā pašā laikā par tik pat procentiem ir samazinājies kopējais būvējumu skaits, tad, visticamāk, tas nenotika tāpēc, ka jaunā metodoloģija ir tik efektīva, bet tāpēc, ka projektā samazinājās izstrāžu skaits, līdz ar to jaunās programmatūras versijas vairs netika izlaistas tik biezi, kā tas bija līdz šim. Attēlā 4.15. var redzēt, kā mainījās kopējais būvējumu skaits projektos pēc *EAF* metodoloģijas ieviešanas.

4.15. att. Kopējā būvējumu skaita starpība

Kā var redzēt grafikā 4.15., tikai projektā «2» būtiski mainījās kopējais būvējumu skaits (samazinājās par 43 %). Šis fakts lika aizdomāties par to, cik nopietni jāuztver projektā «2» esošo regulārā uzturēšanas laika samazināšanu par 7 procentiem. Ja gandrīz par 45 procentiem kļuva mazāk būvējumu un līdz ar to process palika mazāk noslogots, būtu tikai loģiski sagaidīt arī uzturēšanas laika samazināšanos. Taču tas samazinājās tikai par 7 procentiem. Tas liek apšaubīt *EAF* metodoloģijas ieguvumu konkrētajā projektā.

Projektos «1», «3», «4» un «5» būvējumu kopskaits mainījās ne vairāk par 3-4 procentiem pēc absolūtās vērtības. Līdz ar to var secināt, ka šajos projektos konfigurācijas pārvaldības procesu noslodze īpaši nav mainījusies un rādītāji par metodoloģijas ieguvumiem laika ziņā ir uzticamāki.

Attēlā 4.16. ir redzams, kā mainījās kļūdaino būvējumu skaits pēc *EAF* metodoloģijas ieviešanas visos piecos projektos. Grafiks atklāja vēl vienu *EAF* metodoloģijas būtisku ieguvumu: kļūdaino būvējumu skaita samazināšanās. Ieviešot projektos *EAF* metodoloģiju, vidēji par 38 procentiem samazinājās kļūdaino būvējumu skaits.



4.16. att. Kļūdainu būvējumu skaita starpība

4.7. EAF metodoloģijas priekšrocības un trūkumi, balstoties uz eksperimentu rezultātiem

Veicot minētos eksperimentus un analizējot to rezultātus, tika konstatēti šādi *EAF* metodoloģijas ieguvumi:

- metodoloģija samazina laiku, kas ir nepieciešams konfigurācijas pārvaldības procesu ikdienas uzturēšanai. Pateicoties tam, ka visas konfigurācijas pārvaldības darbības tiek izpildītas no centralizētas vietas, pieaug automatizācijas un pārskatāmības līmenis. Visām darbībām ir atbilstošs izejas kods, kas ļauj izvairīties no manuālām darbībām. Veicot eksperimentus piecos projektos, vidēji par 16 procentiem samazinājās laiks, kas bija nepieciešams procesu ikdienas uzturēšanai;

- metodoloģija krietni samazina kļūdaino būvējumu skaitu. Veidojot izejas kodu katrai konfigurācijas pārvaldības darbībai, praksē tiek pārskatīti daži soļi, pievienotas papildu kvalitātes pārbaudes, kas nebija līdz šim, uzlabota kļūdu apstrāde un žurnālifikācijas sistēma. Tas ļāva samazināt kļūdaino būvējumu skaitu vidēji par 38 procentiem;
- konfigurācijas pārvaldības automatizācijas ieviešana aizņem mazāk laika nekā ieviešana pēc vecajiem paņēmieniem ar nosacījumu, ka risinājumu izvēles modulim ir gatavi un notestēti risinājumi atsevišķu konfigurācijas pārvaldības darbību automatizācijai. Eksperimenti parādīja, ka, ja risinājumu izvēles modulis satur implementācijas konfigurācijas pārvaldības darbībām, procesu ieviešanas laiks pēc *EAF* metodoloģijas ir vidēji par 34 procentiem mazāks.

Definējot metodoloģijas trūkumus, tika ņemti vērā gan tehnisko speciālistu viedokļi, gan arī starptautisku konferenču recenzentu ieteikumi, publicējot darba jaunas izstrādes un praktiskos rezultātus [BAR 2015, BAR 2014a, BAR 2014b, BAR 2014c, BAR 2014d, BAR 2014e, BAR 2014f]. Metodoloģijas trūkumi tika numurēti, lai vēlāk, veicot konkrētus uzlabojumus *EAF* metodoloģijā, varētu identificēt, kurš uzlabojums novērš kuru trūkumu. Tabulā 4.10. var redzēt *EAF* metodoloģijas konstatētu trūkumu apkopojumu.

4.10. tabula

***EAF* metodoloģijas trūkumi**

Trūkums kārtas numurs	Apraksts
1	Risinājumu izvēles moduļa struktūra. Eksperimentu rezultāti atklāja, ka strukturēt konfigurācijas pārvaldības izejas kodu pēc konfigurācijas pārvaldības galvenajiem uzdevumiem (*compile*, *deploy*, *prepare baseline*) ir pārāk plaši. Šajā gadījumā funkcijām ir ļoti daudz parametru, un funkcijas ķermenis satur daudz atzarojumu. Ar laiku šādu kodu kļūst ļoti sarežģīti uzturēt, jo ir jāņem vērā projektu specifika, pat tad, ja programmatūras tehnoloģijas ir līdzīgas. Šo trūkumu atzīmēja gan speciālisti, kas palīdzēja īstenot eksperimentus, gan starptautisku konferenču rakstu recenzenti.
2	Vižu modeļa struktūra. Esošā vižu modeļa interpretācija ļoti

	ierobežo projektus. Pirmkārt, vižu modelī jāparedz iespēja, ka programmatūras pārnešanas starp vidēm notiks vairākos notikumos (*Event*) un konfigurācijas plūsmas (*ConfigurationItemFlow*) arī var būt sadalītas sīkāk atkarībā no projektu specifikas. Otrkārt, kā atzīmēja konferenču rakstu recenzenti un tehniskie speciālisti, jēdzieni Notikums (*Event*) un Konfigurācijas plūsma (*ConfigurationItemFlow*) nav intuitīvi saprotami. Līdz ar to būtu nepieciešams atrast veidu, kā vienkāršāk strukturēt konfigurācijas pārvaldības darbības, kas pārnes programmatūras izmaiņas starp vidēm. Papildus tam vižu modelēšanas gaitā konfigurācijas pārvaldniekam būtu jāsniedz iespēja brīvāk strukturēt darbības pa notikumiem un plūsmām. Visbeidzot, jāpārskata notikumu un plūsmu jēdzieni, lai konfigurācijas pārvaldniekiem tie būtu intuitīvi saprotamāki.
3	*PIAM* modeļa būtība. Darbību kopa, kas ir aprakstīta *PIAM* meta-modelī, nav pilnīga. Ir jābūt iespējai pielikt klāt jaunas darbības. Papildus tam transformācija no *EM* uz *PIAM* modeli ļoti ierobežo projektus, kuriem ir jāveic vēl citas darbības, kas nav definētas transformācijas likumos. Iesniedzot modeļu aprakstus zinātniskajai konferencei *MODELSWARD 2015*, tika saņemts ieteikums apvienot *EM* un *PIAM* modeļus, ļaujot lietotājam pašam izvēlēties darbības, kā arī paplašināt darbību kopu meta-modelī.
4	Izejas koda zarošanas modelis neatspoguļo dažādas izejas koda pārvaldības stratēģijas. Ir projekti, kam jau ir citas stratēģijas, un šajā gadījumā *EAF* metodoloģijas ieviešanu apgrūtina fakts, ka metodoloģija paredz noteiktu zarošanas pieeju. Līdz ar to radās ieteikums zarošanas pieeju pasniegt vien kā rekomendāciju, taču atstāt projektiem zināmu brīvību zaru nosaukumu un zarošanas pieejas izvēlē.
5	Servisu modelis neparedz darbību ar vairākām instancēm un tehnoloģijām. Pieņemsim, ir situācija, kad konkrēta programmatūras laidiena aprakstam ir nepieciešama informācija no vairākām dažādām pieteikumu apstrādes sistēmām. Šajā

	gadījumā servisu modelim jābūt pietiekami elastīgam, lai ļautu pieslēgties dažādām sistēmām tā, lai funkcijās nevajadzētu ņemt vērā projektu specifiku. Līdzīga problēma ir arī ar konfigurācijas pārvaldības darbību realizāciju, kad, piemēram, programmatūrai var būt dažādas komponentes un katra komponente ir izstrādāta savā programmēšanas valodā, un līdz ar to būvējumu un instalācijas specifika ir pilnīgi atšķirīga.
6	Prototipa izstrādē netika apsvērtas iespējas formalizēt modeļus un to transformācijas ar kādu no modelēšanas rīkiem, piemēram, *Metaedit+* vai *Eclipse Modelling Framework*. Tas palielina riskus, ka modeļu veidošanā tiks pieļautas kļūdas, kas netiks atklātas modeļu implementācijas laikā.

4.8. *EAF* metodoloģijas iteratīvas izstrādes un atkārtotu eksperimentu nepieciešamības pamatojums

Eksperimentu rezultāti parādīja, ka *EAF* metodoloģija ļauj samazināt konfigurācijas pārvaldības automatizācijas ieviešanas laiku. Ieviešot konfigurācijas pārvaldības automatizāciju piecos projektos, vidēji par 34 % samazinās ieviešanas laiks, salīdzinot ar automatizācijas ieviešanu pēc vecajām metodēm. Šī tendence, no vienas puses, ļautu uzskatīt, ka promocijas darba mērķis ir sasniegts. Taču gan eksperimentu gaitā, gan arī publicējot *EAF* metodoloģijas pamatus starptautisku konferenču rakstu krājumos, tika konstatēti būtiski priekšnosacījumi otrajai izstrādes kārtai:

- risinājumu izvēles modulis. Kad konfigurācijas pārvaldības automatizācija tika ieviesta pēdējā no pieciem projektiem, tika konstatēts, ka atkārtoti izpildāms izejas kods kļuva grūti uzturams. Apspriežot rezultātus ar vadošajiem kompetences grupas speciālistiem, kas piedalījās eksperimentu organizācijā, tika konstatēts fakts, ka pie esošās realizācijas *EAF* metodoloģija nespēj pilnībā sniegt atkartoti lietojamu izejas kodu automatizācijas procesiem. Kļuva skaidrs, ka izejas koda vienībām jābūt mazākām un tām vajadzētu glabāties citādāk. Tika novērota tendence, ka katrā jaunā eksperimentā atkārtoti lietojamais izejas kods pieauga gan

sarežģītības, gan apjoma ziņā. Iezīmējās risks, ka vēl pēc dažiem projektiem risinājumu izvēles modulis kļūst nelietojams;

- recenzenta atsauksmes rakstam [BAR 2015]. Jāatzīmē, ka šis raksts tiks pieteikts konferencei *MODELSWARD 2015*, kas bija veltīta gan *MDA*, gan *MDD* jaunākajiem sasniegumiem. Lai gan raksts tika akceptēts, viens no recenzentiem atzīmēja būtiskus trūkumus *PIAM* modelī, kas būtībā ierobežo programmatūras konfigurācijas pārvaldniekus veidot jaunas darbības, kā arī mainīt to secību. Recenzents ieteica apvienot vižu modeli un no platformas neatkarīgu darbību modeli vienā, lai lietotājs varētu brīvi modelēt ne tikai vides, bet arī darbības. Šis ieteikums rūpīgi tika izvērtēts pēc veiktajiem eksperimentiem un tika pieņemts lēmums ieviest izmaiņas *EAF* metodoloģijā un modeļos.

Sakarā ar to, ka risinājumu izstrādes modulis, *EM* un *PIAM* modeļi ir pamata elementi *EAF* metodoloģijai un šajos elementos bija nepieciešamas izmaiņas, tika pieņemts lēmums organizēt *EAF* metodoloģijas otru izstrādes kārtu. Galvenais mērķis bija uzlabot risinājuma izvēles moduļa struktūru un apvienot *EM* un *PIAM* modeļus.

Ņemot vērā, ka eksperimentu noslēguma brīdī *EAF* metodoloģijas pamati un eksperimentu rezultāti bija publicēti zinātniskajos rakstos un metodoloģijā bija nepieciešams mainīt pamatelementus, tika pieņemts lēmums veikt arī atkārtotus eksperimentus. Otrā eksperimentu kārta ir vajadzīga, jo tiks mainīti vairāki pamatelementi *EAF* metodoloģijai, un tā rezultātā tā kļūs citādāka. Līdz ar to vajadzēs pārliecināties ne tikai par to, vai ir novērsti pirmajā versijā konstatēti trūkumi, bet arī par to, ka nav nograuti kādi no ieguvumiem.

4.9. Nodaļas kopsavilkums

Nodaļā tika aprakstīta jaunās *EAF* metodoloģijas testēšanas gaita. Izstrādājot metodoloģiju, teorētiski pamati tika publicēti starptautisku zinātnisku konferenču rakstu krājumos. Paralēli tika izveidota kompetences grupa uzņēmumā SIA «*Tieto Latvia*» praktisku eksperimentu plānošanai un veikšanai. Papildus tam RTU mācību kursā «Adaptīvas datu apstrādes sistēmas» tika izstrādāts programmatūras prototips *EAF* modeļu ģenerēšanai. Nodaļas sākumā detalizēti tika ilustrēts piemērs, kas parāda *EAF* modeļu lietojumu un ilustrē tikko minēta prototipa darbību. Tika izstrādāti vērtēšanas kritēriji jaunizstrādātajai metodoloģijai. Piecos dažādos projektos tika ieviesta konfigurācijas pārvaldība pēc *EAF*

metodoloģijas. Balstoties uz izstrādātiem kritērijiem, tika fiksēti nepieciešamie rādītāji. Analizējot praktisku eksperimentu rezultātus un zinātnisko rakstu recenzentu ieteikumus, izdevās identificēt *EAF* metodoloģijas ieguvumus un trūkumus.

Katram programmatūras izstrādes projektam, kas piedalījās eksperimentos, tika fiksēti šādi rādītāji:

- konfigurācijas pārvaldības procesu ieviešanas laiks. Šis laiks tika patērēts procesu ieviešanai pēc vecajām metodēm un paņēmieniem, kad *EAF* metodoloģijas vēl nebija. Ieviešanas laika aprēķinam tika izmantoti dati no uzņēmuma SIA «*Tieto Latvia*» darba laika uzskaites sistēmas;
- vidējais laiks nedēļā, kas tika patērēts konfigurācijas pārvaldības procesu regulārai uzturēšanai pirms *EAF* metodoloģijas ieviešanas. Arī šā rādītāja iegūšanai tika izmantota SIA «*Tieto Latvia*» darba laika uzskaites sistēma;
- programmatūras būvējumu kopējais skaits un kļūdaino būvējumu skaits.

Veicot eksperimentus pēc plāna, kas ir aprakstīts nodaļas sākumā, *EAF* metodoloģija tika ieviesta piecos programmatūras izstrādes projektos. Pēc tam, kad pagāja trīs mēneši kopš metodoloģijas ieviešanas, visos projektos atkārtoti tika izmērīti tie paši rādītāji un aprēķināti vērtēšanas kritēriji.

Salīdzinot laiku, kas tika patērēts konfigurācijas pārvaldībai pirms un pēc *EAF* metodoloģijas ieviešanas, tika konstatēti šādi ieguvumi:

- vidēji par 34 % samazinājās konfigurācijas pārvaldības procesu ieviešanas laiks;
- vidēji par 16 % samazinājās laiks konfigurācijas pārvaldības procesu regulārai uzturēšanai;
- vidēji par 38 % samazinājās programmatūras kļūdaino būvējumu skaits.

Promocijas darba nākamajā nodaļā tiks aprakstītas izstrādes, kas papildina un uzlabo šajā nodaļā aprakstītu *EAF* metodoloģiju konfigurācijas pārvaldības ieviešanai. Papildus tam tiks aprakstīti eksperimenti, kas tika veikti jau pēc metodoloģijas uzlabojumiem. Rezultāti tiks salīdzināti ar radītājiem, kas tika iegūti sākotnējos eksperimentos. Visbeidzot, tiks sniegtas praktiskās rekomendācijas modeļvadāmas konfigurācijas pārvaldības ieviešanai, balstoties uz veikto eksperimentu rezultātiem.

5. *EAF* METODOLOĢIJAS UZLABOŠANA UN IEVIEŠANAS REKOMENDĀCIJAS

5.1. *EAF* metodoloģijas uzlabotā versija

Ņemot vērā 4.8. tabulā aprakstītos *EAF* metodoloģijas trūkumus, tika izstrādāta jauna, uzlabota *EAF* metodoloģijas versija. Tajā ir saglabāti pirmās versijas galvenie principi, bet dažas lietas ir pilnībā pārstrādātas. Šajā nodaļā ir definēti tikai jaunie vai pārstrādātie jēdzieni. Elementi, kas, salīdzinot ar iepriekšējo metodoloģijas versiju, nemainījās, šajā nodaļā detalizēti netiks aprakstīti.

a. *EAF* jaunās versijas galvenie principi:

- *EAF* galvenais mērķis ir implementēt konfigurācijas pārvaldības procesu konkrētajā programmatūras izstrādes projektā;
- konfigurācijas pārvaldības procesa implementācijas galvenais nodevums ir konkrētā operētājsistēmā izpildāms izejas kods, kas pilnībā pārvalda konfigurācijas pārvaldības procesu, risinot visus nepieciešamos procesa uzdevumus;
- izejas kods konfigurācijas pārvaldības procesam tiek ģenerēts automātiski, lai mazinātu kļūdu risku cilvēciskā faktora dēļ;
- konfigurācijas pārvaldības izejas kodā nav iešūtas nekādas absolūtās vērtības (angļu val. *hardcodes*). Visas absolūtās vērtības glabājas mainīgajos un konstantēs atbilstoši konkrētai programmēšanas tehnoloģijai;
- *EAF* jaunajā realizācijā papildus tiek ieviesti šādi jēdzieni:
    1. **mainīgais (*Variable*) –** atribūts, kas ir nepieciešams konkrētas darbības (*Action*) izpildei. Piemēram, lai pieslēgtos pie versiju kontroles sistēmas, ir nepieciešama tās adrese, lietotājvārds un parole. Tātad, lai realizētu pieslēgšanās darbību, jābūt definētiem vismaz trim atbilstošajiem mainīgajiem;
    2. **algoritms (*Algorithm*)** – definē aktivitātes, kas realizē kādu konkrētu darbību (*Action*). Vienai darbībai varētu būt vairāki dažādi algoritmi. Konfigurācijas pārvaldības izejas koda ģenerēšanas procesā konfigurācijas pārvaldnieks var izvēlēties vienu no algoritmiem, ar kura palīdzību var realizēt konkrētu darbību (*Action*);

3. **ietvars (*Framework*)** – radniecīgu darbību kopa, kas risina kādu no konfigurācijas pārvaldības uzdevumiem vai kādu daļu no tiem. Piemēram, izejas koda pārvaldībai varētu tikt izmantots ietvars, kas balstās uz kādu no versiju kontroles rīkiem, piemēram, *Subversion*. Šajā gadījumā ietvaru veidotu visas darbības, kas strādā ar *Subversion* versiju kontroles sistēmu, veicot programmatūras izejas koda pārvaldību;
4. **funkcija (*Function*)** – ietvara vienība. Atbilstošās programmēšanas tehnoloģijas programmatūras vienība (funkcija, procedūra), kas saņem noteiktus parametrus un veic vienu konkrētu darbību. Ietvars (*Framework*) ir sava veida bibliotēka, bet funkcijas ir šīs bibliotēkas sastāvdaļas, kas nosaka visu, ko ir spējīgs izdarīt konkrēts ietvars (*Framework*).

Konfigurācijas pārvaldības glabātuvei ir šādi jaunie principi:

- ir uzskaitītas visas platformas, kurās implementē konfigurācijas pārvaldības automatizācijas procesa izejas kodu;
- katrai platformai ir uzskaitīti visi konfigurācijas pārvaldības serveri (*SCMServer*). Obligāti jābūt aprakstam, kas nosaka, kā implementēt konkrētu serveri, ņemot vērā platformas specifiku. Piemēram, viena un tā paša *Jenkins* servera instalācija var atšķirties *Linux* un *Windows* platformām;
- katrai platformai ir uzskaitīti visi ietvari (*Framework*), kurus ir iespējams implementēt;
- katrs ietvars (*Framework*) glabā vismaz šādu informāciju:
    - ietvara nosaukumu;
    - ietvara aprakstu;
    - informāciju par to, kādas papildu programmas ir nepieciešams uzinstalēt un kādas darbības ir nepieciešams veikt konfigurācijas pārvaldības serverī, lai ietvars varētu veiksmīgi strādāt;
    - darbības (*Action*), ko ir iespējams realizēt, izmantojot konkrēto ietvaru;
    - funkcijas (*Function*), ko ir iespējams lietot;
- katra darbība (*Action*) glabā vismaz šādu informāciju:
    - darbības nosaukums;
    - darbības apraksts;
    - nepieciešamie mainīgie (*Variable*) konkrētas darbības realizācijai;

        - iespējamie realizācijas algoritmi (*Algorithm*). Katrai darbībai var būt viens vai vairāki algoritmi, kas nosaka darbības implementāciju;
    - struktūra ļauj brīvi atlasīt visus konfigurācijas pārvaldības serverus un to, kādās platformās tos var uzinstalēt un lietot;
    - struktūra ļauj brīvi atlasīt visas iespējamas darbības (*Action*), kuras ir iespējams realizēt. Atlasot darbības, jābūt iespējai iegūt šādu informāciju:
        - kādās platformās ir iespējams realizēt darbību;
        - kādus ietvarus (*Framework*) ir nepieciešams lietot, lai implementētu darbību;
        - izvēloties platformu un ietvaru konkrētai darbībai, jābūt pieejamai informācijai par algoritmiem un mainīgajiem, kas ir nepieciešami konkrēta algoritma realizācijai;
- *EAF* metodoloģijā tiek realizēts princips «Vides -> Darbības -> Ietvars». Šis princips tiek realizēts šādos posmos.
    1. Lietotājs modelē visas projekta vides un to, kā programmatūras izmaiņas tiks pārnestas starp dažādām vidēm. Projekta vižu modelēšanai tiek izmantota speciāla modelēšanas valoda. Modelēšanas valodai jāsniedz informācija par vidēm, programmatūras izmaiņu plūsmām starp tām un darbībām (*Action*), ko ir nepieciešams veikt, lai realizētu visas tikko minētas plūsmas. Šī posma galvenais nodevums ir vižu modelis, kas ietver informāciju par programmatūras izmaiņu plūsmām un visām darbībām, kas ir nepieciešamas plūsmas realizācijai.
    2. Lietotājs strādā ar darbību sarakstu, kas tika definēts vižu modelī, un konfigurācijas pārvaldības risinājumu glabātuves pārvaldības sistēmu, veicot šādas darbības:
        - izvēlas platformu un atkarībā no izvēlētas platformas veic nākamās darbības;
        - izvēlas vienu no konfigurācijas pārvaldības serveriem;
        - katrai darbībai, kas ir definēta vižu modelī, izvēlas ietvaru, algoritmu un definē atbilstošos mainīgos, kurus atkarībā no izvēles piedāvā definēt glabātuves pārvaldības sistēma.

    Šī posma galvenais nodevums ir struktūra, kurā ir informācija no vižu modeļa, pieliekot klāt visu, ko lietotājs ir definējis ar konfigurācijas pārvaldības

risinājumu pārvaldības sistēmas palīdzību. Šo struktūru sauc par platformas specifisko darbību modeli. Šāds nosaukums ir izvēlēts, jo modeļa galvenais mērķis ir noteikt visu darbību (*Action*) implementāciju atkarībā no platformas. Platformas specifiskā darbību modeļa ģenerācijai tiek izmantota speciāla modelēšanas valoda, kas definē modeļa elementus un to hierarhiju, un algoritms, kas, ņemot vērā tikko minētas modelēšanas valodas likumus un lietotāja definētu informāciju, sastāda gatavu platformas specifisku darbību modeli konkrētam projektam.

3. No gatava platformas specifiskā darbību modeļa automātiski tiek ģenerēts izejas kods, kas ir izpildāms uz attiecīga konfigurācijas pārvaldības servera. Lietotājs atbilstoši uzinstalē un nokonfigurē konfigurācijas pārvaldības serveri, notestē iegūtu izejas kodu un palaiž konfigurācijas pārvaldības procesu.

b. Modeļvadāma konfigurācijas pārvaldības metodoloģijas «*Environment – Action – Framework, EAF*» implementācija

Definīcijas

1. ***PIEM meta-model*** – modelēšanas valoda, kas ir domāta *EAF* metodoloģijas principa «Vides -> Darbības -> Ietvars» pirmā posma realizācijai. Ar šo modelēšanas valodu ir iespējams veidot vižu modeļus.
2. ***PIEM* (*Platform Independent Environment Model*)** – konkrēta projekta vižu modelis, kas ir izveidots no atbilstoša *PIEM meta-model* elementiem.
3. ***PSAM meta-model*** – modelēšanas valoda, kas ir domāta *EAF* metodoloģijas principa «Vides -> Darbības -> Ietvars» otrā posma realizācijai. Ar šo modelēšanas valodu ir iespējams veidot platformas specifiskos darbību modeļus.
4. ***PSAM*** – platformas specifiskais darbību modelis, kas tiek iegūts no *PSAM meta-model* elementiem.
5. ***Code Model*** – koda modelis. Izejas koda failu struktūra, kas ir ģenerēta no *PSAM* modeļa. *EAF* metodoloģijas principa «Vides -> Darbības -> Ietvars» trešā posma galvenais nodevums.
6. ***Solution Database*** – risinājumu pārvaldības datubāze jeb *EAF* elementa *SCMWarehouse* realizācija, kur glabājas visa konkrētā uzņēmuma informācija par konfigurācijas pārvaldības risinājumiem.
7. ***Solution Management System*** – risinājumu pārvaldības aplikācija. Strādā ar risinājumu pārvaldības datubāzi un piedāvā lietotājam izvēlēties risinājumus darbībām

no *PIEM* modeļa. Sistēma tiek izmantota *EAF* metodoloģijas principa «Vides -> Darbības -> Ietvars» otrā posma realizācijā.

8. ***Transformation Algorithm «PIEM -> PSAM»*** – transformācijas algoritms, kas, pateicoties lietotāja ievadītajiem datiem, *PIEM* modeli pārveido *PSAM* modelī.
9. ***TransformationAlgorithm «PSAM -> Code Model»*** – transformācijas algoritms, kas pārveido gatavu *PSAM* modeli koda modelī (*Code Model*), un rezultātā tiek iegūta izejas koda struktūra un instrukcija konfigurācijas pārvaldības servera instalācijai.
10. ***Manager*** – konfigurācijas pārvaldnieks, kas veido modeļus un implementē konfigurācijas pārvaldības procesu.

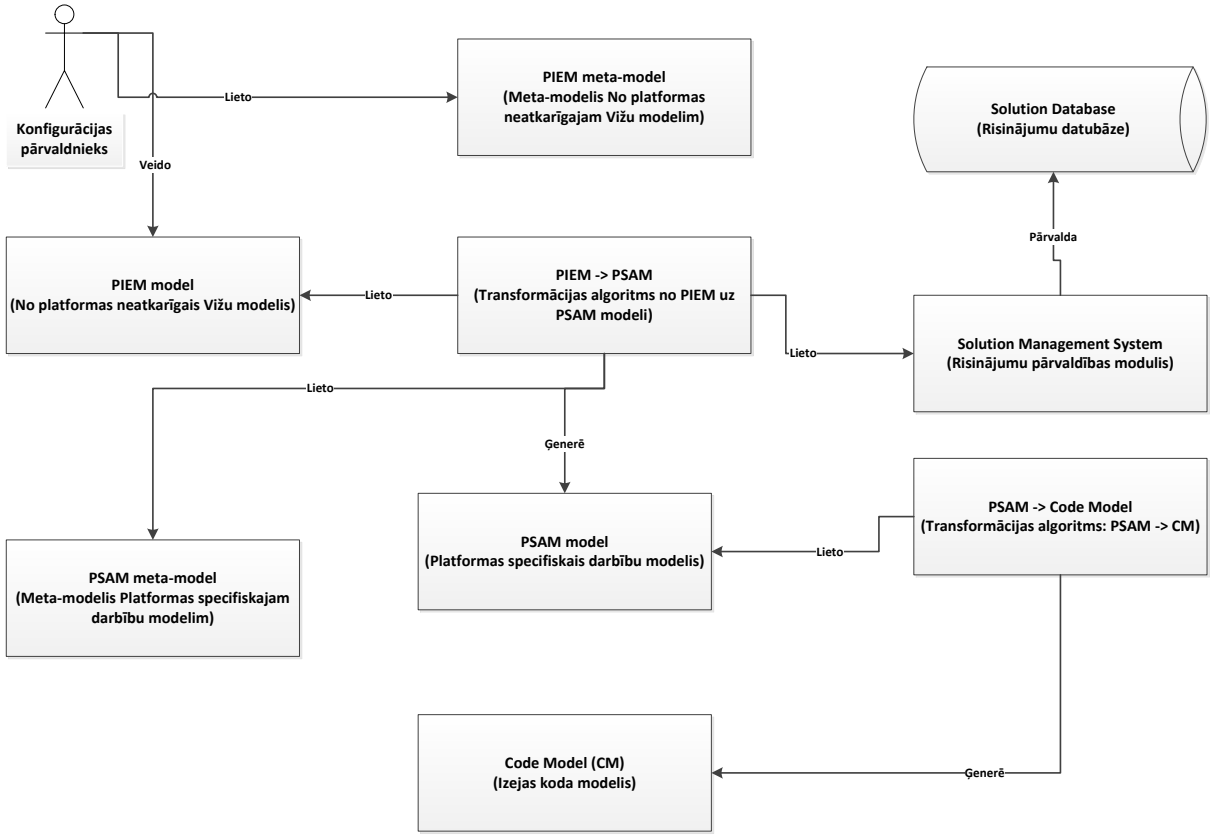
c. *EAF* metodoloģijas jaunās versijas ieviešanas posmi

EAF balstās uz konfigurācijas pārvaldības procesa plānošanu un ieviešanu ar modeļu palīdzību. Ieviešot metodoloģiju kādā projektā, jāveic šādi soļi:

1. ***PIEM* modeļa veidošana.** Šajā posmā lietotājs, izmantojot modelēšanas valodu «*PIEM meta-model*», veido *PIEM* modeli konkrētam projektam. *PIEM* modelis parāda visas projekta vides (*Environment*) un konfigurācijas pārvaldības darbības (*Action*), kuras ir nepieciešamas, pārnesot programmatūras izmaiņas starp vidēm un arī veicot citus konfigurācijas pārvaldības uzdevumus;
2. **transformācijas algoritms «*PIEM -> PSAM*» strādā ar gatavu *PIEM* modeli.** Katrai darbībai (*Action*), kuru lietotājs ir definējis *PIEM* modelī, tiek izvēlēts ietvars no *Solution Database*, ar kura palīdzību darbība tiks implementēta. Ietvara izvēles procesā tiek izmantota *Solution Management System* – aplikācija, kas pārvalda visus pieejamus konfigurācijas pārvaldības risinājumus;
3. ***PSAM* modelis tiek transformēts koda modelī (*Code Model*).** Transformācijas algoritms, balstoties uz gatavu PSAM modeli, uzģenerē izejas koda failu kopu, kuru var izmantot *PIEM* darbību (*Action*) implementācijai.

Kā jau tika minēts promocijas darba trešajā nodaļā, *EAF* metodoloģija ir viena no iespējamām *MTM* (*Model – Transformation – Model*) pieejas realizācijām. Lai varētu formalizēt darbu ar *MTM* pieeju, tika izstrādāta modelēšanas valoda, kas ļauj veidot modeļus, transformācijas un elementus, attēlojot *MTM* pieejas vispārīgu realizāciju. Modelēšanas valoda tika izstrādāta ar *MetaEdit+* rīka palīdzību. Salīdzinot mūsdienīgus modelēšanas valodas veidošanas rīkus [VAS 2013], tika izvēlēts tieši *MetaEdit+*, rīkā var salīdzinoši viegli

mainīt modelēšanas valodu un veidot grafiskus elementus atbilstoši konkrētai situācijai. Vēl viens faktors, kas ietekmēja tieši šā rīka izvēli, ir laba dokumentācijas un lietošanas piemēru pieejamība, kas ļauj ātri apgūt rīku pat iesācējam. Attēlā 5.1. ir redzama *EAF* metodoloģijas jaunās versijas vispārīgā realizācija, kas ir izveidota ar jaunās *MTM* modelēšanas valodas palīdzību.



5.1. att. *EAF* metodoloģijas jaunās versijas vispārīgā realizācija

d. *PIEM* meta-modelis un modeļa piemērs

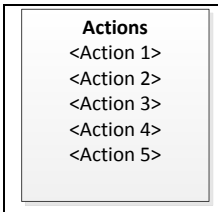
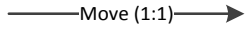
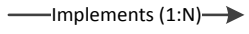
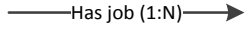
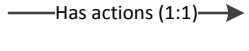
PIEM modeļa mērķis ir parādīt visas projekta vides, starp kurām tiks pārnestas programmatūras izmaiņas un darbības, kas nepieciešamas izmaiņu pārnešanai. Šajā realizācijā tika likvidēti nevajadzīgie atribūti no iepriekšējas *EM* modeļa versijas, darbības no *PIAM* modeļa tika ienestas jaunajā vižu modelī, turklāt darbību kopa vairs nav fiksēta – lietotājs var brīvi pievienot darbības konkrētu programmatūras izmaiņu pārnešanai starp vidēm. Papildus minētajam notikumu un konfigurācijas plūsmu jēdzieni tika aizvietoti ar jēdzieniem «Procesu kopa (*JobSet*)» un «Process (*Job*)». Šie jēdzieni ir intuitīvi labāk saprotami konfigurācijas pārvaldniekiem, kas līdz šim jau ir strādājuši ar nepārtrauktās integrācijas serveriem, tādiem

kā *Bamboo, Jenkins, TeamCity* utt. Tabulā 5.1. ir redzami *PIEM* meta-modeļa elementi ar skaidrojumu.

5.1. tabula

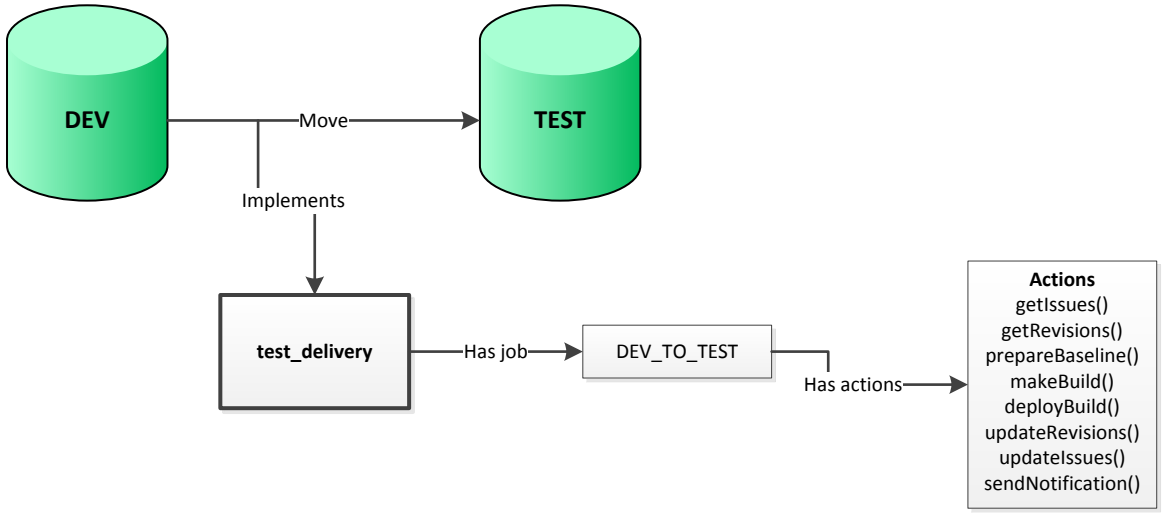
***PIEM* meta-modeļa elementi**

Elements, nosaukums	Apraksts
 **Vide (*Environment*)** Atribūti: • *Name* – vides nosaukums; • *Description* – vides apraksts projekta kontekstā.	Infrastruktūras kopa, kurā atrodas izstrādājama programmatūra (aplikāciju serveri, datubāzes, ārēju sistēmu interfeisi utt.). Katra vide ir paredzēta konkrētai aktivitātei programmatūras izstrādes dzīves ciklā, piemēram, izstrādei, testēšanai, kvalitātes akcepttestēšanai, ekspluatācijai utt. Vides nosaukums parasti ietver informāciju par nozīmi un lietošanu projektā. Piemēram, *DEV* vide – paredzēta izstrādei, *TEST* – testēšanai utt.
 **Procesu kopa (*JobSet*)** Atribūti: • *Name* – procesu kopas nosaukums; • *Description* – procesu kopas apraksts.	Procesu kopa (*JobSet*) modelē procesus, kas tiks darbināti konfigurācijas pārvaldības serverī, lai pārnestu programmatūras izmaiņas no vienas vides uz citu. Vienā procesu kopā var būt viens vai vairāki procesi. To *PIEM* veidošanas laikā nosaka lietotājs. Viena procesu kopa var nodrošināt programmatūras izmaiņu pārnešanu tikai starp divām vidēm.
 **Process (*Job*)** Atribūti: • *Name* – procesa nosaukums; • *Description* – procesa apraksts.	Process, kas veic noteiktas darbības, lai pārnestu programmatūras izmaiņas no vienas vides uz otru. Process obligāti pieder vienai konkrētai procesu kopai.

 **Darbības (*Actions*)**	Darbības modelē konfigurācijas pārvaldības uzdevumus, kas ir jāatrisina, pārnesot programmatūras izmaiņas starp vidēm. Darbības ir abstraktas, un tajās nav nekādas informācijas par implementāciju. Pie darbībām, kā jau minēts, šeit pieskaita konfigurācijas pārvaldības galvenos uzdevumus, piemēram, versiju kontrole, izejas koda pārvaldība, produktu būvējumi, produktu instalācijas, konfigurācijas vienumu statusu uzskaite. Atšķirībā no *EAF* sākotnējas versijas darbību kopa nav fiksēta, un lietotājs varētu tās izvēlēties atkarībā no situācijas.
 **Saikne:** *Mov*. **Saiknes tips:** *1:1*	Programmatūras izmaiņu plūsma (*Flow*) starp divām vidēm. Saiknes tips ir *1:1*, kas nozīmē, ka vienas saiknes ietvaros var pārnest programmatūras izmaiņas tikai uz vienu vidi.
 **Saikne:** *Implements* **Saiknes tips:** *1:N*	Saikne parāda procesu kopas (*JobSet*) piederību konkrētai programmatūras izmaiņu plūsmai (*Flow*). Vienai plūsmai var būt vairākas procesu kopas (*JobSet*), taču katrs konkrēts *JobSet* pieder tikai vienai plūsmai (*Flow*).
 **Saikne:** *Has job* **Saiknes tips:** *1:N*	Saikne parāda procesa (*Job*) piederību konkrētai procesu kopai (*JobSet*). Saikne *1:N* nozīmē, ka vienai procesu kopai var būt vairāki procesi, taču process var piederēt tikai vienai konkrētai kopai (*JobSet*).
 **Saikne:** *Has job* **Saiknes tips:** *1:1*	Saikne parāda elementa *Actions* piederību konkrētam procesam. Saikne *1:1* parāda, ka vienam procesam var būt tikai viena darbību

	secība (*Actions*), savukārt darbību secībai obligāti jāpieder kādam konkrētam procesam.

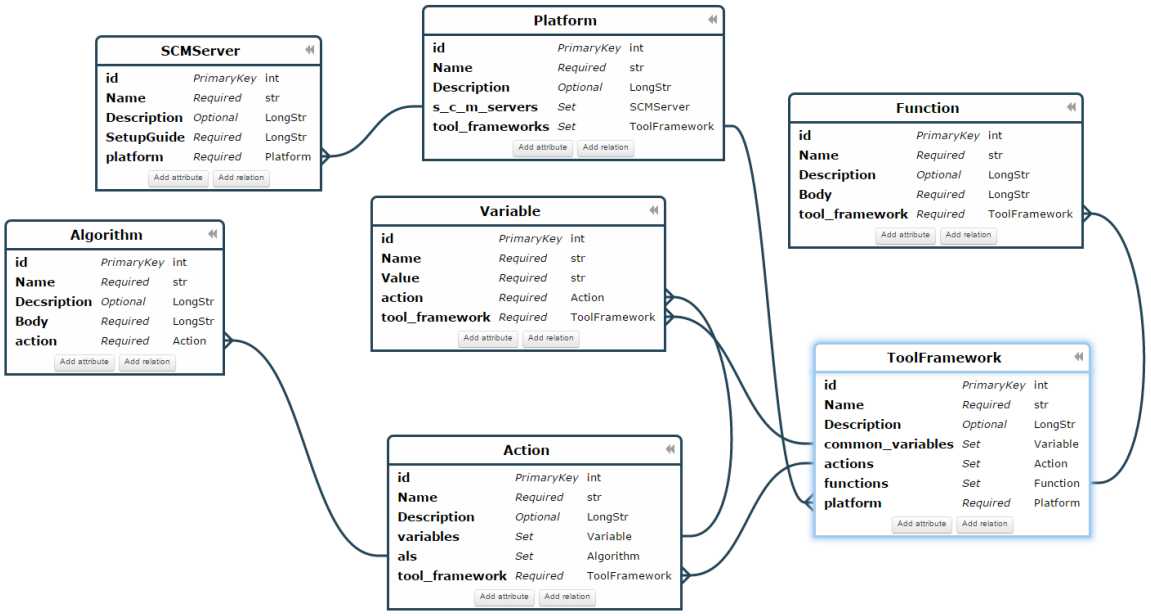
Attēlā 5.2. var redzēt *PIEM* modeļa piemēru.



5.2. att. *PIEM* modeļa piemērs

e. *Solution Database* struktūra un apraksts

Attēlā 5.3. var redzēt *ER* diagrammu *Solution Database* struktūrai. Struktūra ir pilnībā pārveidota, salīdzinot ar *EAF* metodoloģijas sākotnēju versiju, jo, kā parādīja eksperimenti, vecajā struktūrā bija ļoti grūti veidot konfigurācijas pārvaldības darbību risinājumus. Tabulā 5.2. ir dots skaidrojums katrai datubāzes tabulai, kas ietilpst elementā *Solution Database*.



5.3. att. *Solution Database* struktūra

5.2. tabula

Solution Database **tabulas un to apraksts**

Tabulas nosaukums	**Tabulas apraksts**
Platform	Glabā informāciju par platformām jeb operētājsistēmām, kurās ir iespējams realizēt konfigurācijas pārvaldību. Atribūti: ***Name*** – platformas nosaukums ***Description*** – platformas apraksts, papildu informācija
SCMServer	Glabā informāciju par konfigurācijas pārvaldības serveru implementāciju. Atribūti: ***Name*** – nosaukums ***Description*** – apraksts, papildu informācija ***SetupGuide*** – norādījumi par instalāciju, konfigurāciju, papildu programmatūru utt. ***platform*** – atsauce uz tabulu *Platform*, kas norāda platformu, kurā iespējams šo serveri implementēt

ToolFramework	Glabā informāciju par visiem ietvariem (*Framework*), kas ir pieejami uzņēmumā. Atribūti: ***Name*** – nosaukums ***Description*** – apraksts, papildu informācija ***Platform*** – atsauce uz tabulu *Platform*, kas norāda platformu, kurā iespējams šo ietvaru implementēt
Function	Glabā informāciju par visām funkcijām. Atribūti: ***Name*** - nosaukums ***Description*** – apraksts, papildu informācija ***Body*** – funkcijas izejas kods ***tool_framework*** – funkcijas piederība pie konkrēta ietvara
Action	Glabā informāciju par visām darbībām. Atribūti: ***Name*** – nosaukums ***Description*** – apraksts, papildu informācija ***tool_framework*** – darbības piederība pie konkrēta ietvara
Algorithm	Glabā informāciju par visiem algoritmiem. Atribūti: ***Name*** – nosaukums ***Description*** – apraksts, papildu informācija ***action*** – algoritma piederība pie konkrētas darbības
Variable	Glabā informāciju par visiem mainīgajiem. Atribūti: ***Name*** – nosaukums ***Value*** – mainīgā vērtība ***Action*** – mainīgā piederība pie konkrētas darbības

	tool_framework – mainīgā piederība pie konkrēta ietvara

f. *PSAM* meta-modelis un modeļa piemērs

PSAM modelis ir datu struktūra, kas parāda konfigurācijas pārvaldības darbību realizāciju atkarībā no platformas. Datu struktūrai ir šādas daļas:

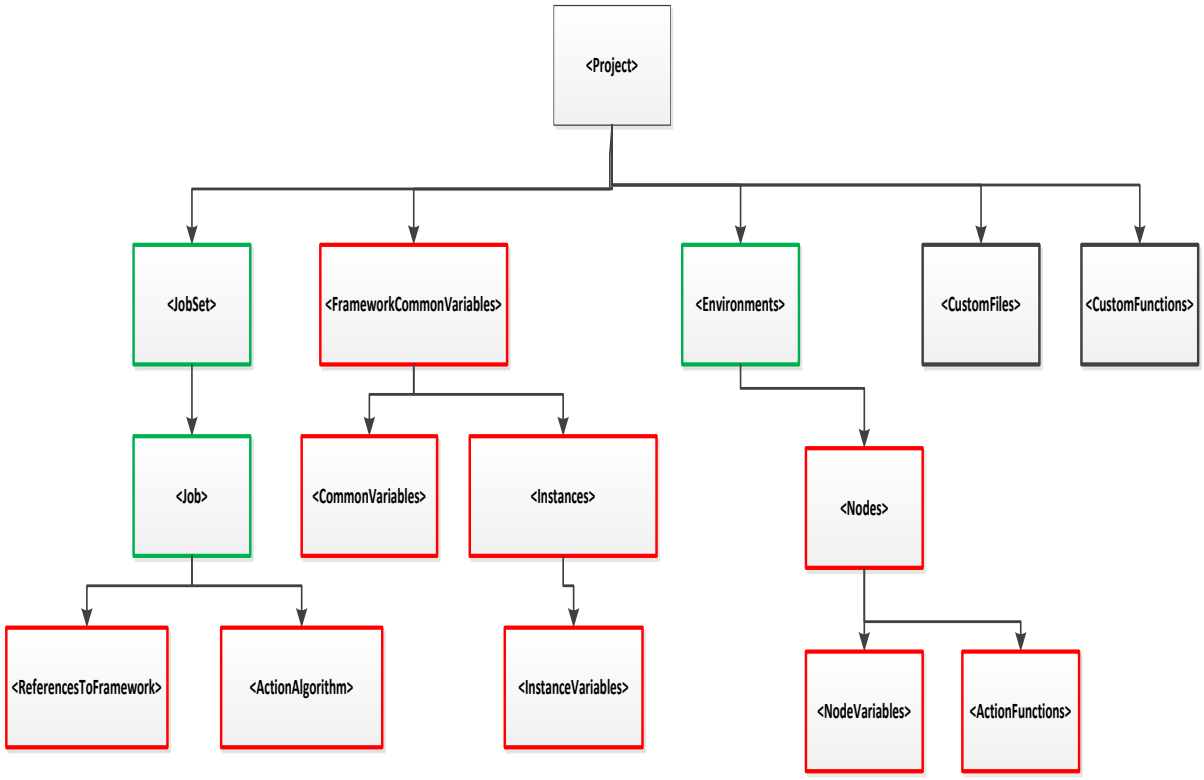
- informācija no *PIEM* modeļa. Šī daļa satur informāciju par vidēm, procesu kopām, procesiem un darbībām. Informācija pilnībā tiek nokopēta no *PIEM* modeļa;
- informācija par platformu, ietvariem, darbību realizācijas algoritmiem un mainīgajiem. Šī daļa aizpildās brīdī, kad lietotājs strādā ar konfigurācijas pārvaldības risinājumu pārvaldības aplikāciju, lai izvēlētos risinājumu katrai darbībai no *PIEM* modeļa;
- projekta specifiskās funkcijas un papildu faili. Šī struktūras daļa *PSAM* modelī paliek tukša un tiek aizpildīta, ieviešot konfigurācijas pārvaldību izejas koda līmenī.

PSAM modeļa struktūra sastāv no *PIEM* modeļa elementiem un no *Solution Database* elementiem, ko izvēlas lietotājs, strādājot ar risinājumu pārvaldības aplikāciju. Papildus tam *PSAM* meta-modelī tika ieviesti divi jauni elementi:

- **mezgls (*Node*)** – programmatūras vienība, kas ir realizēta kādā konkrētā programmēšanas tehnoloģijā un kurai ir vienots būvējumu un instalācijas veidošanas princips. Šo elementu vajadzēja ieviest, lai saglabātu ietvaru funkciju atkārtotas izmantošanas principu un nepieļautu liekas konfigurācijas pārvaldības izejas koda modifikācijas sarežģītos projektos. Piemēram, programmatūras izstrādes projektam var būt vairākas programmatūras vienības, kas ir izstrādātas, izmantojot dažādas programmēšanas valodas un tehnoloģijas. Lai varētu realizēt būvējuma un instalācijas darbības, katrai programmatūras vienībai ievieš savu mezglu un tā ietvaros definē mainīgos, būvējumu un instalācijas funkciju atkarībā programmatūras vienības specifikas. Ja projekts sastāv no programmatūras, kas ir veidota *JAVA* valodā, un citas komponentes, kas ir veidota uz *Oracle* bāzes, būvējumu un instalācijas darbības abām programmatūrām atšķirsies. Mezgla ieviešana katrai programmatūrai ļauj atsevišķi definēt būvējumu un instalācijas algoritmus un attiecīgos mainīgos. Savukārt, piemēram, darbības *build* algoritms ciklā pārskatīs visus mezglus un izsauks definētu funkciju *build()*. Lai gan funkcijas *build()* ķermenis katram mezglam atšķirsies, darbības *build* algoritmu un procesa kopējo kodu nevajadzēs modificēt;

- **instance (*Instance*)** – mainīgo kopa, kas atbilst vienam konkrētam ārējam rīkam, kuram vajadzēs pieslēgties, lai realizētu kādu konfigurācijas pārvaldības darbību (*Action*). Veicot konfigurācijas pārvaldības aktivitātes, dažreiz ir nepieciešams slēgties pie ārējām programmatūrām, lai iegūtu informāciju. Kā piemēru var minēt problēmu un izmaiņu pieprasījumu pārvaldību pieteikumu apstrādes sistēmu. Brīdī, kad testa vidē ir uzinstalēta jauna programmatūras versija, pieteikumu apstrādes sistēmā atbilstošām problēmām un izmaiņu pieprasījumiem nepieciešams papildināt atribūtus. Taču var gadīties tā, ka projektā tiek izmantotas dažādas problēmu pārvaldības sistēmas. Tad katrai tādai sistēmai ievieš savu instanci, kurā definē specifiskus mainīgos un atribūtu papildināšanas funkcijas. Savukārt darbības algoritms ciklā pārskatīs viass instances un tikai izsauc katras instances specifiskās funkcijas.

Attēlā 5.4. var redzēt *PSAM* meta-modeļa elementu hierarhiju. Ar zaļo krāsu ir apzīmēti elementi, kas tiek ņemti no *PIEM* modeļa, ar sarkanu krāsu apzīmēti elementi, ko lietotājs izvēlas no risinājumu pārvaldības datubāzes (*Solution Database*). Tabulā 5.3. ir aprakstīti *PSAM* meta-modeļa elementi.



5.4. att. *PSAM* meta-modeļa elementu hierarhija

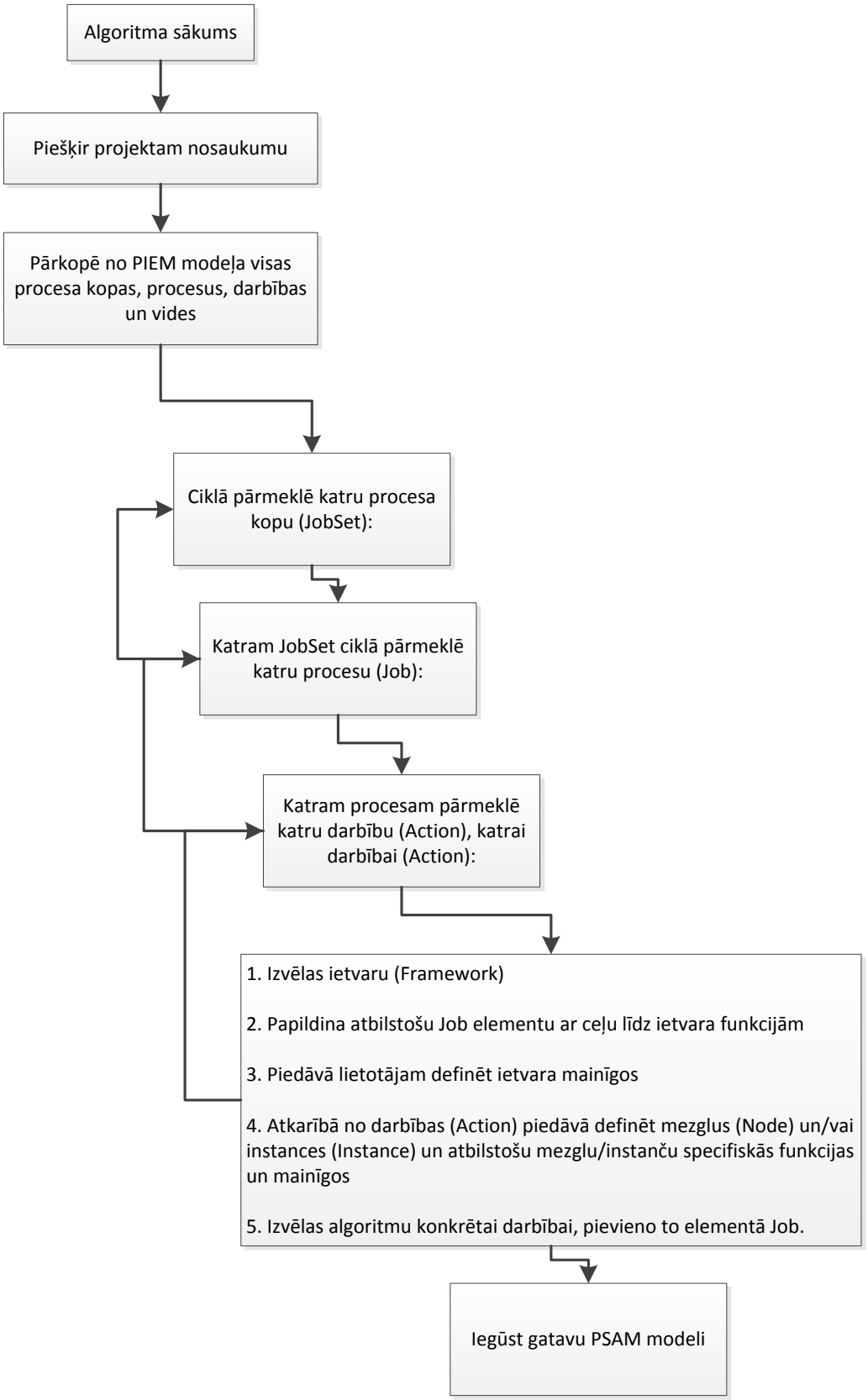
5.3. tabula

***PSAM* meta-modeļu elementu apraksts**

Elements	**Apraksts**
Project	Definē projektu, kurā tiek implementēts konfigurācijas pārvaldības process.
JobSet	Procesu kopa, kas tiek nokopēta no *PIEM* modeļa.
Job	Process. Nokopēts no *PIEM*.
ReferencesToFramework	Ceļš līdz ietvara funkcijām, lai no konkrēta procesa (*Job*) varētu izsaukt ietvara (*Framework*) funkcijas.
ActionAlgorithm	Konkrētas darbības algoritms. Izejas koda gabals, kas veic noteiktu darbību visām atbilstošām instancēm un/vai mezgliem.
FrameworkCommonVariables	Struktūra, kas glabā dažādus mainīgos, kas ir nepieciešami konkrēta ietvara normālai darbībai. Piemēram, ja ietvars slēdzās pie ārējas programmatūras, šajā struktūrā jābūt vides mainīgajiem, kas nodrošina atbilstošas pieslēgšanas iespējas.
CommonVariables	Mainīgie konkrēta ietvara darbību nodrošināšanai.
Instances	Mainīgo kopa, kas raksturo vienu konkrētu programmatūras vienību, ko izmanto atbilstošais ietvars. Piemēram, ja projekts slēdzās pie trijām dažādām problēmu pārvaldības sistēmām un iegūst pieteikumu sarakstu, tad tiks definētas trīs instances un katrai instancei būs atbilstoši mainīgie un funkcija, kas iegūst pieteikumus. Savukārt darbības *getIssues()* algoritms secīgi pārskatīs visas minētās instances un izsauks

	pieteikumu iegūšanas funkciju.
InstanceVariables	Mainīgie vienas konkrētas instances definēšanai.
Environments	Projekta vides. Visas vides tiek nokopētas no *PIEM* modeļa.
Nodes	Programmatūras vienība, kas ir realizēta kādā konkrētā programmēšanas tehnoloģijā un kurai ir vienots būvējumu un instalācijas veidošanas princips.
NodeVariables	Mainīgie viena konkrēta mezgla definēšanai.
ActionFunctions	Mezgla funkcijas, kas izpilda kādu no konfigurācijas pārvaldības darbībām (*Action*), piemēram, *build*, *install*.
CustomFiles	Struktūra, kas glabā papildu failus, kas būs nepieciešami konfigurācijas pārvaldības koda izpildei.
CustomFunctions	Struktūra, kam ir projekta specifiskas funkcijas. *PSAM* modelī šī struktūra ir tukša un tiek aizpildīta tikai konfigurācijas pārvaldības koda modeļa ieviešanas laikā.

PSAM modeļa struktūra aizpildās pēc *PIEM->PSAM* transformācijas algoritma, kas ir redzams attēlā 5.5. Tabulā 5.4. ir dots transformācijas algoritma soļu apraksts.



5.5. att. *PIEM ->PSAM* transformācijas algoritms

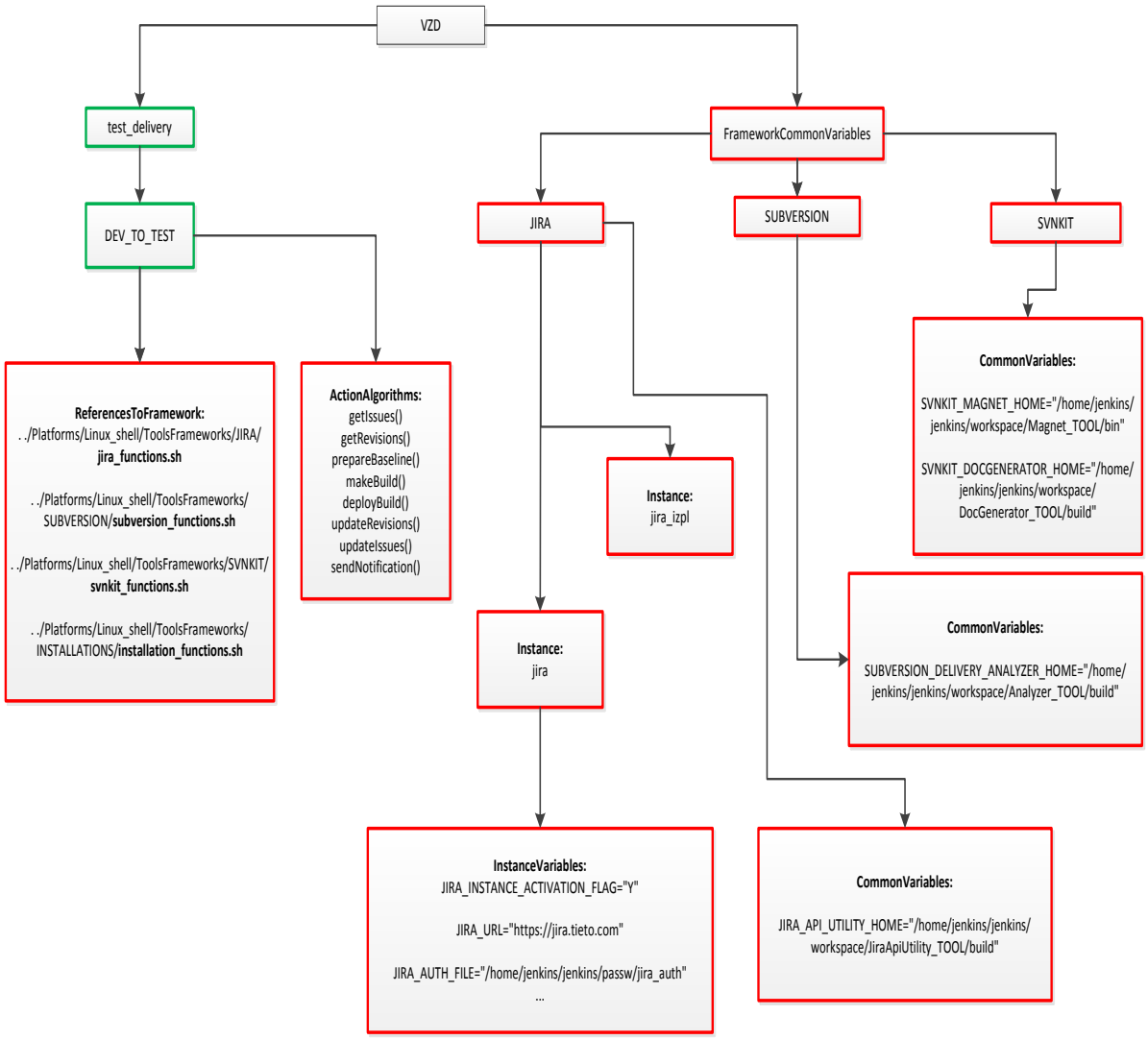
5.4. tabula

***PIEM*->*PSAM* transformācijas algoritma soļu apraksts**

Algoritma solis	**Soļa apraksts**
Piešķir projektam nosaukumu	Piedāvā lietotājam izvēlēties projekta nosaukumu. Šajā brīdī aizpildās elements *Project*.
Pārkopē no *PIEM* modeļa visas procesa kopas (*JobSet*), procesus (*Job*), darbības (*Actions*) un vides (*Environments*)	Transformācijas algoritms izveido sākotnēju struktūru un pārkopē minētus elementus no *PIEM* modeļa. Šajā brīdī aizpildās šādi elementi *PSAM* struktūrā: • *JobSet*; • *Job*; • *Environments*.
Ciklā pārmeklē katru procesa kopu (*JobSet*)	Strādā ar struktūru, kas tika pārkopēta no *PIEM* modeļa iepriekšēja algoritma solī. Katrai procesa kopai izsauc nākamo soli.
Katram *JobSet* ciklā pārmeklē katru procesu (*Job*)	Strādā ar procesu kopu, kas tika iegūta iepriekšējā solī, katram procesam (*Job*) izsauc nākamo soli.
Katram procesam pārmeklē katru darbību (*Action*)	Strādā ar procesu *Job*, kas tiek iegūts iepriekšējā solī. Katrai darbībai (*Action*) izpilda nākamos piecus soļus.
1. Izvēlas ietvaru (*Framework*)	Lietotājs strādā ar risinājumu pārvaldības aplikāciju un veic šādas darbības: • izvēlas platformu; • izvēlas konfigurācijas pārvaldības serveri, kuru iespējams implementēt izvēlētajā platformā. Šajā brīdī lietotājs iegūst konfigurācijas pārvaldības servera implementācijas

	instrukciju; • atlasa visus ietvarus (*Framework*) izvēlētai platformai, ar kuru palīdzību ir iespējams realizēt konkrētu darbību, izvēlas vienu no ietvariem.
2. Papildina atbilstošu *Job* elementu ar ceļu līdz ietvara funkcijām	Atkarībā no ietvara (*Framework*), kas tiek izvēlēts iepriekšējā solī, atbilstošajā procesā (*Job*) tiek ielikts ceļš līdz ietvara funkcijām.
3. Piedāvā lietotājam definēt ietvara mainīgos	Atkarībā no izvēlēta ietvara (*Framework*), lietotājs saņem norādījumus par to, kādas papildu programmas ir jāuzinstalē. Pēc tam lietotājs definē atbilstošos vides mainīgos.
4. Atkarībā no darbības (*Action*) piedāvā definēt mezglus (*Node*) un/vai instances (*Instance*) un atbilstošu mezglu/instanču specifiskās funkcijas un mainīgos	Lietotājs definē darbībai nepieciešamos mezglus (*Node*) vai instances (*Instance*). Atbilstoši darbībai (*Action*) tiek definēti mainīgie un specifiskās funkcijas.
5. Izvēlas algoritmu konkrētai darbībai, pievieno to elementā *Job*	Lietotājs izvēlas vienu no pieejamiem algoritmiem (*Algorithm*), lai izpildītu atbilstošo darbību (*Action*) visiem mezgliem vai visām instancēm.
Iegūst gatavu *PSAM* modeli	Lietotājs saņem gatavu struktūru, kurā katrai darbībai (*Action*) ir visas nepieciešamas detaļas par implementāciju. Šajā brīdī *PSAM* modelis ir gatavs transformācijai koda modelī (*Code Model*), kā arī lietotājam ir konfigurācijas pārvaldības servera implementācijas instrukcija.

Attēlā 5.6. var redzēt *PSAM* modeļa fragmentu.

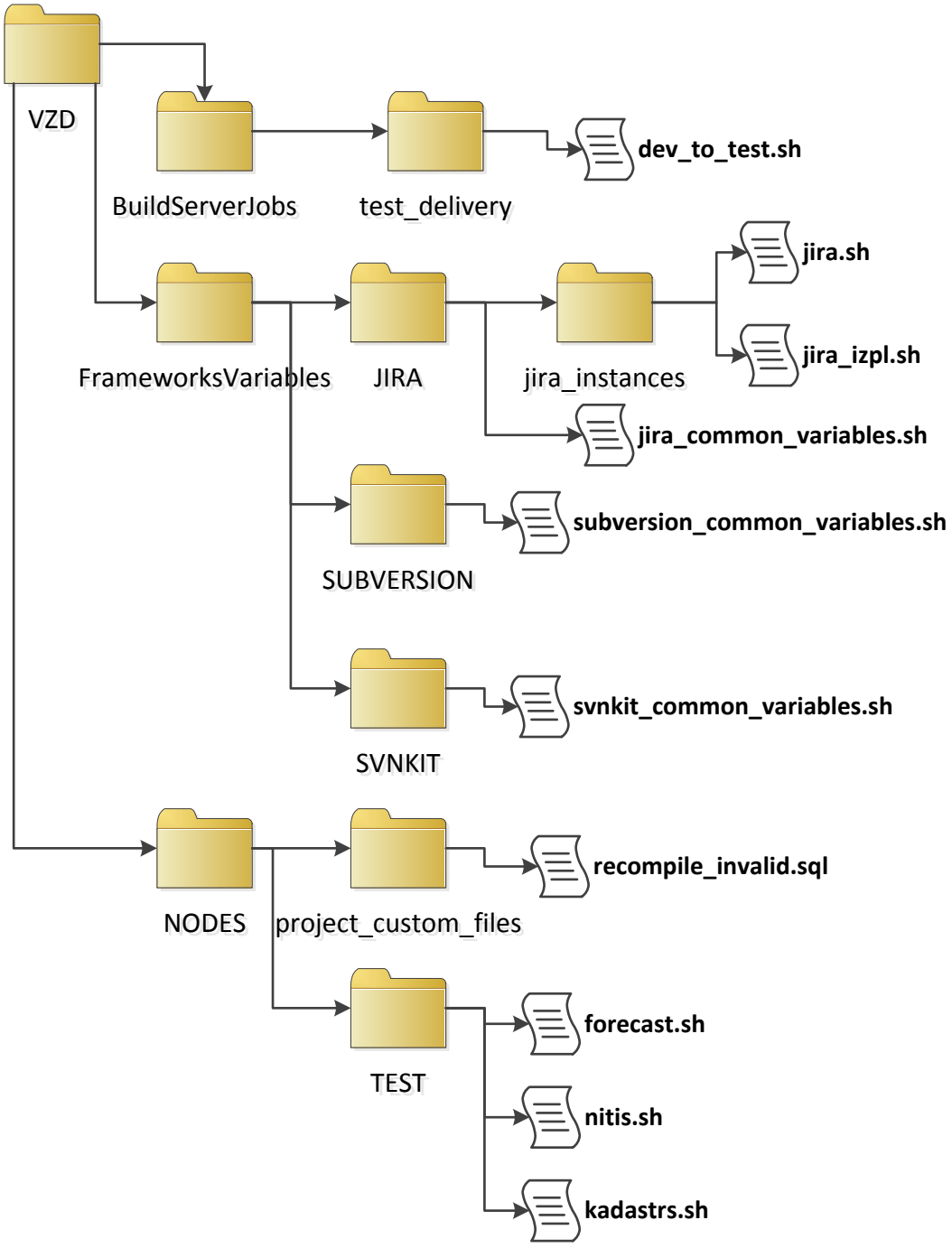


5.6. att. *PSAM* modelis

g. Koda modelis (*Code Model*)

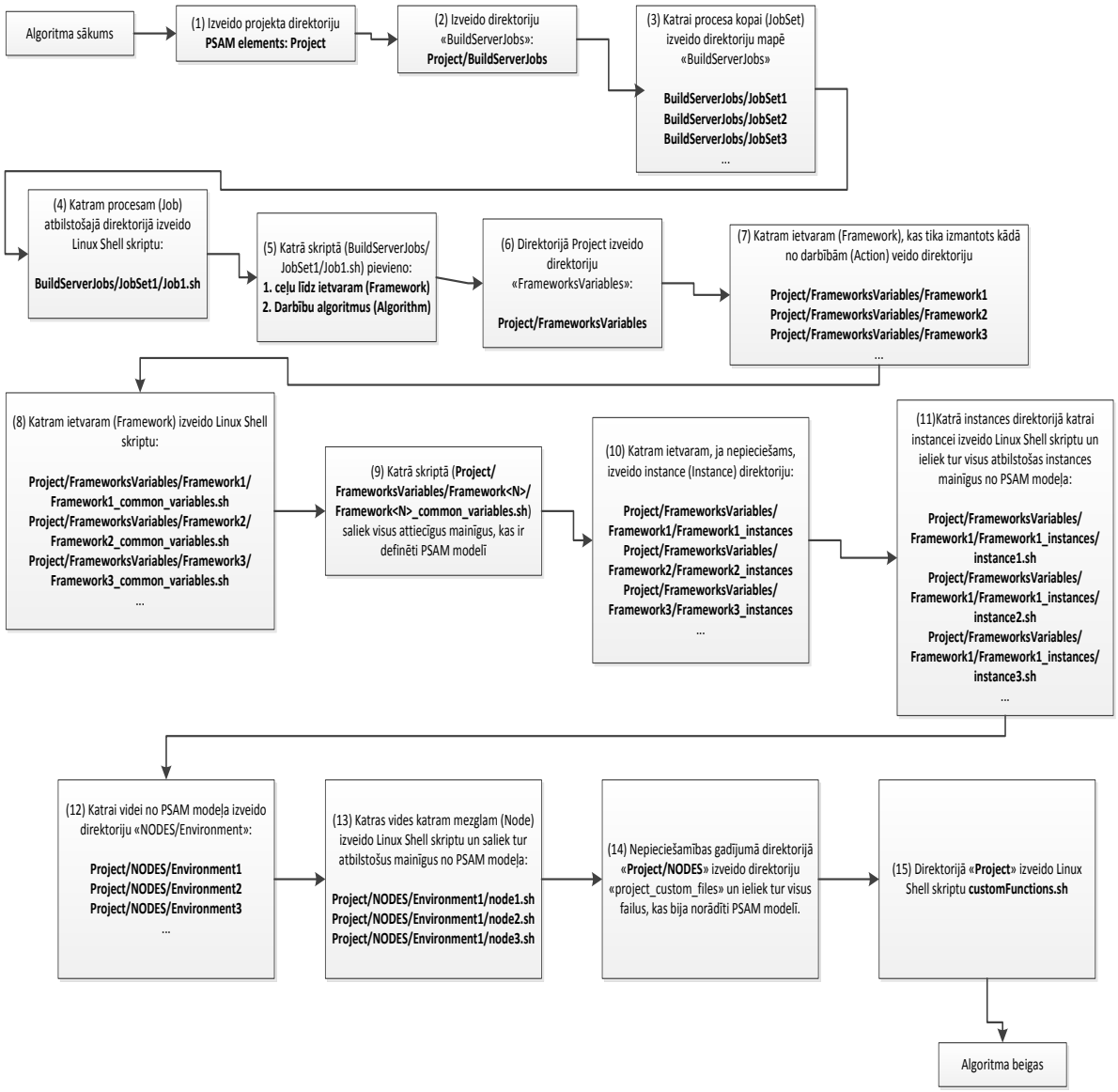
Konfigurācijas pārvaldības *EAF* metodoloģijas koda modelis (*Code Model*) ir ļoti atkarīgs no platformas, kurā tiek realizēts konfigurācijas pārvaldības izejas kods. Transformējot *PSAM* iegūto modeli koda modelī, ir jāievēro gan platformas, gan konkrētās programmēšanas tehnoloģijas likumus. Tāpēc, aprakstot *EAF* metodoloģijas principus un teorētiskos aspektus, nevar dot universālu transformācijas algoritmu *PSAM->Code Model*.

Šajā darbā izejas koda modelis tika uzģenerēts *Linux* platformai. Koda modelis ir *Linux Shell* skriptu komplekts, kas tika ģenerēts, ņemot vērā *Linux Shell* programmēšanas likumus un gatavu *PSAM* modeli. Koda modeļa piemērs atbilstoši iepriekš izveidotam *PSAM* modelim ir redzams attēlā 5.7.



5.7. att. Koda modelis *Linux* platformai

Koda modelis, kas ir redzams 5.7. attēlā, tika veidots, izmantojot transformācijas algoritmu *PSAM->Code Model*, kas tika izstrādāts saistībā ar *EAF* metodoloģiju speciāli *Linux Shell* platformai. Algoritma soļi ir redzami attēlā 5.8.



5.8. att. *Linux* koda modeļa ģenerēšanas algoritms

Izejas koda modelī galvenais skripts, kas tiks izsaukts no nepārtrauktās integrācijas servera, ir *dev_to_test.sh*. Attēlos 5.9. un 5.10. ir parādīti skripta fragmenti, kas piesaista ietvara (*Framework*) mainīgos un funkcijas, kā arī darbības (*Action*) *getRevisions()* algoritmu.

```
1  #=================SUBVERSION=========================================================#
2  . ./Platforms/Linux_shell/ToolsFrameworks/SUBVERSION/subversion_functions.sh
3  . ./Projects/VZD/FrameworksVariables/SUBVERSION/subversion_common_variables.sh
4  export NODES_HOME="./Projects/VZD/NODES/TEST"
5
```

5.9. att. Ietvara implementācija *Linux Shell* skriptā

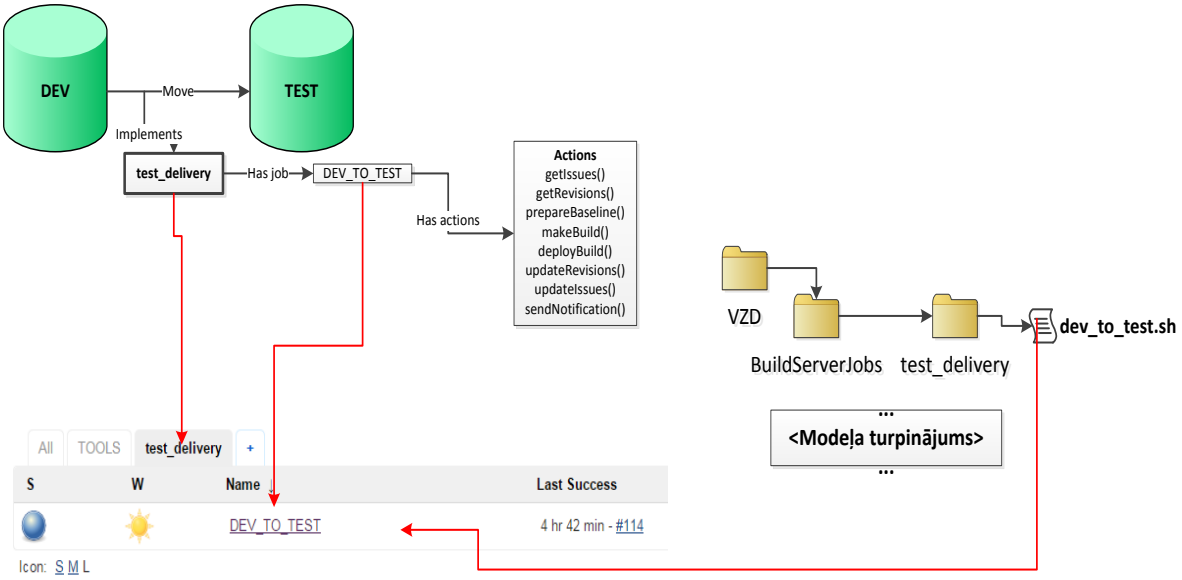
```
#Action getRevisions
#=====================================================================================================

#Iet cauri visām nodem, nolasa attiecīgus parametrus un nosaka uz atbilstošās vides nepiegādātas revīzijas
for nodes in `ls ${NODES_HOME}`
do
    #Inicializē konkrētas daļas mainīgus
    . ${NODES_HOME}/${nodes}
    #nosaka nepiegādātas revīzijas

    echo "Find revisions for: ${SUBVERSION_URL} by attribute ${SUBVERSION_ATRRIBUTE}"

    if [ "${SUBVERSION_GET_REVISION_FLAG}" = "Y" ]; then
        if [ "${SUBVERSION_GET_ALL_REVISIONS_FLAG}" = "Y" ]; then
            SUBVERSION_GET_UNDELIVERED_REVISIONS ${SUBVERSION_URL} ${SUBVERSION_USER} ${SUBVERSION_PASSW} $
        else
            SUBVERSION_GET_UNDELIVERED_REVISIONS_FOR_ISSUES ${SUBVERSION_URL} ${SUBVERSION_USER} ${SUBVERSI
        fi

        if [ -f ${SUBVERSION_TEXT_FILE_FOR_REVISIONS} ]; then
            echo "Nepiegādātas SVN revīzijas ir:"
            echo ""
            cat ${SUBVERSION_TEXT_FILE_FOR_REVISIONS}
            echo ""
        fi
    else
        echo "Nodei ${SUBVERSION_URL} nav nepieciešams meklēt nepiegādātas revīzijas"
    fi
done
# END Action getRevisions
#=====================================================================================================
```

5.10. att. Algoritms *getRevisions()* darbībai *Linux Shell* skriptā

Kad koda modelis ir gatavs, *EAF* metodoloģijā viss ir gatavs konfigurācijas pārvaldības procesu palaišanai. Lai palaistu konfigurācijas pārvaldības procesus, ir nepieciešams veikt šādus soļus:

1. pēc instrukcijas uzinstalē konfigurācijas pārvaldības serveri no *PSAM* modeļa;
2. uz konfigurācijas pārvaldības servera nokonfigurē visas procesa kopas (*JobSet*) un katras kopas visus procesus (*Job*). Informāciju par procesiem ņem no *PSAM* modeļa;
3. veic nepieciešamās papildu instalācijas un pielāgojumus konfigurācijas pārvaldības serverī atkarībā no ietvariem (*Framework*), kas tika izvēlēti *PSAM* modeļa veidošanas laikā;
4. koda modeli papildina ar projekta specifiskām funkcijām un mainīgajiem;
5. izpilda koda modeli, izlabo kļūdas un nepilnības;
6. process ir gatavs funkcionēšanai.

Attēlā 5.11. var redzēt uzinstalētā konfigurācijas pārvaldības servera fragmentu uz *Jenkins* piemēra, kā arī servera elementu saikni ar modeļiem.



5.11. att. Konfigurācijas pārvaldības modeļu un procesu saikne

5.2. Eksperimentu plāns uzlabotās *EAF* versijas testēšanai

Eksperimentu plāns ir līdzīgs eksperimentu pirmajai kārtai. Testējot *EAF* metodoloģijas uzlaboto versiju, tika izmantoti tie paši pieci programmatūras izstrādes projekti.

Gatavojot materiālus eksperimentiem, tika veiktas šādas darbības:

- speciālistu kompetences grupai uzņēmumā SIA «*Tieto Latvia*» tika organizētas papildu apmācības, kurās tika prezentēti visi *EAF* metodoloģijas uzlabojumi un atšķirības no pirmās versijas;
- tika izstrādāts risinājumu pārvaldības modulis atbilstoši jaunajai struktūrai, kas ir orientēta uz platformām un ietvariem (*Framework*);
- par katru no pieciem projektiem no SIA «*Tieto Latvia*» korporatīvas darba laika uzskaites sistēmas tika iegūti šādi dati:
    - laiks, kas tika patērēts konfigurācijas pārvaldības procesu sākotnējai ieviešanai, kad vēl nebija *EAF* metodoloģijas;
    - vidējais laiks nedēļā, kas tika patērēts konfigurācijas pārvaldības procesu regulārai uzturēšanai pirms jebkādas *EAF* metodoloģijas ieviešanas;
    - nedēļu skaits līdz paredzētajam projekta beigu datumam.

No katra projekta konfigurācijas pārvaldības datubāzes tika iegūta šāda informācija:

- programmatūras būvējumu skaits pirms *EAF* ieviešanas;
- programmatūras kļūdaino būvējumu skaits pirms *EAF* ieviešanas.

Katrā no pieciem projektiem tika veikts *EAF* metodoloģijas ieviešanas eksperiments pēc šāda plāna:

- izmantojot programmatūras prototipu, tika izveidots no platformas neatkarīgais vižu modelis (*PIEM*);
- *PIEM* tika papildināts ar informāciju par konfigurācijas pārvaldības darbību realizāciju. Katrai darbībai konfigurācijas pārvaldnieks izvēlējās konkrētās platformas ietvaru un definēja atbilstošos ietvara atribūtus. Ietvari tika izvēlēti no jaunizstrādātā risinājumu pārvaldības moduļa. Rezultātā tika izveidots *PSAM* modelis;
- izveidotais *PSAM* modelis tika transformēts koda modelī (*CM*);
- koda modelis tika implementēts konfigurācijas pārvaldības serverī;
- tika fiksēts laiks, kas tika patērēts, sākot ar *PIEM* veidošanu un beidzot ar *CM* modeļa implementāciju;
- programmatūras konfigurācijas pārvaldības process funkcionēja pēc *EAF* metodoloģijas trīs mēnešu laikā. Pēc tam tika fiksēti šādi rādītāji:
  - vidējais laiks nedēļā, kas bija nepieciešams procesu uzturēšanai un procesa kļūdu labojumiem;
  - programmatūras būvējumu kopējais skaits;
  - programmatūras kļūdaino būvējumu skaits.

Tika organizēta sapulce, kurā kompetences grupas dalībnieki izvērtēja sākotnēji iegūtos un eksperimentu gaitā fiksētos rādītājus par patērēto laiku un būvējumu skaitu. Rādītāji tika salīdzināti ar eksperimentu pirmo kārtu, tika izdarīti secinājumi par ieguvumiem.

5.3. *EAF* metodoloģijas uzlabotās versijas testēšana

Testējot *EAF* metodoloģijas jaunu versiju, tika veikti eksperimenti ar tiem pašiem projektiem, kas tika aprakstīti promocijas darba iepriekšējā nodaļā. Lai eksperimentu rezultātus varētu salīdzināt ar pirmās kārtas eksperimentiem, tika izmantoti tādi paši vērtēšanas kritēriji un tādi paši rādītāji. Tabulā 5.5. ir redzami fiksēti rādītāji pēc *EAF* metodoloģijas jaunās versijas ieviešanas piecos programmatūras izstrādes projektos, kas jau

bija aprakstīti iepriekš. Tabulā 5.6. ir redzami kritēriji eksperimentu otrajai kārtai, kas tika aprēķināti, balstoties uz 5.5. tabulā esošajiem datiem. Aprēķini tika veikti pēc tādām pašām formulām, kas bija minētas iepriekšējā nodaļā. Rādītāji tika fiksēti pēc tam, kad bija pagājuši divi mēneši kopš *EAF* jaunās versijas ieviešanas projektos. Lai neapdraudētu reālo projektu gaitu, arī šajā eksperimentu kārtā tika izmantoti jauni konfigurācijas pārvaldības serveri.

5.5. tabula

Otrās kārtas eksperimentu rādītāji

Rādītāji											
Projekts	***IMPL_TIME_OLD***	***IMPL_TIME_NEW***	***AVG_H_OLD***	***AVG_H_NEW***	***REM_TIME***	***(REM_TIME X AVG_H_OLD)***	***(REM_TIME X AVG_H_NEW) + IMPL_TIME_NEW***	***FAILED_BUILDS_BEFORE***	***FAILED_BUILDS_AFTER***	***BUILD_COUNT_BEFORE***	***BUILD_COUNT_AFTER***
1	160	145	10	8	52	520	561	15	10	84	82
2	300	30	15	14	36	540	534	5	4	160	98
3	76	45	2	1,5	104	208	201	3	1	65	64
4	45	5	8	4	52	416	213	10	3	73	75
5	203	8	3	3	52	156	164	7	3	88	90

5.6. tabula

Otrās kārtas eksperimentu rezultātu apkopojums

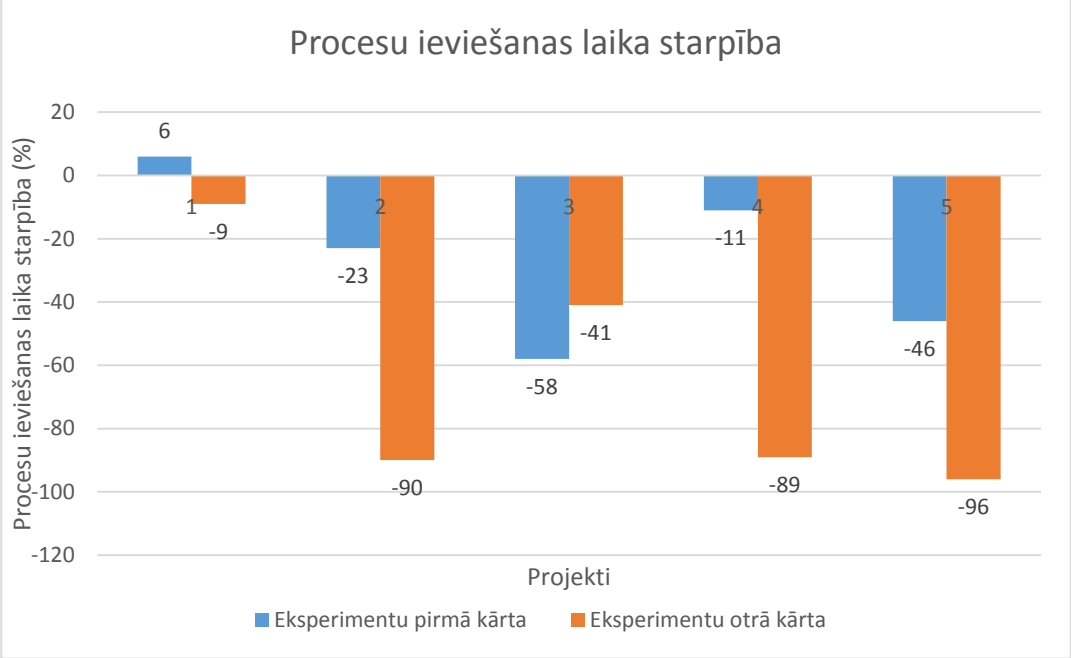
Kritēriji				
Procesu ieviešanas laika starpība (%)	**Regulārā uzturēšanas laika starpība (%)**	**Kopējā uzturēšanas laika starpība (%)**	**Kļūdaino būvējumu starpība (%)**	**Būvējumu kopējā skaita starpība (%)**
–9	–20	8	–33	–2
–90	–7	–1	–20	–39
–41	–25	–3	–67	–2
–89	–50	–49	–70	3
–96	0	5	–57	2

Analizējot otrās kārtas eksperimentu rezultātus, jāsecina, ka nedaudz samazinājās kļūdaino būvējumu skaits, salīdzinot ar pirmo eksperimentu kārtu. Pirmajā kārtā kļūdaino būvējumu skaits samazinājās vidēji par 38 procentiem, savukārt otrajā kārtā – par 49 procentiem. Izmantojot *EAF* metodoloģijas uzlaboto versiju, gandrīz uz pusi samazinās kļūdaino būvējumu skaits. Tas ir panākts, pateicoties uzlabojumiem risinājumu datubāzes struktūrā. Tagad koda vienības, ko izmanto projekti, ir mazākas, vieglāk uzturamas un ir vieglāk organizēt pārdomātu kļūdu apstrādi.

Eksperimentu otrajā kārtā būvējumu kopējais skaits projektos īpaši nav mainījies, salīdzinot ar pirmo kārtu. Līdzīgi kā pirmajā eksperimentu kārtā, arī tagad projektā «2» aptuveni par 40 procentiem samazinājās kopējais būvējumu skaits. Pārējos projektos arī šoreiz kopēju būvējumu skaita starpība svārstās dažu procentu intervālā.

Pateicoties *EAF* metodoloģijas uzlabojumiem, izdevās krietni samazināt procesu ieviešanas laiku. Vidējais rādītājs procesu ieviešanas laika samazināšanai visos piecos

projektos ir 65 procenti, un tas ir ļoti labs rādītājs, ņemot vērā, ka sākotnēji risinājumu datubāze (*Solution Database*) bija tukša. Attēlā 5.12. var redzēt grafiku, kurā tiek salīdzināta ieviešanas laika starpība eksperimentu pirmajā un otrajā kārtā.



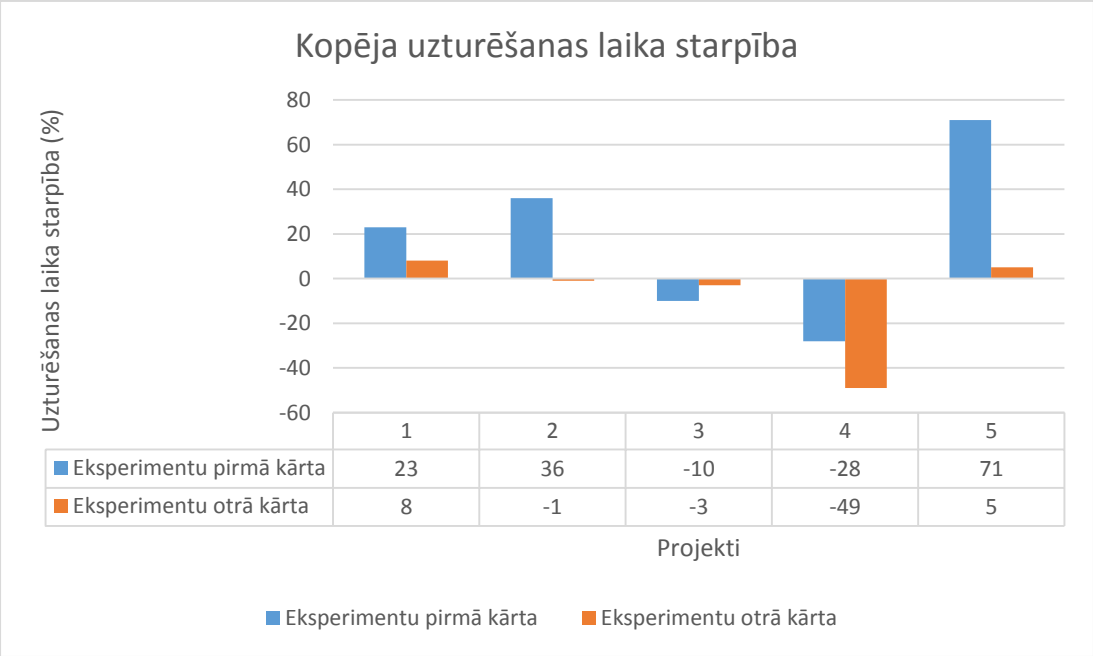
5.12. att. Procesu ieviešanas laika salīdzinājums divās eksperimentu kārtās

Kā redzams attēlā 5.12., arī šoreiz projektā «1» ieviešana aizņēma visvairāk laika, jo arī šoreiz risinājumu datubāze sākotnēji bija tukša. Taču, pateicoties tam, ka izdevās uzlabot risinājumu datubāzi, projektā «2» gandrīz pilnībā izdevās atkārtoti izmantot izejas kodu, kas bija izstrādāts projektam «1». Projektiem «1» un «2» ir līdzīgas tehnoloģijas, līdz ar to risinājumu datubāzē nevajadzēja pievienot klāt jaunus ietvarus (*Framework*). Pavisam citādāka situācija bija ar projektu «3», kur vajadzēja risinājumu datubāzē pielikt klāt jaunu ietvaru (*Framework*), kas varētu strādāt ar *Git* versiju kontroles sistēmu. Rezultātā ieviešanas laika starpība ir –41 %, salīdzinot ar –58 % eksperimentu pirmajā kārtā. Tas nozīmē, ka projektā «3» procesu ieviešana eksperimentu otrajā kārtā aizņēma pat nedaudz vairāk laika, salīdzinot ar pirmo kārtu. Pavisam iepriecinoši rādītāji tika fiksēti projektos «4» un «5», kur procesu ieviešanas laiks, salīdzinot ar vecajām metodēm bez jebkādām metodoloģijām, ir samazinājies attiecīgi par 89 un 96 procentiem. Šāda tendence ir skaidrojama ar to, ka – jo vairāk risinājumu satur risinājumu datubāze (*Solution Database*), jo vairāk izejas koda konfigurācijas pārvaldības procesiem jaunajos projektos var izmantot atkārtoti. Ieviešot konfigurācijas pārvaldību pēc *EAF* metodoloģijas, risinājumi tiek uzlaboti, un tiek novērstas

dažādas kļūdas, līdz ar to *Solution Database* sistēma paliek arvien lietderīgāka jaunajiem projektiem.

Skatoties uz regulārās uzturēšanas laika starpību eksperimentu otrajā kārtā, jāsecina, ka iepriekš atklātā tendence nav mainījusies. Respektīvi, ja projektam pirms jebkādas metodoloģijas ieviešanas bija salīdzinoši zems automatizācijas līmenis, *EAF* metodoloģijas ieviešana to diezgan būtiski uzlaboja, taču, ja arī pirms tam automatizācijas līmenis bija salīdzinoši augsts, *EAF* metodoloģija to būtiski neietekmēja. Otrās kārtas eksperimenti rāda, ka, ieviešot *EAF* metodoloģiju, vidēji par 20 procentiem var samazināt laiku, kas nepieciešams procesu ikdienas uzturēšanai.

Analizējot eksperimentu otrās kārtas rezultātus, visbeidzot tiek salīdzināta procesu kopējā uzturēšanas laika starpība, kas būtībā atbild uz jautājumu, vai konkrētajā projektā ilgtermiņā būtu izdevīgi ieviest *EAF* metodoloģiju. Attēlā 5.13. var redzēt kopējā uzturēšanas laika starpības salīdzinājumu pirmajā un otrajā eksperimentu kārtā.



5.13. att. Kopējā uzturēšanas laika salīdzinājums divām eksperimentu kārtām

Atšķirībā no eksperimentu pirmās kārtas tagad ilgtermiņa ieguvums krietni palielinājies. Izdevās uzlabot pat projekta «5» rādītājus. Pirmajā kārtā ieviest jaunu metodoloģiju bija par 71 procentu ilgāk, nekā uzturēt projekta konfigurācijas pārvaldību pēc vecajiem paņēmieniem, savukārt pēc uzlabotās *EAF* ieviešanas šis cipars pazeminājās līdz 5 procentiem. Papildus jāatzīmē, ka līdz projekta beigām atlika tikai gads. Ja projekts

darbotos ilgāk, *EAF* jaunas versijas ieviešana noteikti dotu ilgtermiņa ieguvumu. Kopumā jāsecina, ka visos projektos, izņemot projektu «3», izdevās uzlabot ilgtermiņa ieguvuma rādītājus. Spiežot pēc eksperimentu otrās kārtas datiem, tikai projektos «1» un «5» *EAF* ieviešana kopumā nav izdevīga, savukārt pirmā *EAF* versija deva pozitīvu ieguvumu tikai divos projektos («3» un «4»). Pirmajā projektā ieviest *EAF* metodoloģiju nebūtu izdevīgi, jo līdz projekta beigām ir atlicis salīdzinoši maz laika (tikai viens gads), taču ieviešana pie tukšās risinājumu datubāzes (*Solution Database*) aizņem diezgan daudz laika. Savukārt neizdevīga lietošana projektā «5» skaidrojama gan ar to, ka līdz projekta beigām ir tikai viens gads, gan arī ar to, ka neizdevās samazināt regulārās uzturēšanas laiku. Analizējot šo gadījumu sīkāk, tika konstatēts, ka projektā vēl pirms jebkādas *EAF* ieviešanas bija visai augsts procesu automatizācijas līmenis. Pēc *EAF* uzlabotās versijas ieviešanas uzturēšanas laiks palika nemainīgs, un tas ietekmēja *EAF* ieviešanas ilgtermiņa ieguvumu.

5.4. *EAF* metodoloģijas trūkumu novēršanas pasākumi

EAF metodoloģijas jaunās versijas izstrāde, kas tika aprakstīta šajā nodaļā, tika izstrādāta ar mērķi novērst trūkumus, kas tika atklāti, veicot pirmās kārtas eksperimentus un publicējot metodoloģijas teorētiskus pamatus zinātniskajos rakstos. Tabulā 5.7. vēlreiz ir apkopoti *EAF* metodoloģijas sākotnējas versijas trūkumi, kā arī pasākumi, kas tika veikti trūkumu novēršanai.

5.7. tabula

***EAF* metodoloģijas sākotnējas versijas trūkumi un to novēršanas pasākumi**

Trūkuma kārtas numurs	**Apraksts**	**Novēršanas pasākumi *EAF* uzlabotās versijas izstrādē**
1	Risinājumu izvēles moduļa struktūra. Eksperimentu rezultāti atklāja, ka strukturēt konfigurācijas pārvaldības izejas kodu pēc konfigurācijas pārvaldības galvenajiem uzdevumiem (*compile, deploy*, *prepare baseline*) ir pārāk plaši. Šajā gadījumā funkcijas satur ļoti daudz parametru, un funkcijas	Pilnībā tika pārstrādāta risinājumu glabātuve (*Solution Database*). Atkārtoti izmantojams izejas kods konfigurācijas pārvaldības uzdevumiem tika sadalīts pa platformām (*Platform*) un ietvariem (*Framework*). Tika ieviesti jaunie jēdzieni: mezgls (*node*) un instance (*instance*), kas atviegloja sarežģītu gadījumu apstrādi bez liekām

	ķermenis satur daudz atzarojumu. Ar laiku šādu kodu kļūst ļoti sarežģīti uzturēt, jo ir jāņem vērā projektu specifika, pat tad, ja programmatūras tehnoloģijas ir līdzīgas. Šo trūkumu atzīmēja gan speciālisti, kas palīdzēja īstenot eksperimentus, gan starptautisku konferenču rakstu recenzenti.	izejas koda pielāgošanām konkrētu projektu specifikai. Atkārtoti lietojamas izejas koda vienības kļuva mazākas un līdz ar to arī vieglāk uzturamas.
2	Vižu modeļa struktūra. Esošā vižu modeļa interpretācija ļoti ierobežo projektus. Pirmkārt, vižu modelī jāparedz iespēja, ka programmatūras pārnešanas starp vidēm notiks vairākos notikumos (*Event*) un konfigurācijas plūsmas (*ConfigurationItemFlow*) arī var būt sadalītas sīkāk atkarībā no projektu specifikas. Otrkārt, kā atzīmēja konferenču rakstu recenzenti un tehniskie speciālisti, jēdzieni notikums (*Event*) un konfigurācijas plūsma (*ConfigurationItemFlow*) nav intuitīvi saprotami. Līdz ar to būtu nepieciešams atrast veidu, kā vienkāršāk strukturēt konfigurācijas pārvaldības darbības, kas pārnes programmatūras izmaiņas starp vidēm. Papildus tam vižu modelēšanas gaitā konfigurācijas pārvaldniekam būtu jāsniedz iespēja brīvāk strukturēt darbības	Tika likvidēts jēdziens *PIAM* modelis. Vižu modelis tika pārveidots, un tajā tika iekļautas konfigurācijas pārvaldības darbības. Rezultātā konfigurācijas pārvaldniekam ir vairāk izvēles brīvības: viņš pats var izlemt kādas darbības ir nepieciešamas programatūras izmaiņu pārnešanai starp vidēm. Tika likvidēti vižu atribūti, kuru sākotnējā versijā bija pārāk daudz un, kā atzīmēja teorētisko pamatu recenzenti, tie nebija intuitīvi saprotami. Notikumu un plūsmu jēdzieni arī tika aizvietoti ar procesu kopas un procesu jēdzieniem, kas ir labāk saprotams konfigurācijas pārvaldniekiem, kas līdz šim ir strādājuši ar nepārtrauktās integrācijas serveriem. Rezultātā tika iegūts *PIEM* modelis (no platformas neatkarīgs vižu modelis), kurā tika apvienotas sākotnējas idejas par *EM* un *PIAM* modeļiem, taču papildus tika veikta virkne uzlabojumu.

	pa notikumiem un plūsmām. Visbeidzot, jāpārskata notikumu un plūsmu jēdzienu, lai konfigurācijas pārvaldniekiem jēdzieni būtu intuitīvi saprotamāki.	
3	*PIAM* modeļa būtība. Darbību kopa, kas ir aprakstīta *PIAM* meta-modelī, nav pilnīga. Ir jābūt iespējai pielikt klāt jaunas darbības. Papildus tam transformācija no *EM* uz *PIAM* modeli ļoti ierobežo projektus, kuriem ir jāveic vēl citas darbības, kas nav definētas transformācijas likumos. Iesniedzot modeļu aprakstus zinātniskajai konferencei *MODELSWARD 2015*, tika saņemts ieteikums apvienot *EM* un *PIAM* modeļus, ļaujot lietotājam pašam izvēlēties darbības, kā arī paplašināt darbību kopu meta-modelī.	Šo trūkumu novērsa trūkuma «2» novēršanas pasākumi.
4	Izejas koda zarošanas modelis neatspoguļo dažādas izejas koda pārvaldības stratēģijas. Ir projekti, kuriem jau ir citas stratēģijas, un šajā gadījumā *EAF* metodoloģijas ieviešanu apgrūtina fakts, ka metodoloģija paredz noteiktu zarošanas pieeju. Līdz ar to radās ieteikums zarošanas pieeju pasniegt vien kā rekomendāciju, taču atstāt projektiem zināmu brīvību zaru nosaukumu un zarošanas pieejas	Tika likvidēts izejas koda zarošanas modelis. Risinājumu datubāzē ietvaros (*Framework*) ir aprakstītas iespējamās izejas koda pārvaldības stratēģijas, taču konkrēti ierobežojumi netika ieviesti. Līdz ar to neatkarīgi no *PIEM* modeļa konfigurācijas pārvaldnieks var pats brīvi izvēlēties izejas koda pārvaldības stratēģiju. Papildus kļuva vieglāk ieviest *EAF* metodoloģiju projektos, kur jau gadiem funkcionē noteikta veida izejas koda pārvaldība un kuru projekts nav

	izvēlē.	gatavs mainīt.
5	Servisu modelis neparedz darbību ar vairākām instancēm un tehnoloģijām. Pieņemsim, ir situācija, kad konkrēta programmatūras laidiena aprakstam ir nepieciešama informācija no vairākām dažādām pieteikumu apstrādes sistēmām. Šajā gadījumā servisu modelim jābūt pietiekami elastīgam, lai ļautu pieslēgties dažādām sistēmām tā, lai funkcijās nevajadzētu ņemt vērā projektu specifiku. Līdzīga problēma ir arī ar konfigurācijas pārvaldības darbību realizāciju, kad, piemēram, programmatūra var sastāvēt no dažādām komponentēm un katra komponente ir izstrādāta savā programmēšanas valodā, un līdz ar to būvējumu un instalācijas specifika ir pilnīgi atšķirīga.	Servisu modeļa jēdziens tika likvidēts, jo jaunā risinājumu datubāze jau atrisina rīku integrācijas problēmas, kas tika konstatētas *EAF* sākotnējā versijā.
6	Prototipa izstrādē netika apsvērtas iespējas formalizēt modeļus un to transformācijas ar kādu modelēšanas rīku, piemēram, *MetaEdit+* vai *Eclipse Modelling Framework*. Tas palielina risku, ka modeļu veidošanā tiks pieļautas kļūdas, kas netiks atklātas modeļu implementācijas laikā.	Izmantojot *MetaEdit+* modelēšanas valodu izstrādes rīku, tika izstrādātas valodas *MTM* pieejas realizācijai, kā arī *PIEM* modeļu veidošanai. Rīks piedāvā iespēju automātiski pārveidot modeli *XML* formātā un apstrādāt to ar aprakstītu prototipu. Tas samazina kļūdu risku, ka modeļi tiks veidoti kļūdaini cilvēciskā faktora dēļ.

5.5. *EAF* metodoloģijas uzlabotās versijas galvenie ieguvumi, atšķirības no citiem konfigurācijas pārvaldības risinājumiem un turpmākās uzlabošanas virzieni

Analizējot otrās kārtas eksperimentu rezultātus, tika konstatēti šādi *EAF* metodoloģijas ieguvumi:

- ieviešot *EAF* metodoloģiju minētajos piecos projektos, vidēji par 65 procentiem samazinājās konfigurācijas pārvaldības automatizācijas ieviešanas laiks. Tas rāda tendenci, ka esošo automatizācijas risinājumu izmantošana tiešām ļauj būtiski samazināt automatizācijas ieviešanu jaunajos projektos. Tas notiek tāpēc, ka metodoloģija iespēju robežās jaunajos projektos atkārtoti izmanto jau esošo izejas kodu, kas realizē konfigurācijas pārvaldības uzdevumus. No jauna tiek izstrādāts tikai projektam specifiskais izejas kods;
- *EAF* metodoloģija palīdz atbrīvot projektu no manuālām darbībām konfigurācijas pārvaldības procesu uzturēšanā. Pateicoties metodoloģijas principam, ka konfigurācijas pārvaldībā izpildāms izejas kods, manuālas darbības vairs netiek veiktas. Eksperimenti rāda, ka projektos ar salīdzinoši zemu automatizācijas līmeni *EAF* metodoloģija krietni to uzlabo. Veiktajos eksperimentos vidēji par 20 procentiem izdevās samazināt laiku, kas ir nepieciešams konfigurācijas pārvaldības procesu regulārai uzturēšanai;
- *EAF* metodoloģija eksperimentālajos projektos par 49 procentiem samazināja kļūdaino būvējumu skaitu. Eksperimenti parādīja, ka, pateicoties pārdomātai kļūdu apstrādei un automatizētai rīku savstarpējai integrācijai, kļūdaino būvējumu skaits projektā samazinās. Jo vairāk risinājumu satur risinājumu glabātuve un jo stabilāki tie ir, jo mazāks ir kļūdaino būvējumu skaits. Ieviešot *EAF* metodoloģiju projektos «4» un «5», kad risinājumu datubāzei bija visi nepieciešamie ietvari (*Framework*), kļūdaino būvējumu skaits samazinājās attiecīgi par 70 un 57 procentiem. Tas parāda tendenci, ka kļūdaino būvēju skaits krietni samazinās, pateicoties *EAF* metodoloģijai.
- *EAF* metodoloģiju ir izdevīgi ieviest projektos, kur līdz šim tā nav ieviesta ar nosacījumu, ka līdz projekta beigām atlicis salīdzinoši daudz laika un procesu automatizācijas līmenis ir salīdzinoši zems (daudz manuālu darbību). Ja līdz

projekta beigām ir maz laika un procesi tāpat ir pietiekami labi automatizēti, pāreja uz *EAF* metodoloģiju varētu būt neizdevīga.

Veicot *MTM* pieejas un *EAF* metodoloģijas izstrādi un praktisko testēšanu, izdevās identificēt atšķirības no citiem konfigurācijas pārvaldības risinājumiem, kas tika aprakstīti un analizēti šajā promocijas darbā:

- gan *MTM* pieeja, gan *EAF* metodoloģija ir orientētas uz konfigurācijas pārvaldības automatizācijas ieviešanas laika samazināšanu, atkārtoti lietojot jau strādājošu izejas kodu;
- *EAF* metodoloģija paredz noteikta veida pieredzes apmaiņu starp projektiem, stingri reglamentē konfigurācijas pārvaldības automatizācijas ieviešanas procedūru ar modeļu palīdzību;
- *EAF* metodoloģija neliek obligāti izmantot konkrētu rīku kādam noteiktam konfigurācijas pārvaldības uzdevumam vai procesam kopumā. Tā vietā metodoloģija sniedz rekomendācijas, kas palīdz strukturēt uzņēmumā esošos automatizācijas risinājumus tā, lai tos iespēju robežās varētu lietot atkārtoti. Līdz ar to teorētiski metodoloģiju var izmantot gan darbam ar jaunākajiem modeļvadāmajiem risinājumiem, tādiem kā *Serena* un *OpenMake* produktiem, gan arī iepriekšējas paaudzes tehnoloģijām (statiski skripti), kas jau gadiem ilgi funkcionē uzņēmumos;
- sākotnēji *MTM* pieeja un *EAF* metodoloģija ir abstraktas. Tika definēta vien koncepcija, kā, izmantojot *MDA* formātu, var implementēt konfigurācijas pārvaldības procesus, katru reizi samazinot automatizācijas ieviešanas laiku. Sākotnējā abstrakcija un modelēšanas valoda, kas ir izstrādāta *MTM* pieejai, ļauj veidot arī citas *MTM* realizācijas ar nosacījumu, ka būs vismaz viens elements konfigurācijas pārvaldības risinājumu glabāšanai;
- *EAF* metodoloģija apskata konfigurācijas pārvaldības automatizāciju kopumā un atšķirībā no daudziem citiem mūsdienīgiem risinājumiem nekoncentrējas tikai uz versiju kontroli, būvējumu vai instalācijas uzdevumiem. Papildus tam metodoloģija ļauj pievienot arī savus uzdevumus un to automatizāciju, paplašinot ietvaru (*Framework*) funkcionalitāti. Projektiem netiek uzspiests kāds noteikts standarts, kas aprobežojas vien ar galveno uzdevumu automatizāciju.

Veicot pirmās un otrās kārtas eksperimentus, *EAF* metodoloģijas ieviešanā tika identificēti šādi riski:

- ietvara funkciju nepieejamības risks. Ja pēkšņi kāda ietvara funkcija darbosies nepareizi, tas ietekmēs uzreiz visus projektus, kur šāda funkcija tiek izmantota. Lai novērstu šo risku, nepieciešams ieviest ļoti stingru kontroli ietvara funkciju izmaiņu veikšanai. Ieteicams obligāti pakļaut ietvaru izejas kodu versiju kontrolei un veikt labojumus tikai ārkārtas nepieciešamības gadījumā. Visas izmaiņas ir jādokumentē, papildus tam vienmēr jābūt iespējai atgriezties ietvara iepriekšējā versijā;
- infrastruktūras ierobežojumu risks. Dažreiz programmatūras pasūtītāji pieprasa, lai konfigurācijas pārvaldības procesi darbotos viņu infrastruktūrā jeb korporatīvajā tīklā. Šajā gadījumā būs apgrūtinoši lietot ietvaru izejas kodu, jo diez vai programmatūras izstrādes kompānija vēlēsies atklāt kādam klientam konfigurācijas pārvaldības izejas kodu, ko lieto arī citu pasūtītāju projekti;
- ietvaru risinājumu pārvaldības risks. Pastāv iespēja, ka ne vienmēr varēs noteikt, kurā ietvarā (*Framework*) ir nepieciešams implementēt kādu darbību. Līdz ar to varētu rasties dažādi ietvari ar ļoti līdzīgu vai pat identisku funkcionalitāti. Šo risku varētu mazināt, piesaistot katram ietvaram atbildīgo personu, kas uztur ietvaru funkciju detalizētu aprakstu un seko, vai citos ietvaros nav funkciju, kas pārklājas ar viņa ietvara funkcijām.

Ņemot vērā veiktu eksperimentu rezultātus, *EAF* metodoloģijai varētu būt šādi attīstības virzieni:

- ietvaru (*Framework*) versionēšanas sistēmas izstrāde. Praksē var rasties situācijas, kad kādam noteiktam projektam var nederēt ietvara jaunāka versija. Jābūt sistēmai, kas uzturēs ietvaru versiju un uzskatāmi parādīs atšķirības starp versijām. Papildus tam jābūt iespējai katram projektam brīvi izvēlēties ietvara versiju neatkarīgi no citiem projektiem;
- vižu sākotnējās instalācijas procesu formalizācija. Šobrīd *EAF* metodoloģija paredz, ka visas vides jau ir izveidotas. Taču reāli programmatūras izstrādes projekta pašā sākumā vides veido no nulles: instalē operētājsistēmas, aplikāciju serverus, datubāzes, konfigurē ugunsmūrus utt. Šeit izpaužas līdzīga problēma, kas tika akcentēta šajā promocijas darbā, ka, ieviešot

procesu vienā projektā, jāprot to pēc iespējas ātrāk ieviest arī citā. Līdz ar to vižu sākotnējai konfigurācijas un instalācijas vadlīnijām un darbībām arī jāglabājas vienā struktūrā, un jābūt vienotai procedūrai, kā jaunajos projektos maksimāli iespējami izmantot jau citu projektu praksi. Līdz ar to nepieciešami *EAF* metodoloģijas papildinājumi ar jaunajiem modeļiem, kas atvieglos jaunu vižu instalācijas un konfigurēšanas procesus;

- jānovērtē *EAF* metodoloģijas atbilstība populārākajiem kvalitātes standartiem un vadlīnijām, piemēram, *CMMI, ISO, ITIL* utt;
- jāizstrādā vēl viens modelis, kas ļautu no *PIEM, PSAM* un *CM* modeļiem automātiski ģenerēt konfigurācijas pārvaldības plānu, kas ir neatņemama nepieciešamo projekta dokumentu sastāvdaļa, ko visbiežāk rakstiski apstiprina pasūtītājs. Savukārt konfigurācijas pārvaldības plāna manuāla veidošana izraisa risku, ka plāns nebūs atbilstošs konfigurācijas pārvaldības modeļiem;
- konfigurācijas pārvaldības modeļu un konfigurācijas pārvaldības problēmvides atgriezeniskās saites izstrāde. Jābūt mehānismam, kas analizē konfigurācijas pārvaldības problēmvidi un identificē vājās vietas konfigurācijas pārvaldības modeļos. Izstrādājot šo saiti, varētu izmantot mākslīgā intelekta metodes un veidot zināšanu bāzi, kurā būs likumi atgriezeniskās saites nodrošināšanai.

5.6. Praktiskas rekomendācijas modeļvadāmajai konfigurācijas pārvaldības ieviešanai programmatūras izstrādes projektā

Pamatojoties uz eksperimentu rezultātiem, kompetences grupas speciālistu atsauksmēm un zinātnisko konferenču recenzentu piezīmēm, tika izstrādāta virkne rekomendāciju un ieteikumu, kas palīdzēs praktiski ieviest konfigurācijas pārvaldību pēc *EAF* metodoloģijas.

1. uzmanīgi izpētīt *EAF* metodoloģijas galvenos principus, kas ir aprakstīti 5. nodaļā;
2. uzmanīgi izpētīt *EAF* metodoloģijas modeļus un risinājumu datubāzes struktūru. Galvenais ir saprast, kā strukturēt esošos konfigurācijas pārvaldības automatizācijas risinājumus atbilstoši *EAF* metodoloģijai. Kad ir labi izprasta

risinājumu datubāzes struktūra, jāizpēta modeļu *PIEM*, *PSAM* un *CM* būtība, transformācijas starp modeļiem;

3. izstrādāt programmatūru, kas strādās ar risinājumu datubāzēm un ļaus pārvaldīt risinājumus. Programmatūrai jānodrošina jaunu platformu (*Platform*), konfigurācijas pārvaldības serveru (*SCMServer*) un ietvaru (*Framework*) pievienošanu, rediģēšanu un dzēšanu. Būtu vēlams risinājumu datubāzi pakļaut versiju kontrolei un dokumentēt visas veiktās izmaiņas. Tas, pirmkārt, ļaus atgriezties iepriekšējās versijās, kā arī ļaus izsekot ietvaru attīstības vēsturei. Tas ļaus labāk saprast, kādas ir ietvara problēmas, kuras no tām ir novērstas, kā arī ļaus spriest par konkrētā ietvara vai tā funkciju stabilitāti. Katram ietvaram (*Framework*) ir obligāti jābūt konfigurācijas pārvaldības darbību (*Action*) sarakstam. Strādājot ar risinājumu pārvaldības aplikāciju, katram ietvaram jābūt iespējams atlasīt visas darbības (*Action*), ko tas prot paveikt. Strādājot ar ietvaru (*Framework*) papildināšanu, būtu vēlams katram ietvaram noteikt atbildīgo personu, kas vislabāk pārzinātu konkrētā ietvara funkcijas un varētu kontrolēt visas izmaiņas, kas tajās jāveic. Tas ļaus samazināt risku, ka nekorekta labojuma dēļ konfigurācijas pārvaldības process ģenerēs kļūdas uzreiz vairākos projektos;
4. izstrādāt programmatūru, kas realizētu *PIEM, PSAM* un *CM* modeļu veidošanu un modeļu transformācijas atbilstoši *EAF* metodoloģijai. Izstrādājot modeļu ģenerēšanas rīkus, ir ieteicams izmantot kādu no populārām modelēšanas valodas izstrādes programmatūrām, piemēram, *MetaEdit+* un *Eclipse Modelling Framework*. Šādu rīku izmantošana samazinās darbu apjomu pie *EAF* metodoloģijas modelēšanas valodas izstrādes, kā arī vēlāk atvieglos modelēšanas valodas testēšanu un validāciju. Eksperimentiem, kas tika veikti saistībā ar promocijas darbu, tika izmantots rīks *MetaEdit+*. Eksperimenti parādīja, ka jaunu modelēšanas valodu varēja izveidot vien dažu stundu laikā, un tas ir ievērojami ātrāk, nekā no nulles izstrādāt pilnīgi jaunu programmatūru;
5. sarīkot apmācības speciālistu grupai, kas strādās pie konfigurācijas pārvaldības ieviešanas, izmantojot *EAF* metodoloģiju. Speciālistiem vajadzētu labi saprast gan *EAF* metodoloģijas darbības principus, gan arī risinājumu pārvaldības programmatūru un modeļu veidošanas rīku. Apmācībās ieteicams izveidot dažus ietvarus (var arī nestrādājošus) un modeļus konfigurācijas pārvaldības procesiem dažos projektos. Tas ļaus labāk saprast jauno rīku darbības principus, kā arī ļaus

konstatēt kļūdas programmatūrā vēl pirms tam, kad modeļi tiks veidoti īstiem programmatūras izstrādes projektiem;

6. *PIEM* modeļu veidošanas posms. Kā jau tika minēts darba trešajā nodaļā, projekta vižu skaits tieši ietekmē projekta resursus. Jo vairāk vižu, jo vairāk nepieciešams gan datorresursu, gan cilvēku resursu to administrēšanai un uzturēšanai. Tāpēc *PIEM* ieteicams veidot kopā ar pasūtītāju, lai rastu kompromisu starp riskiem un resursiem. Svarīgi ir izskaidrot pasūtītājam, kāpēc programmatūras izmaiņu pārnešanu no vides *X* uz vidi *Y* ir svarīgi testēt, kur slēpjas problēmas, kādos gadījumos vide var kļūt nepieejama utt. Ja *IT* uzņēmumam ir vižu modeļi no citiem projektiem, ieteicams riskus izskaidrot ar konkrētiem piemēriem. Jācenšas panākt situāciju, lai pasūtītājs būtu pārliecināts par izpildītāja komandas kompetenci. Tādējādi jācenšas vižu modeļa saskaņošanas dialogu veidot nevis pēc principa «mēs pamēģināsim un, iespējams, sanāks», bet «mēs esam to paveikuši tā un tā, ja jūs izvēlēsieties modeli X, riski būs šādi, ar modeli Y iespējamas citas problēmas, mūsu pieredze rāda, ka modelis Z ir visvairāk piemērots tieši jūsu gadījumiem utt. ». Vižu modeļa veidošana un saskaņošana jāveic projekta pašā sākumā, nevis tad, kad darbs jau rit pilnā sparā. Gan izpildītājam, gan pasūtītājam jābūt skaidrībai par katras vides nozīmi un mērķiem. Jāsaprot, kurā brīdī un kādā nolūkā tur nonāks jaunā konfigurācija, kā konfigurācija tiks identificēta pieteikumu apstrādes sistēmā. Promocijas darba autoram, kurš nodarbojas ar konfigurācijas pārvaldību vairāk nekā sešus gadus, ir pieredze, kad pasūtītājs mēģināja testēt jauno produkta versiju akcepttesta vidē, lai gan izmaiņas bija tikai testa vidē. Pasūtītāju mulsināja relīzes pieteikuma statuss «Testēšanā», no kura bija grūti saprast, kurā vidē testēt risinājumu. Tāpēc šādus un līdzīgus gadījumus svarīgi definēt jau vižu modeļa veidošanas un saskaņošanas stadijā;
7. nepieciešams izstrādāt veidu, kā no *PIEM, PSAM* un *CM* modeļiem automātiski ģenerēt konfigurācijas pārvaldības plānu. Šis pasākums diez vai ļaus ātrāk ieviest konfigurācijas pārvaldību, taču uzskatāmi parādīs to, ka konfigurācijas pārvaldība ir procesu orientēta un dokumentēta. Līdz ar to gan iekšējos, gan ārējos kvalitātes auditos būs krietni vieglāk parādīt, ka process ir dokumentēts atbilstoši reālajai situācijai un rīki strādā atbilstoši aprakstītajam procesam. Promocijas darba autora prakse rāda, ka auditoru biežākais iebildums ir par to, ka konfigurācijas pārvaldības plāns ir pārāk formāls un «nedzīvs», proti, pēc tā

nevar saprast, kā reāli funkcionē konfigurācijas pārvaldība. Šāda dokumenta pievienotā vērtība līdzinās nullei, jo no tā jaunais projekta dalībnieks nevarēs saprast galvenās lietas, kas attiecas uz konfigurācijas pārvaldības realizāciju konkrētajā projektā;

8. kad *EAF* metodoloģija tiks ieviesta projektā, praksē tas izpaudīsies kā rīku savstarpēja informācijas apmaiņa un dažādu darbību (*Action*) izpilde. Šajā brīdī ļoti svarīgi ir izveidot atgriezenisko saiti starp rīku darbībām un modeļiem. Šī saite palīdzēs savlaicīgi identificēt vājās vietas un veikt nepieciešamus labojumus ietvaru funkcijās atkarībā no situācijas. Tieši tāpēc būtisks ieteikums ir, izstrādājot ietvarus, veidot žurnālfailus. Žurnālfailu ierakstos jābūt skaidri redzamam, kāds ietvars tiek izsaukts, kāda ietvara funkcija tiek darbināta, kāda konfigurācijas pārvaldības darbība tiek realizēta un kuram *PSAM* modelim šī darbība pieder. Šāda veida ieraksti kļūdu gadījumā ļauj noteikt vājas vietas *PSAM* modelī un savlaicīgi novērst trūkums, negaidot, piemēram, kad projektā būs 50 nekvalitatīvu būvējumu. Papildus tam ir ieteicams izveidot jebkādu darbību (*Action*) uzskaiti, lai jebkurā brīdī varētu pateikt, piemēram, cik būvējumu bija pagājušajā mēnesī, cik no tiem bija kļūdaini, cik daudz labojumu tika veikts konkrētajā izejas koda zarā, kādi konfigurācijas vienumi produktā tika mainīti visbiežāk utt. Tas ļaus labāk sekot konfigurācijas pārvaldības procesiem, plānot resursus, kas ir nepieciešami procesu uzturēšanai un izdarīt secinājumus par procesu intensitāti un kvalitāti.

Nobeigumā jāatzīmē, ka jauna metodoloģija konfigurācijas pārvaldības automatizācijas ieviešanai tika testēta šādām tehnoloģijām: *Oracle, JAVA, Subversion, Git, JIRA, Jenkins, Hudson, Bamboo, Linux Shell, Ruby*. Eksperimenti parādīja tendenci, ka būtiski samazinās automatizācijas ieviešanas laiks (piecos projektos vidēji par 65 %), aptuveni par 20 % samazinās procesu ikdienas uzturēšanas laiks, kā arī aptuveni uz pusi (veiktajos eksperimentos vidēji par 49 %) samazinās kļūdaino būvējumu skaitu. Lai precīzāk noteiktu ilgtermiņa ieguvumu jaunajai metodoloģijai, nepieciešams vairāk eksperimentu vairākos projektos ar lielākām tehnoloģiju kopām. Tas ir tāpēc, ka metodoloģijas ieviešanu ietekmē arī speciālistu kvalifikācija, projektu apjomi, tehnoloģiju specifika un esošo risinājumu automatizācijas un kvalitātes līmenis. Lielāks eksperimentu skaits nākotnē ļaus iegūt ne tikai precīzākus datus par ilgtermiņa ieguvumu, bet radīs arī idejas, kā formalizēt un skaitliski izteikt citus, tikko minētos faktorus.

DARBA KOPĒJIE REZULTĀTI, SECINĀJUMI UN TURPMĀKI PĒTĪJUMI

Promocijas darba mērķis bija izstrādāt modeļvadāmu pieeju un metodoloģiju konfigurācijas pārvaldības procesu automatizācijas ieviešanai, kas ļautu samazināt automatizācijas ieviešanas laiku un uzlabot automatizācijas procesa kvalitāti. Mērķa sasniegšanai tika veikti šādi soļi:

- izpētīti esošie risinājumi un pieejas konfigurācijas pārvaldības procesu automatizācijai, apzinātas galvenās problēmas un risinājumu attīstības tendences;
- identificēti galvenie ieguvumi un trūkumi jaunākajos konfigurācijas pārvaldības automatizācijas risinājumos, kas atbilst mūsdienīgām attīstības tendencēm;
- izstrādāta pieeja un metodoloģija konfigurācijas pārvaldības procesu automatizācijai, kas ir orientēta uz automatizācijas ieviešanas laika samazināšanu, atkārtoti izmantojot uzņēmumā esošos automatizācijas risinājumus;
- izstrādāts prototips jaunās metodoloģijas ieviešanas automatizācijai;
- izstrādāti kritēriji jaunās metodoloģijas novērtēšanai;
- ieviesta konfigurācijas pārvaldības automatizācija programmatūras izstrādes projektos un pēc izstrādātiem kritērijiem noteikti metodoloģijas ieguvumi un trūkumi;
- izstrādāta metodoloģijas uzlabotā versija, kas novērš eksperimentu rezultātā identificētos trūkumus;
- veikti atkārtoti eksperimenti un iegūti praktiski apliecinājumi tam, ka metodoloģijas uzlabotajā versijā ir novērsti sākotnēji identificētie trūkumi;
- balstoties uz eksperimentu rezultātiem, apkopoti jaunās metodoloģijas ieguvumi, ierobežojumi, ieviešanas riski, kā arī formulētas rekomendācijas metodoloģijas ieviešanai programmatūras izstrādes projektos;
- formulētas jaunās metodoloģijas iespējamie turpmākie attīstības un uzlabošanas virzieni.

Promocijas darbā izstrādāta metodoloģija, un visi jaunie modeļi tika eksperimentāli analizēti, lai varētu pārbaudīt izvirzītās hipotēzes. Eksperimentu rezultāti parādīja:

- pirmā hipotēze tika pierādīta, salīdzinot konfigurācijas pārvaldības automatizācijas ieviešanas laiku pēc vecajām metodēm, kad netika atkārtoti izmantoti esošie risinājumi, un automatizācijas ieviešanas laiku, atkārtoti lietojot jau esošos risinājumus. Eksperimentu rezultāti parādīja tendenci, ka, jaunajos projektos lietojot risinājumus, kas tika izmantoti citos projektos, var ieviest automatizāciju aptuveni divreiz ātrāk. Eksperimentos iegūtais vidējais rādītājs bija 65 %;
- otrā hipotēze tika pierādīta, salīdzinot konfigurācijas pārvaldības automatizācijas ieviešanas laiku pēc jaunās metodoloģijas dažādiem projektiem. Eksperimenti parādīja, ka sākotnēji, kad esošo risinājumu datubāze ir tukša un visus risinājumus jāveido no nulles, ieguvums ir tikai 9 procenti. Savukārt – jo ilgāk risinājumi eksistē datubāzē un attīstās, jo stabilākie tie kļūst, un automatizācijas ieviešanas laiks samazinās. Projektā, kurā tika ieviesta automatizācija, ieviešanas laiks samazinājās par 96 procentiem, salīdzinotar ieviešanu bez EAF modeļvadāmas pieejas. Tieši šajā projektā konfigurācijas pārvaldības automatizācijas ieviešanas brīdī risinājumu datubāzē bija visvairāk gatavu risinājumu, tie bija stabilāki, salīdzinot ar eksperimentu sākumu.

Veicot literatūras analīzi, strādājot ar dažādiem rīkiem konfigurācijas pārvaldības uzdevumu risināšanai un izstrādājot jaunu metodoloģiju, tika secināts:

- mūsdienās bieži konfigurācijas pārvaldības procesus definē nepilnīgi, akcentējot vien dažus uzdevumus, ko min nozares speciālisti, kvalitātes standarti un zinātniskie pētījumi. Piemēram, ir salīdzinoši daudz pētījumu par izejas koda versiju kontroles uzlabošanu. Šādi pētījumi dažreiz sniedz ļoti vērtīgas idejas par to, kā ar modeļvadāmu pieeju palielināt risinājumu efektivitāti, taču tajā pašā laikā citi svarīgi konfigurācijas pārvaldības uzdevumi tiek aizmirsti;
- vēl viena tendence, ko ir ievērojis promocijas darba autors: konfigurācijas pārvaldību mēdz uzskatīt vienkārši par rīku kopu. Dažreiz nozares speciālistiem rodas ilūzija, ka uzinstalējot rīkus, procesus var uzskatīt par ieviestiem un par tiem vairs nav jāuztraucas. Šāda nostāja radīja zaudējumus ne vienam vien projektam gan Latvijā, gan pasaulē. Gan no zinātniskā, gan no praktiskā viedokļa nav svarīgi, kādi rīki projektā ir lietoti, bet ir svarīgi, cik efektīvas ir rīku ieviešanas metodoloģijas, kas spētu efektīvi izvēlēties, nokonfigurēt rīkus, kā arī sniegt rekomendācijas, veicot konfigurācijas pārvaldības procesa darbības.

Promocijas darba autors uzskata, ka lielākais un interesantākais izaicinājums konfigurācijas pārvaldībā tuvākajos gados būs tieši disciplīnas stiprināšana no zinātniskā viedokļa. Tas izpaudīsies jaunu metodoloģiju izstrādē, kas ļaus procesā efektīvāk izmantot jau esošos, gadiem pārbaudītos risinājumus.

Darba rezultāti tika izmantoti zinātniskajos projektos un RTU priekšmeta «Lietišķo datorsistēmu programmatūra» mācību procesā:

- *LZP* projekts «Modeļu un metožu izstrāde lietišķai intelektuālai programmatūrai, pamatojoties uz izkliedētu mākslīgu intelektu, zināšanu pārvaldību un progresīvām tīmekļa tehnoloģijām» izpildē (vad. prof. J. Grundspeņķis); programmatūras pārvaldības modeļvadāmo metožu izstrāde;
- Eiropas Komisijas *7. IP* projektā *eINTERASIA* «*ICT Transfer Concept for Adaptation, Dissementation and Local Exploitation of European Research Results in Central Asia's Countries*», 2013–2015 (projekta koordinators – prof. Leonīds Novickis); programmatūras ietvara pārvaldības modeļu izstrāde;
- priekšmeta «Lietišķo datorsistēmu programmatūra» mācību procesā. Tika sagatavota mācību līdzekļa daļa (Metodiskie norādījumi priekšmetā «Lietišķo datorsistēmu programmatūra»/L. Novickis, V. Kotovs, A. Lesovskis, A. Bartusevičs», RTU, 2012. – 67 lpp,; Nodaļa: Lietišķās programmatūras konfigurācijas pārvaldība);
- valsts pētījumu projektā VVP Y8089 «Kiberfizikālās sistēmas, ontoloģijas un biofotonita drošai un vieglai pilsētai un sabiedrībai (no 2014. g.) – programmatūras konfigurācijas pārvaldība».

Promocijas darba turpmākie attīstības virzieni:

- ietvaru (*Framework*) versionēšanas sistēmas izstrāde. Praksē var rasties situācijas, kad kādam noteiktam projektam var būt nepiemērota ietvara jaunākā versija. Jābūt sistēmai, kas uzturēs ietvaru versijas un uzskatāmi parādīs atšķirības starp versijām. Papildus tam jābūt iespējai katram projektam brīvi izvēlēties ietvara versiju neatkarīgi no citiem projektiem;
- vižu sākotnējas instalācijas procesu formalizācija. Šobrīd *EAF* metodoloģija paredz, ka visas vides jau ir izveidotas. Taču reāli programmatūras izstrādes projekta sākumā vides veido no nulles: instalē operētājsistēmas, aplikāciju serverus, datubāzes, konfigurē ugunsmūrus utt. Šeit izpaužas līdzīga problēma, kas tika akcentēta šajā promocijas darbā, ka, ieviešot procesu vienā projektā,

jāprot to pēc iespējas ātrāk ieviest arī citā. Līdz ar to vižu sākotnējai konfigurācijas un instalācijas vadlīnijām un darbībām arī jāglabājas vienā struktūrā un jābūt vienotai procedūrai, kā jaunajos projektos maksimāli iespējami izmantot jau citu projektu risinājumus. Līdz ar to nepieciešana *EAF* metodoloģijas papildinājumi ar jauniem modeļiem, kas atvieglos jaunu vižu instalācijas un konfigurēšanas procesus;

- jānovērtē *EAF* metodoloģijas atbilstība populārākiem kvalitātes standartiem un vadlīnijām, piemēram, *CMMI, ISO, ITIL* u. c.;
- jāizstrādā vēl viens modelis, kas ļautu no *PIEM, PSAM* un *CM* modeļiem automātiski ģenerēt konfigurācijas pārvaldības plānu, kas ir neatņemama nepieciešamo projekta dokumentu sastāvdaļa, ko visbiežāk rakstiski apstiprina pasūtītājs. Savukārt konfigurācijas pārvaldības plāna manuāla veidošana izraisa risku, ka plāns nebūs atbilstošs konfigurācijas pārvaldības modeļiem;
- konfigurācijas pārvaldības modeļu un konfigurācijas pārvaldības problēmvides atgriezeniskās saites izstrāde. Jābūt mehānismam, kas analizē konfigurācijas pārvaldības problēmvidi un identificē vājās vietas konfigurācijas pārvaldības modeļos. Izstrādājot šo saiti, varētu izmantot mākslīgā intelekta metodes un veidot zināšanu bāzi, kurā būs likumi atgriezeniskās saites nodrošināšanai.

BIBLIOGRĀFISKAIS SARAKSTS

[ABO 2014] About CMMI Institute. 2014. [ONLINE] Available at: http://whatis.cmmiinstitute.com/about-cmmi-institute. [Apskatīts 02 Septembrī 2014].

[AIE 2010] Aiello, R. Configuration Management Best Practices: Practical Methods that Work in the Real World (1st ed.). Addison-Wesley, 2010.

[ALT 2008] Altmanninger K. Models in conflict – towards a semantically enhanced version control system for models. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2008;5002 LNCS:293-304.

[ALT 2010] Bruce Altner, Brett Lewinski. A Roadmap to Continuous Integration. Proceedings of the 2010 IT Summit, NASA, 2010.

[ASN 2010] Asnina, E. & Osis, J. 2010, «Computation independent models: Bridging problem and solution domains», Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modelling Theory-Driven Development, MDA and MTDD 2010, in Conjunction with ENASE 2010, pp. 23.

[AZO 2008] Azoff M., The Benefits of Model Driven Development. MDD in Modern Web-based Systems, Published March, Butler Direct Limited, 2008.

[AZO 2014] Azoff, R., DevOps: Advances in Release Management and Automation. [ONLINE] Available at: http://electric-cloud.com/wp-content/uploads/2014/06/EC-IAR_Ovum-DevOps.pdf [Apskatīts 20. oktobrī 2014].

[BAM 1995] Bamford, R., 1995. *Configuration Management and ISO 9001*. Software Systems Quality Consulting, DO-25 V6, 7., 1995.

[BAR 2012a] Bartusevics A., Kotovs V., Novickis L. A Method for Effective Reuse-Oriented Software Release Configuration and Its Application in Insurance Area. Proceedings of Riga Technical University «Information Tehnology and Management Science», 15th series, RTU Publishing, 2012, Riga, Latvia, pp. 111–115.

[BAR 2012b] Bartusevics A., Kotovs V. Towards the effective reuse-oriented release

configuration process. Proceedings of the 5-th International Scientific Conference «Applied Information and Communication Tehnologies», 2012, Jelgava, Latvia, pp. 99–103.

[BAR 2013] Bartusevics A., AMethodology for Model-Driven Software Configuration Management Implementation and Support. Proceedings of the 6-th International Scientific Conference «Applied Information and Communication Tehnologies», 2013, Jelgava, Latvia, pp. 252–258.

[BAR 2014a] Bartusevičs, A., Novickis, L. Model-Driven Software Configuration management and Environment Model. No: Recent Advances in Electrical and Electronic Engineering. Proceedings of the 3rd International Conference on Systems, Communications, Computers and Applications (CSCCA"14), Itālija, Florence, 22.-24. novembris, 2014. Italy: WSEAS Press, 2014, 132.-140.lpp. ISBN 978-960-474-399-5. ISSN 1790-5117.

[BAR 2014b] Bartusevičs, A., Novickis, L., Leye, S. Implementation of Software Configuration Management Process by Models: Practical Experiments and Learned Lessons. Applied Computer Systems. Nr.16, 2014, 26.–32. lpp. ISSN 2255-8683. e-ISSN 2255-8691. Pieejams: doi:10.1515/acss-2014-0010

[BAR 2014c] Bartusevičs, A., Novickis, L. Models for Implementation of Software Configuration Management. No: Procedia Computer Science. Valmiera, Latvia: 2014, 3.–10. lpp.

[BAR 2014d] Bartusevičs, A., Lesovskis, A., Novickis, L. Model-Driven Software Configuration Management and Semantic Web in Applied Software Development. Proceedings of the 13th International Conference on Telecommunications and Informatics (TELE-INFO '14), IIstanbul, Turkey December 15–17, 2014.

[BAR 2014e] Bartusevics, A., Novickis, L. Towards The Model-Driven Software Configuration Management Process. Iesniegts un akceptēts RTU 55. konferences krājumam.

[BAR 2014f] Bartusevics Arturs, Leonids Novickis and Eberhard Bluemel. 2014. Intellectual Model-Based Configuration Management Conception. Applied Computer Systems. 15(1): 5-41. Retrieved 28 Nov. 2014, from doi:10.2478/acss-2014-0003

[BAR 2015] Bartusevics, A., Novickis, L. Model-Based Approach for Implementation of Software Configuration Management Process. Iesniegts un akceptēts starptautiskās konferences MODELSWARD 2015 (http://www.modelsward.org/) rakstu krājumam.

[BEL 2005] Bellagio, M. What Is Software Configuration Management? Internet http://ptgmedia.pearsoncmg.com/images/0321200195/samplechapter/bellagio_ch01.pdf, 2005.

[BER 2003] Berczuk, Appleton. Software Configuration Management Patterns: Effective TeamWork, Practical Integration (1st ed.). Addison-Wesley, 2003.

[BER 2011] Berziša, S. & Grabis, J. 2011, «Combining project requirements and knowledge in configuration of project management information systems», ACM International Conference Proceeding Series, pp. 89.

[BER 2012] Bērziša, Solvita. Application of Knowledge and Best Practices in Configuration of Project Management Information Systems : promocijas darbs / S. Bērziša; zinātniskais vadītājs J. Grabis; Rīgas Tehniskā universitāte. DATORZINĀTNES UN INFORMĀCIJAS TEHNOLOĢIJAS FAKULTĀTE. Informācijas tehnoloģijas institūts. Vadības informācijas tehnoloģijas katedra. Rīga: [RTU], 2012. – 196 pp.

[BIL 2014] Bill Chamberlin's HorizonWatching. 2014. Top 18 Trends in Application Software Development for 2014. [ONLINE] Available at: http://www.billchamberlin.com/top-18-trends-in-application-software-development-for-2014/. [Apskatīts 20. oktobrī 2014].

[BRA 2008] Bastian Braun. SAVE – Static Analysis on Versioning Entities. ICSE: International Conference on Software Engineering, 2008.

[BRO 2002] Brouse, Peggy S. Configuration Management Interenet: http://www.eolss.net/Sample-Chapters/C15/E1-28-03-02.pdf , 2002.

[BRO 2005] A. B. Brown, A. Keller, and J. L. Hellerstein, A model of configuration complexity and its application to a change management system, IFIP/IEEE International Symposium, Integrated Network Management, pp. 631–644, 2005.

[BRU 2004] Brugge, B., Dutoit, A. Software Configuration Management. Internet https://files.ifi.uzh.ch/rerg/amadeus/teaching/courses/software_engineering

_hs08/folien/Kapitel_23_Addendum_SCM.pdf, 2004.

[BUC 2009] Buchmann T., Dotor A., Westfechtel B. MODEL-DRIVEN DEVELOPMENT OF SOFTWARE CONFIGURATION MANAGEMENT SYSTEMS. ICSOFT 2009 – 4th International Conference on Software and Data Technologies 2009.

[BUS 2011] Bushehrian O., Automatic object deployment for software performance enhancement. The Institution of Engineering and Technology 2011, Vol. 5, Iss. 4, pp. 375–384, 2011.

[CAL 2012] Calhau R., Falbo R. A Configuration Management Task Ontology for Semantic Integration. Proceedings of the 27th Annual ACM Symposium on Applied Computing Pages 348-353 ACM New York, NY, USA, 2012.

[CLE 2012] Clemencic M., Mato P., A CMake-based build and configuration framework. Journal of Physics: Conference Series 396 (2012) 052021, 2012.

[CMC 2014] CMCrossroads | Three Major Trends in Software Release Management You Should Adopt . 2014. [ONLINE] Available at: http://www.cmcrossroads.com/article/three-major-trends-software-release-management-you-should-adopt. [Apskatīts 20. oktobrī 2014].

[COM 2011] Comas J., Mostashari A., Mansouri M., Turner R. A Software Deployment Risk Assessment Heuristic for Use in a Rapidly-Changing Business-to-Consumer Web Environment International Journal of Software Engineering and Its Applications Vol. 5 No. 4, October, 2011.

[CON 2002] Configuration Management Training. Section 1: Explaining Configuration Management, EESA, 2002. Internet: http://esamultimedia.esa.int/docs/industry/SME/Configuration/Section_1-CM.pdf

[CON 2015] The Convergence of DevOps «IT Revolution IT Revolution. [ONLINE] Available at: http://itrevolution.com/the-convergence-of-devops/. [Apskatīts 28. janvārī 2015].

[CRA 2008] Cravino P., Enterprise Software Configuration Management Solutions for Distributed and System z. 1st ed. USA: Redbooks. 2008.

[DAR 2001] Dart, S. Concepts in Configuration Management Systems. Internet http://sceweb.uhcl.edu/boetticher/swen5230/concepts-in-configuration-

management.pdf, 2001.

[DEP 2010] Department of Defense, USA Military Handbook. Configuration management guidance (rev. A) (MIL-HDBK-61A). Retrieved January 5, 2010, from http://www.everyspec.com/MIL-HDBK/MIL-HDBK-0001-0099/MIL-HDBK-61_11531/

[DEV 2014] DevOps Implementation | Giga Promoters. 2014. [ONLINE] Available at: http://gigapromoters.com/offerings/services/it-services/devops-implementation/. [Apskatīts 10. novembrī 2014].

[DOD 2014] Do DevOps tools really exist?. 2014. Do DevOps tools really exist?. [ONLINE] Available at: http://www.scriptrock.com/blog/devops-tools-exist. [Apskatīts 11. novembrī 2014].

[DON 2011] Doniņš Uldis. Topoloģiskā biznesa sistēmu modelēšana un programmatūras sistēmu projektēšana. Metodiskais līdzeklis. – RTU Izdevniecība. Rīga 2011.

[EIL 2006] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, and J. Pershing, «Managing the Configuration Complexity of Distributed Applications in Internet Data Centers,» IEEE Communications Magazine, vol. 44, pp. 166–177, 2006.

[EST 2013] Estler, H.-Christian, Unifying Configuration Management with Merge Conflict Detection and Awareness Systems. In 22nd Australian Conference on Software Engineering. Austalia, 4–7 June 2013. Austalia: IEEE. 201–210.

[FIT 2014] Fitzgerald B., Stol J., Continuous software engineering and beyond: trends and challenges. Proceeding in RCoSE 2014 Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, Pages 1–9. ACM New York, NY, USA, 2014.

[FUG 2014] Fuggetta A., Nitto E., Software process. Proceeding in FOSE 2014 Proceedings of the on Future of Software Engineering, Pages 1–12, ACM New York, NY, USA, 2014.

[GAL 2009] Galup, S. D., Dattero, R., Quan, J. J., Conger, S., An Overview of IT Service Management. *Commun.* ACM, 2009, vol. 52, no. 5, pp. 124–127., 2009.

[GHE 2012] Giacomo Ghezzi, Michael Würsch, Emanuel Giger, Harald Gall. An

Architectural Blueprint for a Pluggable Version Control System for Software (Evolution) Analysis, In: 2nd Workshop on Developing Tools as Plug-ins, Zurich, 03 June 2012–03 June 2012.

[GIE 2009] Giese Holger, Seibel Andreas, Vogel Thomas. A Model-Driven Configuration Management System for Advanced IT Service Management. Available at: http://www.hpi.unipotsdam.de/giese/gforge/publications/pdf/GSV-MRT09_paper_7.pdf, 2009.

[GLO 2012] IT Glossary. Defining The IT Industry. SCM Software Configuration Management. Internet http://www.gartner.com/it-glossary/scm-software-configuration-management/, 2012.

[GRO 2007] H. Gronniger, H. Krahn, B. Rumpe, M. Schindler, and S. Volkel. Textbased Modeling. In 4th International Workshop on Software Language Engineering, 2007.

[GUO 2005] Guozheng Ge, E., Whitehead, Jr. Automatic Generation of Rule-based Software Configuration Management Systems. ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

[HAG 2010] Hagen, S., Kemper, A., Model-Based Planning for State-Related Changes to Infrastructure and Software as a Service Instances in Large Data Centers. Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, On page(s): 11–18, Volume: Issue: , 5–10 July 2010

[HAT 2012] Hideaki Hata, Osamu Mizuno, Tohru Kikuno. Bug Prediction Based on Fine-Grained Module Histories. ICSE: International Conference on Software Engineering, Feb 2012, p 200–210.

[HIS 2014] History of software configuration management – Wikipedia, the free encyclopedia. 2014. [ONLINE] Available at: http://en.wikipedia.org/wiki/History_of_software_configuration_management. [Apskatīts 5. novembrī 2014].

[HIST 2014] A History of Version Control. [ONLINE] Available at: http://ericsink.com/vcbe/html/history_of_version_control.html. [Apskatīts 11. novembrī 2014].

[HUA 2009] Shi-Ming Huang, Chih-Fong Tsai, Po-Chun Huang. Component-based

software version management based on a Component-Interface Dependency Matrix, The Journal of Systems and Software, 2009.

[IEE 2015] IEEE SA - 12207-2008 - Systems and software engineering -- Software life cycle processes. 2015. [ONLINE] Available at: https://standards.ieee.org/findstds/standard/12207-2008.html. [Apskatīts 1. jūlijā 2015].

[ISO 2015] ISO 9001:2008 - Quality management systems -- Requirements . 2015. [ONLINE] Available at: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=46486. [Apskatīts 1. jūlijā 2015].

[JIA 2009] Jiang, L., Eberlein, A., An Analysis of the History of Classical Software Development and Agile Development. Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics San Antonio, TX, USA – October 2009.

[JOH 2011] Johnsen E., Schlatte R. Integrating Aspects of Software Deployment in High-Level Executable Models, presented at the NIK-2011 conference, 2011.

[JUI 2002] Juite Wanga, Yung-I Lin, A fuzzy multicriteria group decision making approach to select configuration items for software development. MathematicsWEB, Fuzzy Sets and Systems, 2002.

[KAN 2005] Ronald Kirk Kandt. Configuration Management Principles and Practices. Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, CA 91 109, USA Internet: http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/10507/1/02-2525.pdf , 2005.

[KAP 2008] Kapitza R, Baumann P, Reiser HP. Using object replication for building a dependable version control system. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2008;5053 LNCS:86-99.

[KAR 2009] G. Karsai, H. Krahn, C. Pinkernell. Design Guidelines for Domain Specific Languages. Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling DSM'09, page 7–13., 2009.

[KEL 2008] S. Kelly and J.-P. Tolvanen. Domain-Specific Modeling: Enabling Full Code Generation. Wiley, 2008.

[KR 2014] Krusche S., Alperowitz L., Introduction of continuous delivery in multi-customer project courses. Proceeding in ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering, Pages 335–343. ACM New York, NY, USA, 2014.

[LAV 2011] Jannik Laval, Simon Denier, Stéphane Ducasse, Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment. Science of Computer Programming, 2011.

[LES 2014] Lesson 11: Devops & Configuration Management Intro – OSU DevOps Bootcamp 0.0.1 documentation. 2014. [ONLINE] Available at: http://devopsbootcamp.readthedocs.org/en/latest/11_devops.html. [Apskatīts 10. novembrī 2014].

[LI 2012] Jingyue Li, Michael D. Ernst. CBCD: Cloned Buggy Code Detector. ICSE: International Conference on Software Engineering, Feb 2012, p 310–320.

[MAL 2012] Malek S. An Extensible Framework for Improving a Distributed Software System's Deployment Architecture. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 38, NO. 1, JANUARY/FEBRUARY 2012.

[MEL 2006] Mellon, K. A Framework for Software Product Line Practice, Version 5.0. Internet: http://www.sei.cmu.edu/productlines/frame_report/config.man.htm, 2006.

[MER 2005] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. Technical Report SEN-E0309, Centrum voor Wiskunde en Informatica, Amsterdam, 2005.

[MET 2002] Anne Mette Jonassen Hass. Configuration Management Principles and Practice, Addison-Wesley Professional. Part of the Agile Software Development Series series, 2002, pages 432.

[MUR 2008] Leonardo Murta, Chessman Correa, Joao Gustavo Prudencio. Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System. ICSE: International Conference on Software Engineering, 2008.

[NIK 2008] Nikulsins, V. & Nikiforova, O. 2008, «Adapting software development process towards the model driven architecture», Proceedings – The 3rd International Conference on Software Engineering Advances, ICSEA 2008, Includes ENTISY 2008: International Workshop on Enterprise

Information Systems, pp. 394.

[NIK 2009] Nikiforova, O., Cernickins, A. & Pavlova, N. 2009, «Discussing the difference between model driven architecture and model driven development in the context of supporting tools the projection of two-hemisphere model into the component model of MDA/MDD», 4th International Conference on Software Engineering Advances, ICSEA 2009, Includes SEDES 2009: Simposio para Estudantes de Doutoramento em Engenharia de Software, pp. 446.

[OPE 2014] OpenMake Products. [ONLINE] Available at: http://www.openmakesoftware.com/build-management. [Apskatīts 22. novembrī 2014].

[OSE 2002] Oject-Oriented Software Engineering Using UML, Patters and JAVA «Software Configuration Management» Internet: http://www.bilkent.edu.tr/~bakporay/cs_413/Bruegge_L28_Configuration Management_ch12lect1.ppt, 2002

[OSI 2008] Osis, J., Asnina, E. & Grave, A. 2008, Formal problem domain modeling within MDA. Proceedings of the 2nd International Conference on Software and Data Technologies, ICSOFT 2007; Barcelona; Spain; Volume 22 CCIS, 2008, Pages 387–398.

[OSI 2010] Osis, J. & Donins, U. 2010, «Platform independent model development by means of topological class diagrams», Proceedings of the 2nd International Workshop on Model-Driven Architecture and Modelling Theory-Driven Development, MDA and MTDD 2010, in Conjunction with ENASE 2010, pp. 13.

[OSI 2011] Osis J., Asnina E. Model-Driven Domain Analysis and Software Development: Architectures and Functions. IGI Global, Hershey – New York, 2011, 514 p.

[PAI 1999] R. Paige, J. Ostroff, and P. Brooke. Principles for Modeling Language Design. Technical Report CS-1999-08, York University, December 1999.

[PAU 2007] Paul M. Duvall, Steve Matyas, and Andrew Glover. Continuous Integration: Improving Software Quality and Reducing Risk. (1st ed.). Addison-Wesley Professional, 2007.

[PFA 1997] P. Pfahler and U. Kastens. Language Design and Implementation by

Selection. In Proc. 1st ACM-SIGPLAN Workshop on Domain-Specific-Languages, DSL ’97, pages 97–108, Paris, France, January 1997. Technical Report, University of Illinois at Urbana-Champaign.

[PIN 2009] Pindhofer Walter, Model Driven Configuration Management. Master work of Wien University, Wien, 2009.

[RAG 2014] Ragan, T., 21st-Century DevOps--an End to the 20th-Century Practice of Writing Static Build and Deploy Scripts, Linux Journal, 230, pp. 116–120, Computers & Applied Sciences Complete, EBSCOhost, viewed 22 October 2014.

[RAZ 2007] Saad Razzaq, Fahad Maqbool, Bilal Anjum. The Challenges & Case for Mining Software Repositories. International MultiConference of Engineers and Computer Scientists, 2007.

[ROS 2010] Alessandro Rossini, Adrian Rutle, Yngve Lamo, UweWolter. A formalisation of the copy-modify-merge approach to version control in MDE. The Journal of Logic and Algebraic Programming, 2010.

[RUA 2003] Ruan Li, Zhong Yong, A New Configuration Management Model for Software Based on Distributed Components and Layered Architecture. Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. 27–29 Aug. 2003.

[SAR 2008] Anita Sarma, David Redmiles, André van der Hoek. Empirical evidence of the benefits of workspace awareness in software configuration management. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, 2008

[SAT 2011] Laika Satish, Identifying the Dissimilarities based on Working of Programs among Versions in DVCS (Distributed Version Control Systems). International Journal of Computer Applications (0975 – 8887) Volume 36– No. 6, December 2011.

[SCH 2010] Holger Schackmann, Horst Lichter. Process assessment by evaluating configuration and change request management systems. Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010

[SCM 2001] Software Configuration Management Internet: http://dogbert.mse.cs.cmu.edu/charlatans/References/Configuration%20Management/0130912972.pdf, 2001.

[SEK 2012] Atsuji Sekiguchi, Kuniaki Shimada, Yuji Wada, Akio Ooba, Ryouji Yoshimi, Akiko Matsumoto. Configuration management technology using tree structures of ICT systems. Proceedings of the 15th Communications and Networking Simulation Symposium
Publisher: Society for Computer Simulation International, 2012

[SER 2014] Serena Deployment Automation Overview. Serena Deployment Automation Overview. [ONLINE] Available at: http://www.serena.com/index.php/en/products/featured-products/serena-deployment-automation/overview/. [Apskatīts 22. novembrī 2014].

[SHI 2010] Shih C., Huang S. Exploring the relationship between organizational culture and software process improvement deployment, Information & Management 47 (2010) 271–281p., 2010.

[SIN 2008] Sindhuja P. N., Surajit Ghosh Dastidar. Software Deployment: Concepts and Technologies. ICFAI Journal of Systems Management, 2008.

[SIN 2010] Sinan Si Alhir. Understanding the Model Driven Architecture (MDA). Available at: http://www.methodsandtools.com/archive/archive.php?id=5, 2010.

[SIY 2008] Harvey Siy, Parvathi Chundi, Daniel J. Rosenkrantz, Mahadevan Subramaniam. A segmentation-based approach for temporal analysis of software version repositories. Jornal of Software Maintenance and Evolution: Research and Practice, 2008.

[SOF 2014] Software configuration management - Wikipedia, the free encyclopedia. 2014. [ONLINE] Available at: http://en.wikipedia.org/wiki/Software_configuration_management. [Apskatīts 5. novembrī 2014].

[STA 2008] Glen Stansberry. 7 Version Control Systems Reviewed. At http://www.smashingmagazine.com/2008/09/18/the-top-7-open-source-version-control-systems/, 2008.

[TAK 2014] Taking Release Management to the Next Level. 2014. [ONLINE] Available at: http://www.slideshare.net/xebialabs/taking-releasemanagementtothenextlevel. [Apskatīts 20. oktobrī 2014].

[TAR 2011] Alexander Tarvo, Thomas Zimmermann, Jacek Czerwonka. An integration resolution algorithm for mining multiple branches in version control

systems. IEEE international conference on software maintenance, ICSM; 2011. 402 p.

[THA 2009] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, Tien N. Nguyen. Clone-Aware Configuration Management. ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009.

[TOL 2005] J. Tolvanen, S Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. Proceeding SPLC'05 Proceedings of the 9th international conference on Software Product Lines Pages 198–209. 2005.

[TRE 2014] Trends in Software Engineering - Dice News. 2014. [ONLINE] Available at: http://news.dice.com/software-engineering-talent-community/trends/. [Apskatīts 20. oktobrī 2014].

[VAC 2006] VACCAPERNA Systems Limited. Software Configuration Management (SCM). Internet http://www.vaccaperna.co.uk/scm/about_scm.html, 2006.

[VAS 2013] Vasiljevics, I., Milosavljevics, G., Dejanovics, I., Filipovics, M., COMPARISON OF GRAFICAL DSL EDITORS. The 6th PSU-UNS International Conference on Engineering and Technology (ICET-2013), Novi Sad, Serbia, May 15–17, 2013.

[WET 2012] Wettinger J., Concepts for Integrating DevOps Methodologies with Model-Driven Cloud Management Based on TOSCA. Institute of Architecture of Application Systems University of Stuttgart, 2012.

[WHA 2014] What Are Current Hot Trends In The Field Of Software Engineering?. 2014. [ONLINE] Available at: http://bloggless.com/it/software-engineering/what-is-currently-popular-in-software-engineering/. [Apskatīts 20. oktobrī 2014].

[WIL 2003] D. Wile. Lessons Learned from Real DSL Experiments. Proceedings of the 36th Hawaii International Conference on System Sciences, 2003.

[WIL 2004] D. Wile. Lessons learned from real DSL experiments. Science of Computer Programming, 51(3):265–290, June 2004.

[WES 2005] Westfechtel, B., Conradi, R. Software Architecture and Software Configuration Management Internet

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.3085&rep=rep1&type=pdf, 2005.

[ЗАМ 2008] Заметки о Software Configuration Management. Управление конфигурацией программного обеспечения. Интернет: http://scm-notes.blogspot.com/p/scm-books.html, 2008.

[ЛАП 2004] Лапыгин, Д. Новичков, А. Конфигурационное управление проектами разработки программного обеспечения. Интернет: http://citforum.ru/SE/quality/configuration_management/, 2004.

[ОРЛ 2011] Орлик, С. Программная инженерия. Конфигурационное управление Перевод главы из SWEBOK с комментариями. Архивировано из первоисточника 14 марта 2012. Проверено 18 июня 2011.

[УДО 2011] Удовиченко, Ю. Управление изменениями и кессонная болезнь проектов. Интернет: http://experience.openquality.ru/software-configuration-management/, 2011.

PIELIKUMI

1. pielikums

Projekta «*Test Solution*» vižu modelis *XML* formātā

```
<?xml version=”1.0” encoding=”UTF-8”?>
<EM_Model>
  <Name=”Testa piemērs” />
  <Description=”Modelis apraksta testa projektu, kur programmatūru izstrādā, testē un nodod akcepttestēšanai pasūtītājam” />
  <Events>
    <Event>
      <Name=”dev” />
      <Description=”Izmaiņu izstrāde DEV vidē un to saglabāšana izejas koda repozitorijā” />
      <AllChangesMoveFlag=”Y” />

      <ConfigurationItemFlows>
        <ConfigurationItemFlow>
          <Name=”Izmaiņu izstrāde un saglabāšana” />
          <Sequence=”1” />
          <OwnerEnvironment=”Programmētājs X” />
          <GoalEnvironment=”DEV” />
          <Description=”Programmētājs veic izmaiņas izstrādes vidē un veicot vienībtestus saglabā izmaiņas versiju kontroles sistēmā” />
        </ConfigurationItemFlow>
      </ConfigurationItemFlows>
    </Event>
    <Event>
      <Name=”test” />
      <Description=”Konfigurācijas pārnešana no izstrādes (DEV) uz testa (TEST) vidi ar nosacījumu, ka izmaiņas piefiksētas izejas koda repozitorijā” />
      <AllChangesMoveFlag=”Y” />

      <ConfigurationItemFlows>
        <ConfigurationItemFlow>
          <Name=”Konfigurācijas pārnešanas notestēšana” />
          <Sequence=”1” />
          <OwnerEnvironment=”DEV” />
          <GoalEnvironment=”Pre_TEST” />
          <Description=”Integrē izmaiņas TEST zarā (merge), uzbūvē produktu un uzinstalē to Pre_TEST vidē” />
        </ConfigurationItemFlow>
        <ConfigurationItemFlow>
          <Name=”Konfigurācijas pārnešana īstajā testa vidē” />
          <Sequence=”2” />
          <OwnerEnvironment=”DEV” />
          <GoalEnvironment=”TEST” />
          <Description=”Paņem būvējumu no iepriekšējas plūsmas un uzinstalē to TEST vidē” />
        </ConfigurationItemFlow>
      </ConfigurationItemFlows>
    </Event>
    <Event>
      <Name=”qa” />
      <Description=”Piegādā tikai notestētas izmaiņas pasūtītājam akceptestēšanai. Pasūtītājs uzinstalē izmaiņas savā akcepttestiem paredzētajā vidē” />
      <AllChangesMoveFlag=”N” />
      <ConfigurationItemFlows>
        <ConfigurationItemFlow>
          <Name=”Izvēlētas konfigurācijas pārnešana no TEST vides uz QA vidi, kas atrodas pie pasūtītāja” />
          <Sequence=”1” />
```

```
                                        <OwnerEnvironment=”TEST” />
                                        <GoalEnvironment=”QA” />
                                        <Description=”Notestētu izmaiņu piegāde klientam” />
                                    </ConfigurationItemFlow>
                                </ConfigurationItemFlows>
                        </Event>
                </Events>
                <Actors>
                        <Actor>
                                <Name=”Programmētājs X” />
                                <Description=”Programmētājs, kas veic izmaiņas programmatūrā” />
                        </Actor>
                </Actors>
                <Environments>
                        <Environment>
                                <Name=”DEV” />
                                <Description=”Izstrādes vide” />
                                <Support_by_customer_flag=”N” />
                                <Development_environment_flag=”Y” />
                                <original_environment_flag=”Y” />
                                <original_environment_name=”” />
                        </Environment>
                        <Environment>
                                <Name=”Pre_TEST” />
                                <Description=”Fiktīva vide, kur notestē konfigurācijas pārnešanas procesu uz
īstu testa vidi” />
                                <Support_by_customer_flag=”N” />
                                <Development_environment_flag=”N” />
                                <original_environment_flag=”N” />
                                <original_environment_name=”TEST” />
                        </Environment>
                        <Environment>
                                <Name=”TEST” />
                                <Description=”Oriģināla testa vide” />
                                <Support_by_customer_flag=”N” />
                                <Development_environment_flag=”N” />
                                <original_environment_flag=”Y” />
                                <original_environment_name=”” />
                        </Environment>
                        <Environment>
                                <Name=”QA” />
                                <Description=”Pasūtītāja akcepttesta vide” />
                                <Support_by_customer_flag=”Y” />
                                <Development_environment_flag=”N” />
                                <original_environment_flag=”Y” />
                                <original_environment_name=”” />
                        </Environment>
                </Environments>
        </EM_Model>
```

Projekta «*Test Solution*» *PIAM* modelis *XML* formātā

```
<?xml version="1.0" encoding="UTF-8"?>
<PIAM_Model>
	<ContinuousIntegrationServer>
		<Platform="value" />
		<ToolName="value" />
		<InstallationNotes="value" />
		<LocationsOfSolutions="value" />
		<Events>
			<Event>
				<Name="dev" />
				<ConfigurationItemFlows>
					<ConfigurationItemFlow>
						<Sequence="1" />
						<Actions>
							<!-- E->P number: 1  -->
							<Action  name="DEVELOPMENT"/>
							<Action  name="COMMIT_CHANGES"/>
						</Actions>
					</ConfigurationItemFlow>
				</ConfigurationItemFlows>
			</Event>
			<Event>
				<Name="test" />
				<ConfigurationItemFlows>
					<ConfigurationItemFlow>
						<Sequence="1" />
						<Actions>
							<!-- E->P number: 7  -->
							<Action
name="PREPARE_BASELINE"/>
							<Action  name="COMPILE_BUILD"/>
							<Action  name="INSTALL_BUILD"/>
						</Actions>
					</ConfigurationItemFlow>
					<ConfigurationItemFlow>
						<Sequence="2" />
						<Actions>
							<!-- E->P number: 8  -->
							<Action  name="INSTALL_BUILD"/>
						</Actions>
					</ConfigurationItemFlow>
				</ConfigurationItemFlows>
			</Event>
			<Event>
				<Name="qa" />
				<ConfigurationItemFlows>
					<ConfigurationItemFlow>
						<Sequence="1" />
						<Actions>
							<!-- E->P number: 10  -->
							<Action
name="PREPARE_BASELINE"/>
							<Action  name="COMPILE_BUILD"/>
							<Action
name="PRODUCT_DELIVERY"/>
							<Action
name="ENV_UPDATE_NOTIFICATION"/>
						</Actions>
```

```
                                        </ConfigurationItemFlow>
                                </ConfigurationItemFlows>
                        </Event>
                </Events>
                <Actions>
                        <Action  name="DEVELOPMENT">
                                <PlatformName="value" />
                                <SolutionName="value" />
                                <NeededTools="value" />
                                <LocationsOfSolutions="value" />
                                <Description="value" />
                        </Action>
                        <Action  name="COMMIT_CHANGES">
                                <PlatformName="value" />
                                <SolutionName="value" />
                                <NeededTools="value" />
                                <LocationsOfSolutions="value" />
                                <Description="value" />
                        </Action>
                        <Action  name="PREPARE_BASELINE">
                                <PlatformName="value" />
                                <SolutionName="value" />
                                <NeededTools="value" />
                                <LocationsOfSolutions="value" />
                                <Description="value" />
                        </Action>
                        <Action  name="COMPILE_BUILD">
                                <PlatformName="value" />
                                <SolutionName="value" />
                                <NeededTools="value" />
                                <LocationsOfSolutions="value" />
                                <Description="value" />
                        </Action>
                        <Action  name="INSTALL_BUILD">
                                <PlatformName="value" />
                                <SolutionName="value" />
                                <NeededTools="value" />
                                <LocationsOfSolutions="value" />
                                <Description="value" />
                        </Action>
                        <Action  name="PRODUCT_DELIVERY">
                                <PlatformName="value" />
                                <SolutionName="value" />
                                <NeededTools="value" />
                                <LocationsOfSolutions="value" />
                                <Description="value" />
                        </Action>
                        <Action  name="ENV_UPDATE_NOTIFICATION">
                                <PlatformName="value" />
                                <SolutionName="value" />
                                <NeededTools="value" />
                                <LocationsOfSolutions="value" />
                                <Description="value" />
                        </Action>
                </Actions>
        </ContinuousIntegrationServer>
</PIAM_Model>
```